Análisis Formal del Sistema de Permisos de Android

Camila Sanz

Tutores

Gustavo Betarte, Juan Diego Campo, Carlos Luna

Informe de Proyecto de Grado presentado al Tribunal Evaluador como requisito de graduación de la carrera Ingeniería en Computación

Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Resumen

Actualmente alrededor de 1700 millones de personas en el mundo eligen dispositivos móviles con Android como sistema operativo. Este liderazgo significativo del mercado lo sitúa como un interesante blanco de ataque para quienes crean aplicaciones maliciosas y además cualquier error que viole la información sensible de los usuarios impacta sobre un gran número de víctimas. Como consecuencia, es de interés analizar su modelo de seguridad y buscar formas de fortalecerlo. Si bien existen diversos estudios sobre el tema, el cambio constante y el surgimiento de nuevos problemas permiten la existencia de líneas muy diversas de investigación que apunten al mismo objetivo. Este trabajo propone un análisis formal completo del modelo de seguridad utilizado por Android, considerando especificaciones e implementaciones existentes, y teniendo en cuenta trabajos previos en el área.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Android es una plataforma de software libre para dispositivos móviles desarrollada por Google [46] junto con la Open Handset Alliance [52]. Provee un sistema operativo, un *middleware* y algunas aplicaciones nativas como el directorio de contactos, la mensajería y el calendario, entre otras [51]. Además de las aplicaciones que están incluidas en el sistema, se permite la instalación de aplicaciones creadas por desarrolladores externos a Android que pueden ser descargadas de *Google Play* [47]. Para ello, Android ofrece el *Software Development Kit* (SDK) que brinda las herramientas necesarias para poder desarrollar aplicaciones en la plataforma utilizando el lenguaje Java.

En agosto del corriente año se liberó al público la versión 7.0 de Android denominada *Android Nougat* [5]. Este trabajo se basa en la versión 6.0 denominada *Android Marshmallow*[6].

Android implementa mecanismos de seguridad tanto a nivel de sistema operativo como a nivel de aplicaciones. A nivel de sistema operativo, al ser un sistema Linux, Android ejecuta cada aplicación en un proceso independiente aislándola del resto. En las aplicaciones, el modelo de seguridad de Android se basa fundamentalmente en su sistema de permisos. Cada aplicación posee un documento, llamado *manifest*, en donde declara cuáles permisos necesita para funcionar correctamente, y cuáles permisos deben poseer otras aplicaciones para poder iniciarla. Además, el sistema solicita explícitamente al usuario la otorgamiento de permisos a la aplicación, la primera vez que ésta los requiere, siempre que ello pueda comprometer la seguridad del dispositivo. Por ejemplo,

cuando el usuario realiza una acción con la cual la aplicación accedería, por primera vez, a la galería de imágenes.

Android está instalado en el 80 % de los dispositivos móviles del mundo, lo que le da una ventaja competitiva frente a los otros dos sistemas operativos líderes: *iOS* y *Windows Phone*, pero a la vez hace que sea más atractivo estudiar cómo violar su modelo de seguridad. Como consecuencia de ello, existen múltiples estudios sobre diversas fallas en el modelo de seguridad de Android y los ataques que dichas fallas permiten [48, 45]. En particular, hay varios trabajos sobre los problemas de seguridad provocados únicamente por la forma de interacción de las aplicaciones [54, 38]. Otras investigaciones destacan que en muchos casos las vulnerabilidades surgen de malas configuraciones por parte de quienes crean la aplicación. En este sentido en [43, 41] se estudia el ataque *privilege escalation* y en qué medida el mismo depende únicamente de malas configuraciones por parte de los desarrolladores.

En este trabajo se desarrolla una especificación formal del modelo de seguridad de Android, haciendo énfasis en el intercambio de mensajes entre aplicaciones (denominados *intents*) y el otorgamiento de permisos en tiempo de ejecución, analizando en profundidad los problemas de seguridad que cada una de estas características conlleva.

Se define un modelo que reformula y profundiza la especificación propuesta por Agustín Romano en [53], quien desarrolla un modelo formal abstracto sobre el que se demuestran propiedades que cumple el modelo de seguridad de Android. Así, el nuevo modelo, provee una especificación funcional que permite disponer de un modelo más cercano al de la plataforma real y agrega funcionalidades no previstas inicialmente, como el manejo de permisos en tiempo de ejecución (propio de Android 6.0) y el envío y recepción de *intents*. Tanto la especificación como el enunciado de teoremas y sus respectivas pruebas formales, disponibles en `https://www.fing.edu.uy/inco/grupos/gsi/documentos/proyectos/pgextension/pgextension.tar.gz`, se desarrollaron en el asistente de pruebas `Coq` [57].

El modelo desarrollado define las condiciones necesarias para que una operación pueda llevarse a cabo en un sistema Android y los cambios que su ejecución genera. Por ejemplo, una condición para que una aplicación pueda ser desinstalada de un dispositivo con Android es estar previamente instalada. Una vez ejecutada la operación de desinstalación, la aplicación y todos los elementos que

la constituyen deberán estar totalmente eliminados del sistema. Además todos los permisos que la aplicación hubiera delegado, deben ser revocados.

El objetivo de este trabajo es analizar formalmente aspectos críticos del modelo de seguridad, detectar y demostrar formalmente escenarios en los cuales se evitan ciertos ataques propios de la comunicación entre aplicaciones y encontrar posibles vulnerabilidades existentes en un sistema Android.

Contribuciones

En este proyecto se realizan las siguientes contribuciones:

- Un estudio detallado del modelo de seguridad de Android.
- Una especificación formal que comprende a los elementos básicos de un sistema Android, haciendo foco en aquellos que influyen en su modelo de seguridad.
- Un análisis de propiedades de seguridad que ayudan a comprender el comportamiento del modelo de seguridad de Android.
- Detección y demostración de que en ciertos escenarios algunos ataques conocidos no pueden llevarse a cabo.
- Un conjunto de problemas de seguridad encontrados sobre la versión 6.0 de Android.

Organización del documento

En el capítulo 2 se presentan las características fundamentales de Android, su arquitectura y conceptos básicos. Además se introduce su modelo de seguridad, tanto a nivel de sistema operativo como de aplicación.

En el capítulo 3 se incluye la especificación formal realizada y en el capítulo 4 las propiedades de seguridad demostradas sobre el mismo.

En el capítulo 5 se mencionan y explican debilidades de seguridad encontradas en la versión 6.0 de Android.

En el capítulo 6 se analizan trabajos relacionados y finalmente en el capítulo 7 se exhiben las conclusiones y líneas de trabajo futuro.

Capítulo 2

Modelo de Seguridad de Android

En este capítulo se describen las características principales de Android y algunos aspectos básicos sobre su modelo de seguridad. La especificación desarrollada, que será explicada en el capítulo 3, se basa en los conceptos explicados en este capítulo.

El trabajo que aquí se presenta está basado en la versión 6.0 de Android por lo que se hará referencia a la misma.

2.1. Descripción General de Android

En esta sección se describen los aspectos generales del sistema operativo Android. En la sección 2.1.1 se detalla su arquitectura, en la sección 2.1.2 los componentes de las aplicaciones y en la sección 2.1.3 cómo dichos componentes intercambian datos e información.

2.1.1. Arquitectura

La arquitectura de Android está dividida en cuatro capas jerárquicas como se muestra en la Figura 2.1. Cada capa provee servicios a las capas de los niveles superiores y necesita de servicios provistos por las capas de niveles inferiores. Las capas más bajas realizan las tareas de interacción con el hardware, mientras que las capas superiores realizan tareas de alto nivel.

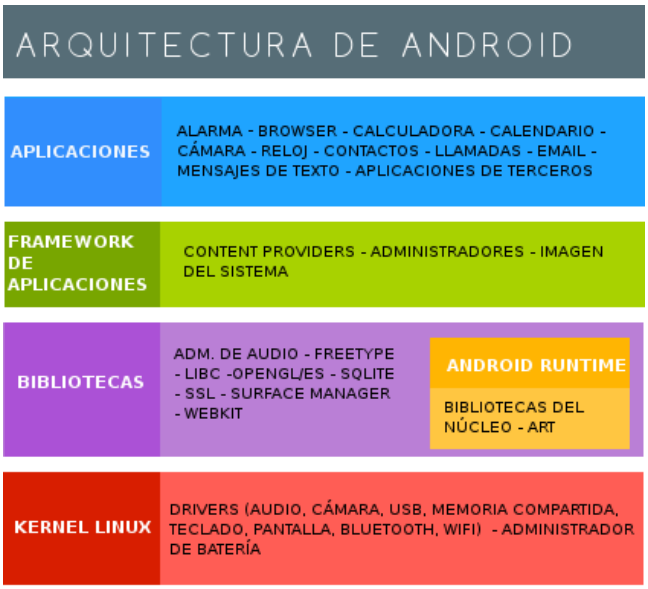

Figura 2.1: Arquitectura de Android

El **Kernel Linux** es una capa de abstracción entre el hardware y el resto de los componentes del sistema [50].

La capa de **Bibliotecas** brinda un conjunto de librerías C/C++ que pueden ser accedidas por los desarrolladores a través del **Framework de Aplicaciones**. Se utilizan para acceder al hardware, a la base de datos y también para llevar a cabo tareas críticas o sensibles. Dentro de la capa **Bibliotecas** hay una sección denominada **Android Runtime**. Ésta agrupa a dos tipos de componentes: las **Bibliotecas del Núcleo** que son la mayor parte de las librerías nativas de Java y una máquina virtual **ART**. Cada aplicación es compilada y ejecutada sobre una instancia propia de esta máquina virtual, que es completamente distinta a las del resto de las aplicaciones [50].

La capa **Framework de Aplicaciones** se encarga de administrar el ciclo de vida de las aplicaciones, de proveer el conjunto de APIs para el desarrollo y de la comunicación entre componentes. Cualquier aplicación puede ofrecer sus servicios a las demás y utilizar servicios de terceros siempre y cuando cuente con los permisos que correspondan [50].

Finalmente, la capa de **Aplicaciones** incluye un conjunto de aplicaciones clave como la cámara de fotos, el correo electrónico, la mensajería y los contactos

entre otras. Todas las aplicaciones desarrolladas se agregan a esta capa [50].

2.1.2. Componentes de una Aplicación

Una aplicación en Android se constituye de bloques denominados **componentes**. Cada componente tiene un rol específico y ayuda a definir el comportamiento general de la aplicación [4].

Existen cuatro tipos de componentes distintos: **actividades**, **servicios**, ***content providers*** y ***broadcast receivers***. Cada uno de ellos tiene un propósito y un ciclo de vida que define cómo el componente es creado y destruido [4].

Actividades

Las actividades proveen las pantallas de una aplicación. Son el punto de interacción del usuario con las aplicaciones. Una aplicación usualmente cuenta con múltiples actividades interconectadas. Por lo general, una de las actividades de cada aplicación es la que el usuario ve al iniciar una aplicación y es denominada "principal"[2]. De todas formas cualquier actividad puede ser iniciada por aplicaciones externas, por lo que no necesariamente se debe iniciar la actividad principal cuando se invoque a la aplicación. Esto permite que puedan existir múltiples instancias en ejecución de una misma actividad [53].

Por ejemplo, un usuario ingresa al menú y selecciona una aplicación que manda mensajes de texto, la actividad principal que va a ver al realizar esta acción es la bandeja de entrada (ver Figura 2.2 [1]). Luego selecciona la opción que le permite mandar un nuevo mensaje, visualizando la actividad correspondiente (ver Figura 2.3). Si ahora el usuario está navegando en la web y quiere compartir algún contenido mediante un mensaje de texto, entonces la actividad que verá en primer lugar es la que le permite mandar un nuevo mensaje (ver Figura 2.3).

[1] Imagen tomada de `https://play.google.com/store/apps/details?id=com.thinkyeah.message&hl=en`

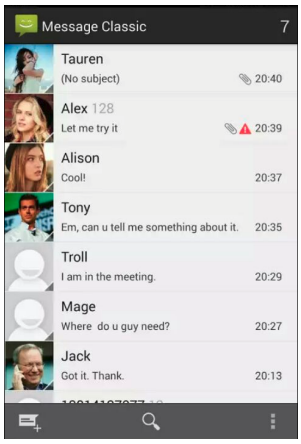

Figura 2.2: Actividad Principal

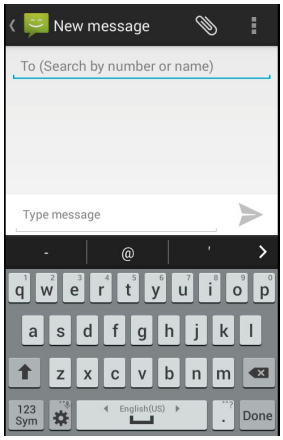

Figura 2.3: Actividad

Servicios

Los servicios son componentes que corren en segundo plano y no tienen interfaz de usuario. Los componentes de otras aplicaciones pueden iniciar un servicio y el mismo continuará ejecutándose en segundo plano aunque el usuario cambie de aplicación [31]. Por lo general los servicios se utilizan para tareas que requieren mucho tiempo de ejecución, pero también se utilizan para trabajar para procesos remotos.

Por ejemplo, si un usuario enciende la radio en su dispositivo y luego abre la aplicación de mensajería para mandar un mensaje, la radio se sigue ejecutando en segundo plano como un servicio.

Content Providers

Los *content providers* administran los datos de la aplicación. Se pueden almacenar datos en el sistema de archivos, en una base de datos SQLite, en la web o en cualquier otro tipo de almacenamiento persistente que pueda ser accedido por la aplicación [4]. Los *content providers* proveen mecanismos para definir la seguridad de los datos [11]. Son la interfaz entre los datos que se encuentran en un proceso con el código que se está ejecutando en otro proceso. Siempre que el *content provider* lo permita, el resto de las aplicaciones podrán acceder a sus recursos, por lo general, mediante el uso de URIs que los identifican [53].

Broadcast Receivers

Los *broadcast receivers* responden a anuncios de tipo *broadcast*. Muchos de estos mensajes son originados por el sistema, por ejemplo el aviso de batería baja o el bloqueo de la pantalla. Las aplicaciones también pueden mandar mensajes de tipo *broadcast*, por ejemplo para notificar que finalizó una descarga y que los datos pueden ser utilizados [4]. Los mensajes de tipo *broadcast* son trasmitidos a todo el sistema, y los *broadcast receivers* son los encargados de decidir cuándo el mensaje deberá llegar a la aplicación a la que pertenecen [53]. Por lo general los *broadcast receivers* actúan de puente entre otros componentes y no realizan grandes cantidades de trabajo [4].

2.1.3. Interacción entre Componentes

El sistema Android corre cada aplicación en un proceso independiente, con permisos que restringen el acceso a otras aplicaciones, por lo tanto, un componente no puede iniciar directamente un componente de otra aplicación [4]. Para realizar este tipo de acción se debe enviar un mensaje al sistema que explicite la intención de iniciar un componente particular. A este tipo de mensajes se le denomina ***intent***.

Un *intent* es un mensaje asincrónico que permite iniciar actividades, servicios y *broadcast receivers*. Es el elemento básico de inter o intra comunicación entre los componentes de las distintas aplicaciones.

Para actividades y servicios, un *intent* define una acción a llevar a cabo y especifica toda la información que el componente iniciado necesite para llevar a cabo la acción. Por ejemplo, un *intent* podría traducirse en un pedido a una actividad para compartir un archivo en una red social.

Las actividades también pueden iniciarse para recibir un resultado retornado en un nuevo *intent*. Por ejemplo, una aplicación podría utilizar un *intent* para que el usuario seleccione una imagen y se la retorne.

Para *broadcast receivers*, un *intent* define un anuncio que deberá ser enviado a todas las aplicaciones. Un ejemplo de un mensaje de tipo *broadcast* es el anuncio de que el dispositivo terminó de encenderse.

Los *intents* se crean mediante objetos `Intent` [18], estableciendo campos que indican qué se espera del *intent* en cuestión. Los campos que pueden incluirse en la creación de un *intent* son [17]:

- **Nombre de componente:** indica el nombre del componente a iniciar. Es un atributo opcional que define de qué tipo será el *intent* (ver Sección 2.1.3.1).

- **Acción:** especifica la acción a llevar a cabo. Determina los campos **data** y **extra**. Se pueden definir acciones internas a una aplicación pero solamente podrán utilizarse para la comunicación de los componentes de dicha aplicación. Por lo general se utilizan las acciones definidas en la clase `Intent`. Por ejemplo, la acción `ACTION_VIEW` se utiliza cuando el *intent* tiene alguna información que se le va a mostrar al usuario, como una imagen.

- **Data:** contiene la URI que referencia los datos sobre los que se debe actuar y/o el tipo de esos datos. Generalmente el tipo de los datos queda definido por la acción. Por ejemplo, si la acción fuera `ACTION_VIEW`, los datos deberían contener el archivo a mostrar.

- **Categoría:** contiene información adicional acerca del componente que debería manejar el *intent*. Un *intent* puede tener múltiples categorías definidas. Por ejemplo, si el *intent* incluye la categoría `CATEGORY_BROWSABLE`, el componente que va a recibirlo debe poder ser iniciado por un browser para mostrar datos referenciados por un link.

- **Extras:** son pares clave-valor que contienen información adicional utilizada para realizar la acción pedida. Algunas acciones utilizan datos contenidos en URIs mediante el atributo **data** y otras usan información contenida en el atributo **extras**. Por ejemplo, si la acción del *intent* es `ACTION_CALL` y se agrega un atributo `EXTRA_PHONE_NUMBER` el mismo indica el número al que se debe realizar la llamada.

- **Banderas:** funcionan como metadatos para el *intent*. Indican al sistema cómo poner en ejecución a un componente o cómo tratarlo una vez que se encuentra corriendo.

2.1.3.1. Tipos de *intents*

Como se mencionó anteriormente, el tipo de un *intent* depende de la presencia del nombre del componente en un *intent*. Existen dos tipos: **implícitos** y **explícitos**.

Un *intent* **implícito** [17] declara la acción que se desea llevar a cabo y no declara ningún componente específico a iniciar. En este caso, el Sistema Android deberá decidir qué aplicaciones pueden realizar dicha acción y por tanto las que serán candidatas a recibir el *intent*. Si la acción debe ser realizada por un componente de tipo actividad, el usuario será quien seleccione la aplicación que realizará la acción. Si la acción debe ser realizada por un componente de tipo servicio, el sistema dirige el *intent* a la aplicación que considere más adecuada. Generalmente se utiliza este tipo de *intents* cuando se sabe qué se quiere hacer, pero no qué componente puede hacerlo. Por ejemplo, cuando se quiere visualizar una imagen adjuntada en un correo electrónico.

Un *intent* **explícito** [17] especifica el nombre del componente a iniciar. Generalmente se utiliza para iniciar un componente dentro de la misma aplicación. Por ejemplo, luego de iniciar sesión en una casilla de correo electrónico, se inicializa la actividad que muestra la bandeja de entrada, y ambas actividades pertenecen a la misma aplicación.

2.1.3.2. Envío de *intents*

Según el tipo de componente al que esté dirigido el *intent* (actividad, servicio, *broadcast receiver*) hay un conjunto de operaciones válidas que permiten el envío del *intent*. Todas las operaciones deberán recibir un *intent* que contenga toda la información necesaria para realizar la acción que corresponda.

Si se quiere iniciar una actividad se debe invocar o bien a la operación `startActivity` o bien a la operación `startActivityForResult`. La diferencia entre las dos operaciones es que la primera únicamente solicita que se ejecute una acción, mientras que la segunda espera obtener cierta información adicional de la actividad que se inicia.

Para iniciar un servicio se debe invocar a la operación `startService` o `boundService`. La diferencia más importante es que un servicio iniciado mediante la operación `startService` puede correr indefinidamente, mientras que un servicio iniciado mediante la operación `boundService` corre mientras exista alguna aplicación que esté ligada a él [31]. Además si una aplicación inicia un servicio mediante la operación `boundService` puede interactuar con el mismo, intercambiando información, lo que no está permitido si se inicia un servicio mediante la operación `startService`. A partir de Android 5.0 (*Android Lollipop*) no se pueden mandar *intents* implícitos mediante la operación `boundService`,

ya que esto podría acarrear problemas de seguridad debido a que se permite el intercambio de información con un servicio que es desconocido para el usuario y que puede ser malicioso.

Para enviar un mensaje de tipo *broadcast* se debe invocar a la operación `sendBroadcast`, `sendOrderedBroadcast` o `sendStickyBroadcast`. La principal diferencia entre ellas es que en la primera se manda el mensaje a todas las aplicaciones y cada una deberá luego aceptarlo o no según corresponda, en la segunda el mensaje se manda según un orden de prioridad que indican los componentes en su *manifest* y en la tercera el mensaje enviado continúa presente aún después que todas las aplicaciones recibieron el mensaje. Las operaciones `sendBroadcast` y `sendOrderedBroadcast` pueden ser protegidas por un permiso adicional cuando se está trabajando con información sensible que hace que solamente aquellas aplicaciones que cuenten con dicho permiso puedan recibir el mensaje. La operación `sendStickyBroadcast` no permite ningún tipo de protección y es obsoleta a partir del nivel 21 del SDK [32].

2.2. El Modelo de Seguridad de Android

En esta sección se describe el modelo de Seguridad de Android. En las secciones 2.2.1 y 2.2.2 se detallan aspectos de seguridad a nivel de sistema operativo y en las secciones 2.2.3, 2.2.4 y 2.2.5 se describen aspectos de seguridad de Android implementados a nivel de aplicación.

2.2.1. *Sandbox* de Aplicaciones

Android implementa el principio del mínimo privilegio forzando a cada aplicación a correr en un ***sandbox***. Esto quiere decir que por defecto las aplicaciones únicamente tienen acceso a los componentes que necesitan para funcionar, lo que crea un entorno seguro en que ninguna aplicación podrá acceder a partes del sistema sobre las que no tenga permisos.

El mecanismo de *application sandbox* se implementa de la siguiente forma [53]:

- El sistema operativo Android es un sistema Linux multi-usuario en el que cada aplicación se corresponde con un usuario diferente.

- Por defecto, el sistema asigna a cada aplicación un único ID de usuario de Linux. Además se establecen permisos para todos los archivos de una aplicación de forma que solo el ID de usuario asignado a esa aplicación pueda accederlo.
- Cada proceso corre en su propia instancia de máquina virtual ART, por lo que cada aplicación se ejecuta aislada del resto.
- Por defecto, cada aplicación corre en su propio proceso Linux. Android inicia el proceso siempre que alguno de sus componentes tenga que ser iniciado y lo mata cuando ya no es necesario o cuando se necesita liberar memoria para otras aplicaciones.

