Descripción y Análisis Formal del Modelo de Seguridad de Android

Tesina de grado presentada por

Agustín Vicente Romano
agustinr88@gmail.com
R-2808/8

al
Departamento de Ciencias de la Computación
en cumplimiento parcial de los requerimientos
para la obtención del grado de

Licenciado en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Av. Pellegrini 250, Rosario, República Argentina

Junio de 2014

Director

Carlos Luna
cluna@fing.edu.uy

Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Julio Herrera y Reissig 565, Montevideo, Uruguay

Co-directores

Gustavo Betarte
gustun@fing.edu.uy

Juan Diego Campo
jdcampo@fing.edu.uy

Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Julio Herrera y Reissig 565, Montevideo, Uruguay

Dedicado a la memoria de Juanjo Priotti

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 lo que una falla en la seguridad de dicha plataforma afectaría a una gran cantidad de usuarios. Este elevado número de víctimas potenciales alienta a los creadores de aplicaciones maliciosas a elegir a Android como objetivo de sus ataques. Es por esto que el análisis y fortalecimiento de su modelo de seguridad se ha convertido en una tarea importante que despierta el interés de numerosos investigadores.

El objetivo de este trabajo es realizar un análisis exhaustivo del modelo de seguridad implementado por Android. Para ello se realiza un estado del arte en el tema, considerando los trabajos más relevantes hasta el momento, y se compara dicho modelo con el implementado en los dispositivos móviles Java (JME-MIDP). Asimismo, se presenta una especificación formal que comprende distintos aspectos sobre la seguridad en Android, poniendo atención, principalmente, en el mecanismo de delegación de permisos y en la interacción con el *framework* de aplicaciones para realizar llamadas al sistema. Dicha especificación está desarrollada con el asistente de pruebas Coq, que es utilizado para demostrar formalmente diferentes propiedades sobre el modelo de seguridad representado.

Índice General

Índice de Figuras

Capítulo 1

Introducción

Android es un sistema operativo *open-source* [21] diseñado, inicialmente, para dispositivos móviles y desarrollado por Google junto con la Open Handset Alliance (OHA) [44]. 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" [42].

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.

Con respecto al modelo de seguridad implementado en Android, a pesar de que el mismo posee características muy variadas, también presenta ciertas limitaciones. Algunos trabajos previos dan cuenta de la rigidez del sistema de permisos a la hora de, por ejemplo, instalar una nueva aplicación [31, 43]. 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 denominado *privilege escalation*, abordado en trabajos anteriores [28, 36] y retomado en el presente trabajo) pero, al mismo tiempo, no existe una documentación precisa que facilite dicha tarea. Adicionalmente, el mecanismo de delegación de permisos ofrecido en dicho modelo presenta características que requieren un mayor análisis, con el objeto de asegurar que no agregan nuevas vulnerabilidades (todavía no descubiertas).

En este trabajo se desarrolla una especificación formal del modelo de seguridad de Android poniendo especial atención en el mecanismo de delegación de permisos y la interacción entre las aplicaciones y el sistema. Dicha especificación, inspirada en los trabajos de Zanella *et al.* [52, 53], está basada en máquinas de estados. Los estados incluyen, en particular, campos para: aplicaciones instaladas, permisos otorgados a las mismas y componentes en ejecución. Dado un estado de la especificación formal, 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, el campo correspondiente a las aplicaciones instaladas contemplará la actualización de dicho campo con la incorporación de la nueva aplicación). De esta forma, el objetivo del trabajo es analizar formalmente distintos aspectos del modelo de seguridad de Android, demostrando propiedades del mismo en el marco de la especificación construida. Tanto la especificación formal como el enunciado y prueba de teoremas sobre el mismo se desarrollaron en Coq [27, 51], por lo que los archivos generados con dicho asistente de pruebas, disponibles en `http://www.fing.edu.uy/inco/grupos/gsi/index.php?page=proygrado&locale=es`, constituyen una parte primordial de este trabajo.

En concreto, esta tesina realiza las siguientes contribuciones:

- Una descripción exhaustiva del modelo de seguridad de Android que incluye un estudio del estado del arte en el análisis del mismo[1].
- Una especificación formal de las definiciones básicas del sistema Android y sus mecanismos de seguridad.
- Un análisis formal de propiedades reunidas de distintas publicaciones del estado del arte sobre la seguridad en Android.
- Demostraciones de nuevas propiedades no estudiadas hasta el momento que describen escenarios omitidos en la documentación oficial, ayudando, de esta forma, a clarificar el verdadero comportamiento del modelo de seguridad en Android.

De las publicaciones disponibles en la literatura, las que más se relacionan con nuestro trabajo son las de Fragkaki *et al.* [37] y Shin *et al.* [48]. En la primera de ellas se desarrolla un modelo formal para analizar los mecanismos de seguridad que implementa Android. En base a este modelo se especifican propiedades deseables para dichos mecanismos y se verifica si las mismas se cumplen. Parte de las propiedades descriptas en este trabajo fueron tomadas para ser demostradas en nuestro propio formalismo, principalmente las relacionadas con la delegación de permisos. Por otro lado,

[1] Parte de los resultados obtenidos en esta etapa fueron plasmados en un *reporte técnico* [47] publicado en la Universidad de la República.

Shin *et al.* también toman como base el trabajo de Zanella *et al.* [53] para analizar el modelo de seguridad de Android mediante un formalismo basado en máquinas de estados desarrollado en Coq. Sin embargo, nuestra especificación cubre aspectos no estudiados en la publicación en cuestión, tales como la delegación de permisos y la interacción con el sistema. Por otro lado, las definiciones utilizadas en ambos trabajos también difieren en gran medida, por lo que, en lugar de ser una extensión a lo hecho anteriormente, la especificación propuesta es más bien una alternativa.

La organización del resto del trabajo es como sigue. El capítulo 2 introduce las características principales de Android, incluyendo su arquitectura y conceptos básicos, y el capítulo 3 ofrece una descripción exhaustiva de su modelo de seguridad junto con un análisis comparativo entre dicho modelo y el implementado en los dispositivos móviles Java (JME-MIDP) [38, 46]. Posteriormente, en los capítulos 4 y 5 se incluyen, respectivamente, la especificación formal del modelo de seguridad de Android y las propiedades demostradas sobre el mismo. Por último, los capítulos 6 y 7 presentan trabajos relacionados, incluyendo las diferencias concretas entre nuestro trabajo y el de Shin *et al.* [48], y las conclusiones obtenidas en esta tesina.

Capítulo 2

Descripción General de Android

A lo largo de este capítulo explicaremos las características principales de Android, incluyendo su arquitectura y conceptos básicos, que se necesitarán para comprender el resto del trabajo. Dado que la versión de Android estudiada es la 4.4.2 (KitKat), tanto el presente capítulo como los posteriores se referirán a esta versión.

2.1. 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 [32, 50].

Los distintos estratos en la arquitectura de Android se describen en la figura 2.1 y, a lo largo de las secciones subsiguientes, se resumen las principales características de cada uno de ellos.

2.1.1. Kernel Linux

"Este estrato funciona como una capa de abstracción entre el hardware y el resto de los componentes del sistema" [42] 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

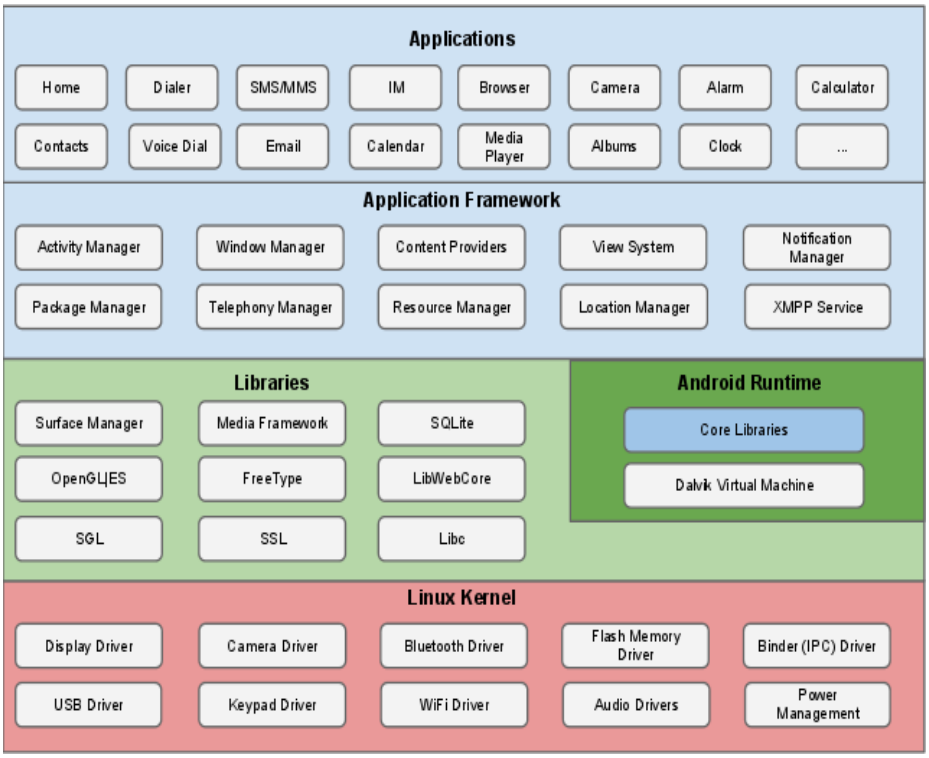

Figura 2.1: Descripción de la arquitectura de Android publicada en *Android Open Source Project* [20].

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.

2.1.2. 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 [42, 50].

- *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" [42]. 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 [3, 50].

- *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" [42].

2.1.3. 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" [34]).

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 [42]. 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.

Otra característica del framework de Android es que "los desarrolladores tienen acceso a las mismas APIs que utilizan las aplicaciones principales" [42] (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 [35, 42]. 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 los capítulos subsiguientes.

2.1.4. 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 [42]. 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 [35].

2.2. 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" [5]. 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" [5]. 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 [5]. 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, 19]. 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

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

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 [5]. Esta posibilidad de reutilizar componentes es una de las características principales del sistema Android.

Por último, cabe señalar que pueden existir múltiples instancias en ejecución de una misma actividad al mismo tiempo. Siguiendo con el ejemplo del párrafo anterior, si dos aplicaciones distintas inician el cliente de correo para redactar un nuevo e-mail, se crearán dos instancias diferentes de la actividad en cuestión.

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 [5]. Si un componente inicia un servicio que ya está ejecutándose, no se vuelve a crear una nueva instancia del mismo sino que se interactúa con la ya existente [8, 17].

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 [5].

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" [5]. 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 [50].

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 el siguiente formato:

`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 [7, 50].

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 [5].

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 [5].

2.3. Interacción entre Componentes

"Tres de los cuatro tipos de componentes, actividades, servicios y *broadcast receivers*, son activados mediante un mensaje asíncrono llamado *intent*" [5]. 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 [5]. 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 [10]:

- **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 3.1.4).

- **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 [30]. 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.

Capítulo 3

El Modelo de Seguridad de Android

En este capítulo se brindará una descripción integral del modelo de seguridad implementado en Android. Para ello, se repasarán los aspectos principales del mismo (sección 3.1) y se lo comparará con el modelo presente en MIDP (sección 3.2).

3.1. Aspectos Principales

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.

3.1.1. *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 [5].

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 3.1.2), un identificador de usuario (UID) distinto con pocos privilegios [35]. 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. 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 hacerlo[1]. Adicionalmente, se ejecuta cada aplicación como un proceso separado del resto, con su propio espacio de direcciones [5]. Debido a esto, valiéndose de los mecanismos de protección de los sistemas Linux, Android garantiza que 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[2]) 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*) [22]. 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 [16]. De la misma forma, para acceder a los recursos del sistema, una aplicación deberá contar con los permisos necesarios para ello (ver sección 3.1.3).

Debido a 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 [20]. Adicionalmente, como se mencionó en la sección 2.1.2, 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.

3.1.2. *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 3.1.3) o autorizar a dos aplicaciones a tener el mismo UID (ver sección 3.1.1) [14]. Varias aplicaciones firmadas con el mis-

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

[2] Sin embargo, como se verá en el capítulo 6, aún teniendo pocos privilegios, una aplicación maliciosa podría influir negativamente en el sistema.

mo 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 [11].

3.1.3. Permisos

Como se explicó en la sección 3.1.1, 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 el conjunto de permisos que necesita para obtener las capacidades adicionales que no tiene por defecto [14]. 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 3.1.2) 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 [31]. Por otro lado, si alguno de los permisos no es concedido, típicamente, la aplicación no puede instalarse[3]; "Android no cuenta con un mecanismo que otorgue permisos dinámicamente (en tiempo de ejecución)" [14].

Cada permiso se identifica con un nombre/texto y, en general, podemos distinguir dos clases de ellos: los definidos por las mismas aplicaciones con el propósito de auto-protección y los que están predefinidos por Android, que controlan el acceso a recursos y servicios del sistema (por ejemplo, acceso a libreta de contactos, envío de SMS). 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 una aplicación solicitante [15]. Existen cuatro niveles posibles, descritos a continuación:

[3]Como lo aclara la documentación de Android [14], existen formas de instalar una aplicación sin exigir que el sistema otorgue todos los permisos solicitados. Sin embargo, si se instala una aplicación por las vías usuales, está garantizado que si una aplicación se instaló, la misma tiene todos los permisos que solicitó. Este último es el tipo de instalación que el usuario ve en una situación normal.

- *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)" [15]. Un ejemplo de este tipo de permisos es `SET_WALLPAPER`, que habilita a la aplicación que lo obtiene a cambiar la imagen de fondo de la pantalla inicial.

- *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" [15]. Si una aplicación solicita cualquier permiso con este nivel de protección, se le pedirá al usuario aprobación explícita para poder concederlo [15]. Por ejemplo, el permiso predefinido por el sistema `INTERNET` tiene asignado el nivel de protección *dangerous* ya que la aplicación que lo posea puede acceder a internet, posibilitando, entre otras cosas, el envío de información a terceros.

- *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" [15]. 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 [50].
  Este nivel de protección fue pensado, principalmente, para ser asignado a permisos definidos por aplicaciones. El objeto del mismo es permitir que aplicaciones pertenecientes al mismo desarrollador puedan compartir información sin la necesidad de la aprobación del usuario [50].

- *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 [15].

Por otro lado, una aplicación también puede declarar estáticamente permisos que exigirá si se quiere acceder a la misma (por ejemplo, si se le quiere enviar un *intent*). Dichos permisos también pueden ser especificados a nivel de componente; es decir, exigir distintos permisos para acceder a componentes diferentes. Si una aplicación no cuenta con el permiso correspondiente

para acceder a un determinado componente externo, cualquier intento por acceder al mismo, de ser descubierto por el sistema, resultará en un error en tiempo de ejecución, que puede o no ser comunicado al usuario [14]. Los momentos en los que el sistema verifica si se cuenta con los permisos adecuados son los siguientes [14]:

- *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 externa, se verifica si los permisos exigidos para acceder a esta última se encuentran contenidos en los permisos otorgados a la aplicación solicitante.

- *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 exigidos 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 exigidos 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 externo o asociarse a uno ya existente, se verifica si los permisos otorgados a la aplicación solicitante incluyen a los exigidos para acceder al servicio en cuestión.

A partir de la versión `HONEYCOMB` de Android, si un componente quiere acceder a otro de la misma aplicación, no se exige ningún permiso para ello [6]; por lo que la verificación explicada en el párrafo anterior no se realiza en ese caso.

Por último, 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.

3.1.4. 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" [34]. 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" [34] 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 [13, 34]. Por otro lado, si una aplicación crea un objeto `PendingIntent`, a pesar de que ésta deje de ejecutarse, todos las aplicaciones que lo recibieron podrán seguir usándolo. Sin embargo, el mismo 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 [13].

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 y un identificador de tipo de operación. Esto otorga a la actividad receptora acceso a los datos identificados por los URIs enviados en el *intent* para realizar la operación especificada en el mismo (lectura y/o escritura), sin importar si ésta no tiene los permisos adecuados para hacerlo[4] [14]. Otra forma de lograr esto es haciendo que la aplicación que cuenta con los permisos correspondientes (o, en realidad, algún componente en ejecución de la misma) invoque el método `grantUriPermission()`, pasando como argumentos el URI del recurso al que se quiere dar acceso, la aplicación a la que se quiere delegar los permisos y el tipo de operación para la que se delegan los mismos (lectura y/o

[4]La documentación oficial sólo menciona a las actividades como posibles receptoras de un *URI permission* por medio de *intents*. Por este motivo, sólo consideraremos este tipo de componente a la hora de describir el mencionado mecanismo de delegación.

escritura). En relación a la revocación, si los permisos fueron concedidos mediante el método `grantUriPermission()`, los mismos se mantienen para todos los componentes de la aplicación receptora hasta que sean revocados a través de `revokeUriPermission()`, que toma como argumentos el URI del recurso y el tipo de operación para la que se quiere realizar la revocación [39]. Luego de ejecutar dicho método, se quitan los permisos para realizar la operación indicada sobre el recurso en cuestión a todas las aplicaciones que los recibieron mediante una delegación. Si a alguna actividad en ejecución le fueron delegados permisos sobre dicho recurso por medio de *intents*, los mismos también son revocados por la ejecución de `revokeUriPermission()`. Adicionalmente, los permisos delegados a través de *intents* se revocan automáticamente al terminar la ejecución de la actividad que los recibió. Antes de ese momento, si no se revocaron explícitamente con el método mencionado, todos los componentes de la aplicación contenedora pueden utilizar los permisos delegados.

Finalmente, para poder hacer uso de este último mecanismo de delegación sobre un *content provider*, se debe especificar en la aplicación contenedora 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*.

3.1.5. *Android Manifest*

Toda aplicación Android debe incluir en su directorio raíz un archivo XML llamado `AndroidManifest` [5]. 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 solicitará la aplicación al momento de instalarse y los que serán exigidos por la misma (ver sección 3.1.3). Adicionalmente, en este archivo se especifica si se autoriza el mecanismo de delegación conocido como *URI permissions*, explicado en la sección 3.1.4, sobre los *content providers* de la aplicación.

Uno de los elementos más importantes del cuerpo de un `AndroidManifest` es <**application**>, en donde, además de especificar algunas características de la aplicación en cuestión, se incluyen los elementos que describen los componentes de la misma. Cada componente, dependiendo de su tipo, se declara utilizando alguno de los siguientes elementos: <**activity**>, <**service**>, <**provider**> y <**receiver**>. En los mismos se especifican las propiedades de cada componente perteneciente a la aplicación, tanto en sus atributos como en los elementos que a su vez pueden llegar a contener [4].

Además del elemento <**application**>, en el cuerpo del `AndroidManifest` se incluyen las siguientes etiquetas [4]:

- **<uses-permission>**
  En este elemento se especifica un permiso, definido por el sistema o por alguna aplicación, que se solicitará al momento de la instalación para obtener capacidades adicionales. Se pueden agregar varios de estos elementos, uno por cada permiso que se quiera solicitar.

