UNIVERSIDAD NACIONAL DE ROSARIO

TESINA DE GRADO
PARA LA OBTENCIÓN DEL GRADO DE
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

Especificación e implementación de un prototipo certificado del sistema de permisos de Android

Autor:
Felipe Gorostiaga

Directores:
Dr. Gustavo Betarte
Dr. Carlos Luna
Co-Director:
Dr. Juan Diego Campo

Departamento de Ciencias de la Computación
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Av. Pellegrini 250, Rosario, Santa Fe, Argentina

28 de octubre de 2016

Índice general

Resumen

La vasta cantidad de dispositivos móviles ejecutando Android y las facilidades de distribución de software de terceros que ofrece hacen de este sistema operativo un objetivo tentador para ciberdelincuentes, cuyas acciones tienen el potencial de afectar a miles de millones de personas. Esta situación ha puesto a Android bajo la lupa de los investigadores, quienes pretenden poner a prueba la robustez de su sistema de seguridad y minimizar los vectores de ataque.

En este trabajo se extendió un modelo preexistente del sistema de seguridad de Android considerando los cambios introducidos en su versión 6.0 y especificando el comportamiento del sistema frente a la ejecución de operaciones erróneas. También se desarrolló un programa que implementa tal modelo y se demostró formalmente su corrección. Además se postularon y demostraron propiedades de seguridad sobre el modelo y el código ejecutable referidos y finalmente se explicó la manera en que la implementación certificada puede utilizarse a modo de oráculo de referencia para verificar la corrección de una plataforma real, y cómo permitiría la posterior programación de un monitor de seguridad capaz de supervisar una plataforma en tiempo real.

Tanto el modelo como el programa ejecutable, la prueba de su corrección y los planteos y demostraciones de las propiedades de seguridad fueron desarrollados haciendo uso del asistente de pruebas Coq.

Introducción

Más de dos mil millones de personas a lo ancho del mundo poseen un teléfono inteligente, número que, se estima, aumentará aún más en los próximos años [27]. De este total, el ochenta por ciento ejecutan el sistema operativo Android [17], dando una cifra estimada de mil seiscientos millones de personas cuya privacidad depende del sistema de seguridad de esta plataforma.

Android es un sistema operativo de código abierto especialmente diseñado para dispositivos móviles iniciado por la empresa Android Inc., la cual fue adquirida por Google en 2005 [18], quien continuó el proyecto, anunciado públicamente en 2007 [24].

En Android, el usuario accede a las funcionalidades de su dispositivo mediante el uso de aplicaciones. Algunas de ellas son distribuidas junto con el propio sistema operativo, mientras que otras son desarrolladas por terceros y descargadas de Internet a voluntad y discreción del dueño del móvil. La posibilidad de ejecutar código propio en tanta cantidad de dispositivos con esta comodidad de distribución plantea un escenario tentador para desarrolladores de aplicaciones maliciosas. Para contrarrestar este efecto y proteger al usuario, la plataforma cuenta con un sistema de seguridad que modera las acciones que las aplicaciones pueden llevar a cabo.

El análisis exhaustivo y formal de la seguridad de un sistema operativo con un alcance masivo sin precedente, que está bajo constante ataque de hackers e incluso de servicios de inteligencia internacionales [23], es de un interés superlativo por la repercusión de las consecuencias que sus resultados pueden arrojar. Asimismo, también es interesante la construcción formal y verificada de una herramienta cuyo comportamiento sea el que se indica en la especificación de tal modelo por las aplicaciones prácticas que puede tener. Particularmente, en el capítulo 4 de este trabajo se analiza la posibilidad de desarrollar un monitor capaz de verificar propiedades de seguridad sobre una plataforma en tiempo real, mientras que en el capítulo 5 se describe cómo utilizar la implementación certificada como oráculo de referencia al aplicar tácticas de testing estructural.

En [35] se desarrolló una especificación formal del sistema de seguridad de Android en su versión 4.4.2 (KitKat) utilizando Coq, un sistema de manejo de pruebas

formales basado en el Cálculo de Construcciones Inductivas [9], el cual ofrece un lenguaje para escribir definiciones matemáticas, algoritmos ejecutables y teoremas, junto con un entorno interactivo para demostrarlos [37].

Como se detalla en el capítulo 1, sección 1.3, existe en Android un sistema de permisos que permite a las aplicaciones efectuar operaciones para las cuales requieren privilegios. En las versiones previas a la 6.0 (Marshmallow), al momento de la instalación de una aplicación en el dispositivo, se le presentaba al usuario la lista de permisos que la misma necesitaba para lograr su funcionamiento. Si el usuario no aceptaba que se le otorgara alguno de esos permisos, la instalación era cancelada. A partir de la versión 6.0, que es la analizada en el presente trabajo, se le permite al usuario otorgar y revocar subconjuntos de estos permisos según lo estime conveniente mientras la aplicación se encuentra en ejecución. Este cambio sustancial en el sistema de permisos se refleja en la extensión del modelo desarrollada en esta tesina. Además se extendió el modelo para contemplar los casos en que se intenta ejecutar una acción erróneamente y el proceder del sistema ante estos escenarios quede formalmente definido. Por último, se implementó un programa cuya corrección respecto al modelo se demostró formalmente. Tanto el programa ejecutable como la demostración de corrección fueron confeccionados mediante el asistente de pruebas Coq. El código del modelo extendido, el programa certificado y su demostración de corrección se encuentran disponibles en [20].

El modelado del manejo de errores en tiempo de ejecución se inspiró en el trabajo de [7]. Asimismo, siguiendo las estrategias allí presentes se demostró la corrección de la implementación codificada. Se utilizó como referencia el trabajo de [8] que provee un análisis formal del modelo de seguridad del sistema operativo, mientras la especificación del mecanismo de otorgamiento, revocación y resolución en tiempo de ejecución de permisos se obtuvo de las guías del sitio oficial de Android [4], recurriendo a pruebas para inferir el comportamiento en situaciones que los documentos no contemplan explícitamente.

El presente informe se organiza de la siguiente manera: el capítulo 1 introduce el sistema operativo, poniendo énfasis en su arquitectura y su sistema de seguridad. En el capítulo 2 se detalla el modelo formalizado y se plantean y demuestran propiedades de seguridad sobre él. El capítulo 3 presenta un programa que se comporta de acuerdo al modelo expuesto en la sección previa, detallando cómo fue formalmente demostrada su corrección. Allí mismo se demuestran propiedades de seguridad sobre la implementación certificada. En el capítulo 4 se sugiere cómo puede usarse el programa presentado en el capítulo anterior para desarrollar un monitor que verifique paso a paso ciertas condiciones de seguridad, utilizando a la implementación certificada como un oráculo. Finalmente, en el capítulo 5 se describen las conclusiones y los trabajos futuros. Se incluye en este informe un anexo en donde constan la semántica de las acciones y los escenarios en donde se deben retornar códigos de errores que no forman parte del cuerpo del trabajo.

Capítulo 1

Android y su sistema de seguridad

Como se mencionó en la Introducción, el sistema operativo Android fue específicamente diseñado para controlar el software en dispositivos móviles, y ha logrado un alcance de miles de millones de usuarios. Estos usuarios interactúan con el dispositivo mediante el uso de las aplicaciones allí instaladas, las cuales recurren al sistema operativo o a otras aplicaciones para alcanzar su funcionalidad. Cada usuario puede instalar y desinstalar aplicaciones según el uso que le dará a su propio dispositivo.

El éxito de Android se debe en parte al inmenso ecosistema de aplicaciones que se encuentran disponibles para ser descargadas desde Internet a través de los servicios de distribución de aplicaciones, como el ampliamente difundido Google Play Store, de Google [19]. Esta es una relación simbiótica, pues los usuarios eligen Android por la cantidad de aplicaciones en su tienda y los desarrolladores eligen implementar sus programas para Android por la cantidad de usuarios que puede alcanzar.

Es por este motivo que los diseñadores del sistema operativo ponen énfasis en fomentar y simplificar el desarrollo y la distribución de aplicaciones para Android por parte de terceros, contando con una comunidad de más de cuatrocientos mil desarrolladores a fines del año 2014 [6] y más de dos millones de aplicaciones en la tienda de Google, Google Play Store, a mediados de 2016 [26]. Con tantas aplicaciones de tantos desarrolladores diferentes pudiendo coexistir en el mismo dispositivo, debió diseñarse un sistema de seguridad con políticas lo suficientemente estrictas para garantizar que ninguna de ellas pueda acceder a datos o interferir en la ejecución de otras sin el consentimiento del usuario.

En este capítulo se describe superficialmente la arquitectura general del sistema de seguridad de Android, y se introducen los componentes críticos que se representan luego en el modelo formal, expuesto en el capítulo 2. Esta tesina continúa el trabajo de [35], en donde puede encontrarse un análisis más profundo de las características del sistema operativo.

1.1. Arquitectura

Android sigue una arquitectura en forma de capas que utilizan servicios de su estrato inmediatamente inferior para ofrecer servicios a su nivel inmediatamente superior, abstrayéndose las capas superiores progresivamente del hardware y software de base. Este esquema se ve representado en la figura 1.1, en donde se pueden apreciar 5 capas, ordenadas desde la más abstracta a la de más bajo nivel:

1. En el nivel superior se encuentran las aplicaciones con las que interactúa directamente el usuario, tanto aquellas que son nativas del sistema operativo (por ejemplo, el calendario o la cámara) como las desarrolladas por programadores externos (por ejemplo, aquellas que se descargan de Internet).

2. Por debajo de ella, se halla la implementación de las interfaces de programación de aplicaciones (API) Java utilizadas por los desarrolladores de apps que habitan en la capa anteriormente descripta. Es en este nivel en donde se resuelven los aspectos de seguridad modelados y analizados en el presente trabajo.

3. En la capa subyacente se encuentran la implementación de librerías nativas en C y C++, utilizadas por componentes de sistema críticos de Android y el entorno de ejecución de aplicaciones de Android 6.0: “Android Runtime” (ART) [3].

4. Un nivel por debajo de ella se aloja la capa de abstracción de hardware, en donde se representan los dispositivos físicos del móvil (como la cámara y el bluetooth).

5. Finalmente, nos encontramos con el núcleo del sistema operativo: un Linux. Esto dota a Android de funcionalidades como ejecución multi-hilo y eficiente manejo de memoria. Además, permite que cada aplicación ejecute con un identificador de usuario y grupo de Linux diferente, características utilizadas para aplicar seguridad [4]. El núcleo contiene los manejadores de hardware y se encarga de la administración de la energía.

1.2. Las aplicaciones

El modelo se centra en el sistema de seguridad que modera las aplicaciones y las interacciones, tanto entre ellas como con el sistema operativo. Por ello, en esta sección se describirán los aspectos más relevantes de las aplicaciones, que serán mencionados en los capítulos posteriores.

Permisos

Por defecto, ninguna aplicación tiene permiso para ejecutar operaciones que puedan tener un impacto adverso en otras aplicaciones, en el sistema operativo en sí o

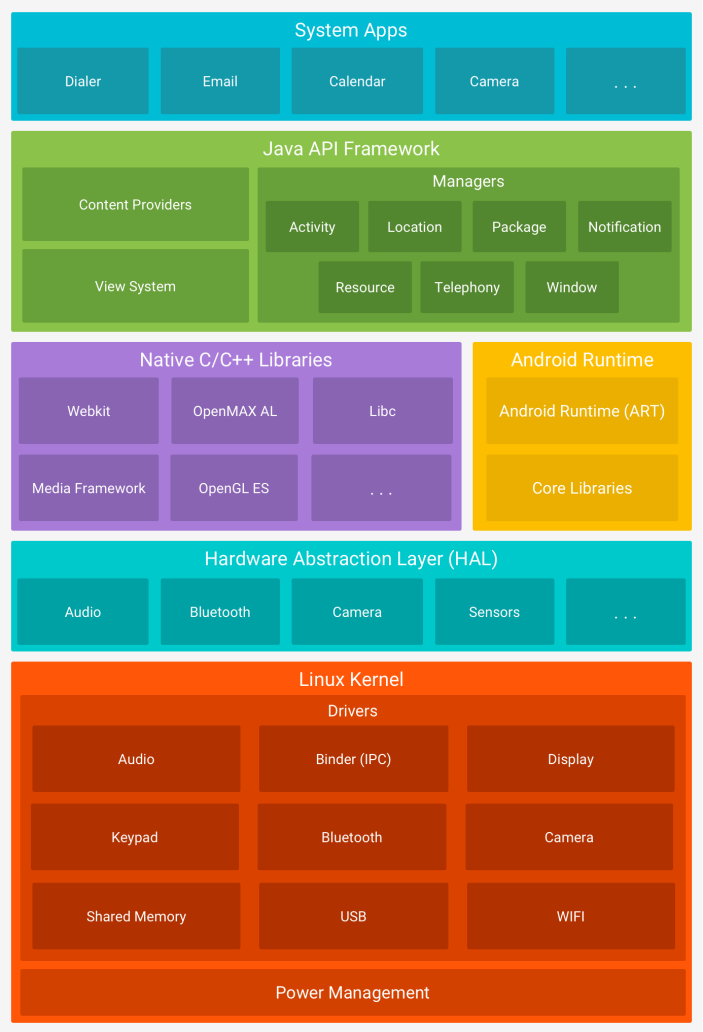

Figura 1.1: Arquitectura de Android [32]

en el propio usuario [4]. Sin embargo, esta restricción es demasiado conservadora y muchas veces una aplicación debe interactuar con otras para lograr su funcionalidad.

Para alcanzar este objetivo, una aplicación puede exponer de forma explícita sus datos y componentes, requiriendo que sólo pueda accederlos quien cuente con los permisos adecuados. Se detallará el sistema de permisos con mayor profundidad en la sección 1.3 .

Algunos permisos son predefinidos del sistema, como el necesario para acceder a la cámara; pero las aplicaciones pueden definir sus propios permisos, indicando su nombre (que funciona como un identificador), el nivel de protección que representa y el grupo al que pertenece, si es que forma parte de alguno. La pertenencia de un

permiso a un grupo será tenida en cuenta en la política de seguridad, pues el usuario podrá otorgar y revocar derechos sobre grupos de permisos. Más información al respecto se brindará en la sección 1.3 .

El nivel de protección de un permiso puede ser “normal”, “peligroso”, “de misma firma” o “de sistema o misma firma”. Nuevamente, la diferencia entre ellos se explicará en la sección 1.3 .

Componentes de una aplicación

Una aplicación consta de un conjunto de componentes, que pueden ser de cuatro tipos:

- **Actividades**:
  Son aquellas que se muestran en la pantalla del dispositivo y con las que el usuario interactúa directamente. Múltiples instancias de la misma actividad pueden ejecutarse en paralelo en el sistema.

- **Servicios**:
  Los servicios se ejecutan en segundo plano y no presentan una interfaz al usuario. Existe a lo sumo una instancia en ejecución de un servicio en el sistema. Suelen ser utilizados para realizar tareas de larga duración que no requieran interacción con el usuario, pues no interfieren con la ejecución de las actividades.

- **Proveedores de contenido**:
  Literalmente: “Los proveedores de contenido manejan el acceso a un conjunto estructurado de datos. Ellos encapsulan la información y proveen mecanismos para aplicar seguridad. Son la interfaz estándar que conecta datos en un proceso con código ejecutándose en otro proceso.” [10]. En otras palabras, un proveedor de contenido actúa como intermediario entre los datos persistidos de la aplicación que lo contiene y el resto de las aplicaciones, moderando el acceso a ellos a través del sistema de permisos. Cada recurso del proveedor de contenido se identifica mediante un identificador uniforme de recursos (URI [34]), y el desarrollador tiene la granularidad suficiente para definir permisos que gobiernen la lectura y la escritura de sus recursos, tanto en general como particularmente. Además, si el proveedor de contenido lo permite, una aplicación que tenga derecho de lectura o escritura sobre un recurso de ella puede otorgar este permiso a otra aplicación que no lo tenga. Más detalles acerca de la delegación de permisos se brindan en la sección 1.3 .

- **Receptores de *intent* de difusión**:
  Estos componentes fueron diseñados para suscribirse y recibir mensajes de difusión emitidos por otra aplicación o por el propio sistema operativo y actuar en consecuencia.

Intents

La interacción entre aplicaciones se logra a través del intercambio de mensajes asincrónicos de tipo *intent*, que pueden ser enviados como un mensaje de difusión (y recibidos por los correspondientes componentes receptores mencionados en la sección anterior), o con un único destinatario.