Hay dos mecanismos para que una aplicación pueda compartir datos con otras y acceder a servicios del sistema. En primer lugar, se pueden implementar aplicaciones que compartan el mismo ID de usuario Linux (ver Sección 2.2.2), y por tanto sus archivos. Las aplicaciones con el mismo ID de usuario también compartirán el proceso Linux y la máquina virtual ART. En segundo lugar, una aplicación puede solicitar permisos para acceder a información sensible y a recursos del dispositivo (ver Sección 2.2.3).

2.2.2. Firma de Aplicaciones

Las aplicaciones deben estar firmadas con un certificado cuya clave privada sea conocida exclusivamente por su desarrollador [29]. El certificado no necesita ser firmado por ninguna entidad particular, sino que típicamente se utilizan certificados firmados por el propio desarrollador (*self-signed certificates*). El objetivo de los certificados es identificar al autor de la aplicación. Esto permite que el sistema otorgue permisos de nivel *signature* y decida si dos aplicaciones pueden tener el mismo ID de usuario.

2.2.3. Permisos

Un permiso es un string que expresa la capacidad de una aplicación de hacer una operación específica [35]. Cada aplicación declara un conjunto de permisos que debe tener para funcionar correctamente y permisos que terceros deben tener para iniciar a sus distintos componentes. En base a los permisos que cada aplicación tiene otorgados, se les permite o denega el acceso a recursos del sistema

o de otras aplicaciones. Cada aplicación declara en su *manifest* (ver Sección 2.2.5) dos tipos de permisos [35]: **permisos declarados** que son definidos por la aplicación y se utilizan para proteger de accesos de terceros a la misma y **permisos requeridos** que son los que la aplicación necesita para poder realizar sus funcionalidades.

Los permisos se clasifican en niveles de protección según el riesgo potencial que se les asocia [53]. La forma en que el sistema decide si un permiso es otorgado o no depende del nivel de protección del mismo. Existen cuatro niveles de protección [20]:

- ***Normal*:** los permisos que están en este nivel de protección cubren los casos en que una aplicación requiere acceder a recursos fuera del *sandbox*, pero que no representan ningún tipo de riesgo para la privacidad del usuario. Los permisos de este tipo serán otorgados automáticamente durante la instalación.

- ***Dangerous*:** los permisos que están en este nivel de protección cubren los casos en que una aplicación requiere acceso a información privada del usuario, o que podría afectar potencialmente a los datos almacenados o al funcionamiento de otras aplicaciones. Por ejemplo, el acceso a la galería de imágenes es un permiso de esta categoría. Los permisos de este tipo deberán ser otorgados por el usuario.

- ***Signature*:** este tipo de permisos son otorgados por el sistema solamente cuando la aplicación que lo solicita está firmada con el mismo certificado que la aplicación declarada en el permiso. Si los certificados coinciden, el sistema otorga el permiso automáticamente sin notificarle al usuario.

- ***Signature or system*:** este tipo de permisos regulan el acceso a recursos o servicios críticos del sistema. Son otorgados solamente a aplicaciones nativas o que están firmadas con el mismo certificado que la aplicación declarada en el permiso.

Hasta la versión 5.0 de Android el sistema de permisos funciona de la siguiente manera [53]: las aplicaciones declaran en su *manifest* los permisos que necesitan para funcionar; cuando se instala una aplicación se otorgan los permisos según su nivel de protección, su certificado y la aprobación por parte del usuario; si en este punto existe algún permiso que no fue otorgado, la aplicación no podrá instalarse.

A partir de la versión 6.0 de Android y dependiendo de la versión del SDK, el usuario otorga permisos de tipo *dangerous* en **tiempo de ejecución** en vez de al momento de instalación [24].

Todos los permisos deberán ser listados en el *manifest*, de igual forma que se hacía en versiones anteriores. Al momento de la instalación, en caso de que el dispositivo utilice la versión 6.0 de Android y una versión del SDK mayor o igual a la 23, se otorgan los permisos que no requieren aprobación por parte del usuario [2]. Los permisos que requieren aprobación por parte del usuario se otorgan a medida que se necesitan para llevar a cabo las funcionalidades de la aplicación. El usuario puede además, revocar cualquier permiso que haya otorgado previamente, por lo que las aplicaciones deben corroborar que tienen los permisos necesarios siempre que se quiera ejecutar una funcionalidad.

Este cambio, además de darle al usuario más control sobre las funcionalidades de la aplicación, hace que el proceso de instalación sea más fluido.

Para ejemplificar el funcionamiento del procedimiento de otorgamiento de permisos, se supone un escenario en donde se está ejecutando una aplicación que tiene, entre otras, la funcionalidad de compartir imágenes en una red social. Todos los permisos de nivel de protección *normal*, *signature* y *signature or system* deben haber sido otorgados previamente, al momento de la instalación. Si el usuario quiere compartir una imagen en una red social, la aplicación debe verificar si el permiso correspondiente fue otorgado previamente. En caso afirmativo, el usuario podrá compartir la imagen deseada sin ninguna interacción extra. En cambio, si el permiso aún no fue otorgado o fue otorgado y luego revocado, la aplicación debe preguntar al usuario si desea otorgar el permiso en cuestión. Si la respuesta es positiva, entonces el usuario puede compartir la imagen sin ninguna otra interacción. Si la respuesta es negativa, la acción no se lleva a cabo y la aplicación debe seguir funcionando con normalidad.

De la misma forma que las aplicaciones declaran los permisos que necesitan para funcionar, también pueden declarar un permiso que las demás aplicaciones deben tener para iniciarla. Esto se puede hacer también a nivel de componente.

[2] Esto se asume ya que la documentación no indica lo contrario.

2.2.4. Delegación de Permisos

Existen dos mecanismos mediante los cuales una aplicación puede delegar sus permisos a otras: **URI *permissions*** y ***pending intents***. Esto permite que las aplicaciones potencialmente puedan realizar acciones para las que no estaban autorizadas originalmente [53].

Un *content provider* puede querer protegerse con permisos de lectura y escritura, mientras que los componentes que cuentan con permisos de acceso pueden necesitar otorgar acceso a URIs específicas para que otros componentes puedan operar con ellas. Un ejemplo es una imagen adjunta en un mail. El acceso al correo electrónico debe estar protegido ya que maneja información sensible, pero si una imagen adjunta se pasa a un visor de imágenes que no tiene permiso de acceso al mail, la misma no podrá ser visualizada. Para solucionar este problema se agregan los **URI *permissions***. Hay dos formas de delegar este tipo de permisos: mediante el método `grantUriPermission` y mediante el uso de *intents*. En ambos casos el *content provider* debe haber autorizado la delegación sobre sus recursos, de lo contrario no se pueden delegar permisos ni de lectura ni de escritura sobre los mismos [53]. Además, para que un componente pueda delegar y revocar permisos a otro debe tener la capacidad para realizar la acción, ya sea mediante un permiso otorgado o mediante un permiso que le haya sido delegado. Si se delega un permiso mediante el método `grantUriPermission`, este permiso se delega de forma permanente hasta que sea revocado a través del método `revokeUriPermission`. Si cuando se inicia un actividad o se devuelve un resultado a una actividad [3] (mediante el envío de un intent) se configura una bandera del *intent* para otorgar permisos de lectura o escritura sobre un recurso, este permiso se delega de forma temporal mientras la aplicación receptora esté en ejecución.

Si se quiere enviar un ***pending intent*** se debe construir un *intent* para realizar determinada acción y a partir de éste crear un objeto `PendingIntent` [21] asociado a la acción. La aplicación receptora del *pending intent* puede realizar cuando quiera la acción disparada por el *intent* tal como si fuera la aplicación original, es decir, contando con su identidad y sus permisos. Un *pending intent* únicamente podrá ser cancelado por la aplicación que lo creó y en dicho caso todas las aplicaciones que lo estén utilizando no podrán seguir haciéndolo [53].

[3] En la documentación solo se incluyen a las actividades como receptoras de URI *permissions* mediante *intents*.

2.2.5. Android *Manifest*

Todas las aplicaciones deben tener en su directorio raíz un archivo XML denominado `AndroidManifest`. El *manifest* contiene información sobre cada aplicación. Entre otras cosas describe a los componentes de una aplicación, declara permisos que la aplicación debe tener para funcionar correctamente, declara permisos que terceros deben tener para poder iniciar a los componentes de la aplicación y define permisos de usuario.

En la Figura 2.4 [4] se observa la estructura general de un *manifest* y todos los elementos que puede contener.

Los únicos elementos que deben aparecer obligatoriamente son `<manifest>` y `<application>` y pueden aparecer una sola vez [19].

Para este trabajo los elementos más relevantes son `<uses-permission>`, `<permission>` y `<application>`.

El elemento `<uses-permission>` [33] solicita un permiso que la aplicación debe tener para funcionar correctamente. Dependiendo de la versión de Android que corre en el dispositivo y del valor del elemento `<uses-sdk>`, que indica la versión del SDK utilizada, el permiso será otorgado al momento de instalar la aplicación o en tiempo de ejecución (ver Sección 2.2.3). Se incluye un elemento de este tipo por cada permiso que se quiera solicitar.

El elemento `<permission>` [25] define un nuevo permiso de seguridad. Dicho permiso podrá ser utilizado tanto por la aplicación que lo define como por otras. Se incluye un elemento de este tipo por cada permiso que se quiera definir.

El elemento `<application>` [9] declara la aplicación. Este elemento contiene sub-elementos que declaran cada uno de los componentes de la aplicación y atributos que pueden afectar a los componentes. El atributo más relevante para la seguridad es `android:permission` [8] que declara a lo sumo un permiso que terceros deben tener para iniciar cualquier componente de la aplicación. Dentro de este elemento se encuentran los sub-elementos `<activity>` [3], `<service>` [30], `<provider>` [26] y `<receiver>` [28] que como se mencionó anteriormente representan a cada uno de los componentes de la aplicación.

Cada uno de estos cuatro elementos puede opcionalmente contener el atributo `android:permission` que establece a lo sumo un permiso que terceros deben tener para iniciar el componente. En caso de que este atributo esté definido,

[4] Imagen tomada de `https://developer.android.com/guide/topics/manifest/manifest-intro.html`

```xml
<?xml version="1.0" encoding="utf-8"?>

<manifest>

    <uses-permission />
    <permission />
    <permission-tree />
    <permission-group />
    <instrumentation />
    <uses-sdk />
    <uses-configuration />
    <uses-feature />
    <supports-screens />
    <compatible-screens />
    <supports-gl-texture />

    <application>

        <activity>
            <intent-filter>
                <action />
                <category />
                <data />
            </intent-filter>
            <meta-data />
        </activity>

        <activity-alias>
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </activity-alias>

        <service>
            <intent-filter> . . . </intent-filter>
            <meta-data/>
        </service>

        <receiver>
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </receiver>

        <provider>
            <grant-uri-permission />
            <meta-data />
            <path-permission />
        </provider>

        <uses-library />

    </application>

</manifest>
```

Figura 2.4: Estructura del *manifest* de Android

prevalece al atributo con el mismo nombre que se define a nivel de aplicación. Es decir, en caso de que ambos atributos estén definidos, se les exigirá a quienes quieran iniciar el componente en cuestión el permiso definido en el propio componente. También pueden contener el atributo `android:exported` [7] que indica cuándo el componente puede ser invocado por terceros.

Los elementos `<activity>`, `<service>` y `<receiver>` pueden contener elementos de tipo `<intent-filter>` [16] que especifican los tipos de *intents* que los distintos componentes pueden recibir. Declara las capacidades que tiene el componente. Dentro de este elemento se pueden encontrar elementos denominados `<action>`, `<category>` y `<data>`. Estos deben cumplir que siempre que

aparezcan elementos `<category>` o `<data>` deberán aparecer elementos de tipo `<action>` también. El elemento `<action>` [1] describe una acción que el componente puede realizar, `<category>` [10] indica que un *intent* con la categoría establecida es capaz de ser resuelto por el componente y `<data>` [12] agrega especificación de datos y puede incluir solamente el tipo de los datos, solamente la URI o ambos.

Los *intent filters* definen el valor por defecto del atributo `android:exported`. Si para un componente el atributo no está presente en el *manifest* y hay por lo menos un *intent filter* definido, entonces el valor por defecto del atributo será verdadero. En cambio si no hay ningún *intent filter* definido el valor por defecto del atributo será falso. La ausencia del atributo `android:exported` lleva a muchas malas configuraciones de las aplicaciones que derivan en posibles problemas de seguridad.

La función fundamental de los *intent filters* es ayudar al sistema a decidir qué aplicaciones pueden recibir un *intent* implícito determinado. Cuando se manda un *intent* implícito se corren tres pruebas que determinan cuál es el conjunto de aplicaciones aptas para recibirlo: ***Action Test***, ***Category Test*** y ***Data Test***.

Un componente pasa el ***Action Test*** [17] si la acción declarada en el *intent* coincide con alguna de las acciones listadas en el filtro o si el *intent* no declara ninguna acción.

Un componente pasa el ***Category Test*** [17] si cada categoría declarada en el *intent* pertenece a la lista de categorías del filtro o si el *intent* no declara ninguna categoría.

Según la información sobre los datos contenida en el *intent*, se define el ***Data Test*** según las siguientes reglas [17]:

- Si el *intent* no contiene URI ni tipo de datos, el componente pasa el test solo si el filtro no especifica ninguna URI ni tipo de datos.
- Si el *intent* contiene URI pero no tipo de datos, el componente pasa el test solo si el filtro especifica una URI equivalente y no especifica ningún tipo de datos.
- Si el *intent* contiene tipo de datos pero no URI, el componente pasa el test solo si el filtro especifica el mismo tipo de datos y no especifica un formato de URI.

- Si el *intent* contiene URI y tipo de datos, el componente pasa el test si el filtro especifica el mismo tipo de datos y la misma URI o si el *intent* tiene URI de tipo *content:* o *file:* y el filtro no especifica ninguna URI.

El elemento `<provider>` puede contener, de forma opcional, tanto al elemento `<grantUriPermission>` como a los atributos `android:readPermission`, `android:writePermission` y `android:grantUriPermission`.

El atributo `android:readPermission` [27] es un permiso que terceros deben tener para poder hacer consultas de lectura sobre los recursos del *content provider*, `android:writePermission` [34] es un permiso que terceros deben tener para poder modificar los recursos del *content provider*, `android:grantUriPermission` [15] indica si las aplicaciones que normalmente no tienen permisos para acceder a un recurso del *content provider* pueden temporalmente ganar acceso.

El elemento `<grantUriPermission>` [14] especifica el conjunto de recursos del *content provider* sobre los que se pueden otorgar permisos.

El valor por defecto del atributo `android:exported` para el elemento de tipo `<provider>` depende de que el valor del atributo `android:minSdkVersion` o del atributo `android:targetSdkVersion`, presentes en el elemento `<uses-sdk>`, sea 16 o menos.

Capítulo 3

Especificación Formal

En este capítulo se describe la especificación realizada, desarrollada en `Coq`, del modelo de seguridad de Android. En la sección 3.1 se explica la metodología utilizada para la realización de este trabajo. En la sección 3.2 se establece la notación que se utilizará en lo que sigue de este informe. En la sección 3.3 se formalizan los elementos básicos de un sistema Android: las aplicaciones, sus componentes, el elemento básico de comunicación entre ellas, etc. En la sección 3.5 se definen las operaciones presentes en el modelo desarrollado. En la sección 3.6 se formaliza la ejecución de una operación, las condiciones necesarias del sistema Android para que se puedan ejecutar y el estado del sistema una vez que fueron ejecutadas. Finalmente, en la sección 3.7 se muestra cómo se probó que la ejecución de todas las acciones definidas en la sección 3.5 da como resultado un sistema válido.

3.1. Metodología

Este modelo extiende el realizado por Agustín Romano [53]. La extensión se basa fundamentalmente en el agregado de *intents* y las operaciones de envío, recepción y resolución de los mismos como forma de comunicación entre los componentes. Además se modificó el modelo existente para otorgar y revocar permisos en tiempo de ejecución adaptándolo a la versión 6.0 de Android.

Este trabajo está basado fundamentalmente en la documentación oficial de Android [13]. En los casos en que la documentación no fue suficiente para

comprender el funcionamiento de los aspectos que se querían modelar, se recurrió a publicaciones académicas que los describieran en mayor detalle.

3.2. Notación Utilizada

Variables y Tipos

Las variables se denotan con tiras alfanuméricas en minúscula y los tipos con tiras que pueden contener mayúsculas. Ambos se escriben en cursiva.

Dados dos tipos A y B el tipo de una función total entre A y B se representa como $A \to B$. Por último $a : A$ indica que la variable a es de tipo A.

Proposiciones

Las proposiciones se construyen con los símbolos $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\exists$ y $\forall$, su significado es el mismo que el de la lógica de predicados. Además $Prop$ es el tipo que le corresponde a todas las proposiciones, por lo tanto y teniendo en cuenta lo definido en el punto anterior, $P : A \to Prop$ representa a un predicado P que toma como parámetro una variable de tipo A.

Tipos Inductivos

Los tipos inductivos se definen listando sus componentes, como se muestra a continuación. Un elemento canónico de un conjunto se obtiene a partir de la combinación de sus constructores.

$$\begin{aligned} T_{ind} \stackrel{def}{=} &\mid f_1 : \ A \to T_{ind} \\ &\mid f_2 : B \to C \to T_{ind} \end{aligned}$$

El tipo ***Set*** es el que le corresponde a todos los dominios.

El tipo inductivo ***list*** se define como:

$$\begin{aligned} list \stackrel{def}{=} &\mid nil : \ list \\ &\mid cons : \ A \to list \to list \end{aligned}$$

Donde A es el tipo de los elementos que componen la lista.

Un elemento de tipo ***option A*** puede representar o bien a algún valor de tipo A o bien a ninguno. Se define como:

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

Registros

Un registro se define especificando sus campos.
$R \stackrel{def}{=} \{\ l_1 :\ A \to B,\ l_2 :\ A\}$

Dado $r : R$, para acceder al campo l_1 del registro, se utilizará la notación $r.l_1$. Además también se hace uso de la notación $r\ =\ (campo_1,\ campo_2)$ para facilitar la referencia a los campos.

Funciones Parciales

Una función parcial de A en B se nota como $A \mapsto B$.

3.3. Definiciones Básicas

3.3.1. Permisos

Un permiso es un objeto que se define mediante el registro *Perm*.

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

$$permLevel \stackrel{def}{=} dangerous \mid normal \mid signature \mid signatureOrSys$$

El campo idP representa al nombre de un permiso y el campo pl a su nivel de protección.

Se cuenta además con el predicado $isSystemPerm$ que será verdadero solamente si el permiso sobre el que fue aplicado es predefinido por el sistema. Se utiliza para distinguir este tipo de permisos de los definidos por el usuario.

3.3.2. Aplicaciones

En Android todas las aplicaciones cuentan con un *manifest* en el que se declaran los componentes que las conforman, así como propiedades y atributos específicos que cumplen cada uno de sus ellos de forma independiente, por ejemplo saber si están o no exportados (ver Capítulo 2).

En el modelo que se presenta, se optó por incluir en cada tipo de componente las propiedades y atributos que le corresponden y eliminarlas del *manifest* de la aplicación. Así, el *manifest* de la aplicación incluye solamente a los elementos que la describen propiamente, como por ejemplo, la lista de los componentes que la conforman. De cualquier forma, es posible recrear una estructura de *manifest* semejante a la de Android a partir de los campos definidos en los componentes y las aplicaciones.

Por tanto, y como la información contenida en los elementos `<activity>`, `<service>`, `<receiver>` y `<provider>` presentes en el *manifest* brindan datos sobre algún componente de la aplicación, no se incluyen en el registro $Manifest$ sino que en la propia definición de cada componente.

$$\begin{aligned} Manifest \stackrel{def}{=} \; & \{cmp : list\ Cmp, \\ & use : list\ Perm, \\ & usrP : list\ Perm, \\ & appE : option\ Perm\} \end{aligned}$$

El campo *cmp* (que se define en la próxima sección) representa a los elementos de tipo `<activity>`, `<service>`, `<receiver>` y `<provider>`. El campo *use* se corresponde con el conjunto de elementos `<uses-permission>` e indica la lista de permisos que la aplicación solicita para poder ejecutar todas sus funcionalidades. El campo $usrP$ se corresponde con el conjunto de elementos `<permission>` y lista los nuevos permisos que se definen en la aplicación. El campo *appE* representa al atributo `android:permission` e indica a lo sumo un permiso que terceros deben tener para iniciar a la aplicación.

Las aplicaciones tienen además un certificado que se representa con el parámetro $Cert$ y un conjunto de recursos que se representan con el parámetro res, ambos de tipo Set.

Toda la información de las aplicaciones será asociada a las mismas una vez que hayan sido instaladas, ya que solo tiene sentido acceder a la información de una aplicación cuando existe en un sistema. Por esta razón es que las aplicaciones antes de ser instaladas solamente cuentan con un identificador que se representa mediante el parámetro *idApp* de tipo Set e identifica al nombre de la aplicación.

3.3.3. Componentes de una Aplicación

3.3.3.1. Definición de Componentes

A efectos de las funcionalidades representadas y de las propiedades estudiadas en este trabajo, es posible clasificar a los componentes en dos grandes grupos: actividades, servicios y *broadcast receivers* por un lado y *content providers* por el otro.

Los componentes del primer grupo pueden recibir *intents* y por lo tanto cuentan con la información necesaria para que el sistema sea capaz de resolver los *intents* implícitos. A su vez, dentro de este grupo cada tipo de componente genera un comportamiento distinto en el sistema al recibir un *intent*.

El componente del segundo grupo maneja los recursos de la aplicación y debe tener información que indique cuándo dichos recursos pueden ser accedidos por terceros.

Por lo mencionado anteriormente, es de interés identificar a cada componente mediante un tipo distinto. Como también existen funcionalidades y propiedades que aplican sobre cualquier tipo de componente, es importante contar con una representación de un componente genérico.