- **<permission>**
  En este elemento se definen los permisos a nivel de aplicación de forma estática. Dichas definiciones tienen que incluir el nivel de protección de los mismos, basándose en lo explicado en la sección 3.1.3. Se pueden agregar varios de estos elementos, uno por cada permiso que se quiera definir.

- **<permission-tree>**
  En este elemento se reserva un espacio de nombres para luego definir permisos a nivel de aplicación en tiempo de ejecución. En uno de sus atributos se especifica una cadena que servirá de prefijo para cualquier permiso que se quiera definir dinámicamente, a través del método `addPermission()`. Se pueden agregar muchos de estos elementos para reservar distintos prefijos.

Adicionalmente, el elemento **<application>** cuenta con el atributo **android:permission** (entre otros) en donde se especifica, a lo sumo, un permiso que será exigido para acceder a cualquier componente de la aplicación [4].

En cuanto a los elementos del `AndroidManifest` que declaran los distintos componentes de la aplicación, podemos destacar dos atributos comunes a todos ellos [4]:

- **android:permission**
  En este atributo se declara, a lo sumo, un permiso que será exigido para acceder al componente en cuestión. El permiso en este atributo (de existir) tiene precedencia sobre el especificado a nivel de aplicación.

- **android:exported**
  Si este atributo está valorizado en `true`, el componente en cuestión estará disponible para ser accedido desde el exterior. En cambio, si el valor es `false`, el componente no podrá ser accedido por una aplicación externa, aún si la misma tiene los permisos exigidos en **android:permission**[5].

En particular, el elemento **<provider>** tiene atributos adicionales a los mencionados anteriormente y, además, puede contener otros elementos [4]:

[5]En realidad, como se verá en el capítulo 5, existen formas para acceder a un componente no exportado desde otra aplicación.

- **android:readPermission**
  En este atributo se declara, a lo sumo, un permiso que será exigido si se quiere leer el *content provider*. El permiso en este atributo (de existir) tiene precedencia sobre el especificado en **android:permission** del *content provider* y la aplicación.

- **android:writePermission**
  En este atributo se declara, a lo sumo, un permiso que será exigido si se quiere escribir sobre algún recurso apuntado por el *content provider*. Este permiso puede ser distinto al especificado en **android:readPermission** y también tiene precedencia sobre el declarado en **android:permission** del *content provider* y la aplicación.

- **android:grantUriPermissions**
  Si este atributo está valorizado en `true`, el *content provider* correspondiente permite la delegación de permisos a través de *URI permissions* (utilizando *intents* o el método `grantUriPermission()`) sobre todos los recursos a los que apunte.

- **<path-permission>**
  Este elemento "define la ruta y permisos requeridos para un subconjunto específico de datos contenidos en un *content provider*" [12] y puede ser agregado varias veces con el fin de especificar múltiples rutas. Por otro lado, también cuenta con sus propios atributos **android:permission**, **android:readPermission** y **android:writePermission** con el mismo significado y precedencia entre ellos que los explicados anteriormente.

- **<grant-uri-permission>**
  Este elemento permite especificar en sus atributos los URIs de los recursos apuntados por el *content provider* para los cuales está permitida la delegación de permisos a través de *URI permissions* (utilizando *intents* o `grantUriPermission()`). Se pueden agregar varios elementos de este tipo para incluir distintos subconjuntos del *content provider* en cuestión. Estos elementos tienen precedencia sobre **android:grantUriPermissions**; es decir, si se incluye al menos uno de ellos, el valor del mencionado atributo es ignorado y sólo se pueden delegar permisos sobre los recursos especificados en dichos elementos.

Ejemplo. La figura 3.1 muestra un archivo `AndroidManifest` (incompleto) en donde se aplica lo explicado anteriormente. Dada esta configuración, la aplicación `com.cpexample` solicita el permiso `SEND_SMS` y define el suyo propio, `cpexample.permission.PERMISO`, con nivel de protección 'normal'. Adicionalmente, la actividad `MainActivity` no exige ningún permiso para ser accedida, por lo que el permiso requerido es el especificado a nivel de aplicación, `SET_WALLPAPER`. Por otro lado, la actividad `SecondActivity`

no está exportada, por lo que, típicamente, sólo está disponible para componentes de la misma aplicación. En cuanto a las operaciones sobre los recursos apuntados por el *content provider* `MiProvider`, se solicitará el permiso `INTERNET` para escribir y `SEND_SMS` para leer. Finalmente, con respecto a la delegación de permisos, `MiProvider` autoriza el mecanismo de *URI permissions* sobre todos sus recursos.

```
<manifest package="com.cpexample" ... >

    ⋮

    <permission android:name="cpexample.permission.PERMISO"
        android:protectionLevel="normal" ... />

    <uses-permission android:name="android.permission.SEND_SMS" />

    <application
        android:permission="android.permission.SET_WALLPAPER" ... >

        <activity
            android:name="com.cpexample.MainActivity" ... >
        </activity>

        <activity
            android:name="com.cpexample.SecondActivity"
            android:permission="android.permission.CALL_PHONE"
            android:exported="false" ... >
        </activity>

        <provider android:name="com.cpexample.MiProvider"
            android:grantUriPermissions="true"
            android:permission="android.permission.SEND_SMS"
            android:writePermission="android.permission.INTERNET" ... >

        </provider>

        ⋮

    </application>

</manifest>
```

Figura 3.1: Ejemplo de un archivo `AndroidManifest`.

3.2. 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 [46] que permiten el desarrollo y posterior ejecución de aplicaciones Java en dispositivos móviles. En particular, MIDP especifica un modelo de seguridad que rige la 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 3.1, y el de MIDP. La información referida a este último modelo fue obtenida tanto de los trabajos de Mazeikis [40, 41] y Zanella [52, 53] como de la especificación informal de MIDP 3.0 [38].

En primer lugar, cabe destacar que la plataforma Java ME, desarrollada en el *Java Community Process* [45], 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 este 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.

Capítulo 4

Especificación Formal

Este capítulo describe la especificación formal del modelo de seguridad de Android, desarrollada en Coq [27, 51]. En la sección 4.1 se explica la metodología seguida para obtener la información a formalizar. La sección 4.2 brinda la notación que se usará para describir el modelo formal construido, siendo la misma una simplificación de la utilizada en Coq para favorecer su comprensión. En la sección 4.3 se formalizan los elementos básicos de Android y su modelo de seguridad, tales como las aplicaciones, componentes de aplicaciones y el archivo `AndroidManifest`. La abstracción elegida para representar un estado del sistema Android es descripta en la sección 4.4 y las operaciones comprendidas en la especificación formal son dadas en la sección 4.5. La semántica de cada una de estas operaciones es explicada en la sección 4.6, en donde se da la pre-condición para poder ejecutar cada una de ellas y el efecto que se causará en el estado del sistema al ejecutarlas. Por último, la sección 4.7, formaliza la ejecución de una operación, utilizando lo expuesto en la sección anterior.

Para favorecer la brevedad del presente capítulo, se decidió omitir ciertas porciones de la especificación real. Cada vez que esto ocurra se hará referencia al nombre del archivo Coq en donde se encuentra la información que fue excluida. Todos estos archivos están disponibles en `http://www.fing.edu.uy/inco/grupos/gsi/index.php?page=proygrado&locale=es`.

4.1. Metodología de Trabajo

La construcción del modelo formal que se desarrollará en las secciones subsiguientes requiere información específica sobre el sistema Android que no está cubierta en los capítulos 2 y 3. Aunque en la mayoría de los casos la información no cubierta en capítulos anteriores es suministrada por la documentación oficial de Android[1], existen ciertos aspectos muy específicos

[1] Es decir, la información disponible en `http://developer.android.com/develop`. Esta misma fuente ya fue citada en reiteradas ocasiones por los capítulos precedentes pero para

que no son abordados en la misma. Es por ello que para la definición de distintas porciones del modelo se utilizan las siguientes fuentes alternativas:

Publicaciones Académicas

La información provista en las distintas publicaciones académicas sobre el sistema Android es la primera alternativa consultada al no encontrar datos suficientes en la documentación oficial. Parte de estas publicaciones también fueron utilizadas en capítulos anteriores con el mismo criterio. Si el artículo consultado describe una versión de Android anterior a la usada en el presente trabajo, se realizarán las verificaciones correspondientes para asegurar que la información extraída siga siendo válida para la versión actual.

Código Fuente

El siguiente recurso a consultar es el código fuente del sistema (en su versión correspondiente a Junio de 2014). El mismo puede ser obtenido mediante la sincronización con el repositorio ofrecido por Google [9] o accediendo a su versión sólo-lectura en `https://github.com/android`.

Ejecución de Aplicaciones

En caso de que la información requerida no pueda ser obtenida de ninguna de las fuentes precedentes, se obtendrá de la ejecución de aplicaciones que se desarrollarán para ese fin. Dichas aplicaciones se correrán en el emulador provisto en el Android SDK, que también incluye, entre otras cosas, la herramienta *Android Debug Bridge* [2]. Esta herramienta, al ser ejecutada con el comando `adb` desde una consola, permite realizar operaciones sobre el emulador que esté corriendo en ese momento (de existir). En la descripción del modelo se utilizará esta herramienta para realizar consultas sobre el estado del sistema al ejecutar las aplicaciones de ejemplo.

4.2. Notación Utilizada

Variables y Tipos

Las variables se notarán con nombres en minúscula y los tipos atómicos con nombres que pueden incluir mayúsculas. Para ambos casos, los nombres se escribirán en cursiva. Por otro lado, dados los tipos A y B, la construcción del tipo de las funciones totales de A en B se representará como $A \to B$. Por último, para especificar que una variable a tiene tipo A se utilizará la notación $a : A$.

describir las características generales de Android.

Proposiciones

Las proposiciones se construirán utilizando los símbolos $\wedge$, $\vee$, $\Rightarrow$, $\exists$ y $\forall$, cuyo significado será el mismo que en la Lógica de Predicados Constructiva. Una proposición P representará, al mismo tiempo, el tipo correspondiente a todas las pruebas de P. De esta forma, $p : P$ denota que la variable p es una prueba de P. Paralelamente, el tipo correspondiente a todas las proposiciones se denominará $Prop$; por lo que $P : Prop$ denota que P es una proposición.

Para poder declarar predicados sobre determinados parámetros, se utilizará el constructor $\rightarrow$, mencionado anteriormente. De esta forma $P : A \rightarrow Prop$ especifica que P es un predicado sobre un parámetro de tipo A. Si, además, se quiere dar una definición del mismo, entonces se hará de la siguiente forma:

$$P\ (a : A) : Prop \stackrel{\text{def}}{=} \ldots$$

Cuando los tipos puedan ser inferidos del contexto, entonces no se incluirán en la definición.

Tipos Dependientes

Sea $P : A \rightarrow Prop$ un predicado sobre elementos de tipo A. Para especificar que una función f toma como argumento un elemento de tipo A que cumpla con el predicado P y devuelve un elemento de tipo B, f deberá tomar los elementos $a : A$ y $p : P\ a$. Esto se representará como $f : \forall\ a : A \bullet P\ a \rightarrow B$. Si, en cambio, se desea que f sea un predicado, se debe reemplazar B por $Prop$.

Tipos Inductivos

Un tipo inductivo se definirá listando sus constructores, como se muestra en el siguiente ejemplo:

$$\begin{array}{l} Ind \stackrel{\text{def}}{=} \\ \mid cons_1 : A_1 \rightarrow B_1 \rightarrow Ind \\ \mid cons_2 : A_2 \rightarrow Ind \end{array}$$

De esta forma, un elemento de tipo Ind sólo podrá ser obtenido por medio de alguno de los constructores $cons_1$ o $cons_2$. Si alguno de ellos no tuviese ningún argumento, podría omitirse el tipo; dejando únicamente el nombre. Adicionalmente, un tipo inductivo puede tomar otro tipo A para completar su definición. En este caso, los constructores correspondientes pueden tomar como argumentos elementos en A.

Pattern Matching

Siguiendo el ejemplo anterior, un elemento i del tipo inductivo Ind se analizará mediante *pattern matching* de la siguiente forma:

`match` i `with`
$|\ cons_1\ a_1\ b_1 \Rightarrow e_1$
$|\ cons_2\ a_2 \Rightarrow e_2$
`end`

Usando este esquema, se contemplan todos los valores posibles para la variable i, tomando la acción apropiada en cada caso. Por otro lado, si en la definición de un predicado algunos de sus parámetros tienen tipos inductivos, también se podrá hacer *pattern matching* sobre ellos.

Tipo *option*

Dado un tipo A, un elemento en *option* A podrá representar o bien un valor en A o ningún valor:

$option\ A \stackrel{\text{def}}{=}$
$|\ None$
$|\ Some : A \to option\ A$

Registros

Un tipo registro se define especificando sus campos y su único constructor de forma análoga a la descripta en el siguiente ejemplo:

$Reg \stackrel{\text{def}}{=} cons\ \{fld_1 : C_1 \to D_1 \to E_1, fld_2 : C_2 \to D_2\}$

Dado un elemento r de tipo Reg, el acceso a su campo fld_i se denotará $r.fld_i$.

Por otro lado, se definirá la relación $\sim$ entre dos elementos del mismo tipo registro de forma tal que $r_1 \sim_A r_2$ se cumplirá si r_1 y r_2 son iguales excepto por, a lo sumo, los campos especificados en A.

Definiciones Locales

Se utilizará el esquema `let` $v := t_1$ `in` t_2, donde v es un identificador y t_1 y t_2 expresiones, para denotar una definición local. De esta forma, el identificador v podrá ser utilizado en t_2 en lugar de la expresión t_1. Este esquema será usado cuando t_1 tenga una longitud considerable y se repita varias veces en t_2.

Funciones Parciales

Un *mapping* o función parcial m de A en B se denotará como $m : A \mapsto B$. Adicionalmente, $m[a]$ representa el acceso al valor correspondiente a $a : A$ por medio de m, de existir. Esto es equivalente a definir un *mapping* m como una función total de A en *option* B.

4.3. Definiciones Básicas

4.3.1. Permisos

Los permisos serán identificados con el registro *Perm*, definido a continuación:

$$Perm \stackrel{\text{def}}{=} perm\ \{idP : idPerm,\ pl : permLevel\}$$

Los campos pl, de tipo $permLevel \stackrel{\text{def}}{=} dangerous \mid normal \mid signature \mid signatureOrSys$, e idP, de tipo $idPerm$, representan, respectivamente, el nivel de protección y el nombre de un permiso.

Además, se contará con un predicado $isSystemPerm$ tal que $isSystemPerm\ p$ se cumplirá si y solo si p es un permiso predefinido por el sistema. El objetivo de dicho predicado es distinguir a este tipo de permisos de los definidos por el usuario.

4.3.2. Componentes de una Aplicación

En cuanto a los cuatro tipos de componentes que puede tener una aplicación, el único que presenta una estructura relevante para nuestro estudio es el de los *content providers*. Por otro lado, existirán propiedades comunes a todos los tipos de componentes como también propiedades que sólo se aplican a un componente en particular (generalmente a los *content providers*). Es por ello que, a pesar de diferenciar las distintas clases de componentes con tipos distintos, también habrá un tipo común que se aplique a todas ellas.

En concreto, los tipos para los componentes actividad, servicio y *broadcast receiver* serán, respectivamente, $Activity \stackrel{\text{def}}{=} activity\ \{idA : idCmp\}$, $Service \stackrel{\text{def}}{=} service\ \{idS : idCmp\}$ y $BroadReceiver \stackrel{\text{def}}{=} broadreceiver\ \{idB : idCmp\}$; donde $idCmp$ es el tipo de los identificadores de componentes[2]. Cada uno de estos tipos consisten en registros de un único campo debido a que, por ahora, no se estudiarán otros aspectos de sus componentes

[2]Un identificador de tipo *idCmp* no se corresponde exactamente con el nombre de un componente, ya que en Android dos componentes pertenecientes a aplicaciones distintas pueden tener el mismo nombre. Por lo tanto, un identificador debe pensarse como el nombre de un componente precedido por la aplicación a la que pertenece.

asociados. Si en trabajos futuros se requiere representar aspectos adicionales de dichos componentes, sólo se tienen que agregar campos a los registros en cuestión. Con respecto a los *content providers*, se contará con los tipos *uri* y *res* correspondientes a los $URIs$ de recursos y los recursos en sí mismos. Como se explica en la sección 2.2, un recurso de un *content provider* podría ser, por ejemplo, una tabla de la base de datos apuntada por éste. Los elementos de tipo *uri* son abstracciones de los *URIs* asociados a cada recurso en donde no se tiene en cuenta su estructura interna; un recurso determinado puede tener distintos $URIs$, uno por cada *content provider* que apunte al mismo. Finalmente, el tipo asignado a los *content providers* en el presente modelo se define de la siguiente forma: $CProvider \stackrel{\text{def}}{=} cp\ \{idC : idCmp,\ map\_res : uri \mapsto res\}$.

A continuación, construimos el tipo correspondiente a un componente genérico como enumeración de los cuatro tipos precedentes:

$$\begin{aligned} Cmp \stackrel{\text{def}}{=} \quad & | \ cmpAct : Activity \to Cmp \\ & | \ cmpSrv : Service \to Cmp \\ & | \ cmpCP : CProvider \to Cmp \\ & | \ cmpBR : BroadReceiver \to Cmp \end{aligned}$$

De esta forma, si se quiere definir una propiedad sobre un componente genérico, se utilizará el tipo Cmp; en cambio, si se quiere definir sobre una actividad, se utilizará el tipo $Activity$. Dado que, por ejemplo, un servicio s de tipo $Service$ es un componente, debería poder ser pasado como parámetro a un predicado sobre elementos de tipo Cmp. Para lograr esto, sólo se tiene que pasar al predicado mencionado el parámetro $cmpSrv\ s$.

4.3.3. Componentes en Ejecución

El tipo correspondiente a una instancia en ejecución de un componente se define como $iCmp \stackrel{\text{def}}{=} icmp\ \{idICmp : idInst\}$; donde $idICmp$ será el identificador de cada elemento. Este tipo se utilizará para representar una instancia de un componente genérico, sin hacer una distinción entre distintas clases de componentes. La razón para ello es que en nuestro modelo no es necesario ese nivel de detalle para analizar las operaciones que se quieren representar. De necesitar modelar este aspecto (o cualquier otro) se pueden agregar campos adicionales al registro correspondiente.

Si bien, como se explicó en la sección 2.2, hay componentes, como las actividades, que pueden tener varias instancias en ejecución corriendo al mismo tiempo, y otros componentes, como los servicios, que tienen a lo sumo una única instancia, en este modelo no se representa dicha particularidad, permitiendo que cualquier componente pueda tener más de una instancia en ejecución corriendo a la vez.

4.3.4. Aplicaciones

Toda aplicación contará con un identificador único, representado por el tipo *idApp*[3], un certificado (ver sección 3.1.2), representado por el tipo *Cert*, y un archivo `AndroidManifest`. Adicionalmente, para simplificar el modelo, suponemos que todos los recursos que utilizará la aplicación (tablas de bases de datos, archivos, etc.) se declararán de forma estática en la misma.