Al crear un *intent*, se especifican:

- El componente destino al que va dirigido (este valor puede omitirse, en cuyo caso la resolución del destinatario se basará en los otros campos)
- La acción que quiere realizarse o, si se enviará como difusión, el evento cuyo suceso se desea notificar, aunque este dato puede omitirse.
- Las categorías a las cuales el *intent* pertenece, que brindan información adicional sobre la acción a ejecutar.
- La información sobre la que debe operar, cuyo tipo (si es de contenido, de un archivo u otro) debe especificarse. Además, puede explicitarse el tipo MIME [33] de estos datos y, de ser necesario, dar la ruta para accederlos.
- Si se desea, información extra del *intent* proveyendo claves y valores que podrán posteriormente ser recuperados por los receptores. Sirve para enriquecer el *intent* con información personalizada que el emisor quiera incluir.
- Opcionalmente, banderas predefinidas del sistema que caracterizan al *intent* y controlan cómo será manejado. Las banderas son definidas por el sistema operativo y los efectos de activar cada una de ellas pueden ser consultados en [25].
- Opcionalmente, un permiso que se les solicitará a los receptores si el *intent* se envía como difusión.

Cuando el componente al cual el *intent* va dirigido no se ha definido, se lo considera un *intent* implícito. Su resolución consiste en decidir, a partir de los otros campos, cuál será el componente destino. Los mensajes de difusión pueden ser enviados a todos los receptores al mismo tiempo (de manera "no ordenada"), o de manera progresiva ("ordenada"), permitiéndoles cancelar su propagación. También pueden ser enviados de manera "pegajosa" ($sticky$), logrando que los componentes que se suscriban al método adecuado reciban, ante su sola suscripción, el último *intent* que se había difundido.

Filtros de intents

Para cada componente definido en una aplicación puede especificarse una lista de filtros de *intents*. A la hora de resolver cuál será el componente destino de un *intent* implícito, el sistema tomará en consideración sólo aquellos componentes para los que exista algún filtro satisfecho por el *intent* en cuestión. Cada filtro puede

especificar qué acción, categoría e información sobre la cual operar debe definir un *intent* para que el componente pueda procesarlo.

Manifiesto y certificado

Una aplicación a instalar incluye un archivo de manifiesto en formato XML [15] en donde provee información precisa acerca de ella. En tal archivo, lista los componentes que define, la versión mínima del sistema operativo necesaria para ejecutarse correctamente, la mínima versión de Android en la cual fue probada (y se garantiza su correcto funcionamiento), la lista de permisos que utilizará en su ejecución, los permisos que ella define y, si lo desea, un permiso que protege a todos sus componentes. En la figura 1.2 se presenta como ejemplo el archivo de manifiesto de una aplicación que utiliza la cámara del dispositivo, accede a su agenda y declara una actividad, llamada "MainActivity", capaz de recibir *intents* de acción android.intent.action.MAIN y categoría android.intent.category.LAUNCHER.

Además, cada aplicación es firmada con el certificado de su desarrollador.

1.3. Modelo de seguridad

Android aprovecha el núcleo Linux del sistema para asignar a cada aplicación un identificador de usuario y grupo diferente [1], y ejecutar cada una de ellas en una máquina virtual distinta. Los archivos creados por las aplicaciones en la memoria interna del dispositivo sólo pueden ser accedidos por su creadora. La consecuencia inmediata de esto es que las aplicaciones no pueden acceder a recursos de los cuales no sean dueñas.

Sistema de permisos

Muchas veces las aplicaciones necesitan acceder a recursos de otras aplicaciones o del sistema operativo para lograr su funcionalidad. Para controlar estos accesos, existe en Android un sistema de permisos que cumple un papel fundamental en su modelo de seguridad.

A grandes rasgos, cuando una aplicación requiere ejecutar una acción que sea considerada peligrosa para el sistema operativo o para la privacidad del usuario — como acceder a la cámara, acceder a los archivos de la memoria externa del dispositivo o leer la agenda de contactos — el sistema operativo analiza si cuenta con los permisos necesarios para hacerlo. De ser así, la acción es llevada a cabo. Caso contrario, la aplicación es finalizada. Una aplicación en ejecución puede chequear si cuenta con los permisos suficientes para llevar a cabo una acción y en caso de que

[1] Dos aplicaciones firmadas con el mismo certificado pueden ser asociadas al mismo id de usuario Linux si así lo requieren

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  Copyright 2015 The Android Open Source Project

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

     http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing,
      software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
      implied.
  See the License for the specific language governing permissions
      and
  limitations under the License.
  -->
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.system.runtimepermissions" >
    <!-- Note that all required permissions are declared here in the
        Android manifest.
     On Android M and above, use of these permissions is only
         requested at run time. -->
    <uses-permission android:name="android.permission.CAMERA"/>

    <!-- The following permissions are only requested if the device
        is on M or above.
     On older platforms these permissions are not requested and will
         not be available. -->
    <uses-permission-sdk-m
        android:name="android.permission.READ_CONTACTS" />
    <uses-permission-sdk-m
        android:name="android.permission.WRITE_CONTACTS" />

    <application
       android:allowBackup="true"
       android:icon="@mipmap/ic_launcher"
       android:label="@string/app_name"
       android:theme="@style/Theme.AppCompat.Light" >
       <activity
          android:name=".MainActivity"
          android:label="@string/app_name" >
          <intent-filter>
             <action android:name="android.intent.action.MAIN" />
             <category
                 android:name="android.intent.category.LAUNCHER" />
          </intent-filter>
       </activity>
    </application>