Se define el tipo genérico de los componentes como:

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

Los componentes de tipo *Activity* se representan mediante el siguiente registro:

$$\begin{aligned} Activity \stackrel{def}{=} \{&idA : idCmp, \\ &expA : option\ bool, \\ &cmpEA : option\ Perm, \\ &intFilterA : list\ IntentFilter\} \end{aligned}$$

Los componentes de tipo *Service* se representan mediante el siguiente registro:

$$Service \stackrel{def}{=} \{idS : idCmp,$$
$$expS : option\ bool,$$
$$cmpES : option\ Perm,$$
$$intFilterS : list\ IntentFilter\}$$

los componentes de tipo *BroadReceiver* se representan mediante el siguiente registro:

$$BroadReceiver \stackrel{def}{=} \{idB : idCmp,$$
$$expB : option\ bool,$$
$$cmpEB : option\ Perm,$$
$$intFilterB : list\ IntentFilter,$$
$$prio : option\ nat\}$$

$$IntentFilter \stackrel{def}{=} \{actFilter : list\ intentAction,$$
$$dataFilter : list\ Data,$$
$$catFilter : list\ Category\}$$

Los campos *idA*, *idS* e *idB* representan al identificador de cada componente.

Los campos *expA*, *expS* y *expB* se corresponden con el atributo `android:exported` que indica si un componente está o no exportado. Este atributo puede estar presente en cada actividad, servicio o *broadcast receiver* declarado en el *manifest* de una aplicación. Los campos *cmpEA*, *cmpES* y *cmpEB* representan al atributo `android:permission` que declara a lo sumo un permiso que las aplicaciones deben tener para poder iniciar al componente.

Los campos de tipo *IntentFilter* representan a los *intent filters* de los componentes y permiten decidir si cada componente cuenta con los medios para recibir un *intent* implícito. El campo *actFilter* del registro *IntentFilter* representa a un conjunto de elementos `<action>`, que agregan una acción que el componente puede llevar a cabo. El campo *dataFilter* del registro *IntentFilter* se corresponde con un conjunto de elementos `<data>`, que agregan tipos de datos que los componentes pueden manejar. El campo *catFilter* del registro *IntentFilter* representa a un conjunto de elementos `<category>`, que agregan una categoría al *intent filter*. Los tipos *intentAction*, *Data* y *Category* utilizados para representar *intent filters*, se definirán en la sección 3.3.4.

Por último, el campo *prio* permite establecer un orden a la hora de mandar mensajes de tipo broadcast mediante la operación `sendOrderedBroadcast`. En caso de utilizar esta operación todos los *broadcast receivers*, que tengan definida la prioridad, recibirán el mensaje en orden de prioridad decreciente.

Los componentes de tipo $CProvider$ se representan mediante el siguiente registro:

$$\begin{aligned} CProvider \stackrel{def}{=} \{&idC : idCmp,\\ &map\_res : uri \mapsto res,\\ &expC : option\ bool,\\ &minSdk : optionnat,\\ &targetSdk : optionnat,\\ &cmpEC : option\ Perm\\ &readE : option\ Perm,\\ &writeE : option\ Perm,\\ &grantU : bool,\\ &uriP : list\ uri\} \end{aligned}$$

El campo idC, representa al identificador de un *content provider*. El campo map_res es una función que para cada URI retorna el recurso del *content provider*. El campo $expC$, se corresponde con el atributo `android:exported`, que puede estar presente en cada *content provider* declarado en el *manifest* de una aplicación e indica si el mismo está disponible para ser accedido por terceros.

El campo $minSdk$ es un entero que indica la versión mínima de SDK en la que una aplicación puede correr. El campo $targetSdk$ es un entero que indica la versión del SDK en la que la aplicación fue probada. Estos dos últimos campos se incluyen en la información de los *content providers* ya que sirven para determinar el valor por defecto del atributo `android:exported` en este tipo de componentes.

El campo $cmpEC$, representa al atributo `android:permission` para un *content provider*, que declara a lo sumo un permiso que las aplicaciones deben tener para poder iniciarlo.

Los campos $readE$, $writeE$, $grantU$ y $uriP$ indican cómo manejar los recursos de la aplicación y cuándo estos pueden ser leídos o modificados por otras aplicaciones. Los campos $readE$ y $writeE$ representan a los atributos

`android:readPermission` y `andorid:writePermission` respectivamente, y establecen a lo sumo un permiso que las aplicaciones deben tener para poder leer o escribir los recursos del *content provider*. El campo *grantU* se corresponde con el atributo `android:grantUriPermissions`, que indica si las aplicaciones que normalmente no tienen permisos para acceder a los recursos del *content provider*, pueden recibir permisos temporales para "sobreescribir"las restricciones establecidas en los atributos `android:readPermission`, `android:writePermission` y `android:permission`. El campo $uriP$ representa al elemento `<grant-uri-permission>` y especifica al conjunto de sus recursos a los que se les puede otorgar permisos.

3.3.3.2. Componentes en Ejecución

Las instancias en ejecución se representan con el tipo $iCmp \stackrel{def}{=} \{idICmp : idInst\}$ donde $idICmp$ es el identificador de cada elemento. No es necesario distinguir las instancias en ejecución de cada tipo de componente, ya que para las propiedades y funcionalidades tratadas en este modelo son indiferentes.

3.3.4. *Intents*

Los objetos de tipo `Intent` deben almacenar toda la información necesaria para que una aplicación pueda llevar a cabo una acción determinada. Es necesario que la representación permita identificar *intents* de tipo implícito y de tipo explícito. Se representa entonces un *intent* mediante el registro $Intent$.

$$
\begin{aligned}
Intent \stackrel{def}{=} \{&idI : idInt,\\
&cmpName : option\ idCmp,\\
&intType : intentType,\\
&action : option\ intentAction,\\
&data : Data,\\
&category : list\ Category,\\
&extra : option\ Extra,\\
&flags : option\ Flag,\\
&brperm : option\ Perm\}
\end{aligned}
$$

$$intentType \stackrel{def}{=} intActivity \mid intService \mid intBroadcast$$

$$intentAction \stackrel{def}{=} activityAction \mid serviceAction \mid broadcastAction$$

$$Data \stackrel{def}{=} \{path : option\ uri,\ mime : option\ mimeType,\ type : dataType\}$$

Se definen además los parámetros $Category$, $Extra$ y $Flag$ de tipo Set.

El campo *idI* indica el identificador del *intent*. Si bien esta información no es necesaria en Android a la hora de crear un objeto de tipo `Intent`, se agrega en el modelo desarrollado para poder identificar dos *intents* que comparten todo el resto de los campos. El campo $cmpName$ permite identificar si el *intent* es implícito o explícito. Si es explícito, el atributo $cmpName$ indica el componente al que está dirigido el *intent*. Si es implícito, el atributo $cmpName$ es $None$. El campo $intType$ indica a qué tipo de componente está dirigido el *intent*. El campo *action* se corresponde con el tipo de acción que se quiere llevar a cabo. El campo *data* puede contener una URI y/o un tipo de datos, brinda información acerca de los recursos sobre los que se va a realizar la acción. El campo *category* agrega información sobre la clase de componentes que son capaces de recibir el *intent*.

Los campos *extra* y $flags$, brindan información adicional que puede estar contenida opcionalmente en un *intent*.

El campo *brperm* representa a un permiso utilizado para enviar un mensaje de tipo broadcast. Este campo no existe en un objeto `Intent` [18] de Android, sino que se agrega al modelo desarrollado para poder saber si se utilizó alguna operación protegida por un permiso para mandar el mensaje broadcast.

3.4. Sistema Android

3.4.1. Definición del Sistema Android

Un sistema particular de Android se define mediante el registro $System$.

$$System \stackrel{def}{=} \{environment : Environment,\ state : State\}$$

El registro $Environment$ representa la parte estática del sistema, es decir todo aquello que sólo pueda modificarse mediante la instalación o desinstalación de una aplicación. Almacena la información propia de las aplicaciones: *manifest*,

certificado y permisos definidos por el usuario. Esta información no puede ser modificada durante el tiempo en que una aplicación se encuentre instalada en el sistema. Además almacena a las aplicaciones nativas del sistema Android que se utilizarán para asignar permisos de tipo *signature or system*.

$$\begin{aligned} Environment \stackrel{def}{=} \{&manifest : idApp \mapsto Manifest, \\ &cert : idApp \mapsto Cert, \\ &systemImage : list\ SysImgApp, \\ &defPerms : idApp \mapsto (list\ Perm)\} \end{aligned}$$

$$\begin{aligned} SysImgApp \stackrel{def}{=} \{&idSI : idApp, \\ &certSI : Cert, \\ &manifestSI : Manifest, \\ &appResSI : list\ res\} \end{aligned}$$

Los campos $manifest$, $cert$,y $defPerms$ son funciones que dado un identificador de aplicación retornan el *manifest*, el certificado y los permisos definidos por el usuario de la misma respectivamente. El campo $systemImage$ representa a las aplicaciones nativas del sistema.

El registro *State* representa a la parte dinámica del sistema, que puede ser modificada por todas las operaciones existentes en el sistema Android modelado. Por ejemplo, los componentes en ejecución serán modificados cada vez que se reciba un *intent*.

$$\begin{aligned} State \stackrel{def}{=} \{&apps : list\ idApp, \\ &perms : idApp \mapsto list\ Perm, \\ &running : iCmp \mapsto Cmp, \\ &delPPerms : (idApp \to CProvider \to uri) \mapsto PType, \\ &delTPerms : (iCmp \to CProvider \to uri) \mapsto PType, \\ &resCont : (idApp \to res) \mapsto Val, \\ &sentIntents : list\ (iCmp \times Intent)\} \end{aligned}$$

$$PType \stackrel{def}{=} Read \mid Write \mid Both$$

El campo *apps* representa al conjunto de aplicaciones instaladas en el sistema. El campo *perms* es una función que a cada aplicación instalada le asocia el conjunto de permisos que le fueron otorgados. El campo *running* es una función que a cada instancia que se está ejecutando le asocia el componente correspondiente. El campo $delPPerms$ representa a los permisos que fueron delegados de forma permanente a una aplicación, $delPPerms\ a\ cp\ u$ es igual a *pt* siempre que la aplicación a tenga permisos delegados de tipo *pt* sobre el recurso u del *content provider cp*. El campo $delTPerms$ representa a los permisos que le fueron delegados de forma temporal a una instancia en ejecución, $delTPerms\ ic\ cp\ u$ es igual a *pt* siempre que la instancia en ejecución ic tenga permisos delegados de tipo *pt* sobre el recurso u del *content provider cp*. El campo $resCont$ hace referencia al valor que tienen los recursos de una aplicación $resCont\ a\ r$ es igual a v sólo cuando el recurso r de la aplicación a tenga asociado el valor v. Finalmente, el campo $sentIntents$ representa al conjunto de *intents* enviados junto con la instancia en ejecución que lo envió.

3.4.2. Validez del Sistema

Un elemento de tipo $System$ deberá cumplir ciertas propiedades para efectivamente representar un sistema Android válido.

Para definir las propiedades que debe cumplir un sistema Android para ser válido se usan los predicados y funciones auxiliares presentados en la Tabla 3.1.

$inApp(c, a, sys)$	Se cumple solamente cuando el componente c pertenece a la aplicación instalada a en un sistema sys.
$equalApps(a1, a2, sys)$	Se cumple cuando las aplicaciones $a1$ y $a2$ tienen el mismo identificador, el mismo *manifest* y el mismo certificado.
$running(ic, c, sys)$	Se cumple cuando el componente c se corresponde con la instancia en ejecución ic en un sistema sys.
$hasDefPerms(a, l, sys)$	Se cumple cuando la lista de permisos definidos por el usuario de la aplicación a es l en un sistema sys.
$hasManifest(a, m, sys)$	Se cumple cuando el *manifest* de la aplicación a es m en un sistema sys.
$hasCert(a, cert, sys)$	Se cumple cuando el certificado de la aplicación a es $cert$ en un sistema sys.
	Continúa en la siguiente página

Tabla 3.1 – continuación de la página anterior

$existsRes(cp, u, sys)$	Es válido cuando en un sistema sys existe un recurso apuntado por la URI u que pertenece al componente cp.
$isActivity(c)$	Es verdadero cuando el componente c es una actividad.
$isService(c)$	Se cumple cuando el componente c es un servicio.
$isBReceiver(c)$	Es verdadero cuando el componente c es un *broadcast receiver*.
$isContentProvider(c)$	Es verdadero cuando el componente c es un *content provider*.
$isIntBReceiver(i)$	Se cumple cuando el *intent* i está destinado a un *broadcast receiver*.
$getCmpId(c)$	Retorna el identificador del componente c.

Tabla 3.1: Predicados y funciones auxiliares - validez

A partir de las definiciones presentadas en las secciones anteriores se establecen las propiedades que tiene que cumplir un elemento sys de tipo $System$ para representar un sistema Android válido. Se considera $sys = (e, s)$ donde $e = (mfsts, certs, img, defPerms)$ y $s = (apps, perms, iCs, delPP, delTP, vals, ints)$.

Propiedad 1 (allCmpDifferent) *No hay dos componentes pertenecientes a aplicaciones instaladas que tengan el mismo identificador.*

$$\forall(c_1\ c_2 : Cmp)(a_1\ a_2 : idApp), inApp(c_1, a_1, sys)\ \wedge\ inApp(c_2, a_2, sys)\ \Rightarrow$$
$$getCmpId(c_1) = getCmpId(c_2)\ \Rightarrow c_1 = c_2$$

Propiedad 2 (notRepeatedCmps) *Un mismo componente no está asociado a dos aplicaciones distintas.*

$$\forall(c : Cmp)(a_1\ a_2 : idApp), inApp(c, a_1, sys)\ \wedge\ inApp(c, a_2, sys)\ \Rightarrow$$
$$equalApps(a_1, a_2, sys)$$

Propiedad 3 (notCPrunning) *Si un componente está corriendo, éste no puede ser un content provider.*

$$\forall(ic : iCmp)(c : Cmp), running(ic, c, sys) \ \Rightarrow \ \neg isContentProvider(c)$$

Propiedad 4 (usrPermsDefined) *Todos los permisos definidos por los desarrolladores están declarados en alguna aplicación instalada.*

$$\forall(a : idApp)(p : Perm)(l : list\ Perm), hasDefPerms(a, l, sys) \ \wedge \ p \in l \ \Rightarrow$$
$$a \in apps \ \wedge$$
$$\exists(m : Manifest), hasManifest(a, m, sys) \ \wedge \ p \in m.usrP$$

Propiedad 5 (existsAppnCPinDel) *Si una aplicación tiene permisos delegados sobre un content provider, entonces tanto la aplicación como el content provider están instalados.*

$$\forall(a : idApp)(cp : CProvider)(u : uri)(pt : PType),$$
$$delPP\ a\ cp\ u = pt \ \Rightarrow$$
$$a \in apps \ \wedge$$
$$\exists(a1 : idApp), inApp((cmpCP\ cp), a1, sys) \ \wedge \ existsRes(cp, u, sys)$$

Propiedad 6 (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.*

$$\forall(ic : iCmp)(cp : CProvider)(u : uri)(pt : PType),$$
$$delTP\ ic\ cp\ u = pt\ \Rightarrow$$
$$\exists(a : idApp), inApp((cmpCP\ cp), a, sys)\ \wedge$$
$$\exists(c : Cmp)(a_1 : idApp), inApp(c, a_1, sys)\ \wedge\ running(ic, c, sys)$$

Propiedad 7 (cmpRunAppIns) *Si una instancia está corriendo, su componente está instalado.*

$$\forall(ic : iCmp)(c : Cmp), running(ic, c, sys)\ \Rightarrow$$
$$\exists(a : idApp), inApp(c, a, sys)$$

Propiedad 8(resContAppInst) *Todo recurso en el sistema pertenece a una aplicación instalada.*

$$\forall(a : idApp)(r : res)(v : Val), vals\ a\ r = v \Rightarrow$$
$$a \in apps$$

Propiedad 9 (actionDefined) *Si un intent filter tiene definido algún elemento de tipo categoría o data, debe tener una acción definida.*

$$\forall(a : idApp), a \in apps \ \Rightarrow$$

$$\exists(m : Manifest), hasManifest(a, m, sys) \ \wedge$$

$$(\forall(c : Cmp), c \in m.cmp \ \Rightarrow$$

$$(isActivity(c) \ \Rightarrow \ \forall(iFil : IntentFilter), iFil \in c.intFilterA \ \Rightarrow$$

$$(iFil.dataFilter \neq nil \ \vee \ iFil.catFilter \neq nil \ \Rightarrow \ iFil.actFilter \neq nil)) \ \wedge$$

$$(isService(c) \ \Rightarrow \ \forall(iFil : IntentFilter), iFil \in c.intFilterS \ \Rightarrow$$

$$(iFil.dataFilter \neq nil \ \vee \ iFil.catFilter \neq nil \ \Rightarrow \ iFil.actFilter \neq nil)) \ \wedge$$

$$(isBReceiver(c) \ \Rightarrow \ \forall(iFil : IntentFilter, iFil \in c.intFilterB \ \Rightarrow$$

$$(iFil.dataFilter \neq nil \ \vee \ iFil.catFilter \neq nil \ \Rightarrow \ iFil.actFilter \neq nil)))$$

Propiedad 10 (statesConsistency) *Toda aplicación instalada en el sistema debe tener un manifest y un certificado que le pertenezcan y viceversa.*

$$\forall(a : idApp), a \in apps \ \Leftrightarrow$$

$$\exists(m : Manifest), hasManifest(a, m, sys) \ \wedge$$

$$\exists(cert : Cert), hasCert(a, cert, sys)$$

Propiedad 11 (intentsPermBroad) *Sólo los intents de tipo broadcast podrán tener definido un permiso.*

$$\forall(i : Intent)(ic : iCmp), \langle ic, i\rangle \in ints \ \wedge$$

$$(\neg \ isIntBReceiver(i) \ \Rightarrow \ i.brperm = None)$$

Por último se deben hacer algunas consideraciones sobre la definición de los campos *apps* y *sentIntents* del registro *State*. Si bien ambos campos se definen

como listas, sólo se hacen controles de pertenencia a la lista, y en el borrado de un elemento se eliminan todas las ocurrencias del mismo, por lo que no se consideraron propiedades de validez con respecto a esto.

Finalmente, un sistema Android válido queda definido como la conjunción de todas las propiedades mencionadas anteriormente, en el predicado *validState*.

3.5. Operaciones

Las operaciones disponibles en el modelo quedan definidas por el tipo *Action* y son las que se pueden observar en la tabla 3.2.

`install` a m $cert$ lr	Instala la aplicación de identificador a cuyo *manifest* es m, su certificado $cert$ y su conjunto de recursos lr en el sistema.
`uninstall` a	Desinstala la aplicación de identificador a del sistema.
`grant` p c	Otorga el permiso p al componente instalado c.
`revoke` p c	Revoca el permiso p al componente instalado c.
`hasPermission` p c	Controla si al componente instalado c le fue otorgado el permiso p
`read` ic cp u	La instancia en ejecución ic lee el recurso asociado a la URI u del *content provider* cp.
`write` ic cp u val	La instancia en ejecución ic escribe el valor val en el recurso asociado a la URI u del *content provider* cp.
`startActivity` i ic	La instancia en ejecución ic envía un *intent* i de tipo *intActivity* mediante la operación `startActivity`.
`startActivityForResult` i n ic	La instancia en ejecución ic envía un *intent* i de tipo *intActivity* cuya identificación de llamada es n mediante la operación `startActivityForResult`.
`startService` i ic	La instancia en ejecución ic envía un *intent* i de tipo *intService* mediante la operación `startService`.

Continúa en la siguiente página

Tabla 3.2 – continuación de la página anterior

`sendBroadcast` *i ic p*	La instancia en ejecución *ic* envía un *intent i* protegido por el permiso *p* de tipo *intBroad* mediante la operación `sendBroadcast`.
`sendOrderedBroadcast` *i ic p*	La instancia en ejecución *ic* envía un *intent i* protegido por el permiso *p* de tipo *intBroad* mediante la operación `sendOrderedBroadcast`.
`sendStikyBroadcast` *i ic*	La instancia en ejecución *ic* envía un *intent i* de tipo *intBroad* mediante la operación `sendStickyBroadcast`.
`resolveIntent` *i a*	El sistema decide si la aplicación *a* es candidata a recibir el *intent* implícito *i* y en caso afirmativo le asigna el *intent* al componente correspondiente.
`receiveIntent` *i ic a*	La aplicación *a* recibe el *intent i* enviado por la instancia en ejecución *ic*.
`stop` *ic*	La instancia *ic* deja de estar en ejecución.
`grantP` *ic cp a u pt*	La instancia en ejecución *ic* delega de forma permanente el permiso *pt*, sobre la URI *u* del *content provider cp*, a la aplicación *a*.
`revokeDel` *ic cp u pt*	La instancia *ic* revoca el permiso delegado *pt* sobre la URI *u* del *content provider cp*.
`call` *ic sac*	La instancia en ejecución *ic* hace una llamada a la API del sistema *sac* (de tipo *SACall*).

Tabla 3.2: Acciones

Las operaciones `startActivity`, `startActivityForResult`, `startService`, `sendBroadcast`, `sendOrderedBroadcast` y `sendStickyBroadcast` representan a las distintas formas existentes de mandar un *intent*. El modelado de la operación `boundService`, mencionada en el Capítulo 2, quedó por fuera del alcance de este proyecto.

3.6. Semántica de las Operaciones

La semántica de las operaciones queda dada por una precondición y una postcondición, es decir lo que debe cumplir un sistema para que se pueda ejecutar determinada operación y cómo queda el sistema luego de ejecutada.

La ejecución de una acción ac sobre un sistema sys válido se representa con la relación $\hookrightarrow$. La notación $sys \overset{ac}{\hookrightarrow} sys'$ significa que la tripla (sys, ac, sys') pertenece a la relación $\hookrightarrow$. Dicha relación se define a continuación.

$$\frac{Pre(sys, ac) \quad Post(sys, ac, sys')}{sys \overset{ac}{\hookrightarrow} sys'}$$

En las Figuras 3.1, 3.2, 3.3, 3.4, 3.5, 3.6 y 3.7 se pueden observar las pre y postcondiciones de las operaciones `install`, `grant`, `revoke`, `startActivity`, `sendBroadcast`, `resolveIntent` y `receiveIntent` respectivamente, que son consideradas las más relevantes para los aspectos de seguridad que interesa estudiar.

Las pre y postcondiciones del resto de las operaciones implementadas se encuentran disponibles en el Anexo I.