A continuación, definimos el tipo para representar a los archivos `AndroidManifest`:

$$
\begin{array}{l}
Manifest \stackrel{\text{def}}{=} \\
\quad mf\ \{\ cmp\ \ : Cmp \to Prop, \\
\qquad\ use\ \ : Perm \to Prop, \\
\qquad\ usrP\ : Perm \to Prop, \\
\qquad\ exp\ \ : \forall c : Cmp \bullet cmp\ c \to Prop, \\
\qquad\ appE\ : option\ Perm, \\
\qquad\ cmpE\ : \forall c : Cmp \bullet cmp\ c \to option\ Perm, \\
\qquad\ readE\ : \forall cp : CProvider \bullet cmp\ (cmpCP\ cp) \to option\ Perm, \\
\qquad\ writeE : \forall cp : CProvider \bullet cmp\ (cmpCP\ cp) \to option\ Perm, \\
\qquad\ grantU : \forall cp : CProvider \bullet cmp\ (cmpCP\ cp) \to Prop, \\
\qquad\ uriP\ \ : \forall cp : CProvider \bullet cmp\ (cmpCP\ cp) \to uri \to Prop\ \}
\end{array}
$$

Los campos del tipo $Manifest$ se corresponden con los elementos de un archivo `AndroidManifest` explicados en la sección 3.1.5. El campo *cmp* es un predicado que abstrae el significado de los elementos <**activity**>, <**service**>, <**provider**> y <**receiver**>; el mismo se cumple si el parámetro que se le pasa es un componente de la aplicación que se está definiendo. Los predicados *use* y *usrP* se corresponden, respectivamente, con los elementos <**uses-permission**> y <**permission**>. El primer predicado se cumplirá si el permiso que toma como argumento es solicitado por la aplicación y el segundo lo hará si el permiso está siendo definido estáticamente en la misma. Por otro lado, *exp c* se cumplirá si el componente *c* está exportado. Este predicado representa el atributo **android:exported** presente en cada elemento que declara un componente. Adicionalmente, *appE* se corresponde con el atributo **android:permission** del elemento <**application**> y especificará, a lo sumo, un permiso que exigirá la aplicación que se está definiendo. El campo *cmpE* es una función que, dado un componente de la aplicación, especifica, a lo sumo, un permiso necesario para acceder al mismo, tal como sucede en el atributo **android:permission** a nivel de componente. Por su parte, *readE* y *writeE* representan los atributos **android:readPermission** y **android:writePermission** especificando, respectivamente, los permisos

[3]En este caso, los elementos del tipo *idApp* sí se corresponden con los nombres de las aplicaciones en Android, ya que éstos no pueden repetirse.

necesarios (de existir) para leer y escribir un *content provider* perteneciente a la aplicación. Por último, los campos $grantU$ y $uriP$ se corresponden, respectivamente, con el atributo **android:grantUriPermissions** y el elemento <**grant-uri-permission**>. De esta forma, $grantU\ cp$ se cumplirá si se habilita la delegación de permisos a través de *URI permissions* sobre el *content provider* cp y $uriP\ cp\ u$ se verificará si se habilita dicho mecanismo de delegación sobre el recurso identificado por u de cp (por ejemplo, si los recursos de un *content provider* cp están identificados por u_1, u_2, u_3 y el predicado $uriP\ cp\ u_j$ se cumple sólo para $j = 1, 2$, una aplicación que tenga los permisos necesarios para leer el contenido del recurso identificado por u_2 podrá habilitar a otra a realizar la misma operación sobre dicho recurso. Sin embargo, esta delegación de permisos no podrá hacerse sobre el recurso identificado por u_3, independientemente de la validez de $grantU\ cp$). En el caso de los elementos <**permission-tree**> y <**path-permission**>, los mismos no se corresponden con ningún campo del tipo $Manifest$ ya que no serán representados en esta versión del modelo. En particular, la razón por la cual no se incluye al elemento <**path-permission**> en la formalización es porque la documentación oficial no ofrece una descripción precisa sobre la semántica del mismo.

De esta forma, una aplicación quedará representada en nuestro modelo como un elemento del siguiente tipo:

$$\begin{aligned} App \stackrel{\text{def}}{=} app\ \{\ & idAp & : idApp, \\ & cert & : Cert, \\ & manifest & : Manifest, \\ & appRes & : res \rightarrow Prop\ \} \end{aligned}$$

En el caso de las aplicaciones, no agregaremos definiciones para representar a las instancias en ejecución de las mismas ya que, por un lado, existirá a lo sumo una única instancia de cada aplicación y, por otro, cualquier acción iniciada por una aplicación será necesariamente iniciada por uno de sus componentes en ejecución.

4.4. Estado del Sistema

A partir de las definiciones básicas explicadas en la sección anterior, podemos representar un estado particular del sistema Android como un elemento del siguiente tipo:

$$
\begin{aligned}
State \stackrel{\text{def}}{=} st \{\ & apps && : App \to Prop, \\
& perms && : App \to Perm \to Prop, \\
& defPerms && : App \to Perm \to Prop, \\
& running && : iCmp \to Cmp \to Prop, \\
& delPPerms && : App \to CProvider \to uri \to PType \to Prop, \\
& delTPerms && : iCmp \to CProvider \to uri \to PType \to Prop, \\
& resCont && : App \to res \to Val \to Prop, \\
& systemImage && : App \to Prop\ \}
\end{aligned}
$$

De esta forma, dado un estado en nuestro modelo, *apps* representará las aplicaciones instaladas, $perms\ a$ los permisos concedidos a la aplicación a al momento de instalarse, $defPerms\ a$ los permisos definidos en la aplicación a y *running* las instancias de componentes que se están ejecutando. Además, $delPPerms\ a\ cp\ u\ pt$ se cumplirá si la aplicación a tiene permisos delegados para realizar la operación pt (de tipo $PType \stackrel{\text{def}}{=} Read \mid Write \mid Both$) sobre el recurso u del *content provider* cp y $delTPerms\ ic\ cp\ u\ pt$ se cumplirá si los permisos se delegaron a la instancia ic (por lo que la validez de los mismos estará limitada por el tiempo en el que dicha instancia se encuentre ejecutándose). Estos dos últimos predicados se relacionan, respectivamente, con las delegaciones hechas a través del método `grantUriPermission()` y las realizadas a través de un *intent* (ver sección 3.1.4). En lo que resta del trabajo nos referiremos a estos dos tipos de delegaciones como permanente y temporal[4]. Paralelamente, $resCont\ a\ r\ val$ se cumplirá si el recurso r de la aplicación a tiene asignado el valor val y el campo $systemImage$ representará las aplicaciones instaladas en la imagen del sistema, relevantes a la hora de otorgar permisos con nivel de protección *signatureOrSystem*. Si bien, en la versión actual de la especificación, estas últimas aplicaciones no se modificarán, las mismas se incluyen en el tipo *State* porque forman parte del estado del sistema.

Por otro lado, un elemento de tipo *State* debe cumplir ciertas propiedades para representar un estado válido del sistema Android. Para definir estas propiedades utilizaremos los predicados auxiliares $inApp$, $isCProvider$, $cmpInstalled$ y $existsRes$, como también la función $getCmpId$. El predicado $inApp\ c\ a$ se satisface si el componente c pertenece a la aplicación a; $isCProvider$ lo hará si el componente que toma como parámetro es un *content provider*; $cmpInstalled\ c\ s$ será válido si c pertenece a una aplicación instalada en el estado s y $existsRes\ u\ cp\ s$ lo será si en s existe el recurso apuntado por el $URI\ u$ en el *content provider* cp. En cuanto a la función $getCmpId$, la misma toma como parámetro un componente genérico (de tipo Cmp) y retorna su identificador. Las definiciones formales de esta última

[4] A pesar de que Android también ofrece el mecanismo de delegación llamado *pending intents* (explicado en la sección 3.1.4), como el mismo no es representado en el presente modelo, sólo nos referiremos al mecanismo conocido como *URI permissions*.

función y los predicados que la preceden se pueden encontrar en el archivo **Estado.v**, disponible en la dirección provista al principio del presente capítulo.

A continuación, utilizando las definiciones anteriores, se describen las propiedades que tiene que satisfacer s, de tipo $State$, para representar un estado válido:

Propiedad 1 (allAppDifferent) *No hay dos aplicaciones instaladas que tengan el mismo identificador:*

$$\forall(a_1\ a_2 : App) \bullet s.apps\ a_1 \wedge s.apps\ a_2 \Rightarrow a_1.idAp = a_2.idAp \Rightarrow a_1 = a_2$$

Propiedad 2 (allCmpDifferent) *No hay dos componentes pertenecientes a aplicaciones instaladas que tengan el mismo identificador:*

$$\forall(c_1\ c_2 : Cmp)(a_1\ a_2 : App) \bullet inApp\ c_1\ a_1 \wedge inApp\ c_2\ a_2 \Rightarrow s.apps\ a_1 \wedge s.apps\ a_2 \Rightarrow getCmpId\ c_1 = getCmpId\ c_2 \Rightarrow c_1 = c_2$$

Propiedad 3 (notRepeatedCmps) *Un mismo componente no está asociado a dos aplicaciones distintas:*

$$\forall(c : Cmp)(a_1\ a_2 : App) \bullet s.apps\ a_1 \wedge s.apps\ a_2 \Rightarrow inApp\ c\ a_1 \wedge inApp\ c\ a_2 \Rightarrow a_1 = a_2$$

Propiedad 4 (notCPrunning) *Si un componente está corriendo, éste no puede ser un* content provider*:*

$$\forall(ic : iCmp)(c : Cmp) \bullet s.running\ ic\ c \Rightarrow \neg\ isCProvider\ c$$

Propiedad 5 (usrPermsDefined) *Todos los permisos definidos por el usuario están declarados en alguna aplicación instalada:*

$$\forall(a : App)(p : Perm) \bullet s.defPerms\ a\ p \Rightarrow s.apps\ a \wedge (a.manifest).usrP\ p$$

Propiedad 6 (existsAppnCPinDel) *Si una aplicación tiene permisos delegados sobre un* content provider*, entonces ambos están instalados:*

$$\forall(a : App)(cp : CProvider)(u : uri)(pt : PType) \bullet s.delPPerms\ a\ cp\ u\ pt \Rightarrow s.apps\ a \wedge cmpInstalled\ (cmpCP\ cp)\ s \wedge existsRes\ cp\ u\ s$$

Propiedad 7 (delTmpRun) *Si una delegación sobre un* content provider *que se realizó a través de* intents *está vigente, entonces la instancia que recibió dicha delegación está ejecutándose y el* content provider *en cuestión está instalado:*

$$\forall(ic : iCmp)(cp : CProvider)(u : uri)(pt : PType) \bullet$$
$$s.delTPerms\ ic\ cp\ u\ pt \Rightarrow$$
$$cmpInstalled\ (cmpCP\ cp)\ s\ \wedge \exists c : Cmp \bullet cmpInstalled\ c\ s \wedge s.running\ ic\ c$$

Propiedad 8 (cmpRunAppIns) *Si una instancia está corriendo, su componente está instalado:*

$$\forall(ic : iCmp)(c : Cmp) \bullet s.running\ ic\ c \Rightarrow cmpInstalled\ c\ s$$

Propiedad 9 (notRepeatedIns) *Una instancia en ejecución puede pertenecer, a lo sumo, a un solo componente:*

$$\forall(ic : iCmp)(c_1\ c_2 : Cmp) \bullet s.running\ ic\ c_1 \wedge s.running\ ic\ c_2 \Rightarrow c_1 = c_2$$

Propiedad 10 (resContAppInst) *Todo recurso en el sistema pertenece a una aplicación instalada:*

$$\forall(a : App)(r : res)(v : Val) \bullet s.resCont\ a\ r\ v \Rightarrow s.apps\ a$$

Propiedad 11 (resContOneVal) *Todo recurso en el sistema tiene asignado un único valor:*

$$\forall(a : App)(r : res)(v_1\ v_2 : Val) \bullet s.resCont\ a\ r\ v_1 \wedge s.resCont\ a\ r\ v_2 \Rightarrow$$
$$v_1 = v_2$$

Para completar la formalización de un estado válido definimos el predicado *validState* como la conjunción de las propiedades descriptas más arriba. Dicho predicado puede encontrarse en el archivo **Estado.v**.

4.5. Operaciones

Las operaciones abstractas disponibles en el modelo son las incluidas en el tipo *Action*, definido a continuación:

$$
\begin{aligned}
Action \stackrel{\text{def}}{=} & \mid install & : App \to Action \\
& \mid uninstall & : App \to Action \\
& \mid start & : iCmp \to Cmp \to Action \\
& \mid stop & : iCmp \to Action \\
& \mid read & : iCmp \to CProvider \to uri \to Action \\
& \mid write & : iCmp \to CProvider \to uri \to Val \to Action \\
& \mid grantT & : iCmp \to CProvider \to Activity \to uri \to \\
& & \quad PType \to Action \\
& \mid grantP & : iCmp \to CProvider \to App \to uri \to PType \to \\
& & \quad Action \\
& \mid revoke & : iCmp \to CProvider \to uri \to PType \to Action \\
& \mid call & : iCmp \to SACall \to Action
\end{aligned}
$$

Informalmente, las operaciones `install` y `uninstall` representan, respectivamente, la instalación y desinstalación de una aplicación en el sistema; `start` representa la operación en la que una instancia en ejecución de un componente inicia la ejecución de otro; por su parte, la operación `stop` interrumpe la ejecución de una instancia de un componente; `read` *ic cp u* representa la lectura del recurso identificado por *u* en el *content provider cp* por parte de la instancia en ejecución *ic* y `write` *ic cp u val* la escritura del valor *val*, de tipo Val, en el recurso identificado por *u* en el *content provider cp* por parte de la instancia *ic*. Por otro lado, `grantP` *ic cp a u pt* representa la delegación permanente de permisos por parte de la aplicación a' a la aplicación a, donde a' es la aplicación a la que pertenece el componente correspondiente a la instancia *ic*. Luego de concretada esta delegación, todos los componentes de a podrán acceder al recurso identificado por *u* del *content provider cp*. El parámetro *pt* es de tipo $PType$, definido en la sección anterior, y restringe el tipo de operación que podrá hacer a sobre *u*. Dicha aplicación podrá hacer uso de los permisos delegados para realizar el tipo de operación *pt* indefinidamente hasta que los mismos sean revocados de forma explícita, mediante la operación `revoke`. Análogamente, `grantT` *ic cp act u pt* representa la delegación temporal de permisos. Por lo tanto, a diferencia de la operación anterior, además de delegar permisos, se pone a correr la actividad *act* mediante la creación de una instancia en ejecución de dicho componente. Los permisos delegados a la aplicación receptora para realizar la operación *pt* sobre el recurso *u* del *content provider cp* caducarán si la instancia en cuestión finaliza su ejecución o se realiza la operación `revoke` sobre *u* y *cp*. Con respecto a esta última operación, `revoke` *ic cp u pt* representa la revocación, por parte del componente en ejecución *ic* (o, en realidad, de la aplicación que contiene a *ic*), de todos los permisos delegados sobre el recurso *u* del *content provider cp* para realizar la acción *pt*. Para que la instancia *ic* pueda revocar los permisos delegados sobre el recurso *u*, la aplicación que la contiene sólo tiene que tener los permisos originales para

acceder a dicho recurso o ser dueña del *content provider* correspondiente; no es necesario que *ic* haya delegado el permiso en cuestión en el pasado. Por último, la operación `call` *ic sac* representa una llamada a la API del sistema *sac*, de tipo $SACall$, por parte de la instancia en ejecución *ic*.

De forma análoga al formalismo construido por Shin *et al.* [48], las operaciones precedentes se definen directamente sobre los elementos involucrados, no sobre sus identificadores. Con esto se busca modelar la interacción de los distintos elementos del sistema a través de las operaciones representadas, sin ocuparnos, al menos en esta versión inicial, de la correspondencia entre identificadores y elementos del sistema. Por ejemplo, la operación `start` *ic c* en nuestro modelo se puede corresponder en el sistema original con el envío de un *intent* explícito a un componente específico como también con el envío de un *intent* implícito en el que el sistema o usuario eligió a *c* como destinatario.

4.6. Semántica de las Operaciones

A continuación describiremos formalmente las operaciones comprendidas en la sección anterior dando la pre-condición que debe cumplir un estado s del sistema, de tipo $State$, para que pueda ejecutarse una operación sobre él y la post-condición que cumplirá el nuevo estado luego de ser ejecutada. Las definiciones completas de las pre- y post-condiciones en cuestión se encuentran en el archivo **Semantica.v**; por lo tanto, las porciones de las mismas que se excluyan del presente informe deben ser consultadas en dicho archivo.

Para representar las pre- y post-condiciones mencionadas en el párrafo anterior se usarán, respectivamente, los predicados $Pre : State \to Action \to Prop$ y $Post : State \to Action \to State \to Prop$.

4.6.1. Semántica de `install`

Si bien Android permite instalar una aplicación cuyo nombre ya se encuentra en el sistema [18], no representaremos dicho aspecto con el fin de simplificar el modelo; por lo tanto, para poder instalar una nueva aplicación a en un estado particular s del sistema se debe verificar que la misma no se encuentre instalada o, lo que es equivalente, que su identificador no exista en s. Al incorporar esta simplificación, no se modelan las verificaciones realizadas por el sistema a la hora de actualizar una aplicación. Dichas verificaciones, que consisten (básicamente) en comparar certificados de aplicaciones y constatar si se solicitan nuevos permisos, no son parte indispensable del modelo de seguridad de Android, sino más bien características secundarias que podrían ser agregadas en una versión posterior de este trabajo.

Siguiendo con la instalación de una nueva aplicación a, para mantener la unicidad de identificadores, hay que verificar que ningún componente de la misma ya se encuentre en el sistema y que todos los componentes tengan identificadores distintos. Además, para que a pueda instalarse, todos los

permisos solicitados por la misma en su archivo `AndroidManifest` deben ser otorgados por el sistema[5], siguiendo el procedimiento correspondiente al nivel de protección de cada uno de estos permisos (ver sección 3.1.3). Sin embargo, como se pudo constatar a través del código fuente, esta última verificación se hace sólo para los permisos solicitados que existen en el sistema (es decir, que estén definidos por alguna aplicación ya instalada o predefinidos por Android). A continuación, se incluye un fragmento del código en donde se manifiesta dicha característica:

Código 4.1: AppSecurityPermissions:352

```
String[] strList = info.requestedPermissions;
...
for (int i=0; i<strList.length; i++) {
  String permName = strList[i];
  ...
  try {
    PermissionInfo tmpPermInfo =
      mPm.getPermissionInfo(permName, 0);
    if (tmpPermInfo == null) {
      continue;
    }
    ...
  } catch (NameNotFoundException e) {
     Log.i(TAG, "Ignoring unknown permission:"+permName);
  }
}
```

El fragmento de código 4.1 es llamado desde la línea 127 de la clase **PackageInstallerActivity**. Esta última clase es la encargada de mostrar al usuario los permisos con nivel de protección *dangerous* que solicita la aplicación a instalar y, de ser autorizados, continuar con la instalación. En el código mostrado, la variable **strList** contiene la lista de permisos solicitados por la aplicación que se está instalando (extraídos desde el `AndroidManifest` correspondiente). Al recorrer esta lista se ignoran los permisos no definidos en el sistema, ya sea abortando las iteraciones en las que el método **getPermissionInfo()** no encuentre un nombre de permiso o capturando la excepción correspondiente. Como resultado, el proceso de instalación no es abortado ante la presencia de permisos inexistentes, sino que sigue normalmente sin mostrar al usuario ninguna notificación.