</manifest>
```

Figura 1.2: Ejemplo de un archivo de manifiesto de Android [5]

esto no sea así, puede solicitarlo al sistema operativo quien, a su vez, puede preguntar al usuario si desea otorgarle tal permiso.

La versión 6.0 (Marshmallow) del sistema operativo contrasta con las versiones anteriores en la forma en que se resuelven los permisos. Anteriormente, al intentar instalar una aplicación en el sistema, se le presentaba al usuario la lista de permisos peligrosos a los cuales ella accedería en tiempo de ejecución. Si el usuario consideraba que la aplicación era demasiado insegura para que le sean otorgados tales permisos, entonces la instalación era cancelada y naturalmente no podría ejecutarse. En Android 6.0, no se requiere que el usuario acepte los permisos a los que posiblemente una aplicación accederá al momento de instalarla, sino que ella misma debe encargarse de solicitar los privilegios necesarios durante su ejecución. Esto permite que un usuario confiera sólo los permisos que estime suficientes para el uso que desea dar a la aplicación. Incluso se le permite que revoque los permisos ya otorgados cuando lo juzgue necesario. Esta flexibilidad representó un cambio basal en las características de seguridad históricas del sistema operativo, y su correcto modelado ha sido uno de los principales focos de atención al desarrollar este trabajo.

Como se mencionó en la sección 1.2, cada permiso tiene asignado un nivel de protección. Los permisos de tipo "normal" sólo requieren que una aplicación los liste como usados para contar con ellos. Los de tipo "peligroso" deben ser listados como usados en el archivo de manifiesto y luego tienen que ser explícitamente otorgados por el usuario: ya sea a todo un grupo, si el permiso a otorgar forma parte de uno; o individualmente si no pertenece a ningún grupo. Los permisos "de misma firma" sólo son otorgados a las aplicaciones que los listen como usados y estén firmadas con el mismo certificado que quien las declara. Finalmente, los permisos "de misma firma o de sistema" se otorgan tanto a las aplicaciones de sistema como a aquellas que las listen como usadas y cuya firma coincida con el certificado de quien las declara.

Delegación de permisos

Una aplicación puede delegar a otra aplicación su derecho de ejecutar determinada acción mediante dos mecanismos diferentes.

Uno se basa en crear un *intent* pero no enviarlo, sino obtener un objeto "*intent* pendiente" asociado a él. Luego, pasar este objeto al componente destino, que al ejecutarlo contará con los mismos permisos de la aplicación que lo envió.

El segundo mecanismo consiste en que la aplicación explícitamente delegue su propio permiso de lectura o escritura sobre un recurso de un proveedor de contenido a otra para que ésta pueda accederlo de igual forma. Cuando se envía un *intent* para iniciar una actividad, puede definirse qué tipo de acceso a qué datos se le quieren delegar. Este segundo tipo de delegación se sostiene hasta que finaliza la actividad

por tal *intent* iniciada o hasta que la delegación sea explícitamente revocada por el delegador. A su vez, una aplicación puede otorgar, a través de una llamada al sistema, permisos permanentes a otra aplicación sobre un recurso de un proveedor de contenido, lo cual se mantendrá hasta que, al igual que en el método anterior, el delegador revoque tal privilegio. Para poder hacer uso de este mecanismo, el proveedor de contenido debe permitirlo en su definición. Caso contrario, no se permitirá que los permisos sobre él sean delegados de esta manera y deberá recurrirse a utilizar "*intents* pendientes" de ser necesario.

Capítulo 2

El modelo

En este trabajo, se extendió una formalización existente del sistema de seguridad de Android 4.4.2 (KitKat) que pone énfasis en la delegación de permisos y en el framework para realizar llamadas al sistema, para abarcar manejo de errores e incluir la delegación y revocación de permisos en tiempo de ejecución presentes en la versión 6.0 (Marshmallow) del sistema operativo. Dicha formalización se desarrolló utilizando el asistente de pruebas Coq, definiendo un componente que representa al sistema y un conjunto de acciones que definen cómo tal componente se modifica frente a la ejecución de cada una de ellas.

2.1. Entorno de trabajo

La especificación del modelo, el programa que lo implementa y la prueba de corrección de ella fueron desarrollados en el Cálculo de Construcciones Inductivas (CIC, por sus siglas en inglés "Calculus of Inductive Constructions"). El Cálculo de Construcciones Inductivas es un lenguaje que apunta a representar programas funcionales y pruebas en lógica de alto orden, basado en el Cálculo de Construcciones [11], un lenguaje funcional tipado polimórfico, pero extendido con definiciones inductivas primitivas [9].

Los tipos disponibles son muy parecidos a aquellos presentes en los lenguajes funcionales fuertemente tipados: tipos definidos por inducción — como secuencias y árboles — y el tipo de las funciones. Un tipo inductivo se define a través de sus constructores y sus elementos se obtienen mediante combinaciones finitas de estos constructores. Los tipos de datos se denominan $Sets$, y se encuentra en disponibilidad una lógica de alto orden que permite predicar sobre los distintos tipos de datos. La interpretación de las proposiciones es constructiva, lo que significa que una proposición se define especificando lo que significa para un objeto ser una prueba de esta proposición. Una proposición es verdadera si y sólo si puede construirse una prueba de su validez [8]. Este es el formalismo subyacente en el asistente de pruebas Coq, que fue la herramienta utilizada en el desarrollo y que permite la extracción

del programa certificado a lenguajes como OCaml y Haskell para su compilación y ejecución.

2.2. Notación

La notación utilizada en este trabajo es similar a la utilizada en [35].

Variables y tipos

Las variables se representan con nombres que comiencen con una letra minúscula mientras que los tipos atómicos, con nombres que contienen letras mayúsculas. El tipo de las funciones totales de A en B, se notará como $A \to \mathrm{B}$. Además, la expresión $a : A$ indica que la variable a es de tipo A.

Funciones parciales

El tipo de una función parcial de A en B se notará $A \twoheadrightarrow B$. Como en la plataforma Coq no pueden utilizarse funciones parciales, las mismas son totalizadas de la siguiente manera: si quiere representarse una función parcial $func : A \twoheadrightarrow B$, entonces para todo $a : A$ el resultado de $f\ a$ será o bien $Value\ (func\ a)$ en caso de que $a \in dom\ func$ o bien $Error\ B\ a$ en caso contrario.

Código insertado

Algunas veces se citarán fragmentos de código Coq, cuya sintaxis respetará la del lenguaje Gallina [13]. También se incluirán fragmentos de código en lenguaje C.

Tuplas

El tipo de una tupla se representa como el producto cartesiano de los tipos que la componen:

$$nTupla = T_1 \times T_2 \times ... \times T_n$$

Una tupla nt de tipo $nTupla$ se escribirá listando los elementos que la conforman entre paréntesis, separados por coma:

$$nt : nTupla = (t_1, t_2, ..., t_n)$$

En donde $t_i : T_i$ para todo $1 \leq i \leq n$.

Definiciones locales

Para evitar la sucesiva reescritura de un término recurrente, se definirá localmente una variable por la cual se reemplazarán las ocurrencias de dicho término en la expresión. El esquema para representar que se define una variable v por la cual se reemplazaron las ocurrencias de t en la expresión e es el siguiente:

$$\mathsf{let}\ v := t\ \mathsf{in}\ e$$

Predicados

Los predicados se expresarán utilizando los símbolos $\wedge, \vee, \implies, \exists, \forall$ de la lógica intuicionista, manteniendo su significado.

Tipos inductivos

Un tipo inductivo se definirá a través de sus constructores, con la siguiente forma:

$$\begin{array}{l} Ind = \\ \quad | \ cons_1 : A_1 \ \to B_1 \ \to Ind \\ \quad | \ cons_2 : A_2 \ \to Ind \end{array}$$

En donde se indica el nombre de los constructores y los tipos de sus argumentos. En el ejemplo, el tipo Ind tiene dos constructores, $cons_1$ y $cons_2$, en donde $cons_1$ requiere un elemento de tipo A_1 y un elemento de tipo B_1 para ser utilizado, mientras que $cons_2$ precisa de un elemento de tipo A_2 para conformar un miembro de Ind. Si algún constructor no tiene argumentos, su tipo puede omitirse.

Pattern matching

El análisis de un elemento $i : Ind$ a través de pattern matching, con sus constructores definidos como en el ejemplo anterior, se notará como:

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

Tipo option

En analogía con la mónada $Maybe$ de Haskell y la clase $Optional$ de Java, se utiliza el tipo $option\ A$ para representar o bien un valor de tipo A o bien la ausencia de valor:

$$\begin{array}{l} option\ A = \\ \quad | \ None \\ \quad | \ Some : A \ \to option\ A \end{array}$$

Tipo list

El tipo que representa a una lista de elementos se notará con el tipo de los elementos que contiene entre corchetes: un elemento $as : [A]$ es una lista de elementos

de tipo A.

La función para construir listas se nota :: . Siguiendo el ejemplo anterior, sea $a : A$, entonces $a :: as : [A]$ es la lista resultado de anteponer el elemento a a as. Una lista puede definirse como la sucesión de sus elementos, separados por coma:

$as = [a_1, a_2, ..., a_n]$
En donde $a_i : A$ para todo $1 \leq i \leq n$.

Cabe notar que las listas son un caso particular de tipo inductivo cuya definición es la siguiente:

$list\ A =$
$\quad | \ nil$
$\quad | :: : A \ \rightarrow list\ A$
Siendo :: un operador infijo.

Tipo nat

El tipo de los números naturales se representa **nat** .

Tipo Prop

El tipo de las proposiciones de Coq mantiene su nombre y se representa **Prop** . Este tipo sirve para razonar acerca del modelo y demostrar la corrección del programa certificado, pero su información es desechada al momento de extraer el código ejecutable.

Registros

Un registro se expresa mediante su nombre y el nombre y tipo de los campos que lo componen, separados por coma:

$Reg = \{$
$\quad f_1 : C_1 \ \rightarrow D_1 \ \rightarrow E_1,$
$\quad f_2 : C_2 \ \rightarrow D_2$
$\}$

Dado un elemento $r : Reg$, el acceso a su campo c se notará $r.c$. En el ejemplo anterior, los registros de tipo Reg se componen de dos campos llamados f_1 y f_2, los cuales son dos funciones de tipo $C_1 \ \rightarrow D_1 \ \rightarrow E_1$ y $C_2 \ \rightarrow D_2$ respectivamente.

Si dos registros r_1, r_2 coinciden en todos sus n campos c_i, $0 \leq i < n$, esta relación entre ellos puede expresarse de la siguiente manera:

$r_1 \equiv_{c_0,c_1,...,c_{n-1}} r_2$

Lemas y teoremas

Las propiedades del sistema se presentarán mediante lemas y teoremas, enunciando su nombre seguido por el planteo de la propiedad demostrada, como se ve en los siguientes ejemplos:

Lema $postInstallCorrect$:
$\forall(s : System)(a : idApp)(m : Manifest)(c : Cert)(lRes : [Res]),$
$pre\ (install\ a\ m\ c\ lRes)\ s \implies$
$validstate\ s \implies$
$post\_install\ a\ m\ c\ lRes\ s\ (install\_post\ a\ m\ c\ lRes\ s)$

Teorema $hasPermissionAndNotGranted$:
$\exists(s : System)(p : Perm)(a : idApp)(permsA : [Perm]),$
$Model.tex - validstate\ s \wedge p.pl = dangerous \wedge$
$s.state.perms\ a = Value\ permsA \wedge p \notin permsA \wedge$
$p \notin getDefPermsForApp\ a\ s \wedge appHasPermission\ a\ p\ s$

2.3. Definiciones básicas del sistema

Algunos tipos que representan los componentes de Android sobre los cuales se predica en la especificación son aquellos listados en [35]. Estos serán descriptos brevemente, pues pueden encontrarse más detalles sobre ellos en el trabajo citado. Se listan las definiciones más interesantes para comprender el modelo y la implementación.

Tipos atómicos

En la tabla 2.1 se presentan los tipos de datos cuya representación es atómica y no ameritan una descomposición más detallada para el desarrollo del modelo, junto con lo que simbolizan en el sistema operativo.

Permisos

El tipo de los permisos se representa a través de un registro

$$Perm = \{\ idP : idPerm,\ \ maybeGrp : option\ idGrp,\ \ pl : PermLevel\ \}$$

En donde idP representa el nombre de un permiso; $maybeGrp$, si está agrupado o no y, en caso de estarlo, el grupo de permisos al que pertenece; y pl, su nivel de protección, que puede ser, como en [35], "peligroso" ($dangerous$), "normal" ($normal$),

Tipo	Representa
$idApp$	Nombres de las aplicaciones, que también son sus identificadores
$idPerm$	Nombres de los permisos, que también son sus identificadores
$idGrp$	Nombres de los grupos de permisos, que también son sus identificadores
$iCmp$	Identificadores de las instancias de componentes actualmente en ejecución
$Cert$	Certificados con los cuales se firman las aplicaciones
Res	Recursos que define una aplicación
uri	Identificadores uniformes de recursos [34]
$mimeType$	Tipos de datos MIME [33]
$idCmp$	Identificadores de los componentes de una aplicación
$Category$	Categorías a las que puede pertenecer un *intent*
$Extra$	Información extra que puede agregar el emisor de un *intent*
$Flag$	Banderas que pueden ser activadas por el emisor de un *intent* e influyen en su manejo
$SACall$	Llamadas al sistema de Android que interfieren en el modelo de seguridad
Val	Valores que pueden escribirse en los recursos de las aplicaciones

Cuadro 2.1: Tipos atómicos y lo que representan

"de misma firma" ($signature$) o "de sistema o misma firma" ($signature/system$). De igual manera existe un predicado $isSystemPerm : Perm \nrightarrow$ **Prop** que se cumplirá para un permiso si y sólo si tal permiso es de sistema.

Data de intent

Al enviar un *intent*, se puede especificar información a la cual el receptor podrá acceder. Para caracterizar esta información se define un tipo $Data$ como un registro:

$$
\begin{aligned}
&Data = \{\\
&\quad path : option\ uri,\\
&\quad mime : option\ mimeType,\\
&\quad type : dataType\\
&\}
\end{aligned}
$$

En donde $path$ especifica la URI del recurso al que el receptor del *intent* accederá, si hay alguno; $mime$ especifica la clase de dato a acceder; y $type$ indica si lo que se quiere acceder (de tipo $dataType$) es contenido ($content$), un archivo ($file$) u otra cosa ($other$).

Intents

Los *intents* son el componente esencial en la manera en la que Android implementa la comunicación entre procesos. El tipo $Intent$ es un record con la siguiente forma:

$$
\begin{aligned}
&Intent = \{\\
&\quad cmpName : option\ idCmp,
\end{aligned}
$$

$intType : intentType$,
$action : option\ intentAction$,
$data : Data$,
$category : [Category]$,
$extra : option\ Extra$,
$flags : option\ Flag$,
$brperm : option\ Perm$
}

En donde $cmpName$ especifica el nombre del componente receptor; $intType$ es la clase de *intent* (de tipo $intentType$), que puede ser de actividad ($intActivity$), de servicio ($intService$) o de difusión ($intBroadcast$); $action$ (de tipo $intentAction$) define la clase de acción del *intent*, si existe, que puede ser de actividad ($activityAction$), de servicio ($serviceAction$) o de difusión ($broadcastAction$); $data$ indica el tipo de dato al que quiere acceder el *intent*; $category$, una lista de categorías a las que pertenece el *intent*; $extra$, si existe, información adicional para el receptor del *intent*; $flags$, si existe, la bandera de datos del *intent*; y $brperm$, si existe, un permiso que debe tener la aplicación que reciba el *intent* si éste es de difusión.

Filtros de Intent

Los componentes de una aplicación definen filtros de *intents* que pueden ser recibidos por ella. Sólo los componentes para los cuales exista un filtro que el *intent* a resolver cumpla podrán ser electos como destinatarios de él. Para representar estos filtros se define un tipo:

$intentFilter = \{$
$actFilter : [intentAction]$,
$dataFilter : [Data]$,
$catFilter : [Category]$,
}

En donde $actFilter$ indica los tipos de acción que puede tenr un *intent* para ser aceptado por este filtro, $dataFilter$ indica los tipos de dato que puede declarar acceder un *intent* para superar el filtro y $catFilter$ indica las categorías a las que puede pertenecer el *intent* para superar el filtro.

Componentes de una aplicación

Entre los componentes de una aplicación pueden encontrarse, como se definió en [35], actividades, servicios, receptores de intents de difusión y proveedores de contenidos. Todos ellos son definidos a través de registros:

$Activity = \{$

$$
\begin{aligned}
&\quad idA : idCmp,\\
&\quad expA : option\ bool,\\
&\quad cmpEA : option\ Perm,\\
&\quad intFilterA : [intentFilter],\\
&\}\\
&Service = \{\\
&\quad idS : idCmp,\\
&\quad expS : option\ bool,\\
&\quad cmpES : option\ Perm,\\
&\quad intFilterS : [intentFilter],\\
&\}\\
&BroadReceiver = \{\\
&\quad idB : idCmp,\\
&\quad expB : option\ bool,\\
&\quad cmpEB : option\ Perm,\\
&\quad intFilterB : [intentFilter],\\
&\}\\
&CProvider = \{\\
&\quad idC : idCmp,\\
&\quad map\_res : uri \twoheadrightarrow res,\\
&\quad expC : option\ bool,\\
&\quad cmpEC : option\ Perm,\\
&\quad readE : option\ Perm,\\
&\quad writeE : option\ Perm,\\
&\quad grantU : bool,\\
&\quad uriP : [uri],\\
&\}
\end{aligned}
$$

En donde los campos idA, idS, idB e idC representan el identificador del componente; $expA$, $expS$, $expB$ y $expC$ indican si el componente figura como exportado; $cmpEA$, $cmpES$, $cmpEB$ y $cmpEC$ indican el permiso por el cual está protegido el componente y finalmente $intFilterA$, $intFilterS$ e $intFilterB$ indican los filtros de *intents* que el componente puede recibir.

A su vez, para los proveedores de contenidos se definen los campos map_res, que indica los recursos a los que apunta cada URI en su dominio; $readE$ y $writeE$, que especifican los permisos necesarios para leer y escribir sobre sus recursos, respectivamente; $grantU$, que indica si el proveedor de contenido permite delegar permisos de lectura y escritura sobre él; y $uriP$, que especifica los URIs presentes en él.

A partir de ellos se construye el tipo de las componentes como un tipo inductivo:

$Cmp =$
$\quad | \; cmpAct : Activity \; \rightarrow Cmp$