Para representar la semántica de las operaciones mencionadas se utilizan los predicados y funciones definidos en la Tabla 3.3 además de los previamente definidos.

$cmpNotInState(c, sys)$	Se cumple cuando el componente c no pertenece a ninguna aplicación instalada en el sistema sys
$cmpDeclareIntentFilterCorrectly(c)$	Se satisface si el componente c define sus *intent filter* de forma correcta, es decir, no existe un *intent filter* que tenga una categoría o un conjunto de datos asociado y no tenga una acción establecida.

Continúa en la siguiente página

Tabla 3.3 – continuación de la página anterior

$authPerms(a, cert, m, sys)$	Es verdadero cuando a la aplicación a se le otorgan los permisos de tipo *normal*, *signature* o *signature or system* en un sistema *sys*. El parámetro *cert* -que se corresponde con el certificado de la aplicación a- se utiliza para decidir cuándo se pueden otorgar permisos de tipo *signature* o *signature or system* y el parámetro m -que se corresponde con el *manifest* de la aplicación- se utiliza para acceder a los permisos requeridos por la aplicación a que se encuentran en el campo use del *manifest*.
$usrAuth(p)$	Se cumple cuando el usuario otorga el permiso p.
$hasGrantedPerms(a, lp, sys)$	Es verdadero cuando la lista de permisos otorgados a la aplicación a en un sistema *sys* es lp.
$cmpRunning(ic, sys)$	Se cumple cuando existe un componente perteneciente a una aplicación instalada en el sistema *sys* que se corresponde con la instancia en ejecución ic.
$actionTest(i, iFil, sys)$	Es verdadero si la acción especificada en el *intent* i coincide con una de las acciones del filter $iFil$, o si la lista del acciones de el *intent filter* $iFil$ no es vacía y el *intent* i no declara ninguna acción.
$categoryTest(i, iFil, sys)$	Se cumple si cada categoría declarada en el *intent* i pertenece a la lista de categorías del *intent filter* $iFil$, o si el *intent* i no declara ninguna categoría.
$dataTest(i, iFil, sys)$	Se verifica si el *intent filter* $iFil$ declara que puede manejar el tipo de datos establecido en el *intent* i.

Continúa en la siguiente página

Tabla 3.3 – continuación de la página anterior

$intentForApp(i, a, c, ic, sys)$	Se verifica si el *intent* i enviado por la instancia en ejecución ic se encuentra en la lista de *intents* enviados del sistema sys y además está destinado al componente c perteneciente a la aplicación a.
$canStart(c_1, c_2)$	Se cumple si c_1 y c_2 pertenecen a la misma aplicación o si c_2 puede ser iniciado por terceros y c_1 cuenta con los permisos para hacerlo.
$preReceiveIntentActivity(i, c)$	Es verdadero si el componente c puede delegar permisos temporales para realizar la acción especificada en el *intent* i.
$preReceiveIntentBroadcast(i, a_1, a_2)$	Se cumple si la aplicación a cuenta con los permisos necesarios para recibir el *intent* i o en el caso de que el permiso contenido en el *intent* i sea de tipo *signature* o *signature or system* y las aplicaciones a_1 y a_2 estén firmadas con el mismo certificado.
$insNotInState(ic, sys)$	Se cumple cuando no existe en el sistema sys la instancia en ejecución ic.
$isIntActivity(i)$	Es verdadero cuando el *intent* i va dirigido a una actividad.
$isIntService(i)$	Es verdadero cuando el *intent* i va dirigido a un servicio.
$addManifest(m, a, sys)$	Retorna el conjunto de *manifests* del sistema sys más el *manifest* m asociado a la aplicación a.
$addCert(cert, a, sys)$	Retorna el conjunto de certificados del sistema sys más el certificado $cert$ asociado a la aplicación a.
$addDefPerms(m, a, sys)$	Retorna el conjunto de permisos definidos por el usuario del sistema sys más el conjunto de permisos definidos por el usuario en el *manifest* m de la aplicación a.

Continúa en la siguiente página

Tabla 3.3 – continuación de la página anterior

$addApp(a, sys)$	Retorna el conjunto de aplicaciones del sistema *sys* al que se le agrega la aplicación *a*.
$addRes(lRes, a, sys)$	Retorna el conjunto de recursos del sistema *sys* al que se le agrega los recursos propios de la aplicación *a*.
$grantInstallPerms(a, m, sys)$	Devuelve el conjunto de permisos otorgados del sistema *sys* más los permisos de nivel *normal*, *signature* o *signature or system* definidos en el *manifest m* que debieron ser otorgados en tiempo de instalación a la aplicación *a*.
$grantDangerousPerm(a, p, m, sys)$	Agrega al conjunto de permisos otorgados del sistema *sys* el permiso p de nivel *dangerous* otorgado en tiempo de ejecución a la aplicación *a*, que debe estar definido en el *manifest m*.
$revokePerm(a, p, m, sys)$	Quita el permiso *p* de nivel *dangerous* del conjunto de permisos otorgados a la aplicación *a* en el sistema *sys*, que debe estar presente en el *manifest m*
$addIntent(i, ic, p, sys)$	Agrega el *intent i* enviado por la instancia *ic* mediante una operación protegida por el permiso *p* al conjunto de *intents* enviados del sistema *sys*.
$implicitToExplicitInt(i, a, sys)$	Le asigna al *intent* enviado *i*, el nombre del componente de la aplicación *a* al que va dirigido. Retorna el conjunto de *intents* enviados del sistema *sys*.
$runCmp(ic, c, sys)$	Agrega la instancia en ejecución *ic* del componente *c* al conjunto de instancias en ejecución del sistema *sys*.
$grantTempPerm(pt, u, cp, ic, sys)$	Se le delegan permisos temporales a la instancia *ic* para realizar la acción *pt* sobre la URI *u* del *content provider cp*.
$intentPath(i)$	Retorna el path de los datos contenidos en el *intent i*.

Continúa en la siguiente página

Tabla 3.3 – continuación de la página anterior

$removeIntent(i, ic, sys)$	Quita al *intent* i enviado por la instancia en ejecución ic del conjunto de *intents* enviados del sistema sys.

Tabla 3.3: Predicados y funciones auxiliares - semántica

Acción `install` a m $cert$ lr

La aplicación a se instala en el sistema.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \land e = (mfsts, certs, defPerms, img) \land \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \land \\ \forall (a_1 : idApp), a_1 \in aps \Rightarrow a_1 \neq a \land \\ \forall (c_1\ c_2 : Cmp), c_1 \in m.cmp\ \land c_2 \in m.cmp \Rightarrow \\ getCmpId(c_1) = getCmpId(c_2)\ \Rightarrow c_1 = c_2 \land \\ \forall (c : Cmp), c \in m.cmp \Rightarrow (cmpNotInState(c, sys) \land \\ cmpDeclareIntentFilterCorrectly(c)) \land \\ authPerms(a, cert, m, sys) \\ \\ addManifest(m, a, sys) = mfst'\ \land addCert(cert, a, sys) = certs' \land \\ addDefPerms(m, a, sys) = defPerms'\ \land addApp(a, sys) = apps' \land \\ addRes(lRes, a, sys) = vals' \land \\ grantInstallPerms(a, m, sys) = perms' \land \\ sys' = (e', s')\ \land e' = (mfst', certs', defPerms', img) \land \\ s' = (apps', perms', iCs, delPP, delTP, vals', ints) \end{array}}{sys \xrightarrow{\texttt{install}\ a\ m\ cert\ lr} sys'}$$

Precondición Para poder instalar la aplicación a en el sistema, no puede existir ninguna otra aplicación con el mismo identificador. No puede haber ningún componente perteneciente a la aplicación a que ya se encuentre instalado en el sistema, ni dos componentes iguales dentro de la misma. Además todos los componentes de a deben haber declarado de forma correcta sus *intent filters*. Por último se le deben otorgar todos los permisos cuyo nivel de protección no sea *dangerous*.

Postcondición La aplicación a se agrega a las apicaciones instaladas del sistema, así como su *manifest*, sus recursos, su certificado y sus permisos definidos. Los permisos otorgados a la aplicación se agregan al campo *perms* del estado.

Figura 3.1: Semántica de la operación `install`

Acción `grant` p c

Se le otorga el permiso p al componente c en tiempo de ejecución.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \land e = (mfsts, certs, defPerms, img) \land \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \land \\ \exists(ic : iCmp), running(ic, c, sys) \land \\ \exists(a : idApp)(m : Manifest), inApp(c, a, sys) \land hasManifest(a, m, sys) \land \\ p \in m.use \land (p.pl = dangerous \Rightarrow usrAuth(p)) \\ \\ \exists(a : idApp)(m : Manifest), inApp(c, a, sys) \land hasManifest(a, m, sys) \land \\ grantDangerousPerm(a, p, m, sys) = perms' \land \\ sys' = (e, s') \land s' = (apps, perms', iCs, delPP, delTP, vals, ints) \end{array}}{sys \xrightarrow{\texttt{grant}\ p\ c} sys'}$$

Precondición Para poder otorgar un permiso p en tiempo de ejecución al componente c, debe existir una instancia en ejecución de c, que requiera el permiso p en su *manifest*, que el mismo tenga un nivel de protección *dangerous* y finalmente que el usuario autorice el otorgamiento del permiso.

Postcondición La aplicación a la que pertenece el componente c tiene ahora al permiso p en su conjunto de permisos otorgados.

Figura 3.2: Semántica de la operación `grant`

Acción `revoke` p c

Se le revoca el permiso p al componente c en tiempo de ejecución.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \land e = (mfsts, certs, defPerms, img) \land \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \land \\ \exists (a : idApp)(lp : list\ Perm), inApp(c, a, sys) \land \\ hasGrantedPerms(a, lp, sys)\ \land p \in lp\ \land\ p.pl = dangerous \end{array}}{sys \xrightarrow{\texttt{revoke}\ p\ c} sys'}$$

$$\frac{\begin{array}{c} \exists (a : idApp)(lp : list\ Perm), inApp(c, a, sys) \land \\ hasGrantedPerms(a, lp, sys) \land \\ \exists (m : Manifest), hasManifest(a, m, sys) \land \\ p \in lp\ \land\ revokePerm(a, p, m, sys) = perms' \land \\ sys' = (e, s')\ \land s' = (apps, perms', iCs, delPP, delTP, vals, ints) \end{array}}{sys \xrightarrow{\texttt{revoke}\ p\ c} sys'}$$

Precondición Para poder revocar el permiso p del componente c en tiempo de ejecución, deberá existir una lista de permisos asociada a la aplicación a la que pertenece c y p deberá pertenecer a dicha lista. Además el p debe tener nivel de protección *dangerous*.

Postcondición Se quita el permiso p de la lista de permisos otorgados a la aplicación a la que pertenece c.

Figura 3.3: Semántica de la operación `revoke`

Acción `startActivity` i ic

La instancia ic manda un *intent* i mediante la operación `startActivity`

Regla

$$\frac{\begin{array}{c} sys = (e, s) \land e = (mfsts, certs, defPerms, img) \land \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \land \\ isIntActivity(i) \ \land i.brperm = None \land \\ cmpRunning(ic, sys) \ \land \ \langle ic, i\rangle \notin ints \\ \\ addIntent(i, ic, None, sys) = ints' \ \land sys' = (e, s') \land \\ s' = (apps, perms, iCs, delPP, delTP, vals, ints') \end{array}}{sys \xrightarrow{\texttt{startActivity}\ i\ ic} sys'}$$

Precondición

Para que la instancia ic pueda mandar el *intent* i mediante la operación `startActivity`, el *intent* i debe ser de tipo actividad y no puede tener declarado ningún permiso en su estructura. Además la instancia en ejecución ic deberá estar corriendo y el par $\langle ic, i\rangle$ no puede estar en el conjunto de *intents* enviados.

Postcondición El *intent* i y la instancia en ejecución ic que lo envió se agregan al conjunto de *intents* enviados del sistema. Además el *intent* no será modificado y deberá seguir sin tener ningún permiso declarado.

Figura 3.4: Semántica de la operación `startActivity`

Acción `sendBroadcast` *i ic p*

La instancia *ic* manda un *intent i* mediante la operación `sendBroadcast`, el *intent* está protegido por el permiso *p* (de tipo *option Perm*)

Regla

$$\frac{\begin{array}{c} sys = (e, s) \wedge e = (mfsts, certs, defPerms, img) \wedge \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \wedge \\ isIntBReceiver(i) \ \wedge i.brperm = None \wedge \\ cmpRunning(ic, sys) \ \wedge \ \langle ic, i\rangle \notin ints \\ \\ addIntent(i, ic, p, sys) = ints' \ \wedge sys' = (e, s') \wedge \\ s' = (apps, perms, iCs, delPP, delTP, vals, ints') \end{array}}{sys \xrightarrow{\texttt{sendBroadcast}\ i\ ic\ p} sys'}$$

Precondición Para que la instancia *ic* pueda mandar el *intent i* mediante la operación `sendBroadcast`, el *intent i* debe ser de tipo *broadcast* y no puede tener declarado ningún permiso en su estructura. Además la instancia en ejecución *ic* deberá estar corriendo y el par $\langle ic, i\rangle$ no puede estar en el conjunto de *intents* enviados.

Postcondición Al *intent i* se le agrega el permiso *p* especificado en la operación y es agregado al conjunto de *intents* enviados del sistema, junto con la instancia *ic* que lo envió.

Figura 3.5: Semántica de la operación `sendBroadcast`

Acción `resolveIntent` i a

El sistema decide si la aplicación a puede recibir el *intent* i.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \land e = (mfsts, certs, defPerms, img) \land \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \land \\ i.cmpName = None \land \\ \exists(ic : iCmp), \langle ic, i\rangle \ \in ints \land \exists(c_1 : Cmp), running(ic, c_1, sys) \land \\ \exists(c_2 : Cmp), inApp(c_2, a, sys) \land \\ \exists(iFil : IntentFilter), \\ (isIntActivity(i) \Rightarrow isActivity(c_2) \ \land \ iFil \in c_2.intFilterA) \land \\ (isIntService(i) \Rightarrow isService(c_2) \ \land \ iFil \in c_2.intFilterS) \land \\ (isIntBReceiver(i) \Rightarrow isBReceiver(c_2) \ \land \ iFil \in c_2.intFilterB) \land \\ actionTest(i, iFil, sys) \ \land categoryTest(i, iFil, sys) \land \\ dataTest(i, iFil, sys) \ \land canStart(c_1, c_2, sys) \\ \\ implicitToExplicitInt(i, a, sys) = ints' \land \\ sys = (e, s') \land s' = (apps, perms, iCs, delPP, delTP, vals, ints') \end{array}}{sys \xrightarrow{\texttt{resolveIntent}\ i\ a} sys'}$$

Precondición Para que una aplicación a sea candidata a recibir el *intent* i el *intent* debe estar en la lista de *intents* enviados del sistema. Debe existir un componente en a cuyo tipo se corresponda con el tipo de i. Debe existir una instancia en ejecución ic que haya enviado el *intent*, y su componente asociado debe poder iniciar al componente de la aplicación a. Por último debe haber un componente perteneciente a la aplicación a que tenga todos los *intents filters* necesarios para realizar la acción exigida por el *intent*.

Postcondición Se define el nombre del componente que recibirá el *intent* i y se incluye en el mismo, es decir que el *intent* i pasa a ser explícito.

Figura 3.6: Semántica de la operación `resolveIntent`

Acción `receiveIntent` *i ic a*

La aplicación *a* recibe el *intent i* enviado por la instancia en ejecución *ic*.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \land e = (mfsts, certs, defPerms, img) \land \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \land \\ \exists (c : Cmp), intentForApp(i, a, c, ic, sys) \land \neg isCProvider(c) \land \\ \exists (c_1 : Cmp)(a_1 : idApp), inApp(c_1, a_1, sys) \land \\ running(ic, c_1, sys) \land \neg isCProvider(c_1) \land canStart(c_1, c) \land \\ (isIntActivity(i) \Rightarrow preReceiveIntentActivity(i, c_1)) \land \\ (isIntBReceiver(i) \land i.brperm <> None \Rightarrow \\ preReceiveIntentBroadcast(i, a, a_1)) \\ \\ \exists (ic1 : iCmp)(c : Cmp), (intentForApp(i, a, c, ic, sys) \land \\ insNotInState(ic_1, sys) \land \\ runCmp(ic_1, c, sys) = iCs' \land \\ (isIntActivity(i) \Rightarrow (\exists (u : uri)(a_1 : idApp)(cp : CProvider), \\ intentPath(i) = Some\ u \land \\ inApp(cp, a_1, sys) \land existsRes(cp, u, sys) \land \\ grantTempPerm(intentActionType(i), u, cp, ic_1, sys) = delTP') \lor \\ intentPath(i) = None \land delTP = delTP') \land \\ (isIntBReceiver(i) \Rightarrow delTP = delTP') \land \\ (isIntService(i) \Rightarrow delTP = delTP')) \land \\ removeIntent(i, ic, sys) = ints' \land \\ sys' = (e, s') \land s' = (apps, perms, iCs', delPP, delTP', vals, ints') \end{array}}{sys \xrightarrow{\texttt{receiveIntent}\ i\ ic\ a} sys'}$$

Precondición Para que una aplicación *a* pueda recibir un *intent i* el mismo debe estar destinado a un componente de la aplicación que no puede ser un *content provider*. Además el componente c_1 que envió el *intent* debe poder iniciar a *c*. En caso de que el *intent* esté destinado a una actividad, el componente c_1 deberá poder delegar los permisos temporales que sean necesarios para poder llevar a cabo la acción pedida por el *intent*. En caso de que el *intent* esté destinado a un *broadcast receiver* y que el mismo haya sido enviado mediante una operación protegida por un permiso, la aplicación *a* deberá contar con el permiso correspondiente.

Postcondición Luego de que un componente *c* de una aplicación recibe un *intent* se crea una instancia en ejecución asociada al mismo. En caso de que el *intent* esté destinado a una actividad se deben haber delegado los permisos temporales correspondientes. En cualquier otro caso los permisos temporales no cambian.

Figura 3.7: Semántica de la operación `receiveIntent`

3.7. Invarianza de la Validez del Sistema

En primer lugar se demostró que la ejecución de toda operación de tipo *Action* sobre un sistema válido resulta en un sistema que también es válido.

Además de asegurar que el modelo construido es válido, la propiedad de invarianza de validez de estado es utilizada para probar propiedades de seguridad introducidas en las secciones 4.1 y 4.2.

Al probar la invarianza de validez de estado se asegura que las propiedades de validez de estado mencionadas en la sección 3.4 se mantienen luego de ejecutada cualquier operación. De existir una operación que no mantenga alguna de dichas propiedades, se podría llegar a inconsistencias en el sistema y las operaciones modeladas no se corresponderían con las operaciones existentes en un sistema Android.

El enunciado del teorema que se debe probar es:

Teorema 1 *Si se ejecuta cualquier acción a en un sistema válido sys se obtendrá un sistema sys′ también válido.*

$$\forall(a : Action)(sys\ sys' : System),\ validState(sys) \Rightarrow$$

$$sys \xhookrightarrow{a} sys' \Rightarrow validState(sys')$$

Para probar este teorema se debe probar que todas las propiedades definida en la sección 3.4 siguen siendo válidas luego de ejecutar cada una de las acciones definidas en la sección 3.5. Para esto se optó por separar la demostración en lemas para cada una de las propiedad que componen el predicado *validState*. Los enunciados de estos lemas son similares a los del **Teorema 1** pero se reemplaza el predicado *validState* de la conclusión por cada una de las propiedades correspondientes. Por ejemplo, para la propiedad *delTmpRun* se define el lema

Lema 1 *Si se ejecuta cualquier acción a en un sistema válido sys se obtendrá un sistema sys′ en el que se cumple delTmpRun(sys′)*

$$\forall(a : Action)(sys\ sys' : System), validState(sys) \Rightarrow$$

$$sys \xhookrightarrow{a} sys' \Rightarrow delTmpRun(sys')$$

Cada uno de estos lemas debe probarse para cada una de las operaciones de tipo *Action* por lo que se modularizó esta prueba separando nuevamente en lemas auxiliares, uno para cada acción. A modo de ejemplo, a continuación se muestran dos de ellos junto con su demostración.

Lema 2 *Dado un intent i, una instancia en ejecución ic, una aplicación a y un sistema válido sys, si se ejecuta la operación $receiveIntent(i, ic, a)$ se obtendrá un sistema sys' en el que se cumple $delTmpRun(sys')$.*

$$\forall(i : Intent)(ic : iCmp)(a : idApp)(sys\ sys' : System),\ validState(s) \Rightarrow$$

$$sys \xrightarrow{\texttt{receiveIntent}\ i\ ic\ a} sys' \Rightarrow delTmpRun(sys')$$

Demostración: Sean $i : Intent$, $ic : iCmp$, $a : idApp$, $sys = (e, s)$ donde $e = (mfsts, certs, defPerms, appRes, img)$ y $s = (apps, perms, iCs, delPP, delTP, vals, ints)$ y $sys' = (e', s')$ donde $e' = (mfsts', certs', defPerms', appRes', img')$ y $s' = (apps', perms', iCs', delPP', delTP', vals', ints')$.
Suponiendo que se cumplen $validState(sys)$ y $sys \xrightarrow{\texttt{receiveIntent}\ i\ ic\ a} sys'$, el objetivo es probar $delTmpRun(sys')$.
De acuerdo a lo definido en la sección 3.4 probar $delTmpRun(sys')$ significa probar que:

$$\forall(ic_1 : iCmp)(cp : CProvider)(u : uri)(pt : PType),$$

$$delTP'\ ic_1\ cp\ u = pt \Rightarrow$$

$$\exists(a : idApp), inApp((cmpCP\ cp), a, sys') \wedge$$

$$\exists(c : Cmp)(a_1 : idApp), inApp(c, a_1, sys') \wedge\ running(ic_1, c, sys')$$

Dados $ic_1 : iCmp$, $cp : CProvider$, $u : uri$, $pt : PType$ tales que se cumple $delTP'\ ic\ cp\ u = pt$, hay que probar que existe una aplicación instalada que contiene a cp y que existe un componente c contenido en una aplicación instalada a_1, que tiene a ic_1 como una de sus instancias en ejecución.