Finalmente, la pre-condición que debe cumplir un estado s para poder instalar una aplicación a se representa a través del predicado de la figu-

[5]Como se aclaró en la sección 3.1.3, existen formas de instalar una aplicación sin que se le concedan todos los permisos solicitados. Sin embargo, en nuestro modelo la aplicación debe tener todos los permisos para poder instalarse, por ser éste el escenario que un usuario ve en una situación normal.

$$
\begin{aligned}
&Pre\ s\ (\texttt{install}\ a) \stackrel{\text{def}}{=} \\
&\quad (\forall a' : App \bullet s.apps\ a' \Rightarrow a.idAp \neq a'.idAp) \wedge \\
&\quad (\forall (c_1\ c_2 : Cmp) \bullet (a.manifest).cmp\ c_1 \wedge (a.manifest).cmp\ c_2 \Rightarrow \\
&\qquad getCmpId\ c_1 \neq getCmpId\ c_2) \wedge \\
&\quad (\forall c : Cmp \bullet (a.manifest).cmp\ c \Rightarrow cmpNotInState\ c\ s) \wedge \\
&\quad authPerms\ a\ s
\end{aligned}
$$

Figura 4.1: Pre-condición de la operación `install`

ra 4.1, que formaliza las características descriptas hasta ahora. Además de la función *getCmpId*, definida en la sección 4.4, este predicado utiliza dos definiciones auxiliares: *cmpNotInState* y *authPerms*. El predicado $cmpNotInState\ c\ s$ se cumplirá si no existe ninguna aplicación instalada en s que contenga un componente con el mismo identificador que c y $authPerms\ a\ s$ lo hará si todos los permisos solicitados por a (que existan en el sistema) son otorgados. Este último predicado formaliza el procedimiento mencionado anteriormente, en donde los requisitos para el otorgamiento de cada permiso dependerá de su nivel de protección. En particular, si uno de los permisos está predefinido por el sistema y su nivel de protección es *signature*, se utilizará el predicado $isManufacturerApp$ para determinar si la aplicación solicitante está firmada con el certificado del fabricante del equipo; y, si el nivel de protección es *dangerous*, se utilizará el predicado $usrAuth$, que se cumplirá si el usuario autoriza el otorgamiento de dicho permiso. Adicionalmente, en la definición de $authPerms$ se utilizará el predicado $usrDefPerm\ p\ s$ para verificar que un permiso p está definido por el usuario en el estado del sistema s. Este último predicado se cumplirá si $s.defPerms\ a'\ p$ es verdadero para alguna aplicación a' instalada en el estado s. A continuación, la definición del predicado $authPerms$:

$$
\begin{aligned}
&authPerms\ a\ s \stackrel{\text{def}}{=} \\
&\forall p : Perm \bullet (a.manifest).use\ p \wedge \\
&\quad isSystemPerm\ p \vee usrDefPerm\ p\ s \Rightarrow \\
&\qquad \texttt{let}\ plevel := p.pl\ \texttt{in} \\
&\qquad\quad \texttt{match}\ plevel\ \texttt{with} \\
&\qquad\quad \mid normal \Rightarrow True \\
&\qquad\quad \mid dangerous \Rightarrow usrAuth\ p \\
&\qquad\quad \mid signature \Rightarrow \\
&\qquad\qquad (\neg isSystemPerm\ p \Rightarrow \\
&\qquad\qquad\quad \exists a' : App \bullet s.apps\ a' \wedge \\
&\qquad\qquad\qquad \texttt{let}\ mfst := a'.manifest\ \texttt{in}
\end{aligned}
$$

$$mfst.usrP\ p \land cert\ a' = cert\ a) \land$$
$$isSystemPerm\ p \Rightarrow isManufacturerApp\ a$$
$$\mid signatureOrSys \Rightarrow$$
$$\exists a' : App \bullet s.systemImage\ a' \land cert\ a' = cert\ a$$
end

Paralelamente, al instalar una aplicación sobre un estado s, se obtiene un nuevo estado s' en donde se agregará la aplicación en cuestión, los recursos que se utilizarán (de existir) y los permisos definidos estáticamente por la misma. Con respecto a estos permisos, si el nombre/identificador de alguno de ellos ya se encuentra en el sistema, el permiso será ignorado y se conservará el original. La descripción de esta particularidad fue obtenida a partir de un fragmento del código fuente de Android, parte del cual se muestra a continuación:

Código 4.2: PackageManagerService:5513

```
for (i=0; i<N; i++) {
    PackageParser.Permission p = pkg.permissions.get(i);
    HashMap<String, BasePermission> permissionMap =
                        p.tree ? mSettings.mPermissionTrees
                        : mSettings.mPermissions;
    ...
    BasePermission bp = permissionMap.get(p.info.name);
    if (bp == null) {
        bp = new BasePermission(p.info.name,
            p.info.packageName, BasePermission.TYPE_NORMAL);
        permissionMap.put(p.info.name, bp);
    }
    ...
    if (bp.sourcePackage == null
          || bp.sourcePackage.equals(p.info.packageName)) {
    ...
    } else {
        Slog.w(TAG, "Permission " + p.info.name
        + " from package " + p.info.packageName
        + " ignored: original from " + bp.sourcePackage);
...
}
```

Este fragmento pertenece a una de las clases que se encarga de la instalación propiamente dicha, **PackageManagerService**. En dicho código se itera sobre los elementos del campo **permissions**, correspondientes a los datos extraídos de las etiquetas <**permission**> y <**permission-tree**>

del `AndroidManifest` de la aplicación a instalar. Dado un permiso definido estáticamente por la aplicación (i.e., con **p.tree** = **false**), se selecciona la lista de permisos presentes en el sistema hasta el momento, **mSettings.mPermissions**, para trabajar con ella. Si el permiso en cuestión no se encuentra en dicha lista, entonces se agrega. Las líneas subsiguientes están abocadas, entre otras cosas, a escribir en el *log* de Android la acción realizada. En caso de que el permiso ya haya existido en el sistema, la aplicación que lo definió será distinta a la actual y, siguiendo el último **if**, se informará que dicho permiso se ignoró.

Por último, para completar el proceso de instalación, se deben registrar en el nuevo estado los permisos que fueron otorgados a la aplicación en cuestión. En caso de solicitarse un permiso inexistente en el sistema pero que está siendo definido por la misma aplicación, el mismo es otorgado automáticamente; en cambio, si dicho permiso no está definido en ninguna parte, el mismo es ignorado sin influir en el proceso de instalación. A continuación, se incluye un fragmento del código Android que se utilizó para deducir este comportamiento del sistema:

Código 4.3: PackageManagerService:6210

```
for (int i=0; i<N; i++) {
  final String name = pkg.requestedPermissions.get(i);
  ...
  final BasePermission bp =
    mSettings.mPermissions.get(name);
  ...
  if (bp == null || bp.packageSetting == null) {
    Slog.w(TAG, "Unknown permission " + name
      + " in package " + pkg.packageName);
    continue;
  }
  ...
}
```

En este otro fragmento de la clase **PackageManagerService** se puede observar que los permisos solicitados (listados en el campo identificado como **requestedPermissions**) que no puedan ser identificados por el sistema son ignorados. Dado que estas líneas se ejecutan luego del código 4.2, los permisos definidos por la aplicación ya fueron agregados al sistema (i.e., a **mSettings.mPermissions**) y, por lo tanto, no están afectados por este filtro. En las líneas que siguen, omitidas en esta descripción, se continúa el procedimiento de otorgamiento para los permisos que no fueron ignorados. Por otro lado, como el código 4.1 es ejecutado antes que el de 4.2, los permisos definidos por la aplicación sí son excluidos de la lista que se le muestra al usuario. Como resultado, los mismos son otorgados a la aplicación en cuestión sin pedir la autorización correspondiente. Para terminar de constatar

$$
\begin{aligned}
&Post\ s\ (\texttt{install}\ a)\ s' \stackrel{\text{def}}{=} \\
&\quad addApp\ s\ s'\ a \wedge \\
&\quad addRes\ s\ s'\ a \wedge \\
&\quad addDefPerms\ s\ s'\ a \wedge \\
&\quad grantPerms\ s\ s'\ a \wedge \\
&\quad s \sim_{apps,perms,defPerms,resCont} s'
\end{aligned}
$$

Figura 4.2: Post-condición de la operación `install`

que una aplicación recibe automáticamente los permisos solicitados definidos por ella misma, se desarrollaron dos aplicaciones, a_1 y a_2, en donde a_1 solicita y define un permiso p con nivel de protección *dangerous*, que no está presente en el sistema, y a_2 sólo exige p para acceder a la misma (en el campo **android:permission** del elemento <**application**>). Al instalar a_1 y, posteriormente, a_2 se observó que: 1. no se le consultó al usuario si autorizaba conceder el permiso p al momento de instalar a_1; y 2. la aplicación a_1 pudo iniciar la actividad principal de a_2 exitosamente. Para completar el caso de prueba, se instaló una tercer aplicación que no solicita el permiso p. Esta aplicación no pudo acceder a la mencionada actividad de a_2.

Utilizando los aspectos descriptos en los párrafos precedentes, en la figura 4.2 se define formalmente el predicado correspondiente a la post-condición que deberá cumplir un nuevo estado s' al instalar una aplicación a en un estado s. Debido a la extensión de este predicado, su descripción en el presente informe se realizó utilizando definiciones auxiliares. La primera de ellas, *addApp*, es un predicado que se cumplirá si las aplicaciones instaladas en s' coinciden exactamente con todas las instaladas en s más la aplicación a. Formalmente:

$$
\begin{aligned}
&addApp\ s\ s'\ a \stackrel{\text{def}}{=} (\forall a' : App \bullet s.apps\ a' \Rightarrow s'.apps\ a') \wedge \\
&(\forall a' : App \bullet s'.apps\ a' \Rightarrow s.apps\ a' \vee a' = a) \wedge \\
&s'.apps\ a
\end{aligned}
$$

El siguiente predicado auxiliar, *addRes*, se define de forma similar a *addApp*, sólo que en este caso los recursos en s' deben consistir en los recursos de s agregando los de a (i.e., r tal que $a.appRes\ r$ sea válido) inicializados en el valor especial $initVal$. En cuanto a $addDefPerms$, este predicado se cumple si en s' se agregan los nuevos permisos definidos por la aplicación a a los ya presentes en s. En nuestro modelo, un permiso p, definido por a, es nuevo con respecto al estado s si se satisface el siguiente predicado:

$$newPerm\ a\ p\ s \stackrel{\text{def}}{=} permDefInApp\ p\ a \land \neg isSystemPerm\ p \land$$
$$(\forall (a' : App)(p' : Perm) \bullet\ s.apps\ a' \land s.defPerms\ a'\ p' \Rightarrow p.idP \neq p'.idP)$$

A su vez, el predicado $permDefInApp\ p\ a$, utilizado en la definición anterior, se cumplirá si el permiso p está definido en el archivo `AndroidManifest` de a. Finalmente, la definición formal de $addDefPerms$ es la siguiente:

$$addDefPerms\ s\ s'\ a \stackrel{\text{def}}{=}$$
$$(\forall (a' : App)(p : Perm) \bullet s.defPerms\ a'\ p \Rightarrow s'.defPerms\ a'\ p)$$
$$(\forall (a' : App)(p : Perm) \bullet s'.defPerms\ a'\ p \Rightarrow s.defPerms\ a'\ p \lor (a' = a \land$$
$$newPerm\ a\ p\ s) \land$$
$$(\forall (p : Perm) \bullet newPerm\ a\ p\ s \Rightarrow s'.defPerm\ a\ p)$$

Por último, $grantPerms\ s\ s'\ a$ se cumplirá si se satisface $s'.perms\ a'\ p$ para toda aplicación a' y permiso p que verifiquen $s.perms\ a'\ p$ o bien $a' = a \land (a.manifest).use\ p \land (isSystemPerm\ p \lor usrDefPerm\ p\ s \lor permDefInApp\ p\ a)$. Esta última condición modela el hecho en el que sólo se conceden los permisos existentes (incluyendo, esta vez, los definidos por la misma aplicación).

4.6.2. Semántica de `uninstall`

Para desinstalar una aplicación se debe verificar que la misma se encuentre efectivamente instalada y que ninguno de sus componentes se encuentre en ejecución.

Por otro lado, cuando una aplicación es borrada se pasa a un nuevo estado del sistema en donde ésta ya no se encuentra entre las aplicaciones instaladas, por lo que también deben borrarse tanto los permisos otorgados y delegados a la aplicación como los definidos por la misma. Esto se sucede de haber ejecutado el siguiente caso de prueba: se instala una aplicación a_1 que define el permiso p (no presente en el sistema) y solicita el permiso p' (distinto a p) con nivel de protección *dangerous*, que es concedido por el usuario. Al completar la instalación, se ejecuta el comando `adb shell pm list permissions -u -f`, que lista los permisos definidos por las aplicaciones en el sistema, dando como resultado que p fue agregado. A continuación, se instala una aplicación a_2 que incluye un *content provider* cp y exige un permiso p'' (distinto a p y p') para acceder al mismo (tanto para leer como escribir). Posteriormente, a_2 delega permisos a a_1 a través del método `grantUriPermissions()` para leer cp y esta última aplicación puede realizar dicha operación sobre cp exitosamente. Al desinstalar a_1 se vuelve a ejecutar el programa `adb` con los mismos parámetros que antes, dando como resultado que el permiso p fue removido del sistema. Por último, se instala

a_1 nuevamente y el sistema vuelve a pedir al usuario que autorice el otorgamiento de p'. En este caso, cuando a_1 intenta leer *cp*, la operación no puede completarse debido a una excepción de seguridad.

A pesar de que los permisos definidos por la aplicación desinstalada son borrados del sistema, los mismos se mantienen en las aplicaciones que están usándolos (ya sea porque les fue asignado o los exigen para permitir el acceso), por lo que, por ejemplo, una aplicación puede tener asignado un permiso que no está en el sistema. Esta particularidad de Android fue descubierta por Shin *et al.* [49] y todavía sigue vigente en la versión estudiada en el presente trabajo.

Siguiendo con el proceso de desinstalación, si bien Android brinda la posibilidad de conservar los recursos de una aplicación para ser utilizados en caso de volverse a instalar, no se representará este aspecto en nuestro modelo[6]. Por lo tanto, al desinstalar una aplicación también se tienen que borrar los recursos usados. Por último, también deben ser revocados todos los permisos delegados a otras aplicaciones e instancias en ejecución para acceder a los *content providers* de la aplicación que se está desinstalando. En versiones anteriores a la estudiada en este trabajo, esta última acción era omitida por Android, provocando que una aplicación que tenía permisos delegados para acceder a otra, podía seguir accediendo luego de reinstalar esta última, aún cuando la misma exigía permisos nuevos [37]. Para constatar que la versión estudiada no adolece de dicho problema, se instalaron dos aplicaciones, a_1 y a_2, tales que a_1 no tiene los permisos necesarios para acceder al *content provider cp* de a_2. Posteriormente, a_2 delega permisos a a_1 través de `grantUriPermissions()` para acceder a *cp* y esta última puede hacerlo exitosamente. Luego de desinstalar y volver a instalar a_2 se observa que a_1 no puede acceder a *cp*.

Finalmente, se formalizará lo descripto hasta ahora con los predicados correspondientes a la pre- y post-condición para la operación `uninstall`. Los mismos se encuentran en la figura 4.3.

Para definir la pre-condición, el único predicado auxiliar utilizado fue *inApp*, introducido en la la sección 4.4. Con respecto a la post-condición, se decidió dividir su definición en predicados auxiliares, como se hizo en la sección 4.6.1, debido a su extensión. El predicado *removeApp* se verificará si las aplicaciones instaladas en el estado s' consisten exactamente en las instaladas en s quitando la aplicación a. Formalmente:

[6]Este aspecto fue excluido de nuestro modelo por no considerarlo una parte fundamental del sistema Android o de las características estudiadas en este trabajo. Por lo tanto, sólo se representa el comportamiento del sistema en el que el usuario elige la opción de borrar los recursos de la aplicación a desinstalar. La inclusión de dicho aspecto al modelo formal puede ser abordado en trabajos futuros (ver capítulo 7).

$$
\begin{aligned}
&Pre\ s\ (\texttt{uninstall}\ a) \stackrel{\text{def}}{=} \\
&\quad s.apps\ a \wedge \\
&\quad \forall (ic : iCmp)(c : Cmp) \bullet s.running\ ic\ c \Rightarrow \neg inApp\ c\ a
\end{aligned}
$$

$$
\begin{aligned}
&Post\ s\ (\texttt{uninstall}\ a)\ s' \stackrel{\text{def}}{=} \\
&\quad removeApp\ s\ s'\ a \wedge \\
&\quad revokePerms\ s\ s'\ a \wedge \\
&\quad revokeSelfPPerms\ s\ s'\ a \wedge \\
&\quad removeDefPerms\ s\ s'\ a \wedge \\
&\quad removeRes\ s\ s'\ a \wedge \\
&\quad revokeOtherTPerms\ s\ s'\ a \wedge \\
&\quad revokeOtherPPerms\ s\ s'\ a \wedge \\
&\quad s.running = s'.running \wedge \\
&\quad s.systemImage = s'.systemImage
\end{aligned}
$$

Figura 4.3: Pre- y post-condición de la operación `uninstall`

$$
\begin{aligned}
&removeApp\ s\ s'\ a \stackrel{\text{def}}{=} \\
&(\forall a' : App \bullet s'.apps\ a' \Rightarrow s.apps\ a') \wedge \\
&(\forall a' : App \bullet s.apps\ a' \Rightarrow s'.apps\ a' \vee a' = a) \wedge \\
&\neg s'.apps\ a
\end{aligned}
$$

Los predicados $revokePerms$ y $revokeSelfPPerms$ representan, respectivamente, la revocación de permisos otorgados y delegados (de forma permanente) a la aplicación a. En cuanto a $removeDefPerms$, el mismo se cumple si los permisos definidos por las aplicaciones en s' consisten en los definidos en s borrando todos los agregados por a. Por otro lado, el predicado $removeRes$ representa la quita de los recursos pertenecientes a la aplicación a en el estado s'. Las formalizaciones de estos últimos cuatro predicados tienen un esquema similar a la definición de *addApp*; por lo tanto, en el presente informe, sólo se describirá el predicado $removeRes$:

$$
\begin{aligned}
&removeRes\ s\ s'\ a \stackrel{\text{def}}{=} \\
&(\forall (a' : App)(r : res)(v : Val) \bullet s'.resCont\ a'\ r\ v \Rightarrow s.resCont\ a'\ r\ v) \wedge \\
&(\forall (a' : App)(r : res)(v : Val) \bullet s.resCont\ a'\ r\ v \Rightarrow s'.resCont\ a'\ r\ v \vee a' = a) \wedge \\
&(\forall (r : res)(v : Val) \bullet \neg s'.resCont\ a\ r\ v)
\end{aligned}
$$

Por último, los predicados $revokeOtherTPerms$ y $revokeOtherPPerms$ representan, respectivamente, la revocación de permisos temporales y per-

manentes para acceder a los *content providers* de la aplicación a desinstalar por parte de cualquier otra. A continuación, se describe únicamente el primero de estos predicados, ya que la definición de $revokeOtherPPerms$ es análoga:

$$
\begin{aligned}
&revokeOtherTPerms\ s\ s'\ a \stackrel{\text{def}}{=} \\
&(\forall (ic : iCmp)(cp : CProvider)(u : uri)(pt : PType) \bullet \\
&\quad s'.delTPerms\ ic\ cp\ u\ pt \Rightarrow s.delTPerms\ ic\ cp\ u\ pt) \wedge \\
&(\forall (ic : iCmp)(cp : CProvider)(u : uri)(pt : PType) \bullet \\
&\quad s.delTPerms\ ic\ cp\ u\ pt \Rightarrow s'.delTPerms\ ic\ cp\ u\ pt \vee \\
&\qquad (a.manifest).cmp\ (cmpCP\ cp)) \wedge \\
&(\forall (ic : iCmp)(cp : CProvider)(u : uri)(pt : PType) \bullet \\
&\quad (a.manifest).cmp\ (cmpCP\ cp) \Rightarrow \neg s'.delTPerms\ ic\ cp\ u\ pt)
\end{aligned}
$$

Como se dijo al principio de la sección 4.6, la definición completa de la post-condición correspondiente a la operación `uninstall` puede encontrarse en el archivo **Semantica.v**.