$\quad | \; cmpSrv : Service \; \rightarrow Cmp$
$\quad | \; cmpCP : CProvider \; \rightarrow Cmp$
$\quad | \; cmpBR : BroadReceiver \; \rightarrow Cmp$

Con un constructor por cada tipo de componente.

Manifiesto

El archivo de manifiesto de una aplicación se representa con el tipo $Manifest$, que tiene la siguiente forma:

$Manifest = \{$
$\quad cmp : [Cmp],$
$\quad minSdk : option$ **nat**,
$\quad targetSdk : option$ **nat**,
$\quad use : [idPerm],$
$\quad usrP : [Perm],$
$\quad appE : option \; Perm$
$\}$

En él se listan sus componentes (en cmp); si se especifica, la versión de Android mínima para ejecutar la aplicación ($minSdk$); si se especifica, la versión de Android para la cual la aplicación fue compilada ($targetSdk$); los permisos que define ($usrP$); los permisos que requiere para su funcionamiento ($useP$); y, si lo hay, el permiso que protege a sus componentes ($appE$).

Aplicaciones

Al instalar una aplicación, se detalla su identificador (de tipo $idApp$), su archivo manifiesto (de tipo $Manifest$), el certificado con el que fue firmada (de tipo $Cert$) y la lista de recursos que define (de tipo $[Res]$).

Aplicaciones del sistema

Algunas aplicaciones no son instaladas por el usuario sino que son distribuidas junto con el propio sistema operativo. Se representan a través de un registro:

$SysImgApp = \{$
$\quad idSI : idApp,$
$\quad certSI : Cert,$
$\quad manifestSI : Manifest,$

$defPermsSI : [Perm]$,
$appResSI : [Res]$
}

En donde se detalla su identificador, el certificado con el que fue firmada, su archivo de manifiesto, y los permisos y recursos que define.

Tipo de acceso a datos

Una aplicación que es capaz de leer y/o escribir el valor de un recurso puede delegar este derecho a otra mediante los mecanismos descriptos en 1.3. El tipo de operación que delega se representa mediante el tipo $PType$:

$PType = \mid Write \mid Read \mid Both$

En donde $Write$ representa la delegación del permiso de escritura, $Read$ representa la delegación del permiso de lectura y $Both$ representa la delegación de ambas.

2.4. El estado del sistema

Siguiendo el modelo desarrollado en [35], se define un sistema ($System$) que encapsula el estado de los aspectos relevantes para la seguridad del dispositivo como un registro de dos componentes, la primera de ellas representando conceptualmente el estado dinámico del sistema, mientras que la segunda componente encapsula información estática acerca de las aplicaciones:

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

En donde un estado ($State$) a su vez también es modelado como un registro:

$State = \{$
$apps : [idApp]$,
$grantedPermGroups : idApp \twoheadrightarrow [idGrp]$,
$perms : idApp \twoheadrightarrow [Perm]$,
$running : iCmp \twoheadrightarrow Cmp$,
$delPPerms : (idApp \times CProvider \times uri) \twoheadrightarrow PType$,
$delTPerms : (iCmp \times CProvider \times uri) \twoheadrightarrow PType$,
$resCont : (idApp \times Res) \twoheadrightarrow Val$,
$sentIntents : [(iCmp \times Intent)]$
}

En él, la componente $apps$ representa la lista de aplicaciones actualmente instaladas en el sistema; $grantedPermGroups$ indica los grupos de permisos otorgados a cada a aplicación; $perms$, los permisos individualmente otorgados a las aplicaciones instaladas; la función parcial $running$ indica a qué componente pertenece

cada instancia en ejecución del sistema; $delPPerms$ y $delTPerms$ representan los permisos sobre recursos de proveedores de contenido concedidos a aplicaciones de manera permanente y a instancias en ejecución de manera temporal, respectivamente; $resCont$ indica los valores de los recursos para cada aplicación; y por último la lista $sentIntents$ contiene los *intents* enviados que aún no han sido recibidos por ningún componente en ejecución.

Por otro lado, el registro que representa el entorno del sistema ($Environment$) en esta formalización es el siguiente:

$$
\begin{aligned}
&Environment = \{ \\
&\quad manifest : idApp \twoheadrightarrow Manifest, \\
&\quad cert : idApp \twoheadrightarrow Cert, \\
&\quad defPerms : idApp \twoheadrightarrow [Perm], \\
&\quad systemImage : [SysImgApp] \\
&\}
\end{aligned}
$$

En donde $manifest$ indica, para cada aplicación instalada, su manifiesto; $cert$, el certificado con el que fue firmada; $defPerms$, los permisos que ella define; y finalmente $systemImage$ representa la lista de aplicaciones que no han sido instaladas, sino que se hallan en la imagen del sistema operativo.

2.5. La validez del sistema

No todos los miembros de $System$ representan el estado de un sistema Android. Sus componentes no son independientes sino que existen propiedades e interrelaciones entre ellas que no son capturadas en la definición del tipo. A un estado cuyas componentes verifican estas características esenciales de la plataforma y sí representa el estado de un sistema Android se lo considera un estado válido.

Las propiedades que debe cumplir un estado para ser considerado válido son las siguientes:

- todos los componentes presentes tanto en aplicaciones instaladas como en aplicaciones de sistema tienen identificadores diferentes
- ningún componente pertenece a dos aplicaciones instaladas diferentes
- ninguna instancia en ejecución es un proveedor de contenido
- todo permiso temporalmente delegado es sobre un proveedor de contenido existente en el sistema y concedido a un componente actualmente en ejecución
- todo componente en ejecución pertenece a una aplicación existente en el sistema

- todas las aplicaciones que den valor a un recurso están en el sistema
- los dominios de las funciones parciales $manifest$, $cert$ y $defPerms$ del entorno del sistema ($environment$) son exactamente los identificadores de las aplicaciones instaladas por el usuario
- los dominios de las funciones parciales $perms$ y $grantedPermGroups$ del estado del sistema ($state$) son exactamente los identificadores de las aplicaciones presentes en la plataforma, tanto las instaladas por el usuario como aquellas en la imagen del sistema
- todas las aplicaciones instaladas tienen identificadores diferentes a las aplicaciones en la imagen del sistema, las cuales a su vez tienen identificadores diferentes entre sí
- todos los permisos definidos por las aplicaciones tienen identificadores diferentes
- todas las funciones parciales son efectivamente funciones, es decir, no existen elementos repetidos en sus dominios
- todos los permisos individualmente otorgados existen en el sistema; y
- todos los intents enviados tienen identificadores diferentes entre sí

Se definió la proposición $validstate : System \rightarrow$ **Prop** , que se verifica sólo para estados válidos. Su definición formal puede ser consultada en el código adjunto accediendo al sitio [20].

2.6. Ejecuciones del sistema

Se definió un conjunto de acciones que modelan operaciones relevantes relativas a la seguridad de un sistema Android (de tipo $System$) como miembros del tipo $Action$, el cual se presenta a continuación:

$$
\begin{aligned}
Action = \\
&| \; install : idApp \rightarrow Manifest \rightarrow Cert \rightarrow [Res] \rightarrow Action \\
&| \; uninstall : idApp \rightarrow Action \\
&| \; grant : Perm \rightarrow idApp \rightarrow Action \\
&| \; revoke : Perm \rightarrow idApp \rightarrow Action \\
&| \; grantPermGroup : idGrp \rightarrow idApp \rightarrow Action \\
&| \; revokePermGroup : idGrp \rightarrow idApp \rightarrow Action \\
&| \; hasPermission : Perm \rightarrow Cmp \rightarrow Action \\
&| \; read : iCmp \rightarrow CProvider \rightarrow uri \rightarrow Action \\
&| \; write : iCmp \rightarrow CProvider \rightarrow uri \rightarrow Val \rightarrow Action \\
&| \; startActivity : Intent \rightarrow iCmp \rightarrow Action
\end{aligned}
$$

$| \ startActivityForResult : Intent \ \rightarrow \textbf{nat} \ \rightarrow iCmp \ \rightarrow Action$
$| \ startService : Intent \ \rightarrow iCmp \ \rightarrow Action$
$| \ sendBroadcast : Intent \ \rightarrow iCmp \ \rightarrow option \ Perm \ \rightarrow Action$
$| \ sendOrderedBroadcast : Intent \ \rightarrow iCmp \ \rightarrow option \ Perm \ \rightarrow Action$
$| \ sendStickyBroadcast : Intent \ \rightarrow iCmp \ \rightarrow Action$
$| \ resolveIntent : Intent \ \rightarrow idApp \ \rightarrow Action$
$| \ receiveIntent : Intent \ \rightarrow iCmp \ \rightarrow idApp \ \rightarrow Action$
$| \ stop : iCmp \ \rightarrow Action$
$| \ grantP : iCmp \ \rightarrow CProvider \ \rightarrow idApp \ \rightarrow uri \ \rightarrow PType \ \rightarrow Action$
$| \ revokeDel : iCmp \ \rightarrow CProvider \ \rightarrow uri \ \rightarrow PType \ \rightarrow Action$
$| \ call : iCmp \ \rightarrow SACall \ \rightarrow Action$

La descripción informal de cada acción se presenta en la tabla 2.2.

La versión 6.0 de Android permite el otorgamiento y la revocación de permisos a las aplicaciones en tiempo de ejecución. Para reflejar esta nueva característica se introdujeron las acciones $grant$, $grantPermGroup$, $revoke$ y $revokePermGroup$. Además, se modificó sustancialmente la semántica de la acción $install$: en las versiones anteriores del sistema operativo el instalar una aplicación implicaba el otorgamiento irrestricto de todos los permisos que listara en su archivo de manifiesto como usado. Actualmente, la instalación de una aplicación inicializa vacías las listas de permisos y grupos de permisos a ella otorgados, debiendo solicitarlos en el momento en que sea necesario para lograr su funcionalidad.

Semántica formal de las acciones del sistema

La semántica de las acciones se especifica dando su precondición y postcondición, las cuales describen, respectivamente, los requerimientos que debe cumplir un sistema para que la acción pueda ejecutarse en él, y las propiedades de un sistema alcanzado por la correcta ejecución de tal acción.

Dados $a : Action$ y $s, s' : System$, las proposiciones $Pre \ s \ a$ y $Post \ s \ a \ s'$ se satisfacen, respectivamente, cuando s cumple con la precondición de a y s' cumple con la postcondición de ejecutar a sobre s. Si para una acción $a : Action$, un sistema $s : System$ cumple la precondición de a y un sistema $s' : System$ cumple la postcondición de a, entonces las tres instancias están relacionadas, y esta relación entre ellas se nota $s \overset{a}{\hookrightarrow} s'$. Más formalmente:

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

A continuación se presentan las semánticas de las acciones $install$, $grant$, $grantPermGroup$, $revoke$ y $revokePermGroup$:

Acción	Descripción
$install\ a\ m\ c\ lRes$	se solicita instalar la aplicación a, cuyo manifiesto es m, está firmada con el certificado c y su lista de recursos es $lRes$
$uninstall\ a$	se solicita desinstalar la aplicación a
$grant\ p\ a$	se requiere otorgar el permiso p a la aplicación a
$revoke\ p\ a$	se desea quitar el permiso p a la aplicación a
$grantPermGroup\ g\ a$	se requiere otorgar el grupo de permisos g a la aplicación a
$revokePermGroup\ g\ a$	se desea quitar el grupo de permisos g a la aplicación a
$hasPermission\ p\ a$	acción que permite chequear si la aplicación a cuenta con el permiso p
$read\ ic\ cp\ u$	la instancia en ejecución ic solicita leer el recurso u del proveedor de contenido cp
$write\ ic\ cp\ u\ val$	la instancia en ejecución ic solicita escribir el recurso u del proveedor de contenido cp asignándole el valor val
$startActivity\ i\ ic$	la instancia en ejecución ic quiere iniciar una actividad especificada mediante el *intent* i
$startActivityForResult\ i\ n\ ic$	la instancia en ejecución ic quiere iniciar una actividad especificada mediante el *intent* i y espera un valor de retorno con token n
$startService\ i\ ic$	la instancia en ejecución ic quiere iniciar un servicio mediante el *intent* i
$sendBroadcast\ i\ ic\ p$	la instancia en ejecución ic quiere enviar el *intent* i como difusión, especificando que sólo puede recibirlo un componente que cuente con el permiso p
$sendOrderedBroadcast\ i\ ic\ p$	la instancia en ejecución ic quiere enviar el *intent* i como una difusión ordenada, especificando que sólo puede recibirla un componente que cuente con el permiso p
$sendStickyBroadcast\ i\ ic$	la instancia en ejecución ic quiere enviar el *intent* i como difusión $sticky$
$resolveIntent\ i\ a$	la aplicación a solicita hacer explícito el *intent* i
$receiveIntent\ i\ ic\ a$	la aplicación a desea recibir el *intent* i enviado por la instancia en ejecución ic
$stop\ ic$	se solicita detener la instancia en ejecución ic
$grantP\ ic\ cp\ a\ u\ pt$	la instancia en ejecución ic delega permisos de manera permanente a la aplicación a. Esta delegación permite a a ejercer la acción pt sobre el recurso u de cp
$revokeDel\ ic\ cp\ u\ pt$	se quitan todos los permisos delegados a la instancia en ejecución ic para ejercer la acción pt sobre el recurso u de cp
$call\ ic\ sac$	la instancia en ejecución ic quiere efectuar la llamada al sistema sac

Cuadro 2.2: Descripción de las acciones

Acción `install` $a\ m\ c\ lRes$

Escenario: se solicita instalar la aplicación a, cuyo manifiesto es m, está firmada con el certificado c y su lista de recursos es $lRes$.

Regla

$$\frac{\begin{array}{l} has\_duplicates\ (map\ getCmpId\ m.cmp) = false \wedge \\ has\_duplicates\ (map\ idP\ m.usrP) = false \wedge \\ \forall c : Cmp, c \in m.cmp \implies cmpNotInState\ c\ s \wedge \\ authPerms\ m\ s \wedge \\ \forall c : Cmp, c \in m.cmp \implies cmpDeclareIntentFilterCorrectly\ c \wedge \\ addManifest\ m\ a\ s\ s' \wedge \\ addCert\ c\ a\ s\ s' \wedge \\ addDefPerms\ a\ m\ s\ s' \wedge \\ addApp\ a\ s\ s' \wedge \\ addRes\ a\ lRes\ s\ s' \wedge \\ initializePermLists\ a\ s\ s' \wedge \\ s.state \equiv_{running,delPPerms,delTPerms,sentIntents} s'.state \\ s.environment \equiv_{systemImage} s'.environment \end{array}}{s \xrightarrow{\texttt{install}\ a\ m\ c\ lRes} s'}$$

Precondición: la aplicación no se encuentra ya instalada ni existe una aplicación en la imagen del sistema con el mismo identificador. Los identificadores de los componentes listados en su manifiesto son diferentes entre sí y diferentes de los componentes ya existentes en el sistema. Los permisos definidos por la aplicación son diferentes entre sí. Todos los permisos declarados por la aplicación no son definidos por otra aplicación, ni de las instaladas por el usuario, ni de sistema. Los filtros de intents de los componentes de la aplicación a instalar están bien construidos.

Postcondición: se agregan el manifiesto, el certificado y los permisos definidos que no fueran de sistema al entorno asociados a su id de aplicación. Se agrega el id de aplicación a la lista de aplicaciones instaladas. Se inicializan los recursos de la aplicación con el valor inicial $initVal$. Se inicializan como vacías las listas de permisos y grupos otorgados a la aplicación. El resto de los componentes del sistema no cambian.

Acción `grant` $p\ a$

Escenario: se requiere otorgar el permiso p a la aplicación a.

Regla

$$\frac{\begin{array}{l}(\exists m : Manifest, isManifestOfApp\ a\ m\ s \wedge p.idP \in m.use) \wedge \\ (isSystemPerm\ p \vee usrDefPerm\ p\ s) \wedge \\ \neg(\exists lPerm : [Perm], s.state.perms\ a = Value\ lPerm \wedge p \in lPerm) \wedge \\ pl\ p = dangerous \wedge maybeGrp\ p = None \wedge \\ grantPerm\ a\ p\ s\ s' \wedge \\ \\ s \equiv_{environment} s' \wedge \\ s.state \equiv_{apps,grantedPermGroups,running,sentIntents} s'.state \wedge \\ s.state \equiv_{delPPerms,delTPerms,resCont} s'.state\end{array}}{s \xrightarrow{\texttt{grant}\ p\ a} s'}$$

Precondición: el permiso p debe estar declarado como usado en el manifiesto de a, debe existir, no debe estar ya otorgado, debe ser de tipo peligroso ($dangerous$) y no debe pertenecer a un grupo de permisos.

Postcondición: el permiso p se agrega a la lista de permisos otorgados a a. El resto de los componentes del sistema no varía.

Acción `grantPermGroup` $g\ a$

Escenario: se requiere otorgar el grupo de permisos g a la aplicación a

Regla

$$\frac{\begin{array}{l}\neg(\exists lGrp : [idGrp], s.state.grantedPermGroups\ a = Value\ lGrp \wedge g \in lGrp) \wedge \\ (\exists (m : Manifest)(p : Perm), isManifestOfApp\ a\ m\ s \wedge \\ p.idP \in m.use \wedge \\ (isSystemPerm\ p \vee usrDefPerm\ p\ s) \wedge \\ p.maybeGrp = Some\ g \wedge \\ p.pl = dangerous) \wedge \\ grantPermGroup\ a\ g\ s\ s' \wedge \\ s \equiv_{environment} s' \wedge \\ s.state \equiv_{apps,perms,running,sentIntents} s'.state \wedge \\ s.state \equiv_{delPPerms,delTPerms,resCont} s'.state\end{array}}{s \xrightarrow{\texttt{grantPermGroup}\ g\ a} s'}$$

Precondición: el grupo g no debe estar ya otorgado, y debe ser el grupo al que pertenece algún permiso peligroso que la aplicación a lista como usado en su manifiesto

Postcondición: el grupo g se agrega a la lista de grupos de permisos otorgados a a. El resto de los componentes del sistema no varía.

Acción `revoke` $p\ a$

Escenario: se desea quitar el permiso p a la aplicación a

Regla

$$\frac{\begin{array}{l}\exists(lPerm : [Perm]), s.state.perms\ a = Value\ lPerm \wedge p \in lPerm \wedge \\ revokePerm\ a\ p\ s\ s' \wedge \\ s \equiv_{environment} s' \wedge \\ s.state \equiv_{apps,grantedPermGroups,running,sentIntents} s'.state \wedge \\ s.state \equiv_{delPPerms,delTPerms,resCont} s'.state\end{array}}{s \xrightarrow{\texttt{revoke}\ p\ a} s'}$$

Precondición: el permiso p aparece entre los otorgados a a.

Postcondición: el permiso p es quitado de la lista de permisos otorgados a a, asegurando que la función $perms$ del nuevo estado es correcta. El resto de los componentes del sistema no varía.

Acción `revokePermGroup` $g\ a$

Escenario: se desea quitar el grupo de permisos g a la aplicación a

Regla

$$\frac{\begin{array}{l}\exists lGrp : [idGrp], s.state.grantedPermGroups\ a = Value\ lGrp \wedge g \in lGrp \\ revokePermGroup\ a\ g\ s\ s' \wedge \\ s \equiv_{environment} s' \wedge \\ s.state \equiv_{apps,perms,running,sentIntents} s'.state \wedge \\ s.state \equiv_{delPPerms,delTPerms,resCont} s'.state\end{array}}{s \xrightarrow{\texttt{revokePermGroup}\ g\ a} s'}$$

Precondición: el grupo de permisos g aparece entre los otorgados a a.