Por hipótesis se cumple $sys \xrightarrow{\texttt{receiveIntent}\ i\ ic\ a} sys'$, entonces vale la postcondición de la operación `receiveIntent` (ver Figura 3.7). En particular, se puede afirmar que:

$$(isIntActivity(i) \Rightarrow (\exists(u : uri)(a_1 : idApp)(cp : CProvider),$$
$$intentPath(i) = Some\ u\ \wedge$$
$$inApp(cp, a_1, sys)\ \wedge existsRes(cp, u, sys) \wedge$$
$$grantTempPerm(intentActionType(i), u, cp, ic_1)sys = delTP') \vee$$
$$intentPath(i) = None\ \wedge delTP = delTP')\ \wedge$$
$$(isIntBReceiver(i) \Rightarrow delTP = delTP') \wedge$$
$$(isIntService(i) \Rightarrow delTP = delTP')$$

por lo tanto, dependiendo del tipo de *intent* de i los permisos delegados temporalmente se verán o no afectados. De esta forma se puede separar la prueba en tres casos, según el tipo de *intent* de i.

- **Caso 1 isIntActivity(i):** a partir de la postcondición de la operación `receiveIntent` se desprende que:

$$(\exists(u : uri)(a : idApp)(cp : CProvider),$$
$$intentPath(i) = Some\ u\ \wedge$$
$$inApp(cp, a_1, sys)\ \wedge existsRes(cp, u, sys) \wedge$$
$$grantTempPerm(intentActionType(i), u, cp, ic_1)sys = delTP') \vee$$
$$intentPath(i) = None\ \wedge delTP = delTP'$$

  Esto quiere decir que se delegaron temporalmente los permisos para realizar la acción en caso que fuera necesario, o no se modificaron los permisos delegados temporalmente. Por lo que se deben probar ambos casos.

- **Caso 1.1 se delegaron temporalmente los permisos para realizar la acción:** para que esto ocurra todos los *content provider*, URIs, instancias en ejecución y tipos de permisos que componen $delTP'$ tienen que o bien que estar en $delTP$ o bien corresponderse con la delegación temporal que se está realizando. En particular ic_1, cp, u y pt tienen que una de estas condiciones. Entonces se tiene que hacer una prueba para cada uno de los casos mencionados.
  - **ic_1, cp, u y pt forman parte de delTP:** como este caso contempla que no haya habido un cambio en los permisos delegados temporalmente, la prueba se reduce a aplicar la validez de estado para el sistema sys. En particular la hipótesis $delTPerms(sys)$.
  - **ic_1, cp, u y pt se corresponden con la nueva delegación:** sean u_1, a_1 y cp_1 obtenidos de suponer $isIntActivity(i)$, entonces $cp = cp_1$, $u = u_1$ y $pt = pt_1$ debido a que son los involucrados en la nueva delegación. En primer lugar hay que probar que existe una aplicación instalada en sys' que contiene a cp. Esto se cumple porque las aplicaciones instaladas no son modificadas por esta acción y por la hipótesis obtenida al suponer $isIntActivity(i)$. Resta probar que existe un componente c_1 contenido en una aplicación instalada a_1 en sys', que tiene a ic_1 como una de sus instancias en ejecución. Como por la pre y postcondición de la operación `receiveIntent` el *intent* va dirigido a la aplicación a, debe existir un componente que pertenezca a dicha aplicación, por lo que existe un componente contenido en una aplicación instalada. A su vez, este componente será el que reciba el *intent*, por lo tanto se comenzará a ejecutar la instancia correspondiente. Dicha instancia será igual a $ic1$ ya que es la que se desprende de la recepción del *intent* y por tanto también es la involucrada en la nueva delegación temporal de permisos.
- **Caso 1.2 no se modificaron los permisos delegados temporalmente:** este caso se desprende directamente de la aplicación de la propiedad de validez de estado $delTPerms$ para el sistema sys.

- **Caso 2 isIntBReceiver(i):** se desprende directamente de la aplicación de la propiedad de validez de estado $delTPerms$ para el sistema sys.

- **Caso 3 isIntService(i):** análogo al anterior.

Lema 3 *Dado un permiso p, un componente c y un sistema válido sys, si se ejecuta la operación grant(p, c) se obtendrá un sistema sys′ en el que se cumple delTmpRun(sys′).*

$$\forall(p : Perm)(c : Cmp)(sys\ sys' : System),\ validState(s) \Rightarrow$$

$$sys \xrightarrow{\texttt{grant}\ p\ c} sys' \Rightarrow delTmpRun(sys')$$

Demostración: Sean $p : Perm$, $c : Cmp$, $sys = (e, s)$ con $e = (mfsts, certs, defPerms, appRes, img)$ y $s = (apps, perms, iCs, delPP, delTP, vals, ints)$ y $sys' = (e', s')$ con $e' = (mfsts', certs', defPerms', appRes', img')$ y $s = (apps', perms', iCs', delPP', delTP', vals', ints')$.
Suponiendo que se cumplen $validState(sys)$ y $sys \xrightarrow{\texttt{grant}\ p\ c} sys'$, el objetivo es probar $delTmpRun(sys')$, que de acuerdo a lo definido en la sección 3.4 significa probar que

$$\forall(ic : iCmp)(cp : CProvider)(u : uri)(pt : PType),$$

$$delTP'\ ic\ cp\ u = pt \Rightarrow$$

$$\exists(a : idApp), inApp((cmpCPcp), a, sys') \wedge$$

$$\exists(c : Cmp)(a1 : idApp), inApp(c, a1, sys') \wedge\ running(ic, c_1, sys')$$

Además como se sabe que $sys \xrightarrow{\texttt{grant}\ p\ c} sys'$ se cumple, entonces la postcondición de la operación `grant` se cumple. En particular, se puede afirmar que $sys' = (e, s')$ y $s' = (apps, perms', iCs, delPP, delTP, vals, ints)$ (ver Figura 3.2). Esto quiere decir que lo único que cambia del sistema sys al sistema sys' son los permisos otorgados. De lo anterior se puede desprender que las instancias en ejecución no cambian luego de la ejecución de la operación:

$$\forall(c : Cmp)(ic : iCmp)running(ic, c, sys') \Leftrightarrow running(ic, c, sys)$$

También se desprende que las aplicaciones instaladas no cambian luego de la ejecución de la operación y por lo tanto todos los componentes que pertenecían a una aplicación antes de la ejecución de la operación lo van a seguir haciendo luego de ejecutada la misma.

$$\forall(a : idApp)(c : Cmp),\ inApp(c, a, sys') \Leftrightarrow inApp(c, a, sys)$$

Por último se sabe que los permisos delegados temporalmente no cambian: $delTP = delTP'$

Aplicando los tres puntos anteriores, para probar la tesis alcanza con probar

$$\forall(ic : iCmp)(cp : CProvider)(u : uri)(pt : PType),$$
$$delTP\ ic\ cp\ u = pt \Rightarrow$$
$$\exists(a : idApp), inApp((cmpCPcp), a, sys) \wedge$$
$$\exists(c : Cmp)(a1 : idApp), inApp(c, a1, sys) \wedge\ running(ic, c_1, sys)$$

y esto, por la definición de $delTmpRun$, es exactamente lo mismo que probar $delTmpRun(sys)$ y se cumple por definición de $validState(sys)$

Capítulo 4

Verificación de Propiedades de Seguridad

En este capítulo se presentan algunas propiedades en relación a la especificación formalizada en el capítulo 3. En la sección 4.1 se enuncian y demuestran algunas propiedades para analizar el funcionamiento del modelo de seguridad de Android. En la sección 4.2 se presentan debilidades del sistema operativo y ciertos escenarios donde se podrían prevenir algunos ataques.

4.1. Propiedades de Seguridad

Para formalizar los lemas a probar en esta sección se usaron los predicados y las funciones auxiliares definidos en la Tabla 4.1 además de los definidos anteriormente.

Cada vez que se mencione la utilización de un elemento para definir una propiedad, de forma implícita se hace referencia a la una variable cuantificada universalmente.

$componentPermission(c)$	Retorna el permiso que protege al componente c en caso de que exista y $None$ en caso contrario.
	Continúa en la siguiente página

Tabla 4.1 – continuación de la página anterior

$applicationPermission(m)$	Retorna el permiso definido en m que protege a la aplicación asociada al mismo en caso de que exista, y $None$ en caso contrario.
$canRead(c, cp, sys)$	Se cumple cuando el componente c tiene permisos otorgados para leer el *content provider* cp.
$delPerms(c, cp, u, pt, sys)$	Es verdadero cuando el componente c tiene permisos delegados para realizar la operación pt sobre el recurso u del *content provider* cp.
$canBeStarted(c)$	Si el componente c es una actividad, un servicio o un *broadcast receiver*, el predicado se cumple cuando c puede ser iniciado por terceros. Si c es un *content provider* el predicado se cumple cuando sus recursos pueden ser accedidos por terceros.
$canGrant(cp, u, sys)$	Se cumple cuando el *content provider* cp permite la delegación de permisos sobre el recurso u.
$intentActionType(i)$	Retorna el tipo de acción que el *intent* i requiere.
$cmpCanCall(c, sac, sys)$	Es verdadero cuando el componente c puede realizar la llamada sac a la API del sistema.

Tabla 4.1: Predicados y funciones auxiliares - verificación

4.1.1. Principio de Mínimo Privilegio

Mediante el uso de *sandbox* Android implementa el Principio de Mínimo Privilegio [4], esto implica que por defecto cada aplicación solo tiene acceso a los componentes que requiere para trabajar. Para poder intercambiar información con componentes externos, las aplicaciones deberán contar con permisos adecuados.

A continuación se describen una serie de propiedades de seguridad que corroboran que el modelo construido cumple con lo mencionado anteriormente.

4.1.1.1. Inicio de Componentes

Se definen un conjunto de lemas que establecen las condiciones para que un componente pueda ser iniciado por otro.

Para la definición de dichos lemas se utilizaron los elementos $c_1,\ c_2 : Cmp$,

$a_1,\ a_2 : idApp$, $ic_1 : iCmp$, i : y un sistema $sys = (s, e)$ válido, donde $s = (apps, perms, iCs, delPP, delTP, vals, ints)$ y $e = (mfsts, certs, defPerms, img)$.

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

$$inApp(c_1, a_1, sys) \ \wedge \ inApp(c_2, a_1, sys) \ \wedge$$
$$(isIntActivity(i) \Rightarrow preReceiveIntentActivity(i, c_1)) \ \wedge$$
$$(isIntBReceiver(i) \wedge i.brperm <> None \Rightarrow$$
$$preReceiveIntentBroadcast(i, a_1, a_1)) \ \wedge$$
$$\langle ic, i\rangle \ \in ints \ \wedge \ i.cmpName = Some\ getCmpId(c_2) \wedge$$
$$\neg isContentProvider(c_2) \wedge running(ic_1, c_1, sys) \Rightarrow$$
$$Pre(sys, \texttt{receiveIntent}(i, ic_1, a))$$

Lema 5 *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.*

$$\forall (m_1 : Manifest)(l : list\ Perm), inApp(c_1, a_1, sys) \ \wedge$$
$$inApp(c_2, a_2, sys) \ \wedge a_1 \neq a_2 \ \wedge \ i.cmpName = Some\ getCmpId(c_2) \ \wedge$$
$$componentPermission(c_2) = Some\ p \ \wedge \ hasManifest(a_1, m_1, sys) \ \wedge$$
$$hasGrantedPerms(a_1, l, sys) \ \wedge \ p \notin l \ \wedge \ running(ic_1, c_1, sys) \Rightarrow$$
$$\neg Pre(sys, \texttt{receiveIntent}(i, ic_1, a_2))$$

Lema 6 *Si c_1 y c_2 pertenecen a aplicaciones distintas, c_2 no exige ningún permiso pero su aplicación sí lo hace y la aplicación correspondiente a c_1 no tiene este último permiso, entonces c_1 no podrá iniciar a c_2.*

$$\forall(m_1\ m_2 : Manifest)(l : list\ Perm), inApp(c_1, a_1, sys)\ \wedge$$

$$inApp(c_2, a_1, sys)\ \wedge a_1 \neq a_2\ \wedge\ i.cmpName = Some\ getCmpId(c_2)\ \wedge$$

$$hasManifest(a_1, m_1, sys)\ \wedge\ hasManifest(a_2, m_2, sys)\ \wedge$$

$$componentPermission(c_2) = None \wedge$$

$$applicationPermission(m_2) = Some\ p \wedge$$

$$hasGrantedPerms(a_1, l, sys)\ \wedge\ p \notin l\ \wedge\ running(ic_1, c_1, sys) \Rightarrow$$

$$\neg Pre(sys, \texttt{receiveIntent}(i, ic_1, a_2))$$

4.1.1.2. Lectura/Escritura de *Content Providers*

Se definen lemas que corroboran que un componente cumpla con las condiciones necesarias para poder acceder a los recursos de un *content provider*.

Todos los lemas definidos en esta sección se cumplen para lectura y escritura de *content provider*, y la única diferencia entre ellos es que se utiliza el predicado $canRead$ o $canWrite$ según corresponda. Por la similaridad entre los lemas, solo se detallarán los de lectura.

Para la definición de dichos lemas se utilizaron los elementos $cp : CProvider$, $c : Cmp$, $ic : iCmp$, $u : uri$, un sistema $sys = (s, e)$ y un sistema $sys' = (s', e')$ válidos.

Lema 7 *Si ic pudo leer el recurso apuntado por u en cp, entonces su componente asociado pertenece a una aplicación que tiene los permisos delegados u otorgados para ello.*

$$running(ic, c, sys)\ \wedge\ sys \xrightarrow{\textbf{\textit{read}}\ ic\ cp\ u} sys' \Rightarrow$$

$$canRead(c, cp, sys) \vee delPerms(c, cp, u, Read, sys)$$

Lema 8 *Si cp pertenece a la misma aplicación que c, entonces c puede leer a cp.*

$$\forall(a : idApp),\ running(ic, c, sys)\ \wedge\ inApp(c, a, sys)\ \wedge$$
$$inApp((cmpCP\ cp), a, sys)\ \wedge\ existsRes(cp, u, sys)\ \Rightarrow$$
$$Pre(sys, \texttt{read}(ic, cp, u))$$

Lema 9 *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.*

$$\forall(a\ a_1 : idApp),\ running(ic, c, sys)\ \wedge\ inApp((cmpCP\ cp), a_1, sys)\ \wedge$$
$$inApp(c, a, sys)\ \wedge\ existsRes(cp, u, sys)\ \wedge$$
$$\neg canBeStarted(cp)\ \wedge\ a \neq a_1 \Rightarrow$$
$$Pre(sys, \texttt{read}(ic, cp, u)) \Leftrightarrow delPerms(c, cp, u, Read, sys)$$

4.1.1.3. Delegación y Revocación de Permisos

En esta sección se prueba que un componente sólo podrá delegar permisos de forma temporal y/o permanente para realizar operaciones que pueda hacer.

Todos los lemas definidos en esta sección se cumplen para lectura y escritura de *content provider*, y la única diferencia entre ellos es que se utiliza el predicado $canRead$ o $canWrite$ según corresponda. Por la similaridad entre los lemas, solo se detallarán los de lectura.

Para la definición de dichos lemas se utilizaron los elementos $ic : iCmp$, $cp : CProvider$, $u : uri$, $c : Cmp$, $i : Intent$ un sistema $sys = (s, e)$ y un sistema $sys' = (s', e')$ válidos, donde $s = (apps, perms, iCs, delPP, delTP, vals, ints)$ y $e = (mfsts, certs, defPerms, img)$.

Lema 10 *Si ic delegó permisos permanentes para leer un recurso apuntado por u de cp, entonces ic puede realizar la operación en cuestión.*

$$\forall(a\ a_1 : idApp),\ running(ic, c, sys)\ \wedge$$

$$inApp((cmpCP\ cp), a_1, sys)\ \wedge\ existsRes(cp, u, sys)\ \wedge$$

$$canGrant(cp, u, s)\ \wedge sys \xrightarrow{\texttt{grantP}\ ic\ cp\ a\ u\ Read} sys'\ \Rightarrow$$

$$canRead(c, cp, sys) \vee delPerms(c, cp, u, Read, sys)$$

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

$$\forall(a : idApp),\ running(ic, c, sys)\ \wedge$$

$$isIntActivity(i) \wedge\ intentActionType(i) = Read\ \wedge$$

$$intentPath(i) = Some\ u \wedge sys \xrightarrow{\texttt{receiveIntent}\ i\ ic\ a} sys'\ \Rightarrow$$

$$\exists(cp_1 : CProvider), canRead(c, cp_1, sys) \vee delPerms(c, cp_1, u, Read, sys)$$

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.*

$$\forall(a\ a_1 : idApp)(c_1 : Cmp)(pt : PType),$$

$$inApp(c, a, sys)\ \wedge\ inApp((cmpCP\ cp), a, sys)\ \wedge$$

$$\langle ic, i\rangle\ \in ints\ \wedge\ inApp(c_1, a_1, sys)\ \wedge i.cmpName = Some\ getCmpId(c_1)\ \wedge$$

$$\neg isCProvider(c_1)\ \wedge\ running(ic, c, sys)\ \wedge$$

$$canStart(c, c_1, sys)\ \wedge canGrant(cp, u, sys)\ \wedge existsRes(cp, u, sys)\ \wedge$$

$$isIntActivity(i) \wedge\ intentActionType(i) = pt\ \wedge\ intentPath(i) = Some\ u \Rightarrow$$

$$Pre(sys, \texttt{receiveIntent}(i, ic, a_1))\ \wedge$$

$$Pre(sys, \texttt{grant}(ic, cp, a_1, u, pt))$$

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

$$running(ic, c, sys) \ \wedge \ sys \xrightarrow{\texttt{revokeDel}\ i\ cp\ u\ Read} sys' \ \Rightarrow$$

$$canRead(c, cp, sys) \vee delPerms(c, cp_1, u, Read, sys)$$

4.1.2. Revocación Irrestricta

Esta propiedad especifica la imposibilidad de revocar un permiso a una aplicación o instancia.

Para la definición de este lema se utilizaron los elementos $ic : iCmp$, $cp : CProvider$, $u : uri$, $pt : PType$, un sistema $sys = (s, e)$ y un sistema $sys' = (s', e')$ válidos, donde $s' = (apps', perms', iCs', delPP', delTP', vals', ints')$ y $e' = (mfsts', certs', defPerms', img')$.

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

$$sys \xrightarrow{\texttt{revokeDel}\ ic\ cp\ u\ pt} sys' \ \wedge$$

$$\forall(ic_1 : iCmp)(c_1 : Cmp), running(ic_1, c_1, sys) \ \wedge \ delTP'\ ic_1\ cp\ u \ <> \ pt \ \wedge$$

$$\forall(a : idApp), a \in apps \Rightarrow delPP'\ a\ cp\ u \ <> \ pt$$

4.1.3. Redelegación de Permisos

En esta sección se prueba que un permiso puede ser delegado por cualquier instancia en ejecución de forma indefinida. Esto provoca que un permiso que fue delegado de forma temporal, pase a estar delegado de forma permanente.

Para la definición de esta propiedad se utilizaron los elementos $c : Cmp$, $a : idApp$, $ic : iCmp$, $i : Intent$, $cp : CProvider$, $u : uri$, $pt : PType$ y un sistema $sys = (s, e)$ válido, donde $s = (apps, perms, iCs, delPP, delTP, vals, ints)$ y $e = (mfsts, certs, defPerms, img)$.

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

$$\forall(a_1 : idApp)(c_1 : Cmp),\ inApp(c, a, sys)\ \wedge\ running(ic, c, sys)\ \wedge$$

$$(delTP\ ic\ cp\ u\ =\ pt\ \vee delPP\ a\ cp\ u\ =\ pt)\ \wedge$$

$$\exists(a_{cp} : idApp),\ inApp((cmpCP\ cp), a_{cp}, sys)\ \wedge inApp(c_1, a_1, sys)\ \wedge$$

$$\neg isCProvider(c_1)\ \wedge\ i.cmpName = Some\ getCmpId(c_1)\ \wedge\ \langle ic, i\rangle\ \in ints \wedge$$

$$isIntActivity(i) \wedge\ intentActionType(i) = pt\ \wedge\ intentPath(i) = Some\ u \wedge$$

$$canStart(c, c_1, sys)\ \wedge canGrant(cp, u, sys)\ \wedge existsRes(cp, u, sys) \Rightarrow$$

$$Pre(sys, \texttt{receiveIntent}(i, ic, a_1))\ \wedge$$

$$Pre(sys, \texttt{grantP}(ic, cp, a_1, u, pt))$$

4.1.4. *Privilege Escalation*

"Este problema se da cuando una aplicación con menos privilegios, puede de todas formas acceder a componentes pertenecientes a una aplicación con más privilegios."[41]

Esto se realiza mediante el uso llamadas intermedias que la aplicación con menos privilegios puede realizar "escalando"las llamadas hasta obtener la información deseada.

Para la formalización de este lema se utilizaron los elementos $c\ c_1 : Cmp$, $sac : SACall$, $ic : iCmp$, $i : Intent$, un sistema $sys = (s, e)$ y un sistema $sys' = (s', e')$ válidos.

Lema 16 *Si ic no puede iniciar un componente c_2 pero inicia a c_1 que sí puede hacerlo, entonces, se podrá iniciar a c_2 a través de una instancia en ejecución de c_1.*

$$\forall(a_1\ a_2 : idApp)(i_1 i_2 : Intent)(c_2 : Cmp),$$

$$inApp(c_2, a_2, sys)\ \wedge\ i_2.cmpName = Some\ getCmpId(c_2) \wedge$$

$$(isIntActivity(i_2) \Rightarrow preReceiveIntentActivity(i_2, c_1))\ \wedge$$

$$(isIntBReceiver(i_2) \wedge i.brperm <> None \Rightarrow$$

$$preReceiveIntentBroadcast(i_2, a_2, a_1))\ \wedge$$

$$\neg isCProvider(c_2)\ \wedge\ running(ic, c, sys)\ \wedge$$

$$\neg canStart(c, c_2, sys)\ \wedge \neg canStart(c, c_2, sys')\ \wedge canStart(c_1, c_2, sys')\ \wedge$$

$$inApp(c_1, a_1, sys)\ \wedge\ i_1.cmpName = Some\ getCmpId(c_1) \wedge$$

$$sys \xrightarrow{\texttt{receiveIntent}\ i_1\ ic\ a_1} sys'\ \Rightarrow$$

$$\exists(ic_1 : iCmp), running(ic_1, c_1, sys)\ \wedge$$

$$(\langle ic_1, i_2\rangle \Rightarrow Pre(sys, \texttt{receiveIntent}(i_2, ic_1, a_2)))$$

Lema 17 *Si ic no puede realizar una llamada al sistema sac pero inicia un componente c_1 que sí puede hacerlo, entonces, luego de iniciar a c_1, se podrá llamar a sac a través de una instancia en ejecución del mismo.*

$$\forall(a_1 : idApp)\ running(ic, c, sys)\ \wedge$$

$$\neg\texttt{pre\_call}(ic, sac, sys)\ \wedge \neg\texttt{pre\_call}(ic, sac, sys')\ \wedge$$

$$cmpCanCall(c_1, sac, sys')\ \wedge\ inApp(c_1, a_1, sys)\ \wedge$$

$$i.cmpName = Some\ getCmpId(c_1)\ \wedge$$

$$sys \xrightarrow{\texttt{receiveIntent}\ i\ ic\ a_1} sys'\ \Rightarrow$$

$$\exists(ic_1 : iCmp),\ running(ic_1, c_1, sys')\ \wedge Pre(sys', \texttt{call}(ic_1, sac))$$

4.2. Debilidades

En esta sección se presentan algunos ataques conocidos a los que el sistema Android es vulnerable y se definen escenarios en los que dichos ataques no pueden darse. A partir de estas dos cosas se prueba que efectivamente en los escenarios encontrados los ataques no pueden llevarse a cabo.