4.6.3. Semántica de `start` y `stop`

Para iniciar la ejecución de un componente en un estado determinado del sistema hay que chequear que el mismo forme parte de una aplicación instalada. Además, en este modelo, un componente sólo puede ser iniciado por una instancia en ejecución de otro. Si bien esta particularidad puede parecer restrictiva, las definiciones en el presente modelo son suficientes para capturar un amplio espectro de comportamientos del sistema Android ya que, en la mayoría de los casos, si un componente comenzó su ejecución es porque fue iniciado por otro, a través de un *intent*. Por ejemplo, el inicio de una aplicación desde el menú se hace a través de un componente en ejecución de la aplicación del sistema `Launcher`, que llama a la actividad principal de la aplicación seleccionada. Adicionalmente, para que una instancia en ejecución pueda iniciar un componente, la aplicación correspondiente a la misma debe tener los permisos necesarios para acceder a dicho componente. Una vez iniciado, el estado del sistema se modifica para agregar una nueva instancia en ejecución del componente en cuestión.

Utilizando las características recién descritas, en la figura 4.4 se da la descripción formal de la pre- y post-condición de la operación `start`.

En la descripción formal de la pre-condición, se utilizaron las definiciones auxiliares $isCProvider$ y $cmpInstalled$, introducidas en la sección 4.4, y el predicado $canStart\ c_1\ c_2$, el cual se cumplirá si la aplicación que contiene a c_1 tiene los permisos necesarios para acceder/iniciar a c_2 y cuya definición formal se da a continuación:

$$
\begin{aligned}
&Pre\ s\ (\texttt{start}\ ic\ c) \stackrel{\text{def}}{=} \\
&\quad cmpInstalled\ c\ s \land \neg isCProvider\ c \land \\
&\quad \exists\ c' : Cmp\ \bullet\ s.running\ ic\ c' \land \\
&\qquad cmpInstalled\ c'\ s \land \neg isCProvider\ c' \land canStart\ c'\ c\ s
\end{aligned}
$$

$$
\begin{aligned}
&Post\ s\ (\texttt{start}\ ic\ c)\ s' \stackrel{\text{def}}{=} \\
&\quad \exists ic : iCmp \bullet insNotInState\ ic\ s \land runCmp\ s\ s'\ ic\ c \land \\
&\quad s \sim_{running} s'
\end{aligned}
$$

Figura 4.4: Pre- y post-condición de la operación `start`

$$
\begin{aligned}
&\quad canStart\ c_1\ c_2\ s \stackrel{\text{def}}{=} \\
&\forall (a_1\ a_2 : App)(H_1 : inApp\ c_1\ a_1)(H_2 : inApp\ c_2\ a_2) \bullet \\
&\quad s.apps\ a_1 \land s.apps\ a_2 \Rightarrow \\
&\qquad \texttt{let}\ m_2 := a_2.manifest\ \texttt{in} \\
&\qquad\quad m_2.exp\ c_2\ H_2 \land \\
&\qquad\quad (\forall p : Perm \bullet m_2.cmpE\ c_2\ H_2 = Some\ p \lor \\
&\qquad\qquad m_2.cmpE\ c_2\ H_2 = None \land m_2.appE = Some\ p \Rightarrow \\
&\qquad\qquad\quad s.perms\ a_1\ p) \lor \\
&\qquad a_1 = a_2
\end{aligned}
$$

Como se ve en este último predicado, un componente c_1 tiene los permisos necesarios para iniciar a otro componente c_2 si pertenecen a la misma aplicación, en cuyo caso no se necesita ningún permiso (ver sección 3.1.3), o si c_2 está exportado y la aplicación que contiene a c_1, identificada por a_1, tiene asignados los permisos que exige c_2. Con respecto a este último caso, como se explicó en la sección 3.1.5, el permiso exigido a nivel de componente (campo *cmpE*) tiene prioridad sobre el exigido a nivel de aplicación (*appE*). Si se exige un permiso para acceder al componente c_2, entonces a_1 debe contar con él, sin necesidad de tener otro; si no se exige ninguno, entonces se debe tener el permiso requerido a nivel de aplicación (de existir).

En cuanto a la formalización de la post-condición, se utilizó el predicado $insNotInState\ ic\ c$, que se cumple si ic no se encuentra corriendo en el estado s, para representar la creación de una nueva instancia. Por último, el predicado *runCmp* representa la ejecución de ic, correspondiente al componente c, en s' y se define de la siguiente forma:

$$
\begin{aligned}
&Pre\ s\ (\texttt{stop}\ ic) \stackrel{\text{def}}{=} \\
&\quad \exists c : Cmp \bullet cmpInstalled\ c\ s \wedge\ s.running\ ic\ c
\end{aligned}
$$

$$
\begin{aligned}
&Post\ s\ (\texttt{stop}\ ic)\ s' \stackrel{\text{def}}{=} \\
&\quad stopIns\ s\ s'\ ic \wedge \\
&\quad revokeTPermsIns\ s\ s'\ ic \wedge \\
&\quad s \sim_{running, delTPerms} s'
\end{aligned}
$$

Figura 4.5: Pre- y post-condición de la operación `stop`

$$
\begin{aligned}
&\quad runCmp\ s\ s'\ ic\ c \stackrel{\text{def}}{=} \\
&(\forall (ic' : iCmp)(c' : Cmp) \bullet s.running\ ic'\ c' \Rightarrow s'.running\ ic'\ c') \wedge \\
&(\forall (ic' : iCmp)(c' : Cmp) \bullet s'.running\ ic'\ c' \Rightarrow \\
&\quad s.running\ ic'\ c' \vee ic = ic' \wedge c = c') \wedge \\
&s'.running\ ic\ c
\end{aligned}
$$

Análogamente, para poder interrumpir la ejecución de un componente, además de ser parte de una aplicación instalada, éste se debe encontrar activo. Al detener un componente, el estado del sistema cambia para registrar que el mismo ya no se encuentra ejecutándose y que, como consecuencia, se revocan los permisos temporales asignados (ver sección 3.1.4).

En la figura 4.5 se brinda la formalización de la pre- y post-condición para la operación `stop`.

A continuación se describe formalmente el predicado $stopIns$, utilizado en la post-condición de `stop`; la definición de $revokeTPermsIns$, correspondiente a la revocación de permisos temporales ligados a ic en s', se omitirá por ser similar:

$$
\begin{aligned}
&\quad stopIns\ s\ s'\ ic \stackrel{\text{def}}{=} \\
&(\forall (ic' : iCmp)(c' : Cmp) \bullet s'.running\ ic'\ c' \Rightarrow s.running\ ic'\ c') \wedge \\
&(\forall (ic' : iCmp)(c' : Cmp) \bullet s.running\ ic'\ c' \Rightarrow s'.running\ ic'\ c' \vee ic = ic') \wedge \\
&insNotInState\ ic\ s'
\end{aligned}
$$

En esta definición se utiliza el predicado $insNotInState$, introducido anteriormente, para representar el hecho en el que la instancia ic ya no se encuentra corriendo en s'.

$$
\begin{aligned}
&Pre\ s\ (\texttt{read}\ ic\ cp\ u) \stackrel{\text{def}}{=} \\
&\quad cmpInstalled\ (cmpCP\ cp)\ s \wedge existsRes\ cp\ u\ s \wedge \\
&\quad \exists c : Cmp \bullet cmpInstalled\ c\ s \wedge s.running\ ic\ c \wedge \\
&\qquad \neg isCProvider\ c \wedge (canRead\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Read\ s)
\end{aligned}
$$

$$
\begin{aligned}
&Post\ s\ (\texttt{read}\ ic\ cp\ u)\ s' \stackrel{\text{def}}{=} \\
&\quad s = s'
\end{aligned}
$$

Figura 4.6: Pre- y post-condición de la operación `read`

4.6.4. Semántica de `read` y `write`

Para representar la lectura/escritura de un recurso identificado por u en el *content provider* cp por parte de una instancia en ejecución ic, el recurso apuntado por u debe existir y la aplicación a la que pertenece ic tiene que tener los permisos correspondientes a la acción a realizar. Paralelamente, si se produce una escritura, el estado del sistema cambia para asignarle el nuevo valor al recurso en cuestión.

A partir de estas descripciones, en las figuras 4.6 y 4.7 se definen las pre- y post-condiciones de las operaciones abstractas `read` y `write`.

Para la definición de las pre-condiciones de ambas operaciones se utilizó el predicado $existsRes$, definido en la sección 4.4, y los predicados auxiliares $canRead\ c\ cp\ s$ y $canWrite\ c\ cp\ s$, que se cumplirán si la aplicación que contiene a c tiene, desde su instalación, los permisos adecuados para leer o escribir, respectivamente, el *content provider* cp en el estado s. Adicionalmente, se utilizó el predicado $delPerms\ c\ cp\ u\ pt\ s$, que se verifica si el componente c tiene permisos delegados para realizar la operación pt sobre el recurso identificado por u del *content provider* cp. Como consecuencia, la combinación de estos predicados en las pre-condiciones representan la posibilidad de delegar permisos para leer un *content provider*. A continuación, se detalla únicamente la definición formal de $delPerms$ y $canRead$, ya que $canWrite$ se define de forma análoga a este último:

$$
\begin{aligned}
&delPerms\ c\ cp\ u\ pt\ s \stackrel{\text{def}}{=} \\
&\forall (a : App) \bullet apps\ s\ a \wedge inApp\ c\ a \Rightarrow \\
&\quad (\exists (ic' : iCmp)(c' : Cmp) \bullet s.running\ ic'\ c' \wedge inApp\ c'\ a \wedge \\
&\qquad s.delTPerms\ ic'\ cp\ u\ Both) \vee \\
&\quad (\exists (ic' : iCmp)(c' : Cmp) \bullet s.running\ ic'\ c' \wedge inApp\ c'\ a \wedge \\
&\qquad s.delTPerms\ ic'\ cp\ u\ pt) \vee \\
&\quad s.delPPerms\ a\ cp\ u\ Both \vee \\
&\quad s.delPPerms\ a\ cp\ u\ pt
\end{aligned}
$$

$$
\begin{array}{l}
\quad canRead\ c\ cp\ s \stackrel{\text{def}}{=} \\
\forall(a_1\ a_2 : App)(H_c : inApp\ c\ a_1)(H_{cp} : inApp\ (cmpCP\ cp)\ a_2)\bullet \\
\quad s.apps\ a_1 \wedge s.apps\ a_2 \Rightarrow \\
\quad \texttt{let}\ m_2 := a_2.manifest\ \texttt{in} \\
\qquad m_2.exp\ (cmpCP\ cp)\ H_{cp} \wedge \\
\qquad (\texttt{let}\ readEn := m_2.readE\ cp\ H_{cp}, \\
\qquad cmpEn := m_2.cmpE\ (cmpCP\ cp)\ H_{cp}, \\
\qquad appEn := m_2.appE\ \texttt{in} \\
\qquad\quad \forall p : Perm \bullet readEn = Some\ p \vee \\
\qquad\qquad readEn = None \wedge cmpEn = Some\ p \vee \\
\qquad\qquad readEn = cmpEn = None \wedge appEn = Some\ p \\
\qquad\qquad\quad \Rightarrow s.perms\ a_1\ p) \vee \\
\quad a_1 = a_2
\end{array}
$$

Como se ve en la definición de *delPerms*, cualquier componente de una aplicación puede hacer uso de los permisos que fueron delegados a alguna instancia en ejecución de la misma [37]. Otra particularidad de este predicado es que el cumplimiento del mismo no depende de si el *content provider* tomado como argumento está exportado o no. Este hecho tiene importantes consecuencias que serán estudiadas en el capítulo 5. Por otro lado, en el predicado *canRead* vuelve a aparecer la precedencia de los elementos del archivo `AndroidManifest` para exigir permisos. Como se explicó en la sección 3.1.5, la mayor precedencia se tiene cuando se exige un permiso para leer un *content provider* (correspondiente a *readEn*), seguido por cuando se especifica un permiso para acceder al componente en cuestión (*cmpEn*), hasta llegar a cuando se exige un permiso para acceder a la aplicación entera (*appEn*). Siguiendo esta jerarquía, el permiso exigido para leer el *content provider* en cuestión será el efectivamente especificado por el elemento de mayor precedencia.

Por último, la post-condición de la operación `read` especifica que los efectos de una operación de lectura no son representados en el presente modelo. En cuanto a la post-condición de `write` *ic cp u val*, la misma representa la asignación del valor *val* al recurso r apuntado por u en el *content provider* *cp*. Para ello, el primer y segundo predicado de la conjunción especifican que los estados s y s' diferirán en, a lo sumo, el valor asignado al recurso r (perteneciente a la aplicación dueña de *cp*); el tercer predicado exige que r tenga el valor *val* en s' y, finalmente, el cuarto predicado exige que si r tenía un valor distinto a *val* en s, ya no lo tenga en s'. Esto último se agregó para evitar que r pueda tener asociados dos valores distintos.

$$
\begin{aligned}
&Pre\ s\ (\texttt{write}\ ic\ cp\ u\ val) \stackrel{\text{def}}{=} \\
&\quad cmpInstalled\ (cmpCP\ cp)\ s \land existsRes\ cp\ u\ s \land \\
&\quad \exists c : Cmp \bullet cmpInstalled\ c\ s \land s.running\ ic\ c \land \neg isCProvider\ c \land \\
&\qquad (canWrite\ c\ cp\ s \lor delPerms\ c\ cp\ u\ Write\ s)
\end{aligned}
$$

$$
\begin{aligned}
&Post\ s\ (\texttt{write}\ ic\ cp\ u\ val)\ s' \stackrel{\text{def}}{=} \\
&\quad (\forall (a' : App)(r : res)(v : Val) \bullet s.resCont\ a'\ r\ v \Rightarrow s'.resCont\ a'\ r\ v \lor \\
&\qquad inApp\ (cmpCP\ cp)\ a' \land s.apps\ a' \land cp.map\_res[u] = r) \land \\
&\quad (\forall (a' : App)(r : res)(v : Val) \bullet s'.resCont\ a'\ r\ v \Rightarrow s.resCont\ a'\ r\ v \lor \\
&\qquad inApp\ (cmpCP\ cp)\ a' \land s.apps\ a' \land cp.map\_res[u] = r \land v = val) \land \\
&\quad (\forall (a' : App)(r : res) \bullet inApp\ (cmpCP\ cp)\ a' \land s.apps\ a' \land cp.map\_res[u] = r \Rightarrow \\
&\qquad s'.resCont\ a'\ r\ val) \land \\
&\quad (\forall (a' : App)(r : res)(v : Val) \bullet inApp\ (cmpCP\ cp)\ a' \land s.apps\ a' \land \\
&\qquad cp.map\_res[u] = r \land s.resCont\ a'\ r\ v \Rightarrow \neg s'.resCont\ a'\ r\ v \lor v = val) \land \\
&\quad s \sim_{resCont} s'
\end{aligned}
$$

Figura 4.7: Pre- y post-condición de la operación `write`

4.6.5. Semántica de `grantT`, `grantP` y `revoke`

Como se explicó en la sección 3.1.4, para que una instancia en ejecución pueda delegar un permiso de forma temporal o permanente sobre un recurso de un *content provider*, este último tiene que permitirlo en el `AndroidManifest` de su aplicación. Si esto se verifica, la instancia que delega los permisos debe poder ser capaz de realizar la operación para la que se está realizando dicha delegación. Además, como la delegación temporal se corresponde con la delegación de permisos a través de *intents* en el sistema Android, para realizar dicha operación se tiene el requisito extra de poder iniciar el componente al que se le quiere delegar el permiso. Por último, como se aclaró en la sección mencionada, en este trabajo sólo nos limitamos a representar las delegaciones temporales a componentes de tipo actividad, ya que es el único caso en el que se ofrece documentación suficiente.

La pre- y post-condición de la operación `grantT` se encuentran disponibles en la figura 4.8 y las correspondientes a `grantP` en la figura 4.9.