Postcondición: el grupo de permisos g es quitado de la lista de grupos de permisos otorgados a a, asegurando que la función $grantedPermGroups$ del nuevo estado es correcta. El resto de los componentes del sistema no varía.

Para conocer la semántica del resto de las acciones, referirse al Anexo.

2.7. Funciones auxiliares

A lo largo del trabajo se presentarán funciones y predicados clarificando y ejemplificando construcciones. En ellos se hace uso de funciones y predicados auxiliares, cuyos significados se describen en la tabla 2.3.

2.8. Resolución de permisos en tiempo de ejecución

La característica más sobresaliente de la última versión de Android es el nuevo sistema de permisos en tiempo de ejecución, en donde se permite otorgar y revocar permisos a una aplicación de manera dinámica.

Para modelar este nuevo sistema de resolución de permisos en tiempo real se definió un predicado $appHasPermission : idApp \rightarrow Perm \rightarrow System \rightarrow$ **Prop** , en donde $appHasPermission\ a\ p\ s$ se verifica cuando la aplicación a cuenta con el permiso p en el sistema s. Informalmente, esto se cumple cuando a tiene el permiso p individualmente otorgado, o el permiso existe en el sistema, es listado como usado en el manifiesto de ella y además se cumple alguna de las siguiente condiciones:

- es definido por ella, o
- es de nivel de peligro $normal$, o
- es de nivel de peligro $dangerous$, agrupado y el grupo ha sido otorgado a la aplicación, o
- es de nivel $signature$ o de nivel $signature/system$ y la aplicación ha sido firmada con el mismo certificado que aquella que definió el permiso, o
- es de nivel $signature/system$ y la aplicación ha sido firmada con el certificado del fabricante del dispositivo.

Formalmente, la definición de $appHasPermission$ es la siguiente:

$$
\begin{array}{l}
appHasPermission\ idapp\ p\ s = \\
(\exists l : [Perm], s.perms.state\ idapp = Value\ l \wedge p \in l) \vee \\
permExists\ p\ s \wedge \\
(\exists m : Manifest, isManifestOfApp\ idapp\ m\ s \wedge idP\ p \in m.use) \wedge \\
(\\
\quad (\exists lPerm : [Perm], s.environment.defPerms\ idapp = Value\ lPerm \wedge \\
\qquad p \in lPerm) \vee \\
\quad pl\ p = normal \vee \\
\quad (pl\ p = dangerous \wedge \\
\qquad (\exists (group : idGrp)(listgrp : [idGrp]), p.maybeGrp = Some\ group \wedge \\
\qquad\quad s.state.grantedPermGroups\ idapp = Value\ listgrp \wedge \\
\qquad\quad group \in listgrp)) \vee \\
\quad ((pl\ p = signature \vee pl\ p = signatureOrSys) \wedge \\
\qquad (\exists c : Cert, appCert\ idapp\ c\ s \wedge certOfDefiner\ p\ c\ s)) \vee \\
\quad (pl\ p = signatureOrSys \wedge \\
\qquad (\exists c : Cert, appCert\ idapp\ c\ s \wedge c = manufacturerCert)) \\
)
\end{array}
$$

2.9. Manejo de errores

Si quiere ejecutarse una acción en un sistema que no cumple con su precondición, debe retornarse un código de error que detalle el motivo. Tales códigos son los constructores del tipo $ErrorCode$:

$ErrorCode =$
$| \ app\_already\_installed \mid duplicated\_cmp\_id \mid duplicated\_perm\_id \mid cmp\_already\_defined$
$| \ perm\_already\_defined \mid faulty\_intent\_filter$
$| \ no\_such\_app \mid app\_is\_running \mid perm\_not\_in\_use \mid no\_such\_perm$
$| \ perm\_already\_granted \mid perm\_not\_dangerous \mid perm\_is\_grouped$
$| \ perm\_wasnt\_granted \mid group\_already\_granted \mid group\_not\_in\_use$
$| \ group\_wasnt\_granted \mid no\_such\_res \mid incorrect\_intent\_type \mid faulty\_intent \mid$
$intent\_already\_sent \mid no\_such\_intt \mid cmp\_is\_CProvider \mid instance\_not\_running$
$| \ a\_cant\_start\_b \mid not\_enough\_permissions \mid no\_CProvider\_fits$
$| \ CProvider\_not\_grantable$

La relación $ErrorMsg : System \rightarrow Action \rightarrow ErrorCode \rightarrow$ **Prop** especifica qué errores son factibles de ser retornados ante el intento de ejecución de una acción en un sistema. $ErorrMsg\ s\ a\ ec$ se verifica cuando el código de error ec es una respuesta aceptable al intentar ejecutar a en s.

En la tabla 2.4 se ilustra qué códigos de errores son aceptables ante cada violación a las precondiciones de las acciones $install$, $grant$, $revoke$, $grantPermGroup$ y $revokePermGroup$. Para ver los códigos de errores factibles de ser arrojados ante fallas en las precondiciones del resto de las acciones, referirse al Anexo.

Relación de ejecución con manejo de errores

Cuando en un estado válido se quiere ejecutar una acción cuya precondición es satisfecha, la especificación indica las características que debe tener el estado alcanzado luego de tal ejecución. En este trabajo se extendió dicha especificación para que se contemplen los casos en los cuales se intenta ejecutar una acción en un estado que no verifica su precondición. Para ello se define el tipo $Response$ inductivamente:

$Response =$
$\quad | \ ok$
$\quad | \ error : ErrorCode \rightarrow Response$

Ejecutar una acción $a : Action$ sobre un estado $s : System$ produce un nuevo estado s' y una respuesta $r : Response$. Ellos están relacionados a través de $Exec : System \rightarrow Action \rightarrow System \rightarrow Response \rightarrow$ **Prop** (y se nota $s \xrightarrow{a/r} s'$) si s es válido, cumple la precondición de a, s' cumple la postcondición de a partiendo de s y la respuesta es ok; o bien, si s es un sistema válido, la respuesta es un código de

error ec, s' es s, y s, a y ec cumplen la relación $ErrorMsg$:

$$\frac{validstate\ s \qquad Pre\ s\ a \qquad Post\ s\ a\ s'}{s \xrightarrow{a/ok} s'}$$

$$\frac{validstate\ s \qquad ErrorMsg\ s\ a\ ec}{s \xrightarrow{a/error\ ec} s}$$

La relación $Exec$ es en donde se captura la semántica del sistema. Un programa respeta el modelo si y sólo si verifica $Exec$ para todas las acciones y los estados.

Esta relación mantiene invariante la validez de un sistema, lo que significa que si se parte de un sistema válido y se ejecuta una acción, tanto ante una ejecución exitosa como ante una ejecución errónea el estado obtenido es válido. Esta noción de invarianza se capturó a través del siguiente teorema, el cual fue demostrado utilizando el asistente de pruebas Coq:

Teorema $validityIsInvariant$:
$\forall (s, s' : System)(act : Action)(r : Response)$,
$validstate\ s \implies s \xrightarrow{act/r} s' \implies validstate\ s'$

2.10. Propiedades de seguridad del modelo

Se presentan a continuación algunas propiedades del modelo de seguridad de Android que pueden resultar interesantes y han sido demostradas sobre la especificación descripta.

Estas propiedades complementan aquellas demostradas en [35], resaltando nuevas características discutibles relativas a la seguridad que son consecuencia de los cambios introducidos en la versión 6.0 de Android y contemplando la extensión de manejo de errores desarrollada en el presente trabajo. Las pruebas formales confeccionadas en Coq se encuentran disponibles en [20].

El sistema de permisos de Android 6.0 no brinda al usuario la granularidad de otorgar un subconjunto propio del conjunto de permisos que pertenecen a un grupo.

Teorema $noGranularityForIndividualPerms$:
$\forall (s, s' : System)(p : Perm)(g : idGrp)(a : idApp)$,
$p.maybeGrp = Some\ g \implies \neg s \xrightarrow{grant\ p\ a/ok} s'$

En un estado válido no puede otorgarse individualmente un permiso peligroso no agrupado.

El resultado de este teorema evidencia una característica discutible del nuevo sistema de permisos de Android 6.0. Por ejemplo, como los permisos peligrosos READ_CONTACTS y WRITE_CONTACTS pertenecen ambos al grupo de permisos CONTACTS, ninguno de ellos puede ser otorgado individualmente. En su lugar, debe permitírsele a la aplicación acceder al grupo de permisos CONTACTS, confiriéndole el derecho tanto de escribir como de leer la agenda del dispositivo.

Su demostración utiliza sencillamente el hecho de que la precondición de la acción $grant$ exige explícitamente que el permiso a otorgar no forme parte de ningún grupo.

A diferencia de lo que ocurría en versiones anteriores, una aplicación a la que nunca se le otorgó cierto permiso peligroso puede contar con él.

Teorema $hasPermissionAndNotGranted$:
$\exists(s : System)(p : Perm)(a : idApp)(permsA : [Perm]),$
$validstate\ s \wedge p.pl = dangerous \wedge$
$s.state.perms\ a = Value\ permsA \wedge p \notin permsA \wedge$
$p \notin getDefPermsForApp\ a\ s \wedge appHasPermission\ a\ p\ s$

Existe un estado válido en el cual una aplicación tiene un permiso peligroso ajeno (es decir, uno que ella no define), sin tenerlo individualmente otorgado.

Complementando el resultado anterior, este teorema formaliza otro punto señalado como débil de la especificación del nuevo sistema de permisos de Android. Siguiendo el ejemplo previo, en un estado válido una aplicación puede tener el permiso de leer contactos (READ_CONTACTS) sin tenerlo otorgado de manera individual (pues tiene derecho sobre todo el grupo de permisos CONTACTS).

Para demostrar este teorema se brinda un testigo en el cual a una aplicación que hace las veces de a se le ha otorgado el grupo de permisos g al que pertenece el permiso p, que es aquél por el cual se preguntará.

Para efectuar una llamada al sistema que requiera únicamente permisos de nivel $normal$, es suficiente listarlos como usados en el archivo de manifiesto de la aplicación.

Teorema $sacProtectedWithNormalPerms$:
$\forall(s : System)(access\_internet : SACall)(c : Cmp)(ic : iCmp)$

$, validstate\ s \implies$
$(\forall(p : Perm)(pIsSystem : isSystemPerm\ p)$
$(pIsRequired : permSAC\ p\ pIsSystem\ access\_internet),$
$p.pl = normal) \implies$
$(\forall(p : Perm)(pIsSystem : isSystemPermp)$
$(pIsRequired : permSAC\ p\ pIsSystem\ access\_internet),$
$p.idP \in (getManifestForApp\ (getAppFromCmp\ c\ s)\ s).use) \implies$
$s.state.running\ ic = Value\ c \implies$
$s \xrightarrow{call\ ic\ access\_internet/ok} s$

En un estado válido, si una llamada al sistema requiere sólo permisos normales, entonces cualquier instancia en ejecución cuya aplicación liste los permisos correspondientes como usados puede ejecutar la llamada.

En Android 6.0, para enviar información a través de la red se necesita contar con el permiso INTERNET. Como este permiso tiene nivel de peligro $normal$, toda aplicación que lo liste como usado en su archivo de manifiesto cuenta con él de manera implícita e irrevocable. Una vez más, esto ha sido criticado por el potencial filtrado de información que permite.

La demostración de este teorema se basa en que $appHasPermission$ se verifica para todo permiso normal que la aplicación liste como usado.

Listar un permiso peligroso como usado no asegura que la aplicación cuente con él indefectiblemente.

Teorema $dangerousPermMissing$:
$\exists(s : System)(p : Perm)(a : idApp),$
$validstate\ s \wedge p.pl = dangerous \wedge permExists\ p\ s$
$\wedge\ a \in s.state.apps \wedge p.idP \in (getManifestForApp\ a\ s).use \wedge$
$\neg appHasPermission\ a\ p\ s$

Existe un estado válido en el que una aplicación instalada no tiene un permiso peligroso existente a pesar de que lo lista como usado.

Este teorema marca una diferencia sustancial entre el sistema de permisos de Android 6.0 y las versiones anteriores de esta plataforma. Históricamente, al instalar una aplicación se le conferían todos los permisos que listara como usados, y éstos no podían ser revocados. Por lo tanto, si una aplicación estaba en ejecución, el programador tenía la certeza de que contaba con todos los permisos que hubiera listado en su archivo de manifiesto. Sin embargo, actualmente esta aserción es equivocada y por lo tanto el desarrollador debe contemplar tales casos y programar consecuentemente

de manera defensiva ante tales circunstancias.

En la demostración de este teorema se presenta un estado testigo en el cual está instalada una aplicación que lista cierto permiso existente, peligroso y no agrupado como usado pero no pertenece a la lista de permisos individualmente otorgados a ella.

Función	Significado
$permExists\ p\ s$	Esta proposición es verdadera cuando el permiso p existe en el sistema s, por ser de sistema o definido por una aplicación
$isManifestOfApp\ a\ m\ s$	La proposición se satisface si m representa el archivo de manifiesto de la aplicación a en s
$certOfDefiner\ p\ c\ s$	Se cumple cuando c es el certificado de quien definió el permiso p en s
$appCert\ a\ c\ s$	Se cumple cuando c es el certificado con el cual fue firmada la aplicación a en s
$manufacturerCert$	Es el certificado con el que el fabricante del dispoisitivo firma sus propias aplicaciones
$getDefPermsForApp\ a\ s$	Función que retorna la lista de permisos definidos por a en s. Es decir, los permisos que ella define en su manifest que no fueran de sistema
$isSystemPerm\ p$	Esta proposición se verifica cuando p es un permiso de sistema
$permSAC\ p\ pIsSystem\ sac$	Proposición que se verifica cuando p es un permiso necesario para ejecutar la llamada al sistema sac
$getAppFromCmp\ c\ s$	Función que retorna la aplicación que contiene al componente c en s
$getManifestForApp\ a\ s$	Función que retorna el manifiesto de la aplicación a en s
$isAppInstalledBool\ a\ s$	Función booleana que retorna verdadero si la aplicación a está instalada en el sistema s, así sea por acción del usuario o porque pertenece al conjunto de aplicaciones de fábrica
$has\_duplicates\ as$	Función booleana que retorna verdadero si la lista as contiene elementos repetidos
$authPermsBool\ m\ s$	Función booleana que retorna falso si existe entre los permisos declarados en m alguno cuyo identificador coincida con otro que ya hubiera sido declarado por otra aplicación del sistema s
$anyDefinesIntentFilterIncorrectly\ cs$	Función booleana que retorna verdadero si entre los componentes de la lista cs alguno define su filtro de *intents* de manera inconsistente
$addNewResCont\ a\ rc\ lRes$	Función que añade al mapa rc los recursos definidos de $lRes$ inicializándolos y asociándolos a a
$nonSystemUsrP\ m$	Función que filtra los permisos declarados en m que sean de sistema
$canGrant\ cp\ u\ s$	Predicado que se cumple si el proveedor de contenido cp permite la delegación de permisos sobre el recurso apuntado por la uri u en s
$existsRes\ cp\ u\ s$	Predicado que se verifica cuando la uri u pertenece al proveedor de contenido cp
$inApp\ c\ a\ s$	Esta proposición es cierta cuando el componente c pertenece a la aplicación a en s
$canStart\ c_1\ c_2\ s$	Proposición que se satisface cuando el componente c_1 tiene los permisos para iniciar al componente c_2 en el sistema s mediante el envío de un *intent* explícito

Cuadro 2.3: Funciones y predicados auxiliares

Acción	**Código de error**	**Falla**
$install\ a\ m\ c\ l$	$app\_already\_installed$	$isAppInstalled\ a\ s$
	$duplicated\_cmp\_id$	$has\_duplicates\ (map\ getCmpId\ m.cmp) = true$
	$duplicated\_perm\_id$	$has\_duplicates\ (map\ idP\ m.usrP) = true$
	$cmp\_already\_defined$	$\neg(\forall c : Cmp, c \in m.cmp \implies cmpNotInState\ c\ s)$
	$perm\_already\_defined$	$\neg authPerms\ m\ s$
	$faulty\_intent\_filter$	$\neg(\forall c : Cmp, c \in m.cmp \implies cmpDeclareIntentFilterCorrectly\ c)$
$grant\ p\ a$	$perm\_not\_in\_use$	$\neg(\exists m : Manifest, s.environment.manifest\ a = Value\ m \wedge p.idP \in m.use)$
	no_such_perm	$\neg(isSystemPerm\ p \vee usrDefPerm\ p\ s)$
	$perm\_already\_granted$	$\exists lPerm : [Perm], s.state.perms\ a = Value\ lPerm \wedge p \in lPerm$
	$perm\_not\_dangerous$	$p.pl <> dangerous$
	$perm\_is\_grouped$	$p.maybeGrp <> None$
$revoke\ p\ c$	$perm\_wasnt\_granted$	$\neg pre\_revoke\ p\ c\ s$
$grantPermGroup\ g\ a$	no_such_app	$a \notin s.state.apps$
	$group\_already\_granted$	$\exists lGrp : [idGrp], s.state.grantedPermGroups\ a = Value\ lGrp \wedge g \in lGrp$
	$group\_not\_in\_use$	$\neg(\exists(m : Manifest)(p : Perm), s.environment.manifest\ a = Value\ m \wedge p.idP \in m.use \wedge (isSystemPerm\ p \vee usrDefPerm\ p\ s) \wedge p.maybeGrp = Someg \wedge p.pl = dangerous)$
$revokePermGroup\ g\ c$	$group\_wasnt\_granted$	$\neg pre\_revokeGroup\ g\ c\ s$

Cuadro 2.4: Tabla de errores de las acciones $install$, $grant$, $revoke$, $grantPermGroup$ y $revokePermGroup$

Capítulo 3

Hacia un monitor de seguridad certificado

3.1. Motivación

La especificación del modelo de seguridad de Android es en sí misma una herramienta teórica muy importante que permite analizar formalmente propiedades sobre ella y señalar errores de seguridad presentes en la plataforma. Incluso, como se verá más adelante, puede utilizarse como fuente de casos de prueba críticos con los cuales se maximiza la posibilidad de encontrar errores de programación durante el desarrollo del sistema o directamente en los sistemas ya desarrollados.

Contar con una implementación cuya corrección haya sido demostrada permitiría comparar un estado de un sistema real con un estado da la implementación correcta y verificar automáticamente que sean equivalentes (definiendo una noción de equivalencia adecuada para el dominio de aplicación). Esto permitiría construir un monitor de la plataforma real que verifique la equivalencia de los sucesivos estados que ésta atraviesa con la traza de estados que recorre el programa certificado. Es con este objetivo que se desarrolló en Coq un programa que modifica el sistema de seguridad según las acciones ejecutadas y se demostró que se comporta de acuerdo al modelo.

3.2. La implementación

Se definió un tipo $Result$ que contiene el resultado retornado por el programa ante una acción:

$$
\begin{aligned}
&Result = \{ \\
&\quad response : Response, \\
&\quad system : System \\
&\}
\end{aligned}
$$

La función principal del programa es $step : System \rightarrow Action \rightarrow Result$, que retorna, dados un sistema y una acción a ejecutar sobre él, un resultado (es decir, un elemento de tipo $Result$) que contiene la respuesta arrojada por el sistema ante la solicitud de tal operación (de tipo $Response$) y el nuevo estado (de tipo $System$) obtenido por ejecutarla.

En realidad, la función $step$ no es más que un despachador de acciones, aplicando pattern matching sobre la acción a ejecutar e invocando a la función correspondiente:

```
step (s:System) (a:Action) : Result =
   match a with
   | install app m c lRes ⇒ install_safe app m c lRes s
   | uninstall app ⇒ uninstall_safe app s
   | grant p app ⇒ grant_safe p app s
  ...
   end.
```

Para cada acción existe una función (por ejemplo $install\_safe$ para la acción $install$) que se encarga de invocar a una función auxiliar que evalúa la precondición en el estado, y retornar un resultado conteniendo o bien el error correspondiente y el estado original si ésta no se cumple; o bien la respuesta ok y el nuevo estado en caso de que la precondición se verifique:

```
install_safe (app:idApp) (m:Manifest) (c:Cert) (lRes : list
    Res) (s:System) : Result =
   match install_pre app m c lRes s with
   | Some ec ⇒ result (error ec) s
   | None ⇒ result ok (install_post app m c lRes s)
   end.
```

De igual manera, por cada acción existe una función que se encarga de evaluar su precondición en cada estado y retornar, si la misma no se cumple, un código de error adecuado y, si por el contrario, la precondición es satisfecha, nada ($None$). Por ejemplo, la función para evaluar la precondición de $install$ es la siguiente:

```
install_pre (app:idApp) (m:Manifest) (c:Cert) (lRes : list
    Res) (s:System) : option ErrorCode =
   if isAppInstalledBool app s then (Some
      app_already_installed) else
   if (has_duplicates idCmp idCmp_eq (map getCmpId (cmp m)))
      then (Some duplicated_cmp_id) else
   if (has_duplicates idPerm idPerm_eq (map idP (usrP m)))
      then (Some duplicated_perm_id) else
   if (existsb (fun c ⇒ cmpInStateBool c s) (cmp m)) then
      (Some cmp_already_defined) else
   if (negb (authPermsBool m s)) then (Some
      perm_already_defined) else
   if (anyDefinesIntentFilterIncorrectly (cmp m)) then (Some
      faulty_intent_filter) else
   None.
```

En el código anterior puede observarse el flujo de control de la función $install\_pre$:

- Si la aplicación ya se encuentra instalada, retorna el error $app\_already\_installed$.
- Si declara dos componentes con el mismo identificador (es decir, si la lista resultado de aplicar la función $getCmpId$ a $m.c$, siendo m su archivo de manifiesto, contiene duplicados), retorna el error $duplicated\_cmp\_id$.
- Si declara dos permisos con el mismo identificador (de manera análoga al paso anterior), entonces retorna el error $duplicated\_perm\_id$.
- Si intenta definir un componente que ya existe en el sistema, entonces retorna el error $cmp\_already\_defined$.
- Si intenta definir un permiso que ya existe en el sistema, entonces retorna el error $perm\_already\_defined$.
- Si alguno de sus componentes define incorrectamente su filtro de *intent*, entonces retorna el error $faulty\_intent\_filter$. Si no, retorna que no hay errores y la precondición se considera satisfecha.

Además, por cada acción existe una tercer función que se encarga de retornar el estado correspondientemente alterado. Por ejemplo, para la acción $install$ se definió la función $install\_post$:

```
install_post (app:idApp) (m:Manifest) (c:Cert) (lRes : list
    Res) (s:System) : System =
   let oldstate := state s in
   let oldenv := environment s in
   sys (st
         (app :: (apps oldstate))
         (map_add idApp_eq (grantedPermGroups oldstate) app
            nil)
         (map_add idApp_eq (perms oldstate) app nil)
         (running oldstate)
         (delPPerms oldstate)
         (delTPerms oldstate)
         (addNewResCont app (resCont oldstate) lRes)
         (sentIntents oldstate)
      )
      (env
         (map_add idApp_eq (manifest oldenv) app m)
         (map_add idApp_eq (cert oldenv) app c)
         (map_add idApp_eq (defPerms oldenv) app
            (nonSystemUsrP m))
         (systemImage oldenv)
      ).
```

Particularmente en el caso de $install\_post$, se antepone la aplicación a instalar a la lista de las aplicaciones ya instaladas, se inicializan como vacías las listas de permisos y grupos otorgados a ella, se inicializan sus recursos, y se agrega la aplicación instalada a los dominios de las funciones parciales $manifest$, $cert$ y $defPerms$ del sistema, asociadas a su archivo manifiesto, el certificado con el que fue firmada, y los permisos que define que no sean de sistema. El resto de los campos no se alteran.

3.3. Corrección

El teorema que certifica la corrección de $step$ respecto al modelo desarrollado, planteado y probado utilizando el asistente de pruebas Coq, es el siguiente:

Teorema $stepIsSound$:
$\forall (s : System)(act : Action)$,
$validstate\ s \implies s \xrightarrow{act/(step\ s\ act).response} (step\ s\ act).system$

Una vez más, la prueba de este teorema parte de un análisis por caso según la acción, probando que cada una de ellas es correcta:

```
Theorem stepIsSound : forall (s:System) (act:Action),
      validstate s → exec s act (system (step s act))
          (response (step s act)).
Proof.
   intros.
   destruct act.
   exact (installIsSound s i m c l H).
   exact (uninstallIsSound s i H).
   exact (grantIsSound s i p H).
...
Qed.
```

Por cada acción se definió un lema que muestra su corrección. Por ejemplo, el lema en donde se demuestra la corrección de la acción $install$ es el siguiente:

Lema $installIsSound$:
$\forall(s : System)(a : idApp)(m : Manifest)(c : Cert)(lRes : [Res]),$
$validstate\ s \implies$
$s \xrightarrow{install\ a\ m\ c\ lRes/(step\ s\ (install\ a\ m\ c\ lRes)).response} (step\ s\ (install\ a\ m\ c\ lRes)).system$

En general, esta prueba hace uso de dos lemas auxiliares: uno demostrando que, si la precondición se cumple, entonces el estado resultado satisface la postcondición de tal acción en el modelo:

Lema $postInstallCorrect$:
$\forall(s : System)(a : idApp)(m : Manifest)(c : Cert)(lRes : [Res]),$
$pre\ (install\ a\ m\ c\ lRes)\ s \implies$
$validstate\ s \implies$
$post\_install\ a\ m\ c\ lRes\ s\ (install\_post\ a\ m\ c\ lRes\ s)$

y otro, demostrando que si la precondición no se cumple, entonces se retorna un código de error válido según la relación $ErrorMsg$:

Lema $notPreInstallThenError$:
$\forall(s : System)(a : idApp)(m : Manifest)(c : Cert)(lRes : [Res]),$
$\neg pre\ (install\ a\ m\ c\ lRes)\ s \implies$
$validstate\ s \implies$
$\exists ec : ErrorCode, (step\ s\ (install\ a\ m\ c\ lRes)).response = error\ ec \wedge$
$ErrorMsg\ s\ (install\ a\ m\ c\ lRes)\ ec \wedge s = (step\ s\ (install\ a\ m\ c\ lRes)).system$

3.4. Encadenando acciones

Al ejecutar una acción a sobre un sistema s se obtiene un sistema resultado s', modificado por la acción ejecutada si su precondición se cumplía. En este nuevo estado alcanzado, a su vez, puede ejecutarse otra acción a' resultando en otro sistema s'' posiblemente diferente de los dos anteriores. Esto es equivalente a ejecutar secuencialmente la lista de acciones $[a, a']$ sobre el estado inicial s, atravesando la traza de estados $[s', s'']$.

Para capturar esta noción de trazas se desarrolló una función $trace$, que dada una lista de acciones y un estado inicial, retorna la lista de sistemas resultado de aplicar las acciones de manera secuencial:

```
trace (s:System) (actions:[Action]) {struct actions}
    :[System] =
match actions with
   | nil ⇒ nil
   | action::rest ⇒ let sys := (system (step s action)) in
      (sys :: trace sys rest)
end.
```

El análisis de la ejecución secuencial de acciones permite observar nuevas propiedades del sistema. Algunas de ellas se describen en la sección 3.5

3.5. Propiedades sobre trazas

En esta sección se listan algunas propiedades de seguridad que se verifican en las trazas de ejecución a partir de un estado inicial válido sobre la implementación certificada. Ellas han sido postuladas y demostradas utilizando el asistente de pruebas Coq, cuyos detalles pueden encontrarse en [20].

En Android 6.0 la revocación de un permiso no invalida las delegaciones de permisos sobre recursos protegidos por él que la aplicación pudo haber efectuado.

Teorema $delegateGrantPRevoke$:
$\forall (s : System)(p : Perm)(a, a' : idApp)(ic, ic' : iCmp)$
$(c, c' : Cmp)(u : uri)(cp : CProvider),$
$validstate\ s \implies (step\ s\ (grant\ p\ a)).response = ok \implies$
$getAppFromCmp\ c\ s = a \implies$
$getAppFromCmp\ c'\ s = a' \implies$
$s.state.running\ ic = Value\ c \implies$
$s.state.running\ ic' = Value\ c' \implies$
$canGrant\ cp\ u\ s \implies$
$existsRes\ cp\ u\ s \implies$
$cp.expC = Some\ true \implies$
$cp.readE = Some\ p \implies$

let $opsResult := trace\ s\ [grant\ p\ a, grantP\ ic\ cp\ a'\ u\ Read, revoke\ p\ a]$ in
$(step\ (last\ opsResult\ s)\ (read\ ic'\ cp\ u)).response = ok$

En todo estado válido, si se le otorga correctamente un permiso p a una aplicación a, uno de cuyos componentes se encuentra ejecutándose con identificador ic; luego este componente en ejecución delega un permiso de lectura a una aplicación a' sobre un recurso de un proveedor de contenido cp cuya lectura está protegida por p, y más tarde se le quita el permiso p a a, posteriormente si un componente de a' intenta leer dicho recurso de cp, podrá hacerlo correctamente.

La demostración de este teorema parte del hecho de que la accion $grant\ p\ a$ sobre el estado inicial es correcta, de lo que se deduce que en el siguiente estado a cuenta con el permiso p. Por lo tanto, puede redelegar este permiso a a' en el paso posterior correctamente, modificando la función $delPPerms$ del estado del sistema (ahora a' tiene derecho de lectura sobre el recurso u de cp). Esta entrada de la función no es quitada al revocársele el permiso p a a y por lo tanto ic' (que es una instancia en ejecución de un componente de a') cumple la precondición de $read$ para u en cp.

En el sistema operativo analizado, la única manera en que una aplicación puede contar con un permiso peligroso no agrupado con el que no contaba previamente, es mediante un otorgamiento explícito.

Teorema $ifPermThenGranted$:
$\forall(initState, lastState : System)(a : idApp)$
$(p : Perm)(l : [Action]),$
$validstate\ initState \implies$
$a \in initState.state.apps \implies$
$p.pl = dangerous \implies$
$p.maybeGrp = None \implies$
$appHasPermission\ a\ p\ lastState \implies$
$\neg appHasPermission\ a\ p\ initState \implies$
$uninstall\ a \notin l \implies$
$last\ (trace\ initState\ l)\ initState = lastState \implies$
$grant\ p\ a \in l$

Para todo estado inicial válido en el cual una aplicación a no tiene un permiso peligroso no agrupado p, si al final de una serie de operaciones a pasa a contar con tal permiso a pesar de nunca haber sido desinstalada, entonces en algún momento le fue otorgado.

En la demostración de este teorema se utiliza el hecho de que mientras la aplicación no sea desinstalada no puede cambiar su manifiesto (por lo que nunca será quien defina el permiso p), y el permiso será siempre no agrupado y peligroso; por lo que

la única manera de lograr que $appHasPermission$ se verifique es agregando p a la lista de permisos individualmente otorgados a a. Y la única manera de lograr esto es mediante la acción $grant$.

Si una aplicación contaba con un permiso que le fue revocado, sólo su reotorgamiento hará que cuente con él nuevamente.

Teorema $revokeAndNotGrant$:
$\forall(initState, sndState, lastState : System)$
$(a : idApp)(p : Perm)(l : [Action])$,
$validstate\ initState \implies$
$p.pl = dangerous \implies$
$p.maybeGrp = None \implies$
$\neg(\exists lPerm : [Perm],$
$initState.environment.defPerms\ a = Value\ lPerm \wedge$
$p \in lPerm) \implies$
$sndState = (step\ initState\ (revoke\ p\ a)).system \implies$
$(step\ initState\ (revoke\ p\ a)).response = ok \implies$
$uninstall\ a \notin l \implies$
$grant\ p\ a \notin l \implies$
$last\ (trace\ sndState\ l)\ sndState = lastState \implies$
$\neg appHasPermission\ a\ p\ lastState$

Si en un estado inicial válido se le revoca correctamente un permiso p a una aplicación a, mientras la aplicación no sea desinstalada ni el permiso reotorgado, la aplicación no contará con él.

La demostración es sencillamente negar la conclusión, utilizar el hecho de que luego de la correcta revocación de p a a, a no tiene el permiso p (pues si a no necesitara a p en su lista de permisos otorgados para contar con él, entonces jamás se le podría haber otorgado en primer lugar) y negar la conclusión para alcanzar las premisas del teorema anterior. Luego, al aplicarlo, se deduce que el otorgamiento de p a a debe ser una acción efectuada, lo cual se contradice con las hipótesis originales.

Algunas aserciones que un desarrollador podía considerar válidas en versiones anteriores de Android, ya no son ciertas en su distribución más reciente. Particularmente, un componente en ejecución puede tener el derecho de iniciar a otro en cierto estado y posteriormente perder este privilegio, aún cuando no se haya desinstalado la aplicación y ni siquiera detenido su proceso.

Teorema $revokeCanStart$:
$\forall(initState : System)(l : [Action])(a_1, a_2 : idApp)$

$(c_1 : Cmp)(act : Activity)(p : Perm),$
$validstate\ initState \implies$
$p.pl = dangerous \implies$
$p.maybeGrp = None \implies$
$a_1 <> a_2 \implies$
$\neg(\exists lPerm : [Perm],$
$initState.environment.defPerms\ a_1 = Value\ lPerm \wedge p \in lPerm) \implies$
$inApp\ c_1\ a_1\ initState \implies$
$inApp\ (cmpAct\ act)\ a_2\ initState \implies$
$act.cmpEA = Some\ p \implies$
$canStart\ c_1\ (cmpAct\ act)\ initState \implies$
$\exists l : [Action], uninstall\ a_1 \notin l \wedge uninstall\ a_2 \notin l \wedge$
$\neg canStart\ c_1\ (cmpAct\ act)\ (last\ (trace\ initState\ l)\ initState)$

En todo estado válido en donde un componente $c1$ tiene la potestad de iniciar a una actividad $c2$ de otra aplicación protegida por un permiso peligroso no agrupado p que no es definido por la aplicación en donde se encuentra $c1$, existen ciertas acciones que hacen que pierda la posibilidad de hacerlo a pesar de que ninguna de las dos aplicaciones haya sido desinstalada.

La lista de acciones que se ofrece como testigo consiste sencillamente en la revocación de p a a_1. Como p es peligroso, no agrupado, no definido por a_1 y sin embargo a_1 cuenta con él (pues uno de sus componentes puede iniciar una actividad por éste protegida), entonces p debe estar individualmente otorgado a a_1. Por lo tanto, su revocación no falla y consecuentemente a_1 ya no contará con tal permiso y sus componentes (en particular c_1) no podrán iniciar a la actividad por él protegida.

Capítulo 4

Monitor de detección de errores

4.1. Habilidades del monitor

Una implementación del sistema de seguridad de Android, cuya corrección respecto al modelo ha sido formalmente demostrada, puede ser utilizada para comparar los resultados de ejecutar una acción sobre un estado en una plataforma a analizar y ejecutar la misma acción sobre el mismo estado en el programa correcto. Esto abre las puertas a la posibilidad de desarrollar un programa que monitoree las acciones efectuadas en un sistema real y evalúe si se cumplen propiedades de interés sobre ella, contando con la implementación certificada para efectuar comparaciones.

Tal monitor se ejecutaría junto a la plataforma a controlar, como un hipervisor pasivo. Debe ser capaz a cada instante de interpretar el estado del sistema y construir un estado equivalente en el modelo desarrollado. A esta operación se la denominará "extraer" el estado real, y se confiará en que haya sido correctamente desarrollada, pues es la manera en que se extrapola la información como la almacena la plataforma real al registro $System$ de Coq con el cual se computará.

Por ejemplo, si el sistema bajo supervisión fue programado en lenguaje C, posiblemente exista un arreglo con los identificadores de las aplicaciones instaladas dentro de un struct que almacena el estado del sistema:

```
struct System {
  int installedApps[MAX_APPS];
  int installedAppsCount;
  ...
};
```

La lista de estos identificadores tomará el lugar de $state.apps$ en el sistema extraído.

Además de la extracción del estado de la plataforma, el monitor debe conocer cuál es la representación en la implementación supervisada de los códigos de errores

modelados. Siguiendo el ejemplo anterior, si en el código fuente de la plataforma se definen las siguientes macros:

```
#define APP_ALREADY_INSTALLED -1
#define DUPLICATED_CMP_ID -2
#define CMP_ALREADY_DEFINED -3
...
```

Entonces el monitor debe conocer que si la ejecución de una acción arroja el código de error -2 en la plataforma real, su equivalente en el modelo es el constructor $duplicated\_cmp\_id : ErrorCode$ (presentado en la sección 2.9). Por último, el monitor debe tener la potestad de interceptar las llamadas a las funciones que procesan las acciones de seguridad modeladas capaces de mutar al sistema, y replicarlas en la implementación certificada.

Continuando con el ejemplo, suponiendo las siguientes declaraciones de tipos:

```
typedef struct System System;
typedef int permission;
typedef int appId;
```

Si existe una función con la firma:

```
void grantPermission(System* s, permission p, appId app);
```

Al efectuarse una llamada a ella, el monitor debe saber que esto es equivalente a ejectuar la acción $grantPermission$ en el sistema modelado del permiso correspondiente a la aplicación correspondiente.

4.2. Su comportamiento

Con todos estas herramientas, la actividad del monitor consiste, al detectarse la llamada a una función que represente la ejecución de una acción modelada, en extraer el estado previo a su ejecución, replicar en el estado extraído la acción efectuada, extraer el estado y el código de error que fueron consecuencia de la ejecución de la operación en la plataforma real, y por último compararlos con el resultado arrojado por la ejecución de la acción en el sistema extraído.

Estos pasos se grafican en la figura 4.1, en donde los círculos azules son datos de la plataforma real (RS y $RRes$, respectivamente el estado de la plataforma supervisada y el resultado en ella de la acción) mientras que los círculos rojos son datos del modelo Coq (CS, $CRes$ y $CRes'$, respectivamente el estado extraído a Coq de RS, el resultado de ejecutar en él la acción, y la representación de la extrapolación a Coq de $RRes$).

Esta secuencia puede repetirse sucesivamente a lo largo de todo el ciclo de ejecución del sistema, dando como resultado un programa que monitorea la veracidad de las propiedades de interés en tiempo real.

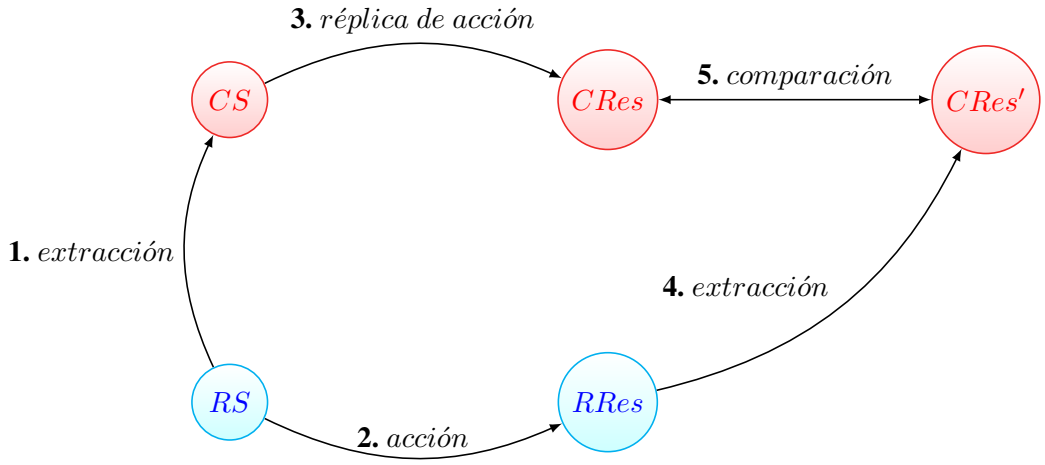

Figura 4.1: Ejecución de monitoreo

4.3. Un monitor configurable

Las comprobaciones a realizar y cómo debe comportarse el monitor cuando ellas no se cumplen es decisión de quien lo configure. En las siguientes subsecciones se listan tres casos de uso del monitor, y cómo llevarlos a cabo.

Verificar validez de estados atravesados

Puede programarse en Coq una función $validstateBool : System \rightarrow bool$ y aplicarla sistemáticamente sobre los estados extraídos de la plataforma. Si esta función verifica que

$$\forall s, validstate\ s \implies validstateBool\ s = true \tag{4.1}$$

entonces si en algún momento la función $validstateBool$ retorna falso, puede concluirse que el sistema alcanzado no cumple con la definición de estado válido descripta en la sección 2.5 y por ende, al haber sido demostrada la invarianza de la validez del estado ante la ejecución de acciones (ver sección 2.5), entonces se deduce que la plataforma no respeta el modelo. Tal situación puede ser considerada una potencial amenaza grave a la seguridad del sistema y el monitor podría actuar deteniendo su ejecución para proteger el dispositivo.

Comprobar la igualdad de permisos otorgados

Cabe notar que para verificar la propiedad anterior no es necesario contar con una implementación correcta del modelo. De hecho, es suficiente con extraer los estados que el sistema atraviesa y computar en ellos la función $validstateBool$.

En cambio, el programa de referencia es de utilidad cuando se pretende validar, por ejemplo, que los permisos otorgados en la plataforma supervisada sean los mismos que los permisos otorgados en el programa certificado. Para analizar tal propiedad, puede ser útil definir dos funciones auxiliares:
$lists\_permuted\_bool : [A] \rightarrow [A] \rightarrow bool$ y $samePermissions : System \rightarrow System \rightarrow bool$. La primera de ellas debe retornar verdadero si y sólo si las dos listas que conforman sus argumentos poseen exactamente los mismos elementos, mientras que la segunda debe cumplir que:

$$samePermissions\ s_1\ s_2 = lists\_permuted\_bool\ s_1.state.apps\ s_2.state.apps$$

es decir, debe retornar verdadero sólo cuando los dos sistemas que recibe como argumento tienen las mismas aplicaciones instaladas. La propiedad se considera verificada cuando los dos sistemas en comparación cumplen $samePermissions$ y además para cada una de las aplicaciones en ellas las funciones $s_1.state.perms$ y $s_2.state.perms$ retornen listas con iguales elementos, y así también lo hagan las listas retornadas por las funciones $s_1.state.grantedPermGroups$ y $s_2.state.grantedPermGroups$ aplicadas a ellas.

La comprobación de que los sistemas extraídos son válidos utilizando la función $validstateBool$ parece ser imperante para lograr resultados satisfactorios, pues una vez alcanzado un estado corrupto (por ejemplo, uno en el cual el dominio de $s.state.grantedPermGroups$ no es exactamente el conjunto de elementos pertenecientes a $s.state.apps$), el chequeo de otras propiedades probablemente carezca de utilidad.

Constatar corrección paso a paso

Por último, supongamos que se quiere utilizar al monitor para verificar que efectivamente el sistema en observación se comporta correctamente respecto al modelo desarrollado. Si los resultados sucesivamente arrojados por la plataforma real coinciden exactamente con aquellos emulados paso a paso, entonces puede concluirse efectivamente que la misma está funcionando como la especificación lo indica.

Sin embargo, la implementación certificada codificada en el presente trabajo es sólo una de las posibles implementaciones correctas: la desigualdad de resultados ante cierta acción no necesariamente implica una violación al modelo. Tal situación se da, por ejemplo, con la acción $install$: la especificación de $install\ a\ m\ c\ lRes$ requiere —entre otras cosas— que si se cumple su precondición, en el nuevo estado alcanzado s' la aplicación a forme parte de la lista $s'.state.apps$ (más detalles acerca de la semántica de $install$ pueden encontrarse en la sección Semántica formal de las acciones del sistema). En la implementación certificada se optó por simplemente anteponer a a la vieja lista de aplicaciones instaladas, pero un sistema que elija anexarla al final de ella es igualmente correcto. Para superar este inconveniente, el usuario del

monitor puede definir una función de equivalencia entre dos sistemas y dos resultados obtenidos luego de ejecutar la misma acción en el mismo estado inicial:
$eq\_res : Action \rightarrow System \rightarrow Result \rightarrow Result \rightarrow bool$
a la cual notaremos $\sim$ y diremos que para dos resultados r_1 y r_2, una acción a y un estado s,

$$r_1 \underset{a,s}{\sim} r_2 \iff eq\_res\ a\ s\ r1\ r2 = true$$

Si además el usuario del monitor demuestra que para toda acción act, sistema s y resultados r_1 y r_2, se cumple la siguiente propiedad :

$$s \xrightarrow{act/r_1.response} r_1.system \implies r_1 \underset{act,s}{\sim} r_2 \implies s \xrightarrow{act/r_2.response} r_2.system$$

entonces puede utilizar el monitor para verificar paso a paso que la plataforma se está comportando correctamente: basta evaluar que a partir del mismo estado, los resultados arrojados por la implementación certificada y el programa bajo observación sean equivalentes, pues como se ha demostrado en la sección 3.3, el programa certificado cumple la relación $Exec$.

La idea detrás de la función de equivalencia es permitir la reutilización de la demostración de que el programa $step$ es correcto, teniendo únicamente que ajustar las diferencias técnicas entre él y la plataforma real que surgen a partir del no determinismo de la relación de especificación. Naturalmente, cuanto más diverja la implementación certificada de la implementación real, menos podrá reusarse de la prueba original y mayor será el trabajo que habrá que invertir en codificar la función de equivalencia y demostrar la propiedad 4.1.

Continuando con el ejemplo anterior, si el instalar una aplicación a resulta en su agregado al final de la lista de aplicaciones instaladas en lugar de ubicarla en la primera posición como ocurre en la implementación certificada, esto deberá ser tenido en cuenta a la hora de desarrollar el caso $eq\_res\ install$ para cuando las respuestas son ok:

```
Definition eq_res (a:Action) (s:System) (r0:Result)
    (r1:Result) : bool :=
match a with
| install a c m lRes ⇒ match response r0 with
  | ok ⇒ response_eq (response r1) ok &&
    let s0 := system r0 in
    let s1 := system r1 in
       environment_eq (environment s0) (environment s1) &&
       lists_permuted_bool (apps (state s0)) (apps (state
          s1)) &&
       grantedPermGroups_eq (grantedPermGroups (state s0))
          (grantedPermGroups (state s1)) &&
    ...
```

El código presentado sugiere una idea de cómo se podría definir la función de equivalencia para el escenario en cuestión, suponiendo que la diferencia resaltada es la única entre ambas implementaciones: si la acción es $install$ y el resultado en r_0 fue ok (es decir, la ejecución de la acción fue correcta), entonces el resultado en r_1 también debe ser ok. Además, los entornos deben ser exactamente iguales (se supone que esto verifica la función $environment\_eq : Environment \rightarrow Environment \rightarrow bool$), y se requiere que las listas de aplicaciones instaladas tengan los mismos elementos, utilizando la ya presentada función $lists\_permuted\_bool$.

Por último, debe verificarse la igualdad del resto de los componentes de $state$ por medio de funciones como $grantedPermGroups\_eq : (idApp \twoheadrightarrow [idGrp]) \rightarrow (idApp \twoheadrightarrow [idGrp]) \rightarrow bool$, que computan la igualdad entre sus componentes.

Utilizando tal relación de equivalencia, debería ser factible demostrar la propiedad 4.1, pues para todo par de listas de aplicaciones $l_0, l_1 : [idApp]$ e id de aplicación $a : idApp$, se cumple que:

$$lists\_permuted\_bool\ l_0\ l_1 \implies a \in l_0 \iff a \in l_1$$

Resultado suficiente para probar que si una lista satisface la postcondición de $install$, también la otra lo hace.

Capítulo 5

Conclusiones y trabajo futuro

La última versión del sistema operativo Android ha introducido cambios sustanciales en sus mecanismos de seguridad con el claro objetivo de mejorar la experiencia de usuario. Sin embargo, modificaciones de tal profundidad traen aparejados nuevos vectores de ataque cuyo estudio es trascendental teniendo en cuenta que en Agosto del presente año el 15 % de los dispositivos Android ya están corriendo dicha versión [14], y sólo cabe esperar que este número vaya en aumento.

Junto con los primeros programas informáticos surgieron los primeros errores de programación. En los albores de la computación esto solía significar en la generalidad de los casos el cierre inesperado del programa, la pérdida de algunas horas de trabajo no guardado o el reinicio esporádico de los equipos. Sin embargo, la masificación de la computadora personal y el advenimiento de Internet convirtieron a los errores de programación más sutiles en vulnerabilidades que dejan al equipo anfitrión a merced de los hackers, agravando las consecuencias de un programa incorrecto.

Esto se refleja en un renovado interés por los métodos formales para asegurar la corrección del software crítico. Un ejemplo de ello es el caso de los ingenieros de la Agencia de Proyectos de Investigación Avanzados de Defensa de Estados Unidos [12], quienes en 2015 se abocaron a la construcción de un drone cuyo programa de control ha sido desarrollado con estos métodos y no pudo ser hackeado por la prestigiosa empresa de seguridad informática “Red Team” [16].

Cabe afirmar que el software que controla la privacidad de miles de millones de personas alrededor del globo puede ser considerado software crítico, y es en este sentido que los avances hacia un programa correcto pueden tener un impacto tangible profundamente positivo en el corto y mediano plazo.

En Agosto de 2016 se presentó la version 7.0 (Nougat) del sistema operativo Android [2] en el cual los directorios privados de las aplicaciones tienen acceso restringido (especificado por la máscara de permisos 0700), y se desalienta fuertemente

la modificación de los permisos UNIX sobre los archivos de estos directorios. En su lugar, se recomienda utilizar un proveedor de contenido cuyo acceso puede ser moderado a través del sistema de permisos de Android, tal y como se presentó a lo largo de esta tesina [1]. No hay cambios de seguridad que afecten el modelo y la implementación presentados en este trabajo.

Haber obtenido un programa cuya corrección respecto al modelo fue demostrada formalmente no es más que el puntapié inicial que permitiría el desarrollo de nuevas herramientas, siendo una de ellas el monitor de propiedades detallado en el capítulo 4. El lenguaje OVAL estandariza la transferencia de información entre herramientas de seguridad definiendo un lenguaje para codificar detalles del sistema observado [29], lo cual exime a otras aplicaciones de la necesidad de proceder a la recolección de datos manualmente, brindándoles la posibilidad de utilizar la información provista para efectuar sus análisis [31]. En este estándar se encuentra disponible un esquema XML (XSD, por sus siglás en inglés "XML Schema Definition" [38]) que describe la estructura de los archivos que detallan el estado de un dispositivo Android [28]. Utilizando una herramienta que implemente este lenguaje, como el intérprete OVAL [30], puede obtenerse la descripción del estado del sistema operativo en formato XML [15]. Además, al contar con la definición de su esquema, puede automarizarse el desarrollo de un intérprete del archivo resultante para su conversión al lenguaje en donde se computarán las propiedades de seguridad a verificar. Particularmente, si el lenguaje elegido para la exportación de la implementación certificada es Haskell [21], la herramienta XsdToHaskell de la colección de utilidades HaXml [22] es capaz de generar un parser para la definición del esquema de Android que al procesar la descripción en XML produce el árbol que representa el estado del dispositivo. De esta manera, sólo resta especificar la conversión de los tipos de datos automáticamente generados por el parser a los tipos automáticamente extraídos de Coq para que finalmente puedan efectuarse las computaciones que examinarán la presencia de vulnerabilidades y corrupciones en la plataforma monitoreada en tiempo real.

La implementación certificada permite a su vez utilizar las técnicas de testing basado en modelos [36], cuyo circuito se representa en la figura 5.1, para efectivamente derivar a partir del propio modelo presentado en este trabajo casos de test que representen puntos críticos capaces de señalar potenciales fallas en una implementación de producción. Una vez obtenidos los casos de test concretos, puede utilizarse para la etapa de comprobación un mecanismo similar a aquél del monitor expuesto en el capítulo 4, solo que en lugar de replicar las acciones que el usuario ejecute en el sistema, deberían ejecutarse las acciones de test correspondientes. Luego se procedería a efectuar las computaciones definidas para intentar desenmascarar errores de implementación, comparando con el resultado arrojado por la implementación certificada como oráculo de referencia cuando se lo considere necesario.

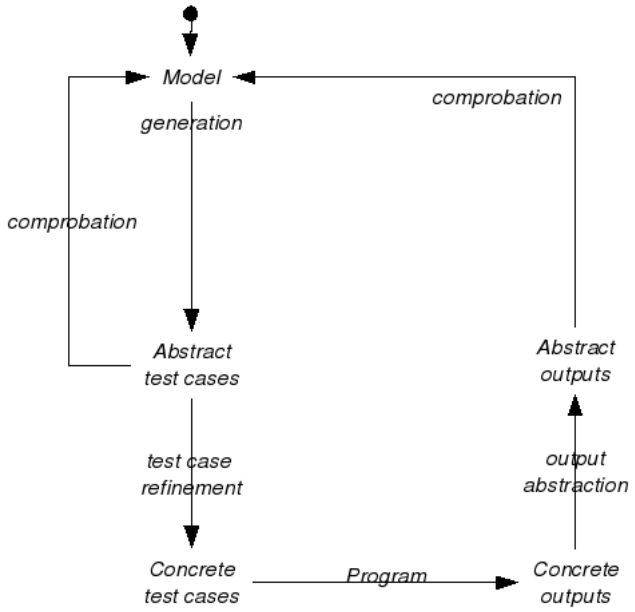

Figura 5.1: Circuito de testing basado en modelos [36]

Bibliografía

[1] *Android 7.0 Behavior Changes — Android Developers*. https://developer.android.com/about/versions/nougat/android-7.0-changes.html#perm. (Visitado 02-09-2016).

[2] *Android - Nougat*. https://www.android.com/versions/nougat-7-0/. (Visitado 02-09-2016).

[3] *Android Runtime*. https://source.android.com/devices/tech/dalvik. (Visitado 02-09-2016).

[4] *Android system permissions*. https://developer.android.com/guide/topics/security/permissions.html. (Visitado 02-09-2016).

[5] *AndroidManifest.xml*. https://developer.android.com/samples/RuntimePermissions/AndroidManifest.html. (Visitado 02-09-2016).

[6] *App Stores Growth Accelerates in 2014*. http://blog.appfigures.com/app-stores-growth-accelerates-in-2014/. (Visitado 02-09-2016).

[7] Gilles Barthe y col. *Formally Verified Implementation of an Idealized Model of Virtualization*. 2013.

[8] G. Betarte y col. "Formal Analysis of Android's Permission-Based Security Model". En: *Scientific Annals of Computer Science* 26.1 (2016), págs. 27-68. DOI: 10.7561/SACS.2016.1.27.

[9] *Calculus of Inductive Constructions*. https://hal.inria.fr/hal-01094195/file/CIC.pdf. (Visitado 02-09-2016).

[10] *Content Providers*. https://developer.android.com/guide/topics/providers/content-providers.html. (Visitado 02-09-2016).

[11] Thierry Coquand y Gérard Huet. *The Calculus of Constructions*. http://www.sciencedirect.com/science/article/pii/0890540188900053. 1988. (Visitado 02-09-2016).

[12] *Defense Advanced Research Projects Agency*. http://www.darpa.mil/. (Visitado 02-09-2016).

[13] *Especificación del lenguaje Gallina*. https://coq.inria.fr/refman/Reference-Manual003.html. (Visitado 02-09-2016).

[14] *Estadísticas de uso de Android*. https://developer.android.com/about/dashboards/index.html. (Visitado 02-09-2016).

[15] *Extensible Markup Language (XML)*. https://www.w3.org/XML. (Visitado 02-09-2016).

[16] *Formal Verification Creates Hacker-Proof Code — Quanta Magazine*. https://www.quantamagazine.org/20160920-formal-verification-creates-hacker-proof-code/. (Visitado 02-09-2016).

[17] *Gartner Says Worldwide Smartphone Sales Grew 9.7 Percent in Fourth Quarter of 2015*. http://www.gartner.com/newsroom/id/3215217. (Visitado 02-09-2016).

[18] *Google Buys Android for Its Mobile Arsenal*. http://www.webcitation.org/5wk7sIvVb. (Visitado 02-09-2016).

[19] *Google Play*. https://play.google.com. (Visitado 02-09-2016).

[20] *Grupo de Seguridad Informática*. http://www.fing.edu.uy/inco/grupos/gsi/index.php?page=proygrado&locale=es. (Visitado 02-09-2016).

[21] *Haskell Language*. https://www.haskell.org/. (Visitado 02-09-2016).

[22] *HaXml: Haskell and XML*. http://projects.haskell.org/HaXml/. (Visitado 02-09-2016).

[23] *How the NSA Accesses Smartphone Data*. http://www.spiegel.de/international/world/how-the-nsa-spies-on-smartphones-including-the-blackberry-a-921161.html. (Visitado 02-09-2016).

[24] *Industry Leaders Announce Open Platform for Mobile Devices*. http://www.openhandsetalliance.com/press_110507.html. (Visitado 02-09-2016).

[25] *Intent flags*. https://developer.android.com/reference/android/content/Intent.html#setFlags(int). (Visitado 02-09-2016).

[26] *Number of apps available in leading app stores as of June 2016*. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. (Visitado 02-09-2016).

[27] *Number of smartphone users worldwide from 2014 to 2019*. http://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. (Visitado 02-09-2016).

[28] *OVAL Android Definition Schema Element Dictionary*. https://oval.mitre.org/language/version5.11/ovaldefinition/documentation/android-definitions-schema.html. (Visitado 02-09-2016).

[29] *OVAL - Open Vulnerability and Assessment Language*. https://oval.mitre.org/. (Visitado 02-09-2016).

[30] *OVAL - OVAL Interpreter*. https://oval.mitre.org/language/interpreter.html. (Visitado 02-09-2016).

[31] *OVAL - OVAL Language Overview*. https://oval.mitre.org/language/about/overview.html. (Visitado 02-09-2016).

[32] *Platform Architecture*. https://developer.android.com/guide/platform/index.html. (Visitado 02-09-2016).

[33] *RFC 2045 - Multipurpose Internet Mail Extensions (MIME)*. https://tools.ietf.org/html/rfc2045. (Visitado 02-09-2016).

[34] *RFC 3986 - Uniform Resource Identifier (URI)*. https://tools.ietf.org/html/rfc3986. (Visitado 02-09-2016).

[35] Agustín Vicente Romano. *Descripción y Análisis Formal del Modelo de Seguridad de Android*. https://www.fceia.unr.edu.ar/lcc/t523/uploads/68.pdf. 2014. (Visitado 02-09-2016).

[36] *Testing basado en modelos*. http://www.fceia.unr.edu.ar/ingsoft/testing-func-a.pdf. (Visitado 02-09-2016).

[37] *The Coq Proof Assistant*. https://coq.inria.fr/. (Visitado 02-09-2016).

[38] *XML Schema Part 0: Primer Second Edition*. https://www.w3.org/TR/xmlschema-0/. (Visitado 02-09-2016).

Anexo

Semántica de las acciones

A continuación se dan las semánticas de las acciones que no han sido presentadas en la sección 2.6. Al igual que en tal sección, la semántica de cada acción se especifica dando su precondición y postcondición, las cuales describen, respectivamente, los requerimientos que debe cumplir un sistema para que la acción pueda ejecutarse en él, y las propiedades de un sistema alcanzado por la correcta ejecución de ella.

Acción `read` $ic\ cp\ u$

Escenario: la instancia en ejecución ic solicita leer el recurso u del proveedor de contenido cp.

Regla

$$\frac{\begin{array}{l} existsRes\ cp\ u\ s \wedge \\ \exists c : Cmp, s.state.running\ ic = Value\ c \wedge \\ (canRead\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Read\ s) \end{array}}{s \xrightarrow{\texttt{read}\ ic\ cp\ u} s}$$

Precondición: el recurso u debe existir en cp. Además, el componente del cual ic es una instancia debe tener derecho de leer el proveedor de contenido cp, o bien debe tener permisos delegados para leer el recurso apuntado por u en cp.

Postcondición: el sistema se mantiene invariante.

Acción `grantP` $ic\ cp\ a\ u\ pt$

Escenario: la instancia en ejecución ic delega permisos de manera permanente a la aplicación a. Esta delegación permite a a ejercer la acción pt sobre el recurso u de cp

Regla

$canGrant\ cp\ u\ s \wedge$
$existsRes\ cp\ u\ s \wedge$
$isAppInstalled\ a\ s \wedge$
$\exists c : Cmp, (s.state.running)\ ic = Value\ c \wedge$
match pt with
$|Read \Rightarrow canRead\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Read\ s$
$|Write \Rightarrow canWrite\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s$
$|Both \Rightarrow (canRead\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Read\ s) \wedge (canWrite\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s)$
$(\forall(a' : idApp)(cp' : CProvider)(u' : uri)(pt' : PType),$
$s.state.delPPerms\ (a', cp', u') = Value\ pt' \implies$
$((a = a' \wedge cp = cp' \wedge u = u') - >$
$s'.state.delPPerms\ (a', cp', u') = Value\ (pt \oplus pt')) \wedge$
$(\neg(a = a' \wedge cp = cp' \wedge u = u') - >$
$s'.state.delPPerms\ (a', cp', u') = Value\ pt')$
$) \wedge$
$(\forall(a' : idApp)(cp' : CProvider)(u' : uri)(pt' : PType),$
$s'.state.delPPerms\ (a', cp', u') = Value\ pt' \implies$
$((a = a' \wedge cp = cp' \wedge u = u') \implies$
match $s.state.delPPerms\ (a', cp', u')$ with
$|Value\ pt'' \Rightarrow pt \oplus pt'' = pt'$
$|\_ \Rightarrow pt = pt'$
end$) \wedge$
$(\neg(a = a' \wedge cp = cp' \wedge u = u') \implies$
$s.state.delPPerms\ (a', cp', u') = Value\ pt')) \wedge$
$s'.state.delPPerms\ (a, cp, u) =$
match $s.state.delPPerms\ (a, cp, u)$ with
$|Value\ pt' \Rightarrow Value\ (pt' \oplus pt)$
$|\_ \Rightarrow Value\ pt$
end$\wedge$
$map\_correct\ s'.state.delPPerms \wedge$
$s \equiv_{environment} s' \wedge$
$s.state \equiv_{apps,grantedPermGroups,perms,running} s'.state \wedge$
$s.state \equiv_{delTPerms,resCont,sentIntents} s'.state$

$$s \xrightarrow{\texttt{grantP}\ ic\ cp\ a\ u\ pt} s'$$

Precondición: El proveedor de contenido cp debe permitir el otorgamiento de permisos sobre su recurso u, dicho recurso debe existir, la aplicación a quien se le delegará el permiso debe estar instalada, y el componente c del cual ic es una instancia en ejecución debe tener el permiso que pretende delegar.

Postcondición: Se agrega el permiso de ejecutar la operación pt sobre el recurso u de cp a la aplicación a. La lista $delPPerms$ del estado del sistema sigue representando a una función parcial (no tiene elementos repetidos en su dominio). Nada más cambia.

Acción `call` $ic\ sac$

Escenario: la instancia en ejecución ic quiere efectuar la llamada al sistema sac

Regla

$$\frac{\begin{array}{c}\exists c : Cmp, s.state.running\ ic = Value\ c \wedge \\ \forall (a : idApp)(p : Perm)(H : isSystemPerm\ p), \\ inApp\ c\ a\ s \implies permSAC\ p\ H\ sac \implies appHasPermission\ a\ p\ s\end{array}}{s \xrightarrow{\texttt{call}\ ic\ sac} s}$$

Precondición: la aplicación a la que pertenece el componente del cual ic es una instancia debe tener todos los permisos de sistema por los cuales la llamada sac está protegida.

Postcondición: el sistema se mantiene invariante.

Acción `uninstall` a

Escenario: se solicita desinstalar la aplicación a

Regla

$$\frac{\begin{array}{l}a \in s.state.apps \wedge \\ (\forall (ic : iCmp)(c : Cmp), s.state.running\ ic = Value\ c \implies \neg inApp\ c\ a\ s) \wedge \\ removeApp\ a\ s\ s' \wedge \\ revokePerms\ a\ s\ s' \wedge \\ revokePermGroups\ a\ s\ s' \wedge \\ removeDefPerms\ a\ s\ s' \wedge \\ removeRes\ a\ s\ s' \wedge \\ revokeOtherTPerm\ a\ s\ s' \wedge \\ revokePPerm\ a\ s\ s' \wedge \\ s.environment \equiv_{systemImage} s'.environment \wedge \\ s.state \equiv_{running,sentIntents} s'.state \wedge \\ s.state \equiv_{delPPerms,delTPerms,resCont} s'.state\end{array}}{s \xrightarrow{\texttt{uninstall}\ a} s'}$$

Precondición: la aplicación que se desea desinstalar debe estar efectivamente instalada, y ninguno de sus componentes debe estar en ejecución.

Postcondición: se elimina a de la lista de aplicaciones instaladas. Además, se la quita de los dominios de las funciones $manifest$, $cert$ y $defPerms$ del entorno del sistema, como así también de los dominios de las funciones parciales que indican los permisos y grupos de permisos otorgados, se revocan todos los permisos otorgados a ella o a cualquiera de sus componentes y se eliminan todos los recursos que ella había definido. El resto del sistema no se modifica.

Acción `hasPermission` p a

Escenario: acción que permite chequear si la aplicación a cuenta con el permiso p

Regla

$$\frac{True}{s \xrightarrow{\texttt{hasPermission}\ p\ a} s}$$

Precondición: no se requieren condiciones para preguntar si una aplicación posee cierto permiso.

Postcondición: el sistema se mantiene invariante.

Acción `write` $ic\ cp\ u\ val$

Escenario: la instancia en ejecución ic solicita escribir el recurso u del proveedor de contenido cp asignándole el valor val

Regla

$$\frac{\begin{array}{l} existsRes\ cp\ u\ s \wedge \\ \exists c : Cmp, s.state.running\ ic = Value\ c \wedge \\ (canWrite\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s) \wedge \\ (\forall (a' : idApp)(r : res)(v : Val), \\ s.state.resCont\ (a', r) = Value\ v \implies s'.state.resCont\ (a', r) = Value\ v \vee \\ (inApp\ (cmpCP\ cp)\ a'\ s \wedge cp.map\_res\ u = Value\ r)) \wedge \\ (\forall (a' : idApp)(r : res)(v : Val), \\ s'.state.resCont\ (a', r) = Value\ v \implies s.state.resCont\ (a', r) = Value\ v \vee \\ ((inApp\ (cmpCP\ cp)\ a'\ s) \wedge cp.map\_res\ u = Value\ r \wedge v = val)) \wedge \\ (\forall (a' : idApp)(r : res), \\ inApp\ (cmpCP\ cp)\ a'\ s \implies cp.map\_res\ u = Value\ r \implies \\ s'.state.resCont\ (a', r) = Value\ val) \wedge \\ map\_correct\ s'.state.resCont \wedge \\ s \equiv_{environment} s' \wedge \\ s.state \equiv_{apps, grantedPermGroups, perms, running} s'.state \wedge \\ s.state \equiv_{delPPerms, delTPerms, sentIntents} s'.state \end{array}}{s \xrightarrow{\texttt{write}\ ic\ cp\ u\ val} s'}$$

Precondición: el recurso apuntado por u debe existir en el proveedor de contenido cp, ic debe ser una instancia en ejecución de algún componente, el cual debe tener permiso de escribir sobre cp o debe tener delegado derecho de escritura sobre el recurso apuntado por u en él.

Postcondición: se sobreescribe con v el valor del recurso apuntado por u en la aplicación a la cual cp pertenece, asegurando que la función $resCont$ del estado del nuevo sistema sea correcta. El resto de los componentes no varían.

Acción `startActivity` $i\ ic$

Escenario: la instancia en ejecución ic quiere iniciar una actividad especificada mediante el *intent* i

Regla

$$\frac{\begin{array}{l} i.intType = intActivity \wedge \\ i.brperm = None \wedge \\ cmpRunning\ ic\ s \wedge \\ \neg(\exists(i' : Intent)(ic' : iCmp), (ic', i') \in s.state.sentIntents \wedge i.idI = i'.idI) \wedge \\ addIntent\ i\ ic\ None\ s\ s' \wedge \\ s \equiv_{environment} s' \wedge \\ s.state \equiv_{apps,grantedPermGroups,perms,running} s'.state \wedge \\ s.state \equiv_{delPPerms,delTPerms,resCont} s'.state \end{array}}{s \xrightarrow{\texttt{startActivity}\ i\ ic} s'}$$

Precondición: el *intent* i a enviar tiene que ser de tipo $intActivity$, no debe especificar ningún permiso que lo proteja, la instancia ic debe estar actualmente en ejecución y no debe existir un *intent* enviado con su mismo identificador.

Postcondición: se agrega el *intent* i a la lista de enviados. El resto de los componentes del sistema no varía.

Acción `startActivityForResult` $i\ n\ ic$

Escenario: la instancia en ejecución ic quiere iniciar una actividad especificada mediante el *intent* i y espera un valor de retorno con token n

Regla

$$\frac{\begin{array}{l} i.intType = intActivity \wedge \\ i.brperm = None \wedge \\ cmpRunning\ ic\ s \wedge \\ \neg(\exists(i' : Intent)(ic' : iCmp), (ic', i') \in s.state.sentIntents \wedge i.idI = i'.idI) \wedge \\ addIntent\ i\ ic\ None\ s\ s' \wedge \\ s \equiv_{environment} s' \wedge \\ s.state \equiv_{apps,grantedPermGroups,perms,running} s'.state \wedge \\ s.state \equiv_{delPPerms,delTPerms,resCont} s'.state \end{array}}{s \xrightarrow{\texttt{startActivityForResult}\ i\ n\ ic} s'}$$

Precondición: el *intent* i a enviar tiene que ser de tipo $intActivity$, no debe especificar ningún permiso que lo proteja, la instancia ic debe estar actualmente en ejecución y no debe existir un *intent* enviado con su mismo identificador.

Postcondición: se agrega el *intent* i a la lista de enviados. El resto de los componentes del sistema no varía.

Acción `startService` $i\ ic$

Escenario: la instancia en ejecución ic quiere iniciar un servicio mediante el *intent* i

Regla

$$\frac{\begin{array}{l} i.intType = intService \wedge \\ i.brperm = None \wedge \\ cmpRunning\ ic\ s \wedge \\ \neg(\exists(i' : Intent)(ic' : iCmp), (ic', i') \in s.state.sentIntents \wedge i.idI = i'.idI) \wedge \\ addIntent\ i\ ic\ None\ s\ s' \wedge \\ s \equiv_{environment} s' \wedge \\ s.state \equiv_{apps,grantedPermGroups,perms,running} s'.state \wedge \\ s.state \equiv_{delPPerms,delTPerms,resCont} s'.state \end{array}}{s \xrightarrow{\texttt{startService}\ i\ ic} s'}$$

Precondición: el *intent* i a enviar tiene que ser de tipo $intService$, no debe especificar ningún permiso que lo proteja, la instancia ic debe estar actualmente en ejecución y no debe existir un *intent* enviado con su mismo identificador.

Postcondición: se agrega el *intent* i a la lista de enviados. El resto de los componentes del sistema no varía.

Acción `sendBroadcast` $i\ ic\ p$

Escenario: la instancia en ejecución ic quiere enviar el *intent* i como difusión, especificando que sólo puede recibirlo un componente que cuente con el permiso p

Regla

$$\frac{\begin{array}{l} i.intType = intBroadcast \wedge \\ i.brperm = None \wedge \\ cmpRunning\ ic\ s \wedge \\ \neg(\exists(i' : Intent)(ic' : iCmp), (ic', i') \in s.state.sentIntents \wedge i.idI = i'.idI) \wedge \\ addIntent\ i\ ic\ p\ s\ s' \wedge \\ s \equiv_{environment} s' \wedge \\ s.state \equiv_{apps,grantedPermGroups,perms,running} s'.state \wedge \\ s.state \equiv_{delPPerms,delTPerms,resCont} s'.state \end{array}}{s \xrightarrow{\texttt{sendBroadcast}\ i\ ic\ p} s'}$$

Precondición: el *intent* i a enviar tiene que ser de tipo $intBroadcast$, no debe especificar ningún permiso que lo proteja, la instancia ic debe estar actualmente en ejecución y no debe existir un *intent* enviado con su mismo identificador.

Postcondición: se agrega el *intent* i a la lista de enviados, protegiéndolo con el permiso p, si se especifica. El resto de los componentes del sistema no varía.

Acción `sendOrderedBroadcast` $i\ ic\ p$

Escenario: la instancia en ejecución ic quiere enviar el *intent* i como una difusión ordenada, especificando que sólo puede recibirla un componente que cuente con el permiso p

Regla

$$\frac{\begin{array}{l} i.intType = intBroadcast \wedge \\ i.brperm = None \wedge \\ cmpRunning\ ic\ s \wedge \\ \neg(\exists(i' : Intent)(ic' : iCmp), (ic', i') \in s.state.sentIntents \wedge i.idI = i'.idI) \wedge \\ addIntent\ i\ ic\ p\ s\ s' \wedge \\ s \equiv_{environment} s' \wedge \\ s.state \equiv_{apps,grantedPermGroups,perms,running} s'.state \wedge \\ s.state \equiv_{delPPerms,delTPerms,resCont} s'.state \end{array}}{s \xrightarrow{\texttt{sendOrderedBroadcast}\ i\ ic\ p} s'}$$

Precondición: el *intent* i a enviar tiene que ser de tipo $intBroadcast$, no debe especificar ningún permiso que lo proteja, la instancia ic debe estar actualmente en ejecución y no debe existir un *intent* enviado con su mismo identificador.

Postcondición: se agrega el *intent* i a la lista de enviados, protegiéndolo con el permiso p, si se especifica. El resto de los componentes del sistema no varía.

Acción `sendStickyBroadcast` i ic

Escenario: la instancia en ejecución ic quiere enviar el *intent* i como difusión $sticky$

Regla

$$\frac{\begin{array}{l} i.intType = intBroadcast \wedge \\ i.brperm = None \wedge \\ cmpRunning\ ic\ s \wedge \\ \neg(\exists(i' : Intent)(ic' : iCmp), (ic', i') \in s.state.sentIntents \wedge i.idI = i'.idI) \wedge \\ addIntent\ i\ ic\ None\ s\ s' \wedge \\ s \equiv_{environment} s' \wedge \\ s.state \equiv_{apps,grantedPermGroups,perms,running} s'.state \wedge \\ s.state \equiv_{delPPerms,delTPerms,resCont} s'.state \end{array}}{s \xrightarrow{\texttt{sendStickyBroadcast}\ i\ ic} s'}$$

Precondición: el *intent* i a enviar tiene que ser de tipo $intBroadcast$, no debe especificar ningún permiso que lo proteja, la instancia ic debe estar actualmente en ejecución y no debe existir un *intent* enviado con su mismo identificador.

Postcondición: se agrega el *intent* i a la lista de enviados. El resto de los componentes del sistema no varía.

Acción `resolveIntent` i a

Escenario: la aplicación a solicita hacer explícito el *intent* i

Regla