Cada vez que se mencione la utilización de un elemento para definir una propiedad, de forma implícita se hace referencia a la una variable cuantificada universalmente.

4.2.1. *Eavesdropping*

El ataque *eavesdropping* refiere al monitoreo no autorizado de información [49]. Al no afectar el normal funcionamiento del dispositivo ni el emisor ni el receptor detecta que se está filtrando información [49]. Si en un dispositivo Android hay alguna aplicación maliciosa instalada, cada vez que se manda un mensaje público de tipo *broadcast* con algún tipo de información sensible el dispositivo es vulnerable a este ataque. Como se mencionó en el capítulo 2, al mandar un mensaje de tipo *broadcast* mediante la operación `sendBroadcast` o `sendOrderedBroadcast` se puede proteger la información contenida en el mensaje mediante un permiso. De esta forma, si cada vez que se manda un *intent* mediante las operaciones mencionadas anteriormente se protege mediante un permiso de tipo *signature* o *signature or system*, se puede asegurar que solo aplicaciones firmadas con el mismo certificado que la aplicación emisora podrán acceder a la información sensible [38]. En estas condiciones el ataque *eavesdropping* no puede realizarse.

Se prueba entonces que en un escenario en que se manda un *intent* mediante la operación `sendBroadcast` o `sendOrderedBroadcast` protegido con un permiso de tipo *signature* o *signature or system* ninguna aplicación que esté firmada con un certificado distinto al de la aplicación emisora podrá recibirlo.

Para la definición de esta propiedad se utilizaron los elementos $c : Cmp$, a $a_1 : idApp$, $ic : iCmp$, $i : Intent$ y un sistema $sys = (s, e)$ válido, donde $s = (apps, perms, iCs, delPP, delTP, vals, ints)$ y $e = (mfsts, certs, defPerms, img)$.

Observar que según la semántica de las operaciones `sendBroadcast` (ver Figura 3.5), `sendOrderedBroadcast` (ver Figura I.7) y `sendStickyBroadcast` (ver Figura I.8), solamente las dos primeras pueden ser protegidas con un permiso,

mientras que en la tercera se exige que no haya ningún permiso. De esta forma esta propiedad queda restringida a las dos primeras operaciones.

Lema 18 *Si un componente c perteneciente a una aplicación a envía un intent de tipo broadcast protegido por un permiso de tipo signature o signature or system entonces si a_1 no tiene el mismo certificado que a no podrá recibirlo.*

$$running(ic, c, sys) \ \wedge \ \neg isCProvider(c) \ \wedge$$
$$inApp(c, a, sys) \ \wedge \ \langle ic, i\rangle \ \in ints \ \wedge$$
$$isIntBReceiver(i) \ \wedge \ \exists(p : Perm),$$
$$(p.pl = signature \ \vee \ p.pl = signatureOrSys) \ \wedge$$
$$i.brperm = Some \ p \ \wedge \ a_1 \in apps \ \wedge$$
$$\exists(cert : Cert), \ hasCert(a, cert, sys) \ \wedge \ \neg \ hasCert(a_1, cert, sys) \ \Rightarrow$$
$$\neg \ Pre(sys, \texttt{receiveIntent}(i, ic, a_1))$$

Demostración: Suponiendo que *ic* está corriendo, que *c* no es un *content provider*, que *c* pertenece a la aplicación *a* y c_1 a a_1, que $\langle ic, i\rangle$ pertenece al conjunto de *intents* enviados, que *i* es de tipo ***broadcast*** y que está protegido por algún permiso de tipo *signature* o *signature or system* y que además el certificado de *a* es distinto al certificado de a_1, se debe probar que no se cumple $Pre(sys, \texttt{receiveIntent}(i, ic, a_1))$.
Esto es equivalente a probar que si a las hipótesis anteriormente mencionadas se agrega que se cumple la precondición de la operación `receiveIntent`, se puede probar un absurdo. Se supone entonces $Pre(sys, \texttt{receiveIntent}(i, ic, a_1))$.
Como se supuso que se cumple $Pre(sys, \texttt{receiveIntent}(i, ic, a_1))$, en particular se cumple (ver Figura 3.7): $isIntBReceiver(i) \wedge i.brperm <> None \Rightarrow preReceiveIntentBroadcast(i, a, a_1)(\circledast)$
Además por hipótesis se cumplen: $isIntBReceiver(i)$ y $\exists(p : Perm), (\ p.pl = signature \ \vee \ p.pl = signatureOrSys) \ \wedge i.brperm = Some \ p \ (\circledast\circledast)$.
De $\circledast$ y $\circledast\circledast$ se puede afirmar que se cumple $preReceiveIntentBroadcast(i, a, a_1)$. Esto quiere decir que o bien *a* cuenta con los permisos necesarios para recibir el *intent i* o bien el permiso contenido en el *intent i* es de tipo *signature* o

signature or system y las aplicaciones a_2 y a_1 están firmadas con el mismo certificado. En ambos casos se debe probar que se llega a un absurdo.

- **Caso 1 a cuenta con los permisos necesarios para recibir el *intent* i:** si a cuenta con los permisos necesarios para recibir el *intent* i, entonces dichos permisos no podrán ser ni de tipo *signature* ni de tipo *signature or system*, como se supuso lo contrario se llega al absurdo.

- **Caso 2 el permiso contenido en el *intent* i es de tipo signature o signature or system y las aplicaciones a_2 y a_1 están firmadas con el mismo certificado:** en primer lugar a_2 surge al asumir la precondición de la operación `receiveIntent` ya que esta exige que exista una aplicación que cumpla determinadas condiciones, en particular que haya un componente c_2 que pertenezca a dicha aplicación y que esté asociado a la instancia ic 3.7. De esto se desprende que $a_2 = a$ porque una instancia en ejecución no puede estar asociada a más de un componente e ic está asociada a c y además un componente no puede pertenecer a más de una aplicación y c pertenece a a. Una vez aclarado esto, en este caso el absurdo se prueba con la igualdad de los certificados de a_1 y a.

4.2.2. *Intent spoofing*

Este ataque consiste en tomar ventaja de un *bug*, una falla de diseño o de configuración para acceder a los recursos de una aplicación que normalmente están protegidos. Como se mencionó en el capítulo 2, que una aplicación pueda ser iniciada por terceros depende del atributo `exported` y de la existencia de elementos `<intent-filter>` en el *manifest*. Las malas configuraciones de las aplicaciones se dan generalmente cuando no se explicita el atributo `exported` pretendiendo que no se permitan invocaciones externas a la aplicación, pero sí se usan elementos de tipo `<intent-filter>` haciendo que el valor por defecto del atributo `exported` sea verdadero. Si una aplicación no fue pensada para poder ser iniciada por terceros pero fue mal configurada puede ser víctima del ataque *intent spoofing* [54].

Una forma sencilla de corroborar que una aplicación no es vulnerable a este tipo de ataque es chequeando de forma estática su *manifest*. De esta forma si el atributo `exported` de una aplicación es falso o si el atributo `exported` no está presente y no se declara ningún elemento de tipo `<intent-filter>` en su

manifest, entonces la aplicación no podrá ser iniciada por terceros. En estas condiciones se evita el ataque *intent spoofing*.

Para la definición de esta propiedad se utilizaron los elementos $c\ c_1 : Cmp$, $a\ a_1 : idApp$, $ic : iCmp$, $i : Intent$ y un sistema $sys = (s, e)$ válido, donde $s = (apps, perms, iCs, delPP, delTP, vals, ints)$ y $e = (mfsts, certs, defPerms, img)$.

Lema 19 *Si un componente c_1 no puede ser iniciado por terceros, entonces la aplicación que lo contiene no podrá recibir un intent dirigido a c_1.*

$$inApp(c, a, sys) \ \wedge inApp(c_1, a_1, sys) \ \wedge a \neq a_1 \ \wedge$$
$$\neg canBeStarted(c_1) \ \wedge \ running(ic, c, sys) \ \wedge \neg isCProvider(c) \ \wedge$$
$$i.cmpName = getCmpId(c_1) \ \Rightarrow$$
$$\neg \ Pre(sys, \texttt{receiveIntent}(i, ic, a_1))$$

Demostración: Suponiendo que c pertenece a la aplicación a y c_1 a a_1, que $a \neq a_1$, que c_1 no puede ser iniciado por terceros, que ic está corriendo, que c no es un *content provider* y que el *intent* i está dirigido a c_1 se debe probar que no se cumple $\neg\ Pre(sys, \texttt{receiveIntent}(i, ic, a_1))$.
Esto es equivalente a probar que si a las hipótesis anteriormente mencionadas se agrega que se cumple la precondición de la operación `receiveIntent`, se puede probar un absurdo. Se supone entonces $Pre(sys, \texttt{receiveIntent}(i, ic, a_1))$.
Por hipótesis el *intent* i está dirigido al componente c_1. Como se supuso que se cumple $Pre(sys, \texttt{receiveIntent}(i, ic, a_1))$, entonces en particular se cumple que c_1 puede ser iniciado por el componente que envía el *intent*, por lo tanto c_1 puede ser iniciado por terceros. Esto es absurdo, ya que por hipótesis c_1 no puede ser iniciado por terceros.
De esta forma se probó que al suponer $Pre(sys, \texttt{receiveIntent}(i, ic, a_1))$ se llega a un absurdo.

Capítulo 5

Observaciones a la Seguridad de Android Marshmallow

Como se mencionó en el capítulo 2, a partir de Android Marshmallow los permisos se otorgan en tiempo de ejecución. Si bien este es el cambio más grande en relación al modelo de seguridad, hay otras modificaciones menores que en conjunto pueden llevar a fallas en la seguridad.

En este capítulo se indican algunos de dichos cambios y se describen los problemas de seguridad que podrían llegar a generarse como consecuencia de ellos.

5.1. Permisos Delegados

Como se mencionó en la sección 2.2.4 hay dos formas mediante las que una aplicación puede realizar acciones para las que originalmente no estaba autorizada: *pending intents* y URI *permissions*. En ambos casos el tipo de problema que se presenta es el mismo: si una aplicación delega un permiso de tipo *dangerous* previamente otorgado por el usuario y éste luego se lo revoca, el permiso delegado continua vigente y ni la aplicación ni el usuario pueden revocarlo. Este problema se ilustra en la Figura 5.1

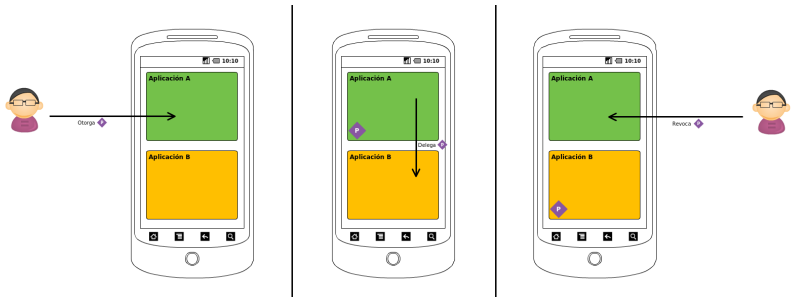

Figura 5.1: Permisos delegados

Pending Intents

Tanto en Android 5.0 como en Android 6.0, cuando una aplicación manda un *pending intent* otorga el derecho de realizar la acción establecida en el mismo con sus permisos e identidad. Un *pending intent* se define como una referencia a los datos que se usaron para crearlo, por lo tanto se guarda el estado de la aplicación del momento en que lo generó [22]. En Android 5.0 el usuario no puede revocar permisos de ningún tipo, por lo que los permisos otorgados a una aplicación no varían durante todo el tiempo que la misma esté instalada. En Android 6.0 los permisos otorgados a una aplicación pueden variar dependiendo de la decisión del usuario.

Al introducir la revocación de permisos en tiempo de ejecución por parte del usuario surge el siguiente posible escenario: una aplicación manda un *pending intent* para realizar una acción protegida por un determinado permiso y posteriormente dicho permiso le es revocado; la aplicación receptora del *pending intent* puede hacer uso del permiso que la aplicación emisora ya no tiene. Más aún, si una aplicación se manda un *pending intent* a si misma se asegura de retener un permiso por tiempo indeterminado.

Supongamos una aplicación para mandar mensajes de texto en ciertos horarios indicados por el usuario y que para resolverlo envía un *pending intent* a alguno de sus componentes para delegarle la tarea. En el momento en que el usuario indica un horario, se genera un *pending intent*. Si la aplicación tiene otorgado el permiso de enviar mensajes, el *pending intent* también lo tendrá. Aunque el usuario le revoque a la aplicación dicho permiso previo a la hora preestablecida, el envío se hará de cualquier manera ya que el *pending intent* sí tiene autorización. Esto puede permitir ataques por parte de aplicaciones maliciosas que “programen“

el envío de mensajes a números de costos elevados mediante *pending intents* y puedan seguir mandándolos mientras la propia aplicación no cancele el *pending intent* correspondiente.

URI *Permissions*

Las aplicaciones pueden delegar permisos sobre recursos de *content providers* a otras aplicaciones, para que se puedan realizar determinadas acciones. El funcionamiento de la delegación de permisos en Android 5.0 es el mismo que en Android 6.0. Cuando una aplicación requiere que una tercera realice una acción sobre un recurso que a priori podría no tener autorización para acceder, le delega el permiso correspondiente para que pueda llevar a cabo la tarea.

Si estos permisos se delegan mediante el envío de intents, entonces no hay ninguna diferencia entre el comportamiento de las aplicaciones entre Android 5.0 y 6.0: la aplicación receptora podrá hacer uso de los permisos delegados mientras esté en ejecución.

Si los permisos se delegan mediante el uso de la operación `grantUriPermission`, solo podrán ser revocados mediante la operación `revokeUriPermission`. Para poder ejecutar la segunda operación, la aplicación tiene que tener permisos otorgados o delegados sobre la acción que quiere revocar. Al igual que en el caso anterior, la principal diferencia entre Android 5.0 es que en Android 5.0 el conjunto de permisos otorgados a una aplicación no cambia, mientras que en Android 6.0 depende de la decisión del usuario.

El problema que se presenta es que si una aplicación delega un permiso otorgado (que el usuario aceptó en algún punto) mediante la operación `grantUriPermission` y luego el usuario revoca dicho permiso, la aplicación receptora va a adquirir el permiso de forma potencialmente permanente. En la documentación oficial [24] no se especifica qué sucede en estos casos con los permisos delegados, por lo que el problema mencionado anteriormente se identifica como una posible falla de seguridad.

Supongamos instaladas una aplicación de correo electrónico que no tiene acceso a la galería de imágenes y una aplicación editora de fotografías que no tiene acceso al correo el electrónico, pero permite mandar las imágenes creadas por mail mediante el uso de la operación `grantUriPermission`. Cuando el usuario solicita el envío de una o varias fotos a través de la aplicación de correo electrónico, se le otorga a ésta última permiso de acceso a los recursos correspondientes. Este

permiso permanecerá vigente hasta que alguna aplicación autorizada ejecute la operación `revokeUriPermission`. Si el usuario le revocara a la aplicación editora de imágenes el permiso de acceso a la galería, ésta no podrá ejecutar la operación `revokeUriPermission` sobre el permiso delegado. Como consecuencia, no se puede garantizar que la aplicación de correo electrónico deje de tener acceso a los recursos seleccionados en algún momento.

5.2. Grupos de Permisos

En Android, todos los permisos de nivel *dangerous* se estructuran en grupos de permisos. El comportamiento del sistema cuando una aplicación requiere el uso de un permiso de nivel *dangerous* en su *manifest*, dependerá de las versiones de Android y del SDK existentes.

Si el dispositivo utiliza la versión 6.0 de Android y 23 o más del SDK, entonces el comportamiento al realizar una funcionalidad es el siguiente [29]:

- Si la funcionalidad requiere un permiso de nivel *dangerous* listado en su *manifest*, y no le fue otorgado previamente ningún otro permiso del grupo de permisos, entonces se le pregunta al usuario si desea otorgar permiso al grupo que se quiere acceder. Por ejemplo si una aplicación quiere mandar un mensaje de texto, deberá pedir el permiso `SEND_SMS`. En este caso el sistema solo le informará al usuario que la aplicación requiere acceso los mensajes. Si el usuario otorga el permiso, el sistema le da a la aplicación el permiso requerido.

- Si la funcionalidad requiere un permiso de nivel *dangerous* listado en su *manifest* y la aplicación ya tiene algún permiso de nivel *dangerous* que pertenece al mismo grupo de permisos, entonces el permiso se otorga sin ninguna interacción con el usuario. Por ejemplo si a la aplicación le fue otorgado el permiso `SEND_SMS` y requiere el permiso `READ_SMS`, el mismo será otorgado de forma automática.

Si el dispositivo utiliza la versión de Android 5.1 o una menor o la versión 22 del SDK o una menor el sistema le pide al usuario que otorgue todos los permisos de nivel *dangerous* al momento de instalar la aplicación [29]. En este punto el sistema también le informa al usuario que la aplicación requiere acceso al grupo de permiso.

La principal diferencia entre la versión 6.0 de Android y las anteriores se da cuando el usuario quiere corroborar qué permisos le fueron otorgados a una aplicación en particular.

Por ejemplo, supongamos el caso de la aplicación *Facebook* que requiere entre otros permiso para acceder a la mensajería. En las versiones anteriores a la 6.0 al momento de instalación el usuario deberá conceder permiso para acceder al grupo `SMS`, mientras que en la versión 6.0 cuando se quiera realizar una acción que involucre el acceso a la mensajería el usuario deberá otorgar el permiso para acceder al grupo `SMS`. En versiones anteriores a las 6.0 si el usuario consulta los permisos que utiliza la aplicación *Facebook* verá una imagen similar a la de la Figura 5.2. En donde claramente se ve que la aplicación solo tiene permisos para leer los mensajes. Si se realiza lo mismo en un dispositivo con la versión 6.0 de Android, ser verá una imagen similar a la de la Figura 5.3 [1]. En este caso no se especifica qué tipo de permiso se le otorgó a la aplicación, sino que sigue diciendo que la aplicación tiene acceso al grupo `SMS`.

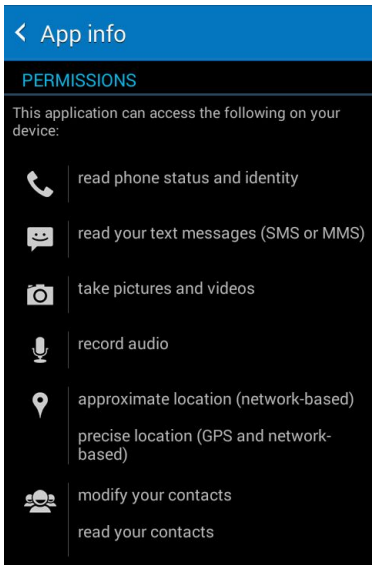

Figura 5.2: Android Lollipop Acceso SMS

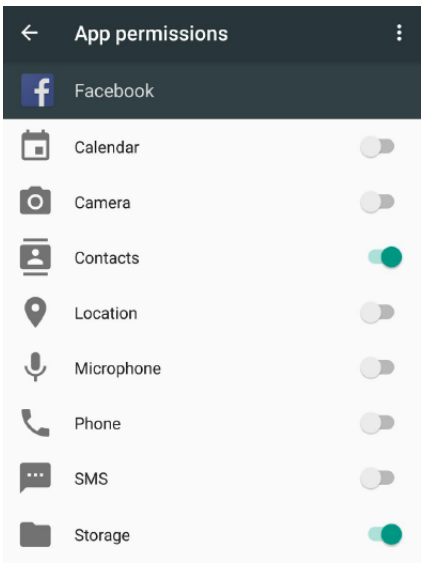

Figura 5.3: Android Marshmallow Acceso SMS

En un escenario donde se instala una aplicación maliciosa que se encargue de

[1] Imagen tomada de `https://motorola-global-portal.custhelp.com/app/answers/prod_answer_detail/a_id/108954/p/1449,9582`

mandar al usuario una alerta especial cada vez que reciba un mensaje de texto, es razonable que dicha aplicación tenga el permiso `RECEIVE_SMS`. Sin embargo, al otorgar este permiso el usuario está implícitamente otorgando, entre otros, el permiso `READ_SMS` que pertenece al mismo grupo de permisos. En Android 6.0 el usuario nunca se entera que otorgó este permiso y la aplicación maliciosa podría estar accediendo a los mensajes de texto y por ende a información sensible como credenciales bancarias que se hayan enviado por mensaje de texto.

5.3. Acceso a Internet Automático

Como se mencionó anteriormente, los únicos permisos que requieren aprobación por parte del usuario, son los de tipo *dangerous*. Uno de los cambios más importantes de la versión 6.0 de Android es que el permiso de acceso a Internet ya no es de tipo *dangerous* sino que pasó a ser de tipo *normal* [23]. Esto implica que todas las aplicaciones podrían tener acceso a Internet si así lo quisieran.

Una de las principales razones de este cambio, es que una gran parte de los desarrolladores financian sus aplicaciones mediante publicidad [39]. Si los usuarios son capaces de desactivar el permiso de acceso a Internet, entonces estas aplicaciones no podrían seguir financiándose y la oferta de aplicaciones gratuitas disminuiría significativamente.