En cuanto a las pre-condiciones de estas operaciones, para representar formalmente que se permite la delegación de permisos sobre un *content provider*, utilizamos el predicado *canGrant*, que toma un *content provider* y un identificador de recurso y se verifica si se pueden delegar permisos sobre él:

$$
\begin{array}{l}
Pre\ s\ (\texttt{grantT}\ ic\ cp\ act\ u\ pt) \stackrel{\text{def}}{=} \\
\quad cmpInstalled\ (cmpCP\ cp)\ s \wedge canGrant\ cp\ u\ s \wedge existsRes\ cp\ u\ s \wedge \\
\quad cmpInstalled\ (cmpAct\ act)\ s \wedge \\
\quad \exists c : Cmp \bullet cmpInstalled\ c\ s \wedge s.running\ ic\ c \wedge \\
\qquad canStart\ c\ (cmpAct\ act)\ s \wedge \\
\qquad \texttt{match}\ pt\ \texttt{with} \\
\qquad \mid Read \Rightarrow canRead\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Read\ s \\
\qquad \mid Write \Rightarrow canWrite\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s \\
\qquad \mid Both \Rightarrow (canRead\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s) \wedge \\
\qquad\quad (canWrite\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s) \\
\qquad \texttt{end}
\end{array}
$$

$$
\begin{array}{l}
Post\ s\ (\texttt{grantT}\ ic\ cp\ act\ u\ pt)\ s' \stackrel{\text{def}}{=} \\
\quad \texttt{let}\ cAct := cmpAct\ act\ \texttt{in} \\
\quad \exists iact : iCmp \bullet insNotInState\ iact\ s \wedge \\
\qquad runCmp\ s\ s'\ iact\ cAct \wedge \\
\qquad grantTPerm\ s\ s'\ iact\ cp\ u\ pt \wedge \\
\quad s \sim_{running, delTPerms} s'
\end{array}
$$

Figura 4.8: Pre- y post-condición de la operación `grantT`

$$
\begin{array}{l}
Pre\ s\ (\texttt{grantP}\ ic\ cp\ a\ u\ pt) \stackrel{\text{def}}{=} \\
\quad cmpInstalled\ (cmpCP\ cp)\ s \wedge canGrant\ cp\ u\ s \wedge existsRes\ cp\ u\ s \wedge \\
\quad s.apps\ a \wedge \\
\quad \exists c : Cmp \bullet cmpInstalled\ c\ s \wedge s.running\ ic\ c \wedge \\
\qquad \texttt{match}\ pt\ \texttt{with} \\
\qquad \mid Read \Rightarrow canRead\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Read\ s \\
\qquad \mid Write \Rightarrow canWrite\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s \\
\qquad \mid Both \Rightarrow (canRead\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s) \wedge \\
\qquad\quad (canWrite\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s) \\
\qquad \texttt{end}
\end{array}
$$

$$
\begin{array}{l}
Post\ s\ (\texttt{grantP}\ ic\ cp\ a\ u\ pt)\ s' \stackrel{\text{def}}{=} \\
\quad grantPPerm\ s\ s'\ a\ cp\ u\ pt \wedge \\
\quad s \sim_{delPPerms} s'
\end{array}
$$

Figura 4.9: Pre- y post-condición de la operación `grantP`

$$
\begin{aligned}
&\quad canGrant\ cp\ u\ s \stackrel{\text{def}}{=} \\
&\forall(a : App)(H : inApp\ (cmpCP\ cp)\ a) \bullet \\
&\quad s.apps\ a \Rightarrow \\
&\qquad \texttt{let}\ m := a.manifest\ \texttt{in} \\
&\qquad\quad m.uriP\ cp\ H\ u \vee \\
&\qquad\qquad \neg(\exists u' : uri \bullet m.uriP\ cp\ H\ u') \wedge m.grantU\ cp\ H
\end{aligned}
$$

La definición de este predicado formaliza las descripciones realizadas en la sección 3.1.5 sobre el atributo **android:grantUriPermissions** y el elemento <**grant-uri-permission**>, correspondientes a un archivo `AndroidManifest`. Como consecuencia, el mecanismo de delegación de permisos sobre el *content provider* *cp* sólo estará habilitado para los identificadores especificados en $m.uriP\ cp$. De no especificarse ninguno, entonces lo estará para todos los recursos del *content provider* en cuestión si $m.grantU\ cp$ se cumple. Por otro lado, como Android permite la redelegación de permisos [37], ambas pre-condiciones incluyen el predicado $delPerms$, definido en la sección 4.6.4, al momento de verificar si el componente correspondiente a ic puede realizar la operación pt. Si dicho predicado no estuviese, entonces sólo se podrían delegar permisos solicitados en tiempo de instalación.

Por último, para agregar la delegación realizada al estado s', en las post-condiciones de `grantT` y `grantP` se utilizaron, respectivamente, los predicados auxiliares $grantTPerm$ y $grantPPerm$. Adicionalmente, la post-condición de `grantT` cuenta con el predicado $runCmp$, definido en la sección 4.6.3, para iniciar la ejecución de la actividad receptora en el estado s'. A continuación, se brinda únicamente la definición formal de $grantTPerm$; la especificación del predicado restante se puede encontrar en **Semantica.v**:

$$
\begin{aligned}
&\quad grantTPerm\ s\ s'\ ic\ cp\ u\ pt \stackrel{\text{def}}{=} \\
&(\forall(ic' : iCmp)(cp' : CProvider)(u' : idRes)(pt' : PType) \bullet \\
&\quad s.delTPerms\ ic'\ cp'\ u'\ pt' \Rightarrow s'.delTPerms\ ic'\ cp'\ u'\ pt') \wedge \\
&(\forall(ic' : iCmp)(cp' : CProvider)(u' : idRes)(pt' : PType) \bullet \\
&\quad s'.delTPerms\ ic'\ cp'\ u'\ pt' \Rightarrow s.delTPerms\ ic'\ cp'\ u'\ pt' \vee \\
&\qquad ic = ic' \wedge cp = cp' \wedge u = u' \wedge pt = pt') \wedge \\
&s'.delTPerms\ ic\ cp\ u\ pt
\end{aligned}
$$

Por otra parte, si una instancia en ejecución quiere revocar permisos que fueron delegados para realizar el tipo de operación pt sobre cierto recurso u del *content provider* *cp*, su aplicación correspondiente debe ser la dueña de *cp* ó tener los permisos adecuados para realizar la operación en cuestión desde su instalación. Finalmente, al realizarse la revocación, el estado del sistema

$$
\begin{array}{l}
Pre\ s\ (\texttt{revoke}\ ic\ cp\ u\ pt) \stackrel{\text{def}}{=} \\
\quad cmpInstalled\ (cmpCP\ cp)\ s \wedge existsRes\ cp\ u\ s \wedge \\
\quad \exists c : Cmp \bullet cmpInstalled\ c\ s \wedge s.running\ ic\ c \wedge \\
\qquad \texttt{match}\ pt\ \texttt{with} \\
\qquad \mid Read \Rightarrow canRead\ c\ cp\ s \\
\qquad \mid Write \Rightarrow canWrite\ c\ cp\ s \\
\qquad \mid Both \Rightarrow canRead\ c\ cp\ s \wedge canWrite\ c\ cp\ s \\
\qquad \texttt{end}
\end{array}
$$

$$
\begin{array}{l}
Post\ s\ (\texttt{revoke}\ ic\ cp\ u\ pt)\ s' \stackrel{\text{def}}{=} \\
\quad revokeTPerm\ s\ s'\ cp\ u\ pt \wedge \\
\quad revokePPerm\ s\ s'\ cp\ u\ pt \wedge \\
\quad s \sim_{delTPerms, delPPerms} s'
\end{array}
$$

Figura 4.10: Pre- y post-condición de la operación `revoke`

cambia dejando sin efecto todas las delegaciones, temporales y permanentes, para realizar la operación *pt* sobre el identificador *u* de *cp*; sin poder revocar un permiso a una aplicación en particular [37].

En la figura 4.10 se puede ver la definición de la pre- y post- condición de la operación `revoke`, que formaliza lo descripto en el párrafo anterior.

A diferencia de las operaciones `grantT` y `grantP`, la pre-condición de `revoke` no incluye el predicado *delPerms* a la hora de verificar si el componente *c* está en condiciones de realizar la operación involucrada en la revocación. De esta forma, como se adelantó en la descripción informal, los permisos delegados no son suficientes para permitir la revocación. En cuanto a la post-condición de `revoke`, la definición formal del predicado *revokePPerm* es la siguiente:

$$
\begin{array}{l}
\quad revokePPerm\ s\ s'\ u\ cp\ pt \stackrel{\text{def}}{=} \\
(\forall (a' : App)(cp' : CProvider)(u' : idRes)(pt' : PType) \bullet \\
\quad s'.delPPerms\ a'\ cp'\ u'\ pt' \Rightarrow s.delPPerms\ a'\ cp'\ u'\ pt') \wedge \\
(\forall (a' : App)(cp' : CProvider)(u' : idRes)(pt' : PType) \bullet \\
\quad s.delPPerms\ a'\ cp'\ u'\ pt' \Rightarrow s'.delPPerms\ a'\ cp'\ u'\ pt' \vee \\
\qquad pt = Both \wedge cp' = cp \wedge u' = u \vee \\
\qquad pt \neq Both \wedge cp' = cp \wedge u' = u \wedge pt' = pt) \wedge \\
(pt = Both \Rightarrow \forall (a' : App)(pt' : PType) \bullet \neg s'.delPPerms\ a'\ cp\ u\ pt') \wedge \\
(pt \neq Both \Rightarrow \forall a' : App \bullet \neg s'.delPPerms\ a'\ cp\ u\ pt)
\end{array}
$$

$$
\begin{array}{l}
Pre\ s\ (\texttt{call}\ ic\ sac) \stackrel{\text{def}}{=} \\
\quad \exists c : Cmp \bullet s.running\ ic\ c \wedge \\
\quad\quad \forall (a : App)(p : Perm)(H : isSystemPerm\ p) \bullet \\
\quad\quad\quad s.apps\ a \wedge inApp\ c\ a \wedge permSAC\ p\ H\ sac \Rightarrow \\
\quad\quad\quad\quad s.perms\ a\ p
\end{array}
$$

$$
\begin{array}{l}
Post\ s\ (\texttt{call}\ ic\ sac)\ s' \stackrel{\text{def}}{=} \\
\quad s = s'
\end{array}
$$

Figura 4.11: Pre- y post-condición de la operación `call`

Este último predicado representa la revocación de permisos permanentes en el estado s'. Si dicha revocación involucra ambas operaciones (representado por $pt = Both$), entonces, se quitan las delegaciones permanentes hechas sobre el identificador u de cp para realizar cualquier operación; de lo contrario, sólo se revocan los permisos para realizar la operación especificada. En cualquiera de los dos casos, la revocación se extiende a todas las aplicaciones con permisos delegados sobre el identificador en cuestión. Análogamente, el predicado restante, $revokeTPerm$, representa la eliminación de delegaciones temporales en s'. La definición formal de este último predicado se omitirá en el presente informe para favorecer su brevedad.

4.6.6. Semántica de `call`

Por último, representamos un llamado a la API del sistema por parte de una instancia en ejecución exigiendo que la aplicación correspondiente tenga los permisos necesarios para ello. Esto último se especifica en la pre-condición de `call` de la figura 4.11 utilizando el predicado $permSAC$, de tipo $\forall p : Perm \bullet isSystemPerm\ p \rightarrow SACall \rightarrow Prop$, tal que $permSAC\ p\ H\ sac$ se cumplirá si el permiso predefinido por el sistema p es requerido para llamar a sac.

4.7. Ejecución de Operaciones

La ejecución de una operación act, de las descritas en la sección 4.5, sobre un estado válido se representa con la relación $\hookrightarrow$, perteneciente a $State \times Action \times State$. A continuación se brinda la definición de dicha relación, en donde $(s, act, s') \in \hookrightarrow$ se nota como $s \overset{act}{\hookrightarrow} s'$:

$$\frac{validState\ s \qquad Pre\ act \qquad Post\ s\ act\ s'}{s \overset{act}{\hookrightarrow} s'}$$

Intuitivamente, $s \overset{act}{\hookrightarrow} s'$ se cumplirá si el estado (válido) s reúne las condiciones necesarias para la ejecución de act y s' representa el efecto producido por la misma. De esta forma, esta relación modela una transición de estados en el sistema.

Capítulo 5

Verificación

En este capítulo se probarán formalmente algunas propiedades con respecto a la especificación descripta en el capítulo 4. Dichas propiedades se dividieron en dos grandes grupos; el primero de ellos es desarrollado en la sección 5.1 y consiste en una única propiedad que asegura que la correspondencia entre el modelo formal y el sistema real se mantiene a lo largo de las ejecuciones de las operaciones representadas; el segundo grupo, descripto en la sección 5.2, comprende un conjunto de propiedades destinadas a analizar distintos aspectos del modelo de seguridad de Android.

El desarrollo y demostración de las propiedades incluidas en el presente capítulo también fueron hechas con el asistente de pruebas Coq, por lo que los archivos correspondientes a las mismas completarán lo expuesto en las secciones subsiguientes. Al igual que los archivos referidos en el capítulo 4, los mismos se encuentran en `http://www.fing.edu.uy/inco/grupos/gsi/index.php?page=proygrado&locale=es`.

5.1. Invarianza sobre la Validez de Estado

Lo primero que se demostró formalmente con respecto al modelo construido es que si se ejecuta cualquier operación del tipo *Action* sobre un estado válido, el estado resultante también será válido. De esta forma, nos aseguramos que dichas operaciones mantienen la correspondencia planteada en la sección 4.4 entre el modelo y el sistema Android, evitando que se llegue a estados del modelo que no podrían reproducirse en el sistema que se está representando.

Formalmente, el enunciado del teorema demostrado es el siguiente:

Teorema 1 $\forall(act : Action)(s\ s' : State) \bullet s \overset{act}{\hookrightarrow} s' \Rightarrow validState\ s'$

Para probar este teorema se debe mostrar que cada una de las propiedades que componen el predicado *validState* (ver sección 4.4) se cumple en los estados resultantes de aplicar cada operación. La metodología escogida

para lograrlo fue dividir la demostración en distintos lemas, uno por cada propiedad presente en *validState*. Los enunciados de los mismos son, en su mayoría, similares al del teorema 1, reemplazando el predicado *validState* por la propiedad correspondiente. Por ejemplo, el lema asociado a la propiedad $allAppDifferent$ se enuncia de la siguiente forma:

Lema 1 $\forall(act : Action)(s\ s' : State) \bullet s \overset{act}{\hookrightarrow} s' \Rightarrow allAppDifferent\ s'$

Teniendo demostrados los lemas descriptos anteriormente, éstos pueden ser utilizados para probar cada una de las propiedades en $validState\ s'$ del teorema 1. En cuanto a las hipótesis de estos lemas, las mismas son satisfechas por la del teorema en cuestión.

De forma análoga, el lema 1 muestra que para dar una prueba del mismo se deben considerar todas las operaciones especificadas en el tipo *Action*. Dado que esta particularidad se extiende al resto de los lemas, la demostración de todos ellos es también dividida en otros lemas auxiliares; uno por cada valor posible de $act : Action$. A continuación, se muestran dos ejemplos de estos últimos junto con sus demostraciones.

Lema 2 $\forall(a : App)(s\ s' : State) \bullet s \xrightarrow{\texttt{install}\ a} s' \Rightarrow allAppDifferent\ s'$

Demostración Sean $a : App$ y $s, s' : State$ tales que verifican $s \xrightarrow{\texttt{install}\ a} s'$, nuestro objetivo es probar $allAppDifferent\ s'$, cuya definición es la siguiente (ver sección 4.4):

$$\forall(a_1\ a_2 : App) \bullet s'.apps\ a_1 \land s'.apps\ a_2 \Rightarrow a_1.idAp = a_2.idAp \Rightarrow a_1 = a_2$$

Dadas dos aplicaciones a_1 y a_2 instaladas en s' tales que $a_1.idAp = a_2.idAp$, si probamos que a_1 y a_2 son en realidad la misma aplicación, habremos concluido la demostración.

Como primer paso, utilizaremos la hipótesis $s \xrightarrow{\texttt{install}\ a} s'$ y la definición de $\hookrightarrow$ (ver sección 4.7) para concluir que a, s y s' cumplen con la postcondición de `install` a. Como consecuencia, el predicado $addApp\ s\ s'\ a$ de la misma también se verifica (ver figura 4.2), por lo que podemos asegurar lo expresado en la segunda fórmula de este último predicado:

$$\forall a' : App \bullet s'.apps\ a' \Rightarrow s.apps\ a' \lor a' = a \tag{2.1}$$

Es decir, cada aplicación instalada en s' o bien ya estaba en s o se trata de la aplicación a recién instalada con la operación `install`. En particular, a_1 y a_2 son dos aplicaciones instaladas en s', por lo que se verifica la proposición $(s.apps\ a_1 \lor a_1 = a) \land (s.apps\ a_2 \lor a_2 = a)$. Aplicando la propiedad distributiva concluimos que al menos uno de los cuatro casos pertenecientes a la disyunción resultante debe cumplirse y, por lo tanto, dividimos la prueba en dichos casos:

Caso 1. $s.apps\ a_1 \wedge s.apps\ a_2$

Utilizando nuevamente la hipótesis $s \xrightarrow{\texttt{install}\ a} s'$, podemos concluir que se cumplen $validState\ s$ y, por formar parte de dicho predicado, $allAppDifferent\ s$. A partir de esta última fórmula (cuya definición ya fue expandida anteriormente para s') y de las hipótesis $s.apps\ a_1 \wedge s.apps\ a_2$ y $a_1.idAp = a_2.idAp$, se sigue inmediatamente que $a_1 = a_2$.

Caso 2. $s.apps\ a_1 \wedge a_2 = a$

En este caso, a_1 es una aplicación que ya estaba instalada en el estado s y a_2 es la aplicación a recién instalada. Apelando una vez más a la hipótesis $s \xrightarrow{\texttt{install}\ a} s'$ y la definición de $\hookrightarrow$, concluimos que s cumple la pre-condición de `install` y, en particular, también una de las propiedades que la componen:

$$\forall a' : App \bullet s.apps\ a' \Rightarrow a.idAp \neq a'idAp \tag{2.2}$$

Es decir, como sabemos que a se instaló exitosamente a partir de s, entonces su identificador no puede estar asignado a una aplicación en dicho estado. Consecuentemente, como a_1 está instalada en s, se cumple que $a.idAp \neq a_1.idAp$. Sin embargo, también sabemos por hipótesis que $a_1.idAp = a_2.idAp$ y que, en este caso, $a_2 = a$ (por lo que $a_2.idAp = a.idAp$). Como resultado, se cumplen simultáneamente $a.idAp = a_2.idAp = a_1.idAp$ y $a.idAp \neq a_1.idAp$, llegando así a un absurdo.

Dado que en este caso las hipótesis se contradicen entre sí, el mismo no puede ocurrir y es descartado.

Caso 3. $a_1 = a \wedge s.apps\ a_2$

Utilizando un razonamiento análogo al del caso anterior, se concluye que éste tampoco puede darse y es descartado.

Caso 4. $a_1 = a \wedge a_2 = a$

En este caso se prueba trivialmente que $a_1 = a = a_2$.

□

Lema 3 $\forall(ic : iCmp)(c : Cmp)(s\ s' : State) \bullet s \xrightarrow{\texttt{start}\ ic\ c} s' \Rightarrow allAppDifferent\ s'$

Demostración Sean $ic : iCmp$, $c : Cmp$ y $s, s' : State$ tales que verifican $s \xrightarrow{\texttt{start}\ ic\ c} s'$ y, como consecuencia de la definición de $\hookrightarrow$ y $validState$, $allAppDifferent\ s$. Expandiendo esta última definición, obtenemos la siguiente hipótesis equivalente:

$$\forall(a_1\ a_2 : App) \bullet s.apps\ a_1 \wedge s.apps\ a_2 \Rightarrow a_1.idAp = a_2.idAp \Rightarrow a_1 = a_2 \tag{3.1}$$

Por otro lado, dado que $s \xrightarrow{\texttt{start}\ ic\ c} s'$ se cumple, entonces $Post\ s$ (`start` $ic\ c$) s' se verifica. Como consecuencia, se puede asegurar que $s.apps = s'.apps$ (ver figura 4.4); es decir, que las aplicaciones instaladas se mantienen inalteradas de un estado al otro.

Combinando este último hecho con la fórmula 3.1, obtenemos:

$$\forall(a_1\ a_2 : App) \bullet s'.apps\ a_1 \wedge s'.apps\ a_2 \Rightarrow a_1.idAp = a_2.idAp \Rightarrow a_1 = a_2$$

Por último, sólo resta notar que este último predicado es exactamente la definición de $allAppDifferent\ s'$, que es lo que se quería probar.

□

Como la prueba completa del teorema 1 tiene una extensión considerable, concluiremos esta sección dando únicamente lo explicado hasta ahora. La demostración en cuestión se encuentra disponible en el archivo **Inv.v** que, al mismo tiempo, depende de los lemas asociados a las propiedades de *validState*. El nombre de los archivos que contienen cada uno de estos lemas es igual al de su propiedad asociada (por ejemplo, el archivo correspondiente al lema 1 es **allAppDifferent.v**). Dado que todas las pruebas se desarrollaron en Coq, la validez de las mismas fue chequeada por dicho asistente.

5.2. Propiedades de Seguridad

En esta sección se realiza un análisis formal del modelo de seguridad de Android mediante la descripción de distintas propiedades que se cumplen en el sistema representado. Estas propiedades se agruparán según el aspecto del sistema que se quiera describir y, en su mayoría, no estarán acompañadas de sus demostraciones para favorecer la brevedad de la presente sección. Sin embargo, las pruebas de estas propiedades pueden ser obtenidas de los archivos Coq disponibles en la dirección provista al principio del presente capítulo.

Para la formalización de las propiedades del sistema se declaran los elementos $ic : iCmp$, $a : App$, $c : Cmp$, $cp : CProvider$, $u : uri$, $act : Activity$ y $s, s' : State$, asumiendo que se cumplen $validState\ s$, $s.running\ ic\ c$, $inApp\ c\ a$, $s.apps\ a$, $existsRes\ cp\ u\ s$ y $cmpInstalled\ (cmpAct\ act)\ s$. De esta forma, todas las propiedades en esta sección tendrán cuantificados universalmente (de manera implícita) dos estados s y s' tales que s es un estado válido en el que está corriendo la instancia ic del componente c, perteneciente a la aplicación a también instalada en dicho estado. Adicionalmente, se asume que el identificador u del *content provider* cp apunta a un recurso existente en s y la actividad act pertenece a una aplicación instalada. En caso de necesitar otros elementos para la formulación de ciertas propiedades, los mismos serán declarados en la sección o enunciado correspondiente.

5.2.1. Principio de Mínimo Privilegio

Como se explicó en la sección 3.1.1, la propiedad más importante enunciada por Android sobre su modelo de seguridad es que cumple con el principio de *mínimo privilegio*; es decir, que "cada aplicación, por defecto, tiene acceso únicamente a los componentes que requiere para trabajar" [5], sin poder acceder a sectores del sistema para los que no tiene los permisos adecuados. A continuación, se enunciarán distintos lemas que apuntan evaluar la validez de esta propiedad al iniciar un componente, leer/escribir un *content provider* o delegar y revocar permisos en diversos escenarios posibles.

Inicio de Componentes

Los siguientes lemas describen un conjunto de condiciones suficientes y necesarias para poder iniciar un componente en Android. Para la formulación de los mismos contaremos con los elementos $c_1, c_2 : Cmp$, $a_1, a_2 : App$ y $ic_1 : iCmp$, asumiendo que $s.apps\ a_1$ y $s.apps\ a_2$ se verifican.

Lema 4 *Si c_1 y c_2 pertenecen a la misma aplicación, entonces c_1 podrá iniciar a c_2:*

$$s.running\ ic_1\ c_1 \land inApp\ c_1\ a_1 \land inApp\ c_2\ a_1 \land \neg isCProvider\ c_2 \Rightarrow Pre\ s\ (\texttt{start}\ ic_1\ c_2)$$

Lema 5 *Si c_2 no está exportado y c_1 pertenece a otra aplicación, entonces c_1 no podrá iniciar a c_2:*

$$\forall (H : inApp\ c_2\ a_2) \bullet s.running\ ic_1\ c_1 \land inApp\ c_1\ a_1 \land a_1 \neq a_2 \land \neg (a_2.manifest).exp\ c_2\ H \Rightarrow \neg Pre\ s\ (\texttt{start}\ ic_1\ c_2)$$

Lema 6 *Si c_1 y c_2 pertenecen a aplicaciones distintas y c_2 exige un permiso que la aplicación de c_1 no tiene, c_1 no podrá iniciar a c_2:*

$$\forall (H_1 : inApp\ c_1\ a_1)(H_2 : inApp\ c_2\ a_2)(p : Perm) \bullet a_1 \neq a_2 \land (a_2.manifest).cmpE\ c_2\ H_2 = Some\ p \land \neg s.perms\ a_1\ p \land s.running\ ic_1\ c_1 \Rightarrow \neg Pre\ s\ (\texttt{start}\ ic_1\ c_2)$$

Lema 7 *Asumiendo que c_1 y c_2 pertenecen a aplicaciones distintas, si c_2 no exige ningún permiso pero su aplicación sí lo hace y, paralelamente, la aplicación correspondiente a c_1 no tiene este último permiso, c_1 no podrá iniciar a c_2.*

$$\forall (H_1 : inApp\ c_1\ a_1)(H_2 : inApp\ c_2\ a_2)(p : Perm) \bullet a_1 \neq a_2 \land (a_2.manifest).cmpE\ c_2\ H_2 = None \land (a_2.manifest).appE = Some\ p \land \neg s.perms\ a_1\ p \land s.running\ ic_1\ c_1 \Rightarrow \neg Pre\ s\ (\texttt{start}\ ic_1\ c_2)$$

Todos las propiedades del sistema enunciadas por los lemas precedentes se encuentran en concordancia con el principio de mínimo privilegio. Las demostraciones de los mismos se encuentran en el archivo **canStart.v** y su validez fue chequeada por Coq.

Lectura/Escritura de *Content Providers*

La primera propiedad probada en este caso es una condición necesaria para la lectura de un *content provider* en un escenario general.

Lema 8 *Si ic pudo leer el recurso apuntado por u en cp, entonces su componente asociado pertenece a una aplicación que tiene los permisos para ello, ya sea desde su instalación o por medio de una delegación.*

$$s \xrightarrow{\texttt{read}\ ic\ cp\ u} s' \Rightarrow canRead\ c\ cp\ u\ s \vee delPerms\ c\ cp\ u\ Read\ s$$

La condición necesaria para realizar una operación de escritura se define de forma análoga a la anterior, cambiando la operación `read` y el predicado *canRead* por `write` y $canWrite$ respectivamente, por lo que el enunciado del lema correspondiente será omitido. Debido a que esta similitud se repite en todos los lemas probados en la sección, sólo se describirán las propiedades que involucran lecturas.

A continuación, se analiza el sistema de permisos para la operación de lectura en dos escenarios específicos.

Lema 9 *Si cp pertenece a la misma aplicación que c, entonces c puede leer a cp:*

$$inApp\ (cmpCP\ cp)\ a \Rightarrow Pre\ s\ (\texttt{read}\ ic\ cp\ u)$$

Lema 10 *Si cp no está exportado y c pertenece a otra aplicación, entonces cp puede ser leído por c si y sólo si su aplicación correspondiente tiene permisos delegados para hacerlo:*

$$\forall(a' : App)(H : inApp\ (cmpCP\ cp)\ a') \bullet$$
$$s.apps\ a' \wedge a \neq a' \wedge \neg(a'.manifest).exp\ (cmpCP\ cp)\ H \Rightarrow$$
$$Pre\ s\ (\texttt{read}\ ic\ cp\ u) \iff delPerms\ c\ cp\ u\ Read\ s$$

Demostración Sean $a, a' : App$, $ic : iCmp$, $c : Cmp$, $cp : CProvider$, $u : uri$ y $s, s' : State$ tales que, además de cumplir lo asumido al principio de la sección 5.2, verifican $s.apps\ a'$, $a \neq a'$ y $\neg(a'.manifest).exp\ (cmpCP\ cp)\ H$, donde H es una prueba de $inApp\ (cmpCP\ cp)\ a'$, que también se verifica.

($\Rightarrow$) Suponiendo la validez de $Pre\ s$ (`read` $ic\ cp\ u$), nos proponemos demostrar $delPerms\ c\ cp\ u\ Read\ s$.

Expandiendo la pre-condición de `read` (ver figura 4.6), podemos asegurar que existe un componente c^* que verifica $s.running\ ic\ c^*$ y $canRead\ c^*\ cp\ u\ s$

$\vee$ $delPerms$ c^* cp u $Read$ s. Como s es un estado válido, entonces, según la propiedad 9, una misma instancia en ejecución no puede pertenecer a dos componentes distintos en s. Por lo tanto, dado que se cumplen $s.running\ ic\ c$ (suposición hecha al principio de la sección 5.2) y $s.running\ ic\ c^*$ al mismo tiempo, entonces $c = c^*$ y, consecuentemente, se verifica $canRead\ c\ cp\ u\ s$ $\vee$ $delPerms\ c\ cp\ u\ Read\ s$. Como queremos demostrar que únicamente la segunda proposición se cumple, probaremos que la primera es falsa. Para ello hay que tener en cuenta que si se cumple $canRead\ c\ cp\ u\ s$, entonces, por su definición, debe valer $(a'.manifest).exp\ (cmpCP\ cp)\ H$ o $a = a'$, contradiciendo, en ambos casos, las hipótesis hechas al principio.

($\Leftarrow$) Suponiendo la validez de $delPerms\ c\ cp\ u\ Read\ s$, nos proponemos demostrar $Pre\ s$ (`read` $ic\ cp\ u$).

A continuación, probaremos cada proposición perteneciente a la precondición de `read` (ver figura 4.6):

- $cmpInstalled\ (cmpCP\ cp)\ s$. Para que esta proposición sea válida, debe existir una aplicación instalada en s que contenga a cp. Dado que la aplicación a' cumple, por hipótesis, con el predicado $s.apps\ a'$ y existe la prueba $H : inApp\ (cmpCP\ cp)\ a'$, podemos concluir que la proposición en cuestión se verifica.

- $existsRes\ cp\ u\ s$. Se cumple por lo asumido al principio de la sección 5.2.

- $\exists c^* : Cmp \bullet s.running\ ic\ c^* \wedge \neg isCProvider\ c^* \wedge (canRead\ c^*\ cp\ u\ s \vee delPerms\ c^*\ cp\ u\ Read\ s)$. Para probar esta cuantificación existencial usaremos el componente c de testigo. De esta forma, $s.running\ ic\ c$ se cumple por lo asumido al principio de la sección 5.2 y $canRead\ c\ cp\ u\ s$ $\vee$ $delPerms\ c\ cp\ u\ Read\ s$ por suponer la validez de $delPerms\ c\ cp\ u\ Read\ s$ para este caso. Por último, utilizando nuevamente la validez el estado s, podemos aplicar la propiedad 4 para concluir que, como $s.running\ ic\ c$ se verifica, entonces $\neg isCProvider\ c$ también.

□

La validez del lema 10 muestra que aunque un componente no esté exportado, el mismo podría ser accedido por aplicaciones externas.

Los enunciados y demostraciones de todos los lemas ligados a esta sección se encuentran en el archivo **canReadWrite.v**.

Delegación y Revocación de Permisos

En primer lugar, se probaron propiedades para asegurar que una aplicación sólo puede delegar permisos (temporales o permanentes) para realizar operaciones que ella misma está autorizada a hacer. Sólo se describe el lema

correspondiente a la operación de lectura ya que para los casos de la operación de escritura y ambas operaciones, el enunciado es análogo. Además, a lo largo de esta sección se asumirá la validez de *canStart c* (*cmpAct act*) *s*; es decir, que el componente *c* tiene los permisos necesarios para iniciar la actividad *act*.

Lema 11 *Si ic delegó permisos, temporales o permanentes, para leer un recurso apuntado por u de cp, entonces ic puede realizar la operación en cuestión:*

$$\forall a' : App \bullet s \xrightarrow{\texttt{grantT}\ ic\ cp\ act\ u\ Read} s' \vee s \xrightarrow{\texttt{grantP}\ ic\ cp\ a'\ u\ Read} s' \Rightarrow canRead\ c\ cp\ u\ s \vee delPerms\ c\ cp\ u\ Read\ s$$

Por otro lado, se probó que un componente puede habilitar a una aplicación externa para realizar una operación determinada sobre cualquier *content provider* de la misma aplicación, sin necesidad de contar con los permisos adecuados para realizar la operación en cuestión.

Lema 12 *Si c y cp pertenecen a la misma aplicación y cp autoriza la delegación sobre u, entonces ic puede delegar permisos, temporales o permanentes, sobre u:*

$$\forall (a' : App)(pt : PType) \bullet inApp\ (cmpCP\ cp)\ a \wedge s.apps\ a' \wedge canGrant\ cp\ u\ s \Rightarrow Pre\ s\ (\texttt{grantT}\ ic\ cp\ act\ u\ pt) \wedge Pre\ s\ (\texttt{grantP}\ ic\ cp\ a'\ u\ pt)$$

Como consecuencia de esta última propiedad y del lema 10 (junto con sus dos lemas análogos), existe la posibilidad de que cualquier aplicación externa obtenga permisos delegados para acceder a un *content provider* no exportado. Esta posibilidad no es contemplada por la documentación oficial de Android, referida en la sección 3.1.5, a la hora de describir el significado de un componente no exportado.

Para el caso de la revocación de permisos, describimos el siguiente lema correspondiente a la operación de lectura.

Lema 13 *Si ic revocó permisos para leer el recurso apuntado por u en cp, entonces ic puede realizar la operación en cuestión:*

$$s \xrightarrow{\texttt{revoke}\ ic\ cp\ u\ Read} s' \Rightarrow canRead\ c\ cp\ u\ s$$

Al igual que para el caso de las delegaciones, las propiedades asociadas al resto de las operaciones serán omitidas por formularse de manera análoga.

Los enunciados y pruebas de los lemas de esta sección se encuentran en los archivos **canGrant.v** y **canRevoke.v**.

5.2.2. Revocación Irrestricta

A continuación, se describe una propiedad que destaca la imposibilidad de revocar permisos delegados a una aplicación/instancia en particular.

Lema 14 *Si ic revocó los permisos sobre el recurso apuntado por u de cp, entonces se eliminarán todos los permisos delegados asociados a u:*

$$s \xrightarrow{\texttt{revoke}\ ic\ cp\ u\ pt} s' \Rightarrow$$
$$(\forall (ic' : iCmp)(c' : Cmp) \bullet s.running\ ic'\ c' \Rightarrow \neg s'.delTPerms\ ic'\ cp\ u\ pt) \wedge$$
$$(\forall (a' : App) \bullet s.apps\ a' \Rightarrow \neg s'.delPPerms\ a'\ cp\ u\ pt)$$

Una consecuencia directa de esta propiedad es que un componente en ejecución puede revocar permisos que no fueron delegados por él mismo. Esto da lugar a que aplicaciones que no están relacionadas unas con otras puedan revocarse permisos entre sí, habilitando escenarios potencialmente confusos [37].

La demostración formal de la propiedad referida anteriormente se encuentra en el archivo **rvkCoarseGrnd.v**.

5.2.3. Redelegación de Permisos

En esta sección se formaliza el hecho en el que un permiso puede ser redelegado indefinidamente por cualquier componente en ejecución.

Lema 15 *Si ic o la aplicación a' tienen un permiso delegado, éste puede ser redelegado, tanto de forma temporal como permanente:*

$$\forall (a'\ a'' : App) \bullet s.apps\ a'' \wedge s.delTPerms\ ic\ cp\ u\ pt \vee$$
$$s.delPPerms\ a'\ cp\ u\ pt \Rightarrow$$
$$Pre\ s\ (\texttt{grantT}\ ic\ cp\ act\ u\ pt) \wedge Pre\ s\ (\texttt{grantP}\ ic\ cp\ a''\ u\ pt)$$

Como corolario de este lema, si a un componente en ejecución le fue delegado un permiso de forma temporal, entonces este componente o cualquier otro de la misma aplicación puede redelegar el permiso permanentemente a dicha aplicación. De esta forma, si se tiene un permiso delegado temporalmente, también se lo puede tener permanentemente [37]. Esta particularidad provoca que ambos tipos de delegación no sean muy diferentes entre sí en la práctica. Por ejemplo, un componente puede recibir un permiso delegado a través de un *intent* con el fin de que dicho permiso valga mientras dure la ejecución del componente en cuestión. Sin embargo, este último tiene la posibilidad de extender la vigencia del permiso para que sólo pueda ser revocado a través del método `revokeUriPermission()`; incumpliendo el objetivo original.

Además, combinando la propiedad descripta en esta sección con los lemas 10, 12 y sus derivados, concluimos que el acceso a un *content provider* no exportado no es exclusivo para aplicaciones que recibieron la delegación

directamente de la dueña del mismo, sino que se extiende a otras aplicaciones a través de la redelegación.

La prueba del lema 15 se encuentra disponible en el archivo **canRedel.v**.

5.2.4. *Privilege Escalation*

La problemática conocida como *privilege escalation* ocurre cuando una aplicación puede realizar tareas para las que no cuenta con los permisos correspondientes, utilizando como intermediaria a una tercera que sí cuente con ellos, denominada *representante*. "En este escenario, el usuario le concede al *representante* un determinado permiso. El *representante* define una interfaz pública que expone parte de su funcionalidad. Una aplicación maliciosa que no posee el permiso que tiene el *representante* invoca la interfaz del mismo, causando que este último haga una llamada a la API del sistema. El sistema aprueba y ejecuta la llamada a la API proveniente del *representante* porque el mismo tiene el permiso adecuado para hacerlo. De esta forma, la aplicación maliciosa logra ejecutar una llamada a la API del sistema que no hubiese podido hacer directamente."[36]

A continuación, se enuncia el lema en donde se afirma que este problema está presente en Android.