Desde Google el cambio se justifica diciendo que el acceso a Internet por si solo no genera exposición de información sensible ni ningún otro problema de seguridad [39]. Pero las aplicaciones que por las funcionalidades que realizan necesitan acceder a información sensible pero no a Internet, podrían solicitar el permiso de acceso a Internet sin que el usuario se entere ni pueda hacer nada para evitarlo.

Este es un cambio que se introdujo en Android 6.0, pero afecta a todas las versiones. Cuando se descarga una aplicación en una versión de Android menor a la 6.0 todos los permisos de nivel *dangerous* deben otorgarse al momento de la instalación. El permiso de acceso a Internet, al ser ahora de nivel *normal* ya no aparece entre los permisos solicitados por las aplicaciones.

En estas condiciones, es muy fácil disfrazar aplicaciones maliciosas de inofensivas. Hasta antes de realizar este cambio, una aplicación editora de imágenes que solicita acceso a la galería de fotos pero no a Internet parecía segura, a partir del cambio introducido en Android 6.0 el usuario no puede saber si esa

aplicación además está accediendo a Internet.

En la Figura 5.4 se puede observar una aplicación linterna disponible en *Google Play* con aproximadamente 10 millones de descargas, que en la versión 4.4.2 de Android el único permiso que solicita al momento de la instalación es el acceso a cámara. Esto es razonable debido a que las aplicaciones linterna utilizan el flash de la cámara. Sin embargo, en la Figura 5.5 se muestra el *manifest* de la aplicación, donde en el cuadro naranja se puede ver que la misma declara que utiliza el permiso de acceso a Internet. En esta aplicación en particular, el usuario no puede inferir que se va a solicitar este permiso.

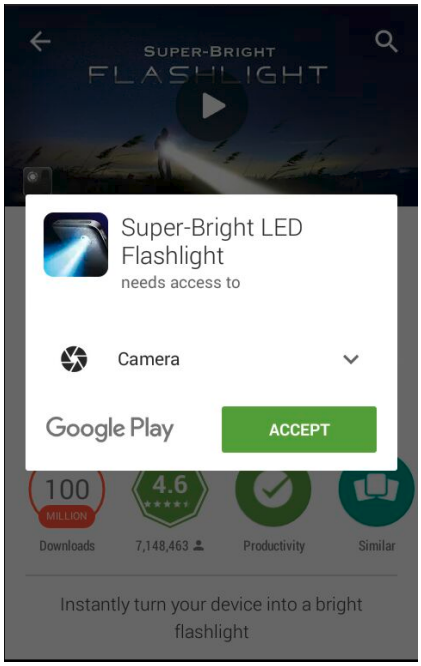

Figura 5.4: Aplicación Linterna Permisos

Figura 5.5: Aplicación Linterna *Manifest*

En la Figura 5.6 se muestra el juego *Unblock Me* disponible en *Google Play* con aproximadamente 50 millones de descargas. En este caso a diferencia del anterior, la aplicación solicita permiso para realizar compras en la aplicación por lo que el usuario podría inferir que también se solicita permiso de acceso a Internet. En la Figura 5.7 se observa el *manifest* de la aplicación *Unblock Me* donde en el cuadro naranja se puede ver como efectivamente se solicita permiso

de acceso a Internet.

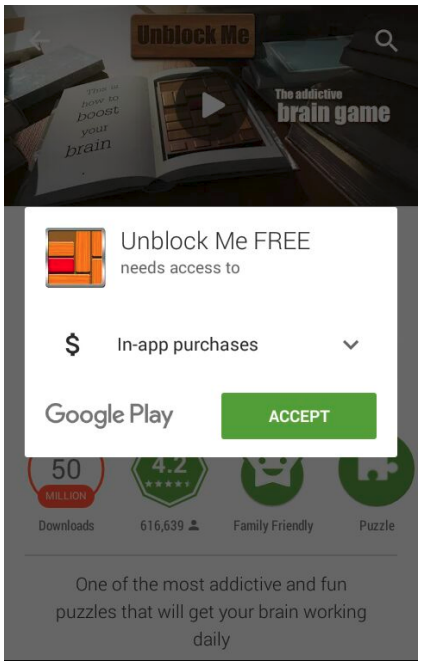

Figura 5.6: Unblock Me Permisos

Figura 5.7: Unblock Me *Manifest*

5.4. Adaptación de Aplicaciones Maliciosas

Uno de los objetivos principales del cambio en el sistema de permisos es que el usuario tenga más conciencia de los permisos que cada aplicación requiere. Si el usuario recibe una notificación con todos los permisos que una aplicación requiere y en el momento en que quiere instalarla, es factible que los acepte todos sin cuestionarse por qué se solicitan. En cambio, si los permisos se le presentan de forma individual cuando quiere realizar determinada acción el usuario puede evaluar si el permiso que se solicita suena necesario para realizar la acción.

Si bien es una iniciativa razonable, las condiciones bajo las cuales los permisos se solicitan al usuario en tiempo de ejecución son totalmente manejables por los desarrolladores. Si el dispositivo corre una versión de Android menor o igual a la 5.1 o si la versión del SDK es menor o igual a 22, los permisos siguen siendo solicitados en tiempo de instalación [24]. El desarrollador no puede manejar la

versión de Android que va a correr en el dispositivo que descargue su aplicación, pero sí puede fijar el elemento `<uses-sdk>` en el *manifest* de la misma. Este atributo indica el nivel del SDK sobre el que fue testeada la aplicación y de ser menor que el existente en el dispositivo, habilita un modo de compatibilidad que permite que las aplicaciones funcionen con normalidad. De todas maneras, el desarrollador no puede evitar que el usuario manualmente le revoque permisos a su aplicación en tiempo de ejecución.

A aproximadamente un año de liberada la versión 6.0 de Android existen aplicaciones maliciosas que ya fueron adaptadas para sortear las dificultades impuestas por el nuevo sistema de permisos: fijan el elemento `<uses-skd>` a un valor menor o igual a 22 y chequean antes de realizar cada operación que los permisos sigan estando vigentes ante una posible revocación explícita por parte del usuario.

Un ejemplo de esto es *Bankosy* [55] que es un trojano financiero que permite hacer llamadas, interceptar y borrar mensajes de texto, borrar datos y habilitar o deshabilitar el modo silencio. El ataque se basa en robar credenciales y luego realizar transacciones fraudulentas en las cuentas de la víctima.

5.5. Permisos Faltantes

Como se mencionó anteriormente, la versión 6.0 de Android permite que el usuario le quite un permiso a una aplicación en cualquier momento. Las aplicaciones deben poder continuar ejecutándose aún cuando no se les hayan sido otorgados todos los permisos que requieren. Los desarrolladores deben ser conscientes de esto y hacer los controles correspondientes en caso que sea necesario. Aún en aplicaciones que indiquen una versión del SDK menor a 23, el usuario podrá revocar los permisos que quiera y es más factible que se generen errores en la ejecución.

Aunque esto no significa un riesgo de filtrado de información, la experiencia del usuario no va a ser del todo satisfactoria y podría llevar a la desinstalación de las aplicaciones que cumplan estas características.

Capítulo 6

Trabajos Relacionados

En este capitulo se describen otros trabajos que estudian el modelo de seguridad de Android. Los trabajos incluidos o bien detectan y corrigen fallas puntuales sobre la seguridad, o bien aplican métodos formales para probar ciertas propiedades sobre el modelo de seguridad de la plataforma.

En relación a la seguridad de Android, Armando *et al.* [36] estudian problemas de seguridad generados por la interacción entre la capa aplicaciones y el kernel Linux. Su trabajo concluye que el *Android Security Framework* no discrimina entre los distintos invocadores a funcionalidades del kernel Linux, permitiendo que cualquier aplicación instalada en el dispositivo tenga acceso al mismo. Esto habilita a que aplicaciones maliciosas puedan acceder a información sensible del usuario.

El trabajo de Sbîrlea *et al.* [54] desarrolla un método de inspección estática del código de las aplicaciones en conjunto con el flujo de interacciones que genera una invocación a una API del sistema, detectando dónde se generan fallas de seguridad en algún tipo de comunicación entre aplicaciones. Este análisis permite identificar que casi un 1 % de las aplicaciones estudiadas vulneran información sensible del usuario. Por ejemplo la aplicación *Adobe Photoshop Express*, que cuenta con aproximadamente 50 millones de descargas, compromete la información de los contactos del usuario.

En la misma línea, el trabajo de Chin *et al.* [38] extiende el tipo de comunicación para el cual detectar vulnerabilidades. Además de detectar posibles violaciones a la información sensible del usuario, este trabajo reconoce y advierte

sobre formas de comunicación inter aplicaciones que pueden resultar en una falla de seguridad.

Cozzette *et al.* [40] proponen un manejo de Intents alternativo para evitar posibles filtrados de información sensible para el usuario y fallas en las aplicaciones por recepciones inesperadas de intents implícitos. Modifican la resolución de estos intents forzando a que, en caso de ser posible, se considere a la aplicación que envía el intent como receptora. Además endurecen los controles requeridos para que terceros puedan invocar a una aplicación, protegiendo a las aplicaciones de intents peligrosos.

Mediante inspección del código fuente y del flujo que genera una invocación a una API del sistema la investigación de Felt *et al.* [43] advierte que el ataque denominado *privilege escalation* depende en gran medida de una mala configuración de las aplicaciones por parte de los desarrolladores. Este trabajo no solo permite la detección de aplicaciones vulnerables a este tipo de ataque sino que además provee un mecanismo de prevención del mismo.

También trabajando sobre el ataque *privilege escalation*, Davi *et al.* [41] logran efectuar un ataque sobre una aplicación con algún *bug* menor que permite escalar los permisos de una aplicación maliciosa.

Felt *et al.* [42] construyen una herramienta de análisis estático que determina el número máximo de permisos que una aplicación necesita para funcionar correctamente. Este número en comparación con la cantidad de permisos requeridos en el *manifest* establece cuándo las aplicaciones solicitan permisos que nunca utilizan. Ésto habilita a que una aplicación acceda a información teóricamente fuera de su alcance.

En relación a trabajos que aplican métodos formales para el estudio del modelo de seguridad de Android, Armando *et al.* [35] y Bugliesi *et al.* [37] formalizan elementos básicos del sistema Android como aplicaciones, componentes, permisos, *manifest* y algunas operaciones importantes para definir un sistema de tipos y efectos. La diferencia principal entre estos dos trabajos es que mientras Armando *et al.* [35] demuestran que un sistema de tipos y efectos es apropiado para modelar formalmente el sistema Android, Bugliesi *et al.* [37] se basan en los trabajos de Davi *et al.* [41] para verificar formalmente si una aplicación es o no vulnerable al ataque *privilege escalation*.

Fragkaki *et al.* [44] proponen un sistema formal de transiciones abstracto sobre un modelo de permisos que es instanciado para representar a Android.

Se contemplan solamente algunas operaciones como la revocación y delegación de permisos y el efecto que el reinicio del dispositivo tiene sobre los mismos. Enuncian y demuestran propiedades de seguridad que permiten comprender y ayudar a encontrar fallas y malos comportamientos del modelo de seguridad de la plataforma.

Shin *et al.* [56] proponen un modelo formal del sistema de permisos de Android desarrollado en
`Coq`. Este trabajo presenta una simplificación del sistema Android, representando todos los componentes mediante un tipo único. Como consecuencia no es posible probar propiedades sobre *content providers*, ni trabajar con la interacción entre componentes mediante *intents*.

Agustín Romano [53] propone un modelo formal abstracto del sistema de seguridad de Android basado en una máquina de estados. Este modelo contempla todos los tipos de componentes y un conjunto de operaciones significativo para el estudio en profundidad del sistema operativo. Desarrolla pruebas sobre los recursos de los content providers que permiten mejorar la comprensión del funcionamiento de la plataforma y no habían sido estudiados con anterioridad.

Este trabajo es una extensión del realizado por Agustín Romano [53]. Presenta una especificación más cercana al sistema Android, modificando la forma de representarlo. Para ello, el modelo distingue entre una parte estática (que solo es modificada por las operaciones de instalación y desinstalación) y una parte dinámica (modificada por todas las operaciones). Se evita el uso de predicados para representar a los elementos presentes en el sistema, cambiándolos por listas y funciones, con el objetivo de poder representar el sistema mediante estructuras de datos (acercando la definición a la plataforma real) y no predicar sobre componentes abstractos del sistema. Para que cada componente contenga las propiedades que le corresponden del *manifest*, como por ejemplo el atributo *exported*, se desarma la estructura existente en el trabajo original. Una vez adaptado el modelo, se demuestra que el cambió en la representación no tiene impacto sobre las propiedades probadas y que todas continúan siendo válidas. Finalmente, se expande el modelo permitiendo representar el envío, resolución y recepción de intents y el manejo de permisos en tiempo de ejecución. A partir de esta formalización es posible realizar pruebas relacionadas a ataques conocidos a la plataforma que surgen del intercambio de intents.s.

Capítulo 7

Conclusiones y Trabajos Futuros

Aunque en las nuevas versiones de Android se puede observar un esfuerzo importante por mejorar y aumentar los mecanismos de seguridad del sistema, la documentación y especificación oficial no es exhaustiva, de forma involuntaria, esto puede llevar a que los desarrolladores aplicaciones que no sean seguras. Como consecuencia surgen nuevos problemas de seguridad que comprometen la información sensible de los usuarios. Si bien existen muchos estudios que abordan dicha problemática y ofrecen mecanismos de detección y resolución de diversos problemas de seguridad, en su mayoría son empíricos y están probados en un número limitado de aplicaciones.

Este trabajo se focaliza en dos aspectos principales. En primer lugar, realiza una descripción informal sobre el modelo de seguridad de Android que permite una mejor comprensión del funcionamiento del sistema. En segundo lugar, contribuye al desarrollo de una especificación formal completa del modelo de seguridad de Android.

El objetivo principal del trabajo se centró en la transición de un modelo de seguridad muy abstracto a un modelo funcional que se corresponde con la plataforma real, incluyendo aspectos clave del funcionamiento de Android que impactan directamente en su modelo de seguridad, como por ejemplo el manejo de intents implícitos. Además se adaptaron demostraciones formales a propiedades enunciadas en trabajos previos (mencionadas en 4.1), concluyendo

Modelo formal	1116
Invarianza de validez de sistema	5578
Propiedades de seguridad	1216
Análisis de Debilidades	128
Total	8038

Figura 7.1: Líneas de código de la especificación en `Coq`

que todas ellas son válidas en el modelo aquí construido.

Finalmente, se estudiaron ataques existentes relacionados con el envío y recepción de intents y se enunciaron y demostraron propiedades que muestran formalmente que existen escenarios en los que un sistema Android es inmune a algunos ataques como *intent spoofing* y *eavesdropping*.

Es de interés remarcar que si bien el sistema Android es amplio y complejo, el modelo formal del mismo en conjunto con las demostraciones de las propiedades de seguridad estudiadas, pudieron ser especificadas mediante el uso de `Coq` en tan solo 8038 líneas de código distribuidas como se muestra en la Figura 7.1.

Por último, se detectaron y estudiaron informalmente algunos problemas de seguridad introducidos en Android 6.0 al adaptar el sistema de permisos de Android 5.0 para soportar la solicitud de permisos en tiempo de ejecución. Estas vulnerabilidades fueron descriptas en el capítulo 5.

Como Android 6.0 fue liberado en Octubre de 2015 y aún no abarca a un porcentaje significativo de los dispositivos Android, no existen demasiados estudios relacionados con el nuevo sistema de permisos. De todos modos, este fue uno de los cambios más importantes que se le realizó al modelo de seguridad de Android en los últimos tiempos, por lo que su análisis resulta de gran interés.

Si bien el alcance de este proyecto no abarcó la demostración formal de las vulnerabilidades encontradas, permite abrir nuevas líneas de investigación relacionadas con el sistema de permisos de Android 6.0.

Aunque el modelo desarrollado brinda las bases necesarias para hacer un análisis significativo sobre el modelo de seguridad de Android, hay algunos aspectos que no fueron incluidos dentro del alcance de este proyecto.Estos aspectos podrían ser de interés no únicamente para encontrar potenciales problemas de seguridad de la plataforma, sino que también para lograr un modelo formal que abarque todos los conceptos y elementos presentes en un sistema Android. En

este sentido, se destacan las siguientes líneas de trabajo por las que se podría expandir el modelo desarrollado:

- **Pending intents:** los pending intents se utilizan para la delegación temporal de permisos. Sería útil considerarlos en una posible extensión al modelo que se presenta en este trabajo. También es importante formalizar y demostrar que las vulnerabilidades encontradas en el capítulo 5 relacionadas con el uso de pending intents; son realmente una amenaza para el modelo de seguridad de Android.

- **Servicios ligados:** resultaría interesante agregar al modelo la operación `boundService()` y evaluar si existe algún ataque específico relacionado con ella. Por ejemplo, estudiar qué ocurre si una aplicación tiene permisos para iniciar un servicio, lo hace mediante esta operación y posteriormente el usuario revoca el permiso que permite iniciar el servicio.

- **Actualización de aplicaciones:** si bien no tiene gran impacto sobre la seguridad de Android, incluir esta funcionalidad en el modelo desarrollado contribuiría a la creación de un modelo más completo y que comprenda todos los aspectos funcionales de la plataforma.

- **Cambios introducidos por Android 7.0:** debería estudiarse en particular la manera de compartir archivos entre aplicaciones. A partir de la versión 7.0 de Android no se puede exponer un archivo mediante su URI fuera de la aplicación. En caso de que se mande un intent que contenga la URI de un archivo, la aplicación falla y lanza una excepción de seguridad. Dado que esta modificación al sistema cambia el modo de delegar permisos de forma permanente, se debería agregar al modelo y estudiar los problemas de seguridad involucrados.

- **Manejo de errores:** para lograr un modelo más completo y robusto, con mayor similitud con el sistema Android.

Bibliografía

[1] Android Developers. *<action>*. Disponible en: `http://developer.android.com/intl/es/guide/topics/manifest/action-element.html`. Último acceso: Agosto 2016.

[2] Android Developers. *Actividades*. Disponible en: `https://developer.android.com/guide/components/activities.html`. Último acceso: Agosto 2016.

[3] Android Developers. *<activity>*. Disponible en: `https://developer.android.com/guide/topics/manifest/activity-element.html`. Último acceso: Agosto 2016.

[4] Android Developers. *Android Fundamentals*. Disponible en: `https://developer.android.com/guide/components/fundamentals.html?hl=es`. Último acceso: Agosto 2016.

[5] Android Developers. *Android Marshmallow*. Disponible en: `https://www.android.com/versions/nougat-7-0/`. Último acceso: Agosto 2016.

[6] Android Developers. *Android Nougat*. Disponible en: `https://www.android.com/versions/marshmallow-6-0/`. Último acceso: Agosto 2016.

[7] Android Developers. *android : exported*. Disponible en: `https://developer.android.com/guide/topics/manifest/activity-element.html#exported`. Último acceso: Agosto 2016.

[8] Android Developers. *android : permission*. Disponible en: `https://developer.android.com/guide/topics/manifest/application-element.html#prmsn`. Último acceso: Agosto 2016.

[9] Android Developers. <*application*>. Disponible en: `https://developer.android.com/guide/topics/manifest/application-element.html`. Último acceso: Agosto 2016.

[10] Android Developers. <*category*>. Disponible en: `http://developer.android.com/intl/es/guide/topics/manifest/category-element.html`. Último acceso: Agosto 2016.

[11] Android Developers. *Content Providers*. Disponible en: `https://developer.android.com/guide/topics/providers/content-providers.html`. Último acceso: Agosto 2016.

[12] Android Developers. <*data*>. Disponible en: `http://developer.android.com/intl/es/guide/topics/manifest/data-element.html`. Último acceso: Agosto 2016.

[13] Android Developers. *Documentación Oficial*. Disponible en: `https://developer.android.com/index.html`. Último acceso: Agosto 2016.

[14] Android Developers. <*grant-uri-permission*>. Disponible en: `http://developer.android.com/intl/es/guide/topics/manifest/grant-uri-permission-element.html`. Último acceso: Agosto 2016.

[15] Android Developers. *grantUriPermission*. Disponible en: `http://developer.android.com/intl/es/guide/topics/manifest/provider-element.html#gprmsn`. Último acceso: Agosto 2016.

[16] Android Developers. <*intent-filter*>. Disponible en: `https://developer.android.com/guide/topics/manifest/intent-filter-element.html`. Último acceso: Agosto 2016.

[17] Android Developers. *Intent Filters*. Disponible en: `https://developer.android.com/guide/components/intents-filters.html`. Último acceso: Agosto 2016.

[18] Android Developers. *IntentObject*. Disponible en: `https://developer.android.com/reference/android/content/Intent.html`. Último acceso: Agosto 2016.

[19] Android Developers. *Manifesto*. Disponible en: `https://developer.android.com/guide/topics/manifest/manifest-intro.html`. Último acceso: Agosto 2016.

[20] Android Developers. *Nivel de Permisos*. Disponible en: `https://developer.android.com/guide/topics/manifest/permission-element.html?hl=es#plevel`. Último acceso: Agosto 2016.

[21] Android Developers. *Pending Intent Object*. Disponible en: `https://developer.android.com/reference/android/app/PendingIntent.html?hl=es`. Último acceso: Agosto 2016.

[22] Android Developers. *Pending Intents*. Disponible en: `https://developer.android.com/reference/android/app/PendingIntent.html`. Último acceso: Agosto 2016.

[23] Android Developers. *Permiso de Acceso a Internet*. Disponible en: `https://developer.android.com/reference/android/Manifest.permission.html#INTERNET`. Último acceso: Agosto 2016.

[24] Android Developers. *Permisos en Tiempo de Ejecución*. Disponible en: `https://developer.android.com/intl/es/training/permissions/requesting.html`. Último acceso: Agosto 2016.

[25] Android Developers. <*permission*>. Disponible en: `http://developer.android.com/intl/es/guide/topics/manifest/permission-element.html`. Último acceso: Agosto 2016.

[26] Android Developers. <*provider*>. Disponible en: `https://developer.android.com/guide/topics/manifest/provider-element.html`. Último acceso: Agosto 2016.

[27] Android Developers. *readPermission*. Disponible en: `http://developer.android.com/intl/es/guide/topics/manifest/provider-element.html#rprmsn`. Último acceso: Agosto 2016.

[28] Android Developers. <*receiver*>. Disponible en: `https://developer.android.com/guide/topics/manifest/receiver-element.html`. Último acceso: Agosto 2016.

[29] Android Developers. *Seguridad en Android*. Disponible en: `https://developer.android.com/guide/topics/security/permissions.html`. Último acceso: Agosto 2016.

[30] Android Developers. *<service>*. Disponible en: `https://developer.android.com/guide/topics/manifest/service-element.html`. Último acceso: Agosto 2016.

[31] Android Developers. *Servicios*. Disponible en: `https://developer.android.com/guide/components/services.html`. Último acceso: Agosto 2016.

[32] Android Developers. *Sticky Broadcast*. Disponible en: `https://developer.android.com/reference/android/content/Context.html`. Último acceso: Agosto 2016.

[33] Android Developers. *<uses-permission>*. Disponible en: `http://developer.android.com/intl/es/guide/topics/manifest/uses-permission-element.html`. Último acceso: Agosto 2016.

[34] Android Developers. *writePermission*. Disponible en: `http://developer.android.com/intl/es/guide/topics/manifest/provider-element.html#wprmsn`. Último acceso: Agosto 2016.

[35] Alessandro Armando, Gabriele Costa, and Alessio Merlo. Formal modeling and reasoning about the android security framework. In Catuscia Palamidessi and MarkD. Ryan, editors, *TGC'12: 7th International Symposium on Trustworthy Global Computing*, volume 8191 of *LNCS*. Springer Berlin Heidelberg, 2013. URL: `http://www.ai-lab.it/armando/pub/tgc2012.pdf`, `doi:10.1007/978-3-642-41157-1_5`.

[36] Alessandro Armando, Alessio Merlo, and Luca Verderame. An empirical evaluation of the android security framework. In Lech J. Janczewski, Henry B. Wolfe, and Sujeet Shenoi, editors, *SEC*, volume 405 of *IFIP Advances in Information and Communication Technology*, pages 176–189. Springer, 2013. URL: `http://dblp.uni-trier.de/db/conf/sec/sec2013.html#ArmandoMV13`.

[37] Michele Bugliesi, Stefano Calzavara, and Alvise Spanò. Lintent: Towards security type-checking of android applications. In *Proceedings of Formal*

Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013, pages 289–304, Berlin, Heidelberg, 2013. Springer.

[38] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1999995.2000018, doi:10.1145/1999995.2000018.

[39] Cody Toombs. *Acceso a Internet en Android 6.0.* Disponible en: http://www.androidpolice.com/2015/06/06/android-m-will-never-ask-users-for-permission-to-use-the-internet-and-thats-probably-okay/. Último acceso: Agosto 2016.

[40] Adam Cozzette, Kathryn Lingel, Steve Matsumoto, Oliver Ortlieb, Jandria Alexander, Joseph Betser, Luke Florer, Geoff Kuenning, John Nilles, and Peter L. Reiher. Improving the security of android inter-component communication. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Ghent, Belgium, May 27-31, 2013*, pages 808–811, 2013. URL: http://dblp.uni-trier.de/db/conf/im/im2013.html#CozzetteLMOABFKNR13.

[41] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1949317.1949356.

[42] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/2046707.2046779, doi:10.1145/2046707.2046779.

[43] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses.

In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=2028067.2028089.

[44] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. Modeling and enhancing android's permission system. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Proceedings of the 17th European Symposium on Research in Computer Security, ESORICS 2012*, volume 7459 of *LNCS*, pages 1–18. Springer, 2012.

[45] Gartner. *Distribución de SO en dispositivos móviles*. Disponible en: http://www.gartner.com/technology/home.jsp. Último acceso: Agosto 2016.

[46] Google. *Google*. Disponible en: https://www.google.com/about/. Último acceso: Agosto 2016.

[47] Google Play. *Play Store*. Disponible en: https://play.google.com/store?hl=en. Último acceso: Agosto 2016.

[48] Gordon Kelly. *Distribución del malware*. Disponible en: http://www.forbes.com/sites/gordonkelly/2014/03/24/report-97-of-mobile-malware-is-on-android-this-is-the-easy-way-you-stay-safe/#19a0a5707d53. Último acceso: Agosto 2016.

[49] Info Sec. *Eavesdropping*. Disponible en: http://www.infosec.gov.hk/english/promotion/files/Script_Eavesdropping.pdf. Último acceso: Agosto 2016.

[50] Masoumeh Al. Haghighi Mobarhan. Formal Specification of Selected Android Core Applications and Library Functions. Master's thesis, Chalmers University of Technology, University of Gothenburg, Gotemburgo, Suecia, 2011.

[51] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1755688.1755732, doi:10.1145/1755688.1755732.

[52] Open Handset Alliance. *Open Handset Alliance*. Disponible en: `http://www.openhandsetalliance.com/`. Último acceso: Agosto 2016.

[53] A. Romano. Descripción y análisis formal del modelo de seguridad de android. Technical report, Master's thesis, Universidad Nacional de Rosario, 2014. Disponible en: `www.fing.edu.uy/inco/grupos/gsi/index.php?page=proygrado&locale=es`.

[54] D. Sbîrlea, M. G. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar. Automatic detection of inter-application permission leaks in android applications. *IBM J. Res. Dev.*, 57(6):2:10–2:10, November 2013. URL: `http://dx.doi.org/10.1147/JRD.2013.2284403`, `doi:10.1147/JRD.2013.2284403`.

[55] SecurityWeek News. *Bankosy*. Disponible en: `http://www.securityweek.com/android-trojans-exploit-marshmallows-permission-model`. Último acceso: Agosto 2016.

[56] Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka. A formal model to analyze the permission authorization and enforcement in the Android framework. In *2010 IEEE 2nd International Conference on Social Computing*, pages 944–951, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[57] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.4pl3*, 2014. URL: `http://coq.inria.fr`.

Anexo I

Formalización de operaciones

En esta sección se definen las pre y postcondiciones de todas las operaciones no mencionadas en la sección 3.5.

Para poder hacer las especificaciones correspondientes se utilizaron los predicados y las funciones auxiliares que se encuentran en la Tabla I.1.

$removeApp(a, sys)$	Se quita la aplicación a del sistema sys
$removeManifest(a, sys)$	Se quita el *manifest* de la aplicación a del sistema sys
$removeCert(a, sys)$	Se quita el certificado de la aplicación a del sistema sys
$revokeGrantedPerms(a, sys)$	Se quitan los permisos otorgados a la aplicación a del sistema sys
$removeDefPerms(a, sys)$	Se quitan los permisos definidos en la aplicación a del sistema sys
$removeRes(a, sys)$	Se quitan los recursos correspondientes a la aplicación a del sistema sys
$revokeOtherTPerm(a, sys)$	Se quitan los permisos temporales delegados a la aplicación a del sistema sys
$revokePPerms(a, sys)$	Se quitan los permisos permanentes delegados a la aplicación a del sistema sys
	Continúa en la siguiente página

Tabla I.1 – continuación de la página anterior

$stopIns(ic, sys)$	La instancia ic deja de pertenecer al conjunto de instancias en ejecución del sistema.
$revokeTPermsIns(ic, sys)$	Si la instancia ic tiene algún permiso temporal delegado, se revoca.
$hasPermToDelOrRev(pt, c, cp, u, sys)$	Se cumple si el componente c puede realizar la acción pt sobre el recurso u del content provider cp en el sistema sys. Para esto puede tener permisos delegados o permisos otorgados.
$grantPPerm(a, cp, pt, u, sys)$	La aplicación a pasa a tener permisos para realizar la acción pt sobre el recurso u del *content provider cp*.
$revokePPerm(cp, pt, u, sys)$	Se revocan todos los permisos permanentes delegados para realizar la acción pt sobre el recurso u del *content provider cp*.
$revokeTPerm(cp, pt, u, sys)$	Se revocan todos los permisos temporales delegados para realizar la acción pt sobre el recurso u del *content provider cp*.
$permSAC(p, isSystemPerm(p), sac)$	Se cumple si el permiso p predefinido por el sistema es necesario para realizar la llamada *sac*.
$canWrite(c, cp, sys)$	Se cumple si el componente c tiene permisos otorgados de escritura sobre el *content provider cp*.
$writeValue(cp, u, v, sys)$	Retorna todos los valores existentes en sys asignándole el valor v al recurso u del *content provider cp* .

Tabla I.1: Predicados y funciones auxiliares - anexo

Acción `uninstall` a

La aplicación a se desinstala del sistema.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \wedge e = (mfsts, certs, defPerms, img) \wedge \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \wedge \\ a \in aps \wedge \\ \forall (ic : iCmp)(c : Cmp), running(ic, c, sys) \Rightarrow \\ \neg inApp(c, a, sys) \\ \\ removeApp(a, sys) = apps' \wedge \\ removeManifest(a, sys) = mfst' \wedge \\ removeCert(a, sys) = certs' \wedge \\ revokeDefPerms(a, sys) = defPerms' \wedge \\ revokeGrantedPerms(a, sys) = perms' \wedge \\ removeRes(a, sys) = vals' \wedge \\ revokeOtherTPerm(a, sys) = delTP' \wedge \\ revokePPerm(a, sys) = delPP' \wedge \\ sys' = (e', s') \ \wedge e' = (mfst', certs', defPerms', img) \wedge \\ s' = (apps', perms', iCs, delPP', delTP', vals', ints) \end{array}}{sys \xrightarrow{\texttt{uninstall}\ a} sys'}$$

Precondición Para poder desinstalar la aplicación a del sistema, la misma debe estar previamente instalada y no ninguno de sus componentes puede estar en ejecución.

Postcondición La aplicación a se remueve del sistema, junto con su *manifest*, su certificado, sus recursos y los permisos que tenía otorgados. Además se revocan todos los permisos delegados por la aplicación y los definidos estáticamente en la misma.

Figura I.1: Semántica de la operación `uninstall`

Acción `hasPermission` c p

El componente c tiene otorgado el permiso p.

Regla

$$\frac{\begin{array}{c} sys = (e,s) \wedge e = (mfsts, certs, defPerms, img) \wedge \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \wedge \\ \exists (a : idApp)(ic : iCmp), inApp(c, a, sys) \wedge \\ running(ic, c, sys) \end{array} \qquad sys' = (e,s)}{sys \xrightarrow{\texttt{hasPermission}\ c\ p} sys'}$$

Precondición Para corroborar si el componente c tiene el permiso p es necesario que exista una aplicación a instalada que lo contenga y una instancia en ejecución de dicho componente.

Postcondición No se realiza ninguna modificación en sistema.

Figura I.2: Semántica de la operación `hasPermission`

Acción `read` *ic cp u*

La instancia en ejecución *ic* lee la URI *u* del *content provider cp*.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \land e = (mfsts, certs, defPerms, img) \land \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \land \\ \exists(a : idApp), inApp((cmpCP\ cp), a, sys) \land \\ existsRes(cp, u, sys) \land \\ (\exists(a : idApp)(c : Cmp), inApp(c, a, sys) \land \\ running(ic, c, sys) \land \neg isCProvider(c) \land \\ (canRead(c, cp, sys) \lor delPerms(c, cp, u, Read, sys))) \end{array}}{sys \xrightarrow{\texttt{read}\ ic\ cp\ u} sys'}$$

Precondición Para que la instancia *ic* pueda leer el recurso *u* del *content provider cp* debe existir un componente instalado asociado a dicha instancia que pueda leer el recurso o tenga los permisos delegados para hacerlo. Además *cp* debe pertenecer a una aplicación instalada.

Postcondición No se realiza ninguna modificación en sistema.

Figura I.3: Semántica de la operación `read`

Acción `write` *ic cp u v*

La instancia en ejecución *ic* escribe el valor *v* en la URI *u* del *content provider cp*.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \wedge e = (mfsts, certs, defPerms, img) \wedge \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \wedge \\ \exists(a : idApp), inApp((cmpCP\ cp), a, sys) \wedge \\ existsRes(cp, u, sys) \wedge \\ \exists(a : idApp)(c : Cmp), inApp(c, a, sys) \wedge \\ running(ic, c, sys) \wedge \neg isCProvider(c) \wedge \\ (canWrite(c, cp, sys) \vee delPerms(c, cp, u, Write, sys)) \\ \\ writeValue(cp, u, v, sys) = vals' \wedge \\ sys' = (e, s') \ \wedge s' = (apps, perms, iCs, delPP, delTP, vals', ints) \end{array}}{sys \xleftarrow{\texttt{write}\ ic\ cp\ u\ v} sys'}$$

Precondición Para que la instancia *ic* pueda leer el recurso *u* del *content provider cp* debe existir un componente instalado asociado a dicha instancia que pueda leer el recurso o tenga los permisos delegados para hacerlo. Además *cp* debe pertenecer a una aplicación instalada.

Postcondición No se realiza ninguna modificación en sistema.

Figura I.4: Semántica de la operación `write`

Acción `startActivityForResult` i n ic

La instancia ic manda un *intent* i mediante la operación `startActivityForResult`. En natural n representa al código de respuesta.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \land e = (mfsts, certs, defPerms, img) \land \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \land \\ isIntActivity(i) \ \land i.brperm = None \land \\ cmpRunning(ic, sys) \land \ \langle ic, i\rangle \notin ints \\ \\ addIntent(i, ic, None, sys) = ints' \ \land sys' = (e, s') \land \\ s' = (apps, perms, iCs, delPP, delTP, vals, ints') \end{array}}{sys \xrightarrow{\texttt{startActivityForResult}\ i\ n\ ic} sys'}$$

Precondición

Para que la instancia ic pueda mandar el *intent* i mediante la operación `startActivityForResult`, el *intent* i debe ser de tipo actividad y no puede tener declarado ningún permiso en su estructura. Además la instancia en ejecución ic deberá estar corriendo y el par $\langle ic, i\rangle$ no puede estar en el conjunto de *intents* enviados.

Postcondición El *intent* i y la instancia en ejecución ic que lo envió se agregan al conjunto de *intents* enviados del sistema. Además el *intent* no será modificado y deberá seguir sin tener ningún permiso declarado.

Figura I.5: Semántica de la operación `startActivityForResult`

Acción `startService` i ic

La instancia ic manda un *intent* i mediante la operación `startService`.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \land e = (mfsts, certs, defPerms, img) \land \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \land \\ isIntService(i) \ \land i.brperm = None \land \\ cmpRunning(ic, sys) \land \ \langle ic, i\rangle \notin ints \\ \\ addIntent(i, ic, None, sys) = ints' \ \land sys' = (e, s') \land \\ s' = (apps, perms, iCs, delPP, delTP, vals, ints') \end{array}}{sys \xrightarrow{\texttt{startService}\ i\ ic} sys'}$$

Precondición

Para que la instancia ic pueda mandar el *intent* i mediante la operación `startService`, el *intent* i debe ser de tipo servicio y no puede tener declarado ningún permiso en su estructura. Además la instancia en ejecución ic deberá estar corriendo y el par $\langle ic, i\rangle$ no puede estar en el conjunto de *intents* enviados.

Postcondición El *intent* i y la instancia en ejecución ic que lo envió se agregan al conjunto de *intents* enviados del sistema. Además el *intent* no será modificado y deberá seguir sin tener ningún permiso declarado.

Figura I.6: Semántica de la operación `startService`

Acción `sendOrderedBroadcast` i ic p

La instancia ic manda un *intent* i, el *intent* está protegido por el permiso p (de tipo $option\ Perm$) mediante la operación `sendOrderedBroadcast`.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \land e = (mfsts, certs, defPerms, img) \land \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \land \\ isIntBReceiver(i) \ \land i.brperm = None \land \\ cmpRunning(ic, sys) \land \ \langle ic, i\rangle \notin ints \\ \\ addIntent(i, ic, p, sys) = ints' \ \land sys' = (e, s') \land \\ s' = (apps, perms, iCs, delPP, delTP, vals, ints') \end{array}}{sys \xrightarrow{\texttt{sendOrderedBroadcast}\ i\ ic\ p} sys'}$$

Precondición Para que la instancia ic pueda mandar el *intent* i mediante la operación `sendOrderedBroadcast`, el *intent* i debe ser de tipo *broadcast* y no puede tener declarado ningún permiso en su estructura. Además la instancia en ejecución ic deberá estar corriendo y el par $\langle ic, i\rangle$ no puede estar en el conjunto de *intents* enviados.

Postcondición Al *intent* i se le agrega el permiso p especificado en la operación y es agregado al conjunto de *intents* enviados del sistema, junto con la instancia ic que lo envió.

Figura I.7: Semántica de la operación `sendOrderedBroadcast`

Acción `sendStickyBroadcast` i ic

La instancia ic manda un *intent* i mediante la operación `sendStickyBroadcast`.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \land e = (mfsts, certs, defPerms, img) \land \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \land \\ isIntBReceiver(i) \ \land i.brperm = None \land \\ cmpRunning(ic, sys) \land \ \langle ic, i\rangle \notin ints \\ \\ addIntent(i, ic, None, sys) = ints' \ \land sys' = (e, s') \land \\ s' = (apps, perms, iCs, delPP, delTP, vals, ints') \end{array}}{sys \xrightarrow{\texttt{sendStickyBroadcast}\ i\ ic} sys'}$$

Precondición Para que la instancia ic pueda mandar el *intent* i mediante la operación `sendStickyBroadcast`, el *intent* i debe ser de tipo *broadcast* y no puede tener declarado ningún permiso en su estructura. Además la instancia en ejecución ic deberá estar corriendo y el par $\langle ic, i\rangle$ no puede estar en el conjunto de *intents* enviados.

Postcondición Al *intent* i se le agrega el permiso p especificado en la operación y es agregado al conjunto de *intents* enviados del sistema, junto con la instancia ic que lo envió.

Figura I.8: Semántica de la operación `sendStickyBroadcast`

Acción `stop` ic

La instancia ic deja de estar en ejecución.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \land e = (mfsts, certs, defPerms, img) \land \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \land \\ \exists (a : idApp)(c : Cmp), inApp(c, a, sys) \land \\ running(ic, c, sys) \\ \\ stopIns(ic, sys) = iCs' \land \\ revokeTPermsIns(ic, sys) = delTP' \land \\ sys' = (e, s') \ \land s' = (apps, perms, iCs', delPP, delTP', vals, ints) \end{array}}{sys \xrightarrow{\texttt{stop}\ i} sys'}$$

Precondición Para que una instancia pueda dejar de estar en ejecución, tiene que haberlo estado previamente.

Postcondición Se remueve la instancia ic del conjunto de instancias del sistema, y se revocan todos los permisos temporales delegados por dicha instancia.

Figura I.9: Semántica de la operación `stop`

Acción `grantP` *ic cp a u pt*

La instancia *ic* le delega a la aplicación *a* permisos permanentes de tipo *pt* sobre la URI *u* del *content provider cp*.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \land e = (mfsts, certs, defPerms, img) \land \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \land \\ a \in apps \ \land \exists(a_1 : idApp), inApp((cmpCP\ cp), a_1, sys) \land \\ canGrant(cp, u, sys) \land existsRes(cp, u, sys) \land \\ \exists(a_2 : idApp)(c : idApp), inApp(c, a_2, sys) \land \\ hasPermToDelOrRev(pt, c, cp, u, sys) \\ \\ grantPPerm(a, cp, u, pt, sys) = delPP' \land \\ sys' = (e, s') \ \land s' = (apps, perms, iCs, delPP', delTP, vals, ints) \end{array}}{sys \xrightarrow{\texttt{grantP}\ ic\ cp\ a\ u\ pt} sys'}$$

Precondición Para que la instancia *ic* pueda delegar permisos permanentes de tipo *pt* a la aplicación *a* sobre el recurso *u* del *content provider cp*, *a* debe pertenecer a las aplicaciones del sistema, *ic* tiene que estar en ejecución y además debe poder realizar la operación para la que quiere delegar permisos.

Postcondición La aplicación *a* puede realizar la operación *pt* sobre el recurso *u* del content provider *cp*.

Figura I.10: Semántica de la operación `grantP`

Acción `revokeDel` *ic cp u pt*

La instancia *ic* revoca el permiso pt sobre la URI *u* del *content provider cp*.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \wedge e = (mfsts, certs, defPerms, img) \wedge \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \wedge \\ \exists (a : idApp) inApp((cmpCP\ cp), a, sys) \wedge \\ existsRes(cp, u, sys) \wedge \\ \exists (c : Cmp)(a_1 : idApp), inApp(c, a_1, sys) \ \wedge \ running(ic, c, sys) \wedge \\ hasPermToDelOrRev(pt, cp, c, sys) \\ \\ revokeTPerm(cp, u, pt, sys) = delTP' \wedge \\ revokePPerm(cp, u, pt, sys) = delPP' \wedge \\ sys' = (e, s') \ \wedge s' = (apps, perms, iCs, delPP', delTP', vals, ints) \end{array}}{sys \xrightarrow{\texttt{grantP}\ ic\ cp\ a\ u\ pt} sys'}$$

Precondición Para que una instancia *ic* pueda revocar todos los permisos de tipo *pt* sobre la URI *u* del *content provider cp*, se debe cumplir que la instancia este en ejecución, que el recurso efectivamente exista y que la instancia tenga los permisos para realizar la operación correspondiente.

Postcondición Se revocan los permisos de tipo *pt* delegados de forma temporal o permanente sobre el recurso *u* del *content provider cp*.

Figura I.11: Semántica de la operación `revokeDel`

Acción `call` *ic sac*

La instancia en ejecución *ic* hace un llamado a una API del sistema *sac*.

Regla

$$\frac{\begin{array}{c} sys = (e, s) \land e = (mfsts, certs, defPerms, img) \land \\ s = (apps, perms, iCs, delPP, delTP, vals, ints) \land \\ \exists(c : Cmp), running(ic, c, sys) \land \\ \forall(a : idApp)(p : Perm)(H : isSystemPerm(p)), inApp(c, a, sys) \land \\ permSAC(p, H, sac) \Rightarrow \\ \exists(lp : list\ Perm), hasGrantedPerms(a, lp, sys) \ \land \ p \in lp \end{array} \quad sys' = (e, s)}{sys \xrightarrow{\texttt{call}\ ic\ sac} sys'}$$

Precondición Para que la instancia *ic* pueda hacer un llamado a la API *sac* debe estar en ejecución. Además la aplicación que contiene al componente asociado a la instancia deberá contar con los permisos necesarios para hacerlo.

Postcondición No se realiza ninguna modificación en sistema.

Figura I.12: Semántica de la operación `call`