Lema 16 *Si ic no puede realizar una llamada al sistema sac pero inicia un componente c' que sí puede hacerlo, entonces, luego de iniciar a c', se podrá llamar a sac a través de una instancia en ejecución del mismo:*

$$\begin{aligned}\forall(c' : Cmp)(sac : SACall)\bullet\ \neg Pre\ s\ (\texttt{call}\ ic\ sac) \wedge \neg Pre\ s'\ (\texttt{call}\ ic\ sac)\\ \wedge\ cmpCanCall\ c'\ sac\ s' \wedge s \xrightarrow{\texttt{start}\ ic\ c'} s' \Rightarrow\\ \exists ic' : iCmp \bullet s'.running\ ic'\ c' \wedge Pre\ s'\ (\texttt{call}\ ic'\ sac)\end{aligned}$$

Para la formulación del lema anterior se utilizó el predicado auxiliar *cmpCanCall c sac s*, que se cumplirá si el componente *c* está instalado en el estado del sistema *s* y su aplicación correspondiente puede realizar la llamada *sac* a la API del sistema. Formalmente:

$$\begin{aligned}&cmpCanCall\ c\ sac\ s \stackrel{\text{def}}{=} cmpInstalled\ c\ s\ \wedge\\ &(\forall(a : App)(p : Perm)(H : isSystemPerm\ p) \bullet s.apps\ a \wedge inApp\ c\ a\ \wedge\\ &\quad permSAC\ p\ H\ sac \Rightarrow s.perms\ a\ p)\end{aligned}$$

Al probar el lema 16 podremos asegurar que dado un componente en ejecución *ic* que no puede realizar una determinada llamada al sistema (en términos de la definición informal, una aplicación maliciosa) y un componente c' que sí puede hacerlo (el *representante*), si *ic* inicia a c' (equivalente en nuestro modelo a "invocar la interfaz pública del *representante*"), entonces va a existir un componente en ejecución de c' que pueda realizar la llamada

a la API que no puede hacer *ic*. Dentro del conjunto de componentes en ejecución de c' se encuentra el iniciado por *ic* (es por ello que siempre va a existir al menos uno); por lo tanto, si tenemos en cuenta dicha instancia, *ic* logró, poniendo a c' como intermediario, generar las condiciones para que se pueda ejecutar, no necesariamente de forma inmediata, la llamada al sistema que tenía prohibida, como se menciona en la descripción coloquial.

Paralelamente, en otros trabajos publicados [28, 37] se extiende la definición de *privilege escalation* a las operaciones entre componentes. En esta nueva definición, se formula el problema en el que un componente en ejecución *ic* que no tiene los permisos adecuados para iniciar un componente c'', puede hacerlo indirectamente por medio de c'. Como se menciona en los trabajos citados, Android también padece este último problema. A continuación, proveemos el lema correspondiente a esta propiedad junto con su demostración.

Lema 17 *Si ic no puede iniciar un componente c'' pero inicia a c' que sí puede hacerlo, entonces, se podrá iniciar a c'' a través de una instancia en ejecución de c':*

$$\forall(c'\ c'' : Cmp)\bullet\ cmpInstalled\ c''\ s \land \neg isCProvider\ c'' \land \neg canStart\ c\ c''\ s$$
$$\neg canStart\ c\ c''\ s' \land canStart\ c'\ c''\ s' \land s \xrightarrow{\texttt{start}\ ic\ c'} s' \Rightarrow$$
$$\exists ic' : iCmp\bullet\ s'.running\ ic'\ c' \land Pre\ s'\ (\texttt{start}\ ic'\ c'')$$

Demostración Sean $ic : iCmp$, $c', c'' : Cmp$ y $s, s' : State$ tales que verifican $\neg isCProvider\ c''$, $cmpIntalled\ c''\ s$, $canStart\ c'\ c''\ s'$ y $s \xrightarrow{\texttt{start}\ ic\ c'} s'$, nuestro objetivo es probar que existe efectivamente una instancia $ic' : iCmp$ del componente c' que verifique $Pre\ s'$ (`start` $ic'\ c''$).

Utilizando la hipótesis $s \xrightarrow{\texttt{start}\ ic\ c'} s'$ junto con la definición de $\hookrightarrow$ (ver sección 4.7), concluimos que $Post\ s$ (`start` $ic\ c'$) s' se cumple. Expandiendo este último predicado (ver figura 4.4), podemos asegurar que existe una nueva instancia de c' corriendo en s'. Esta instancia será el testigo $ic' : iCmp$ seleccionado para probar la existencia planteada en el párrafo anterior; por lo tanto, el resto de la demostración consiste en probar cada predicado perteneciente a la conjunción de $Pre\ s'$ (`start` $ic'\ c''$):

- $cmpInstalled\ c''\ s'$. El cumplimiento de $Post\ s$ (`start` $ic\ c'$) s' también implica que $s.apps = s'.apps$; por lo tanto, si c'' pertenecía a una aplicación instalada en s (hipótesis $cmpIntalled\ c''\ s$), lo mismo se cumple en s'.

- $\neg isCProvider\ c''$. Se cumple por hipótesis.

- $\exists c^* : Cmp\bullet s'.running\ ic'\ c^* \land cmpInstalled\ c^*\ s' \land \neg isCProvider\ c^* \land canStart\ c^*\ c''\ s'$. Para probar esta cuantificación existencial usaremos de testigo a c', componente correspondiente a ic' en s'. Luego,

s.running ic′ c′ se cumple por la misma definición de *ic′* y *canStart c′ c″ s′* por hipótesis. Utilizando nuevamente la proposición $s \xrightarrow{\texttt{start}\ ic\ c'} s'$, obtenemos que *Pre s* (`start` *ic c′*) se verifica, por lo que, en particular, también se cumple $\neg isCProvider\ c'$. Adicionalmente, la validez de la pre-condición implica que *c′* está instalado en *s* y, utilizando el hecho $s.apps = s'.apps$ derivado más arriba, concluimos que también se cumple $cmpInstalled\ c'\ s'$.

□

Análogamente al lema 16, *ic* es el componente malicioso, *c′* el *representante* y *c″* el componente que se quiere iniciar. Enfocándonos en la instancia de *c′* que inició *ic* (la única que podemos asegurar que existe a partir de las premisas), se genera la posibilidad de acceder a *c″*.

Las definiciones y demostraciones desarrollados en Coq correspondientes a esta sección se encuentran en el archivo **privEscalation.v**.

Capítulo 6

Trabajos Relacionados

El objetivo del presente capítulo es dar una amplia descripción del estado del arte para poder comparar el trabajo presentado con el resto de las publicaciones consultadas. Este objetivo se alcanzará describiendo un conjunto de trabajos disponibles en la literatura que también analizan el modelo de seguridad de Android, tomando enfoques distintos al utilizado en la presente tesina. Los trabajos más alejados de nuestra metodología son los que reportan fallas puntuales que fueron detectadas sin utilizar métodos formales y los más cercanos son los que, si bien realizan un análisis formal, utilizan modelos o herramientas que difieren de las nuestras.

Entre los trabajos que reportan fallas puntuales en el modelo de seguridad de Android[1], Armando *et al.* [24] muestran, a través de pruebas empíricas sobre el código fuente, que la implementación del modelo de seguridad no discrimina de dónde provienen las llamadas al sistema 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 a través de código nativo 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 [23] 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 sis-

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

tema `socket` y `connect`. Para realizar este ataque sólo basta con que dicha aplicación cuente con el permiso para acceder a internet[2]. En las capas superiores de la arquitectura, por ejemplo, los trabajos de Davi *et al.* [33] y Felt *et al.* [36] exponen la problemática conocida como *privilege escalation*. La descripción aportada por este último trabajo se tomó de referencia en la sección 5.2.4 al probar formalmente que dicho problema ocurre efectivamente en nuestro modelo. Por último, Chin *et al.* [30] 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.* [35] realizan un análisis detallado del diseño e implementación del sistema de permisos en Android. Dicho análisis informal 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.* [31] realizan críticas al modelo de seguridad 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.* [43] 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.* [22] proponen un modelo para representar los elementos y la semántica operacional del sistema Android, poniendo atención 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* [26] 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 cual-

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

quier comportamiento que surja de la expresión. Por lo tanto, si se prueban propiedades sobre la *history expression* (utilizando técnicas ya conocidas [25]), 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.* [28] 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 [29] 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.

Por último, las publicaciones de Fragkaki *et al.* [37] y Shin *et al.* [48] fueron introducidas en el capítulo 1 por ser las más relevantes para nuestro trabajo. En cuanto a esta última publicación, el formalismo ofrecido en la misma, a diferencia de lo presentado en la presente tesina, no diferencia los distintos tipos de componentes (todos están representados por el mismo tipo $COMPS$) y no representa las instancias en ejecución de los mismos. Estas limitaciones impiden modelar características que involucren a un tipo de componente particular, como se hace en nuestra especificación. Por otro lado, también consideramos que los tipos que representan un estado del sistema y un archivo `AndroidManifest` tienen campos innecesarios, por lo que la definición de los mismos también difieren con nuestro trabajo. Por ejemplo, el tipo $STATE$ tiene dos campos distintos destinados a representar tanto las aplicaciones como los componentes instalados en un estado determinado; dado que los componentes instalados pueden ser obtenidos de los archivos `AndroidManifest` de las aplicaciones presentes en el sistema, se optó por omitir este último campo en la presente tesina. Paralelamente, la ejecución de una acción se representa como la adición de una operación entre dos componentes instalados al estado del sistema. En nuestro enfoque, dicha ejecución no es registrada en el sistema, por lo que no es necesario definir una operación para indicar la finalización de la misma. Por último, vale la pena volver a destacar que el trabajo en cuestión no estudia la lectura/escritura de *content providers*, el mecanismo de delegación de permisos conocido como *URI permissions* y las llamadas a la API del sistema por parte de una aplicación; aspectos comprendidos en nuestra especificación. Tomando en

cuenta las diferencias concretas mencionadas (y las que todavía restan por describir), se puede respaldar lo asegurado en el capítulo 1; a saber, que la especificación propuesta en este trabajo incluye y extiende los resultados de la publicación de Shin *et al.*, utilizando para ello un enfoque alternativo.

Capítulo 7

Conclusiones

A pesar de la creciente relevancia que tienen los mecanismos de seguridad implementados en el sistema Android, no existe una especificación oficial rigurosa (ni siquiera informal) de los mismos. Esto tiene, por lo menos, dos consecuencias directas: la primera de ellas es que no se puede realizar una verificación exhaustiva del modelo de seguridad en donde se analicen las distintas relaciones entre sus elementos; la segunda es que los desarrolladores de nuevas aplicaciones no cuentan con una documentación precisa para poder interactuar correctamente con el modelo de seguridad, por lo que dichas aplicaciones pueden llegar a ser inseguras.

La presente tesina realiza distintos aportes orientados a mitigar la problemática descripta en el párrafo anterior. En primer lugar, la descripción informal ofrecida en los capítulos 2 y 3 complementan la información brindada por la documentación oficial con la provista en publicaciones académicas (mencionadas, en su mayoría, en el capítulo 6) y libros especializados. De esta forma, estos capítulos aportan una visión global y a la vez exhaustiva del modelo de seguridad en Android que pretende ser tanto un punto de referencia como también un disparador inicial para trabajos futuros sobre el mismo. Otra de las principales contribuciones de esta tesina es una especificación formal del modelo de seguridad de Android. Dicha especificación constituye, en principio, el primer formalismo desarrollado en Coq dedicado al estudio del mecanismo de delegación de permisos y la interacción con la API del sistema. Finalmente, en la sección 5.2 se reunieron y probaron propiedades enunciadas en distintas publicaciones (por ejemplo, las enunciadas en las secciones 5.2.2, 5.2.3 y 5.2.4) y se demostraron propiedades no estudiadas hasta el momento (por ejemplo, la propiedad/lema 12 y sus derivados). Estas nuevas propiedades destacan que ningún *content provider* puede estar completamente aislado del resto de las aplicaciones, a pesar de la posibilidad de construir componentes no exportados. Este hecho debe ser tenido en cuenta por los desarrolladores de aplicaciones para no incurrir en la contradicción de declarar un *content provider* como no exportado y luego

Definiciones Auxiliares	458
Especificación Formal	537
Invarianza sobre la Validez de Estado	2892
Propiedades de Seguridad	1072
Comentarios	636
Total	5595

Figura 7.1: Líneas de código de los archivos desarrollados en Coq

delegar permisos sobre él.

Visto como caso de estudio, este trabajo muestra que se puede construir una especificación relativamente breve (ver figura 7.1) para analizar formalmente, mediante la demostración de teoremas, características importantes de la seguridad en Android. Esto resalta la factibilidad de aplicar métodos formales a proyectos reales, haciendo que estos últimos sean más confiables.

Como trabajos futuros, una alternativa posible es completar la especificación formal iniciada en esta tesina representando otros aspectos de la seguridad en Android. A continuación, se listan algunos de ellos:

- Implementación subyacente: Utilizando un nivel de abstracción más bajo, se pueden analizar las consecuencias de utilizar como base el sistema de permisos de Linux y los posibles problemas que pueden surgir a partir de ello.

- Envío y recepción de *broadcasts*: La especificación actual no comprende el envío o recepción de *broadcasts*. Además de incorporar estos elementos al modelo, se puede analizar en detalle el mecanismo de distribución de los mismos para asegurar que no incorpora nuevas vulnerabilidades.

- *Intents* implícitos: Otro aspecto que no es representado en nuestra especificación es el referido a los *intents* implícitos. La operación `start` en el modelo actual se corresponde con el escenario en el que un componente en ejecución inicia a otro; sin embargo, no se tiene en cuenta si dicha acción se realizó a través de un *intent* explícito o implícito. Si se quiere profundizar en este último caso, hay que analizar el mecanismo en el que el sistema elige el destinatario de dichos *intents*.

- Firmas y certificados: Los aspectos relativos a las firmas de las aplicaciones y los certificados de las claves públicas no son analizados en profundidad en el presente trabajo, por lo que el estudio de los mismos constituye una extensión posible.

- *Pending intents*: El único mecanismo de delegación considerado en el presente trabajo es el denominado *URI permissions*, por lo que una posible extensión consiste en analizar el mecanismo de *pending intents*.

Al hacer esto, además de completar el análisis de todos los mecanismos de delegación de permisos ofrecidos en Android, se estudiarían las vulnerabilidades que pueden surgir a partir de las características introducidas por los *pending intents*, como por ejemplo, la posibilidad de que una aplicación le conceda su propia identidad a otra para realizar una operación determinada.

- Definición dinámica de permisos: La definición de permisos en tiempo de ejecución y la semántica del elemento <**permission-tree**> de un archivo `AndroidManifest` son cuestiones a agregar al modelo actual.

- Actualización de aplicaciones: Una de las simplificaciones realizadas para la construcción de nuestra especificación formal fue excluir la posibilidad de actualizar una aplicación. Si bien no lo consideramos como una característica indispensable del modelo de seguridad de Android, su representación haría que la especificación sea más completa.

- Conservación de recursos: Otros de los aspectos excluidos del modelo es la posibilidad que ofrece Android de conservar los recursos (i.e., datos) de una aplicación al desinstalarla. Al representar esta posibilidad, se podrían detectar nuevas vulnerabilidades que surjan a partir de ella.

A medida que se vaya completando la especificación iniciada en esta tesina, se contará con más medios para analizar formalmente el modelo de seguridad de Android de manera integral.

Bibliografía

[1] Android Developers: *Activities*. http://developer.android.com/guide/components/activities.html. Último acceso: Junio 2014.

[2] Android Developers: *Android Debug Bridge*. http://developer.android.com/tools/help/adb.html. Último acceso: Junio 2014.

[3] Android Developers: *Android NDK*. http://developer.android.com/tools/sdk/ndk/index.html. Último acceso: Junio 2014.

[4] Android Developers: *App Manifest*. http://developer.android.com/guide/topics/manifest/manifest-intro.html. Último acceso: Junio 2014.

[5] Android Developers: *Application Fundamentals*. http://developer.android.com/guide/components/fundamentals.html. Último acceso: Junio 2014.

[6] Android Developers: *Build.VERSION_CODES*. http://developer.android.com/reference/android/os/Build.VERSION_CODES.html#HONEYCOMB. Último acceso: Junio 2014.

[7] Android Developers: *Content Providers*. http://developer.android.com/guide/topics/providers/content-providers.html. Último acceso: Junio 2014.

[8] Android Developers: *Context*. http://developer.android.com/reference/android/content/Context.html#startService(android.content.Intent). Último acceso: Junio 2014.

[9] Android Developers: *Downloading the Source*. http://source.android.com/source/downloading.html. Último acceso: Junio 2014.

[10] Android Developers: *Intents and Intents Filters*. http://developer.android.com/guide/components/intents-filters.html. Último acceso: Junio 2014.

[11] Android Developers: *<manifest>*. `http://developer.android.com/guide/topics/manifest/manifest-element.html#uid`. Último acceso: Junio 2014.

[12] Android Developers: *¡path-permission¿*. `http://developer.android.com/guide/topics/manifest/path-permission-element.html`. Último acceso: Junio 2014.

[13] Android Developers: *PendingIntent*. `http://developer.android.com/reference/android/app/PendingIntent.html`. Último acceso: Junio 2014.

[14] Android Developers: *Permissions*. `http://developer.android.com/guide/topics/security/permissions.html`. Último acceso: Junio 2014.

[15] Android Developers: *R.styleable*. `http://developer.android.com/reference/android/R.styleable.html#AndroidManifestPermission_protectionLevel`. Último acceso: Junio 2014.

[16] Android Developers: *Security Tips*. `http://developer.android.com/training/articles/security-tips.html`. Último acceso: Junio 2014.

[17] Android Developers: *Services*. `http://developer.android.com/guide/components/services.html`. Último acceso: Junio 2014.

[18] Android Developers: *Signing Your Applications*. `http://developer.android.com/tools/publishing/app-signing.html#strategies`. Último acceso: Junio 2014.

[19] Android Developers: *Starting an Activity*. `http://developer.android.com/training/basics/activity-lifecycle/starting.html`. Último acceso: Junio 2014.

[20] Android Open Source Project: *Android Security Overview*. `http://source.android.com/devices/tech/security/index.html`. Último acceso: Junio 2014.

[21] Android Open Source Project: *Licenses*. `http://source.android.com/source/licenses.html`. Último acceso: Junio 2014.

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

[23] 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`.

[24] 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`.

[25] 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.

[26] 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`.

[27] 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.

[28] 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.

[29] 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`.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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`.

[40] 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.

[41] Mazeikis, Gustavo, Gustavo Betarte y Carlos Luna: *Formal Specification and Analysis of the MIDP 3.0 Security Model.* En *SCCC*, páginas 59–66, 2009.

[42] 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.

[43] 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`.

[44] Open Handset Alliance: *Open Handset Alliance.* `http://www.openhandsetalliance.com`. Último acceso: Junio 2014.

[45] Oracle: *Java Community Process.* `http://jcp.org/en/home/index`. Último acceso: Junio 2014.

[46] Oracle: *Java ME Technology Overview.* `http://www.oracle.com/technetwork/java/javame/java-me-overview-402920.html`. Último acceso: Junio 2014.

[47] Romano, Agustín y Carlos Luna: *Descripción y análisis del modelo de seguridad de Android.* Informe técnico, Universidad de la República, 2013. `www.fing.edu.uy/inco/pedeciba/bibliote/reptec/TR1308.pdf`. Último acceso: Junio 2014.

[48] 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`.

[49] 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.

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

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

[52] 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.

[53] Zanella Béguelin, Santiago, Gustavo Betarte y Carlos Luna: *A Formal Specification of the MIDP 2.0 Security Model.* En Dimitrakos, Theodosis, Fabio Martinelli, Peter Y. A. Ryan y Steve A. Schneider (editores): *Formal Aspects in Security and Trust*, volumen 4691 de *Lecture Notes in Computer Science*, páginas 220–234. Springer, 2006, ISBN 978-3-540-75226-4. `http://dblp.uni-trier.de/db/conf/ifip1-7/fast2006.html#BeguelinBL06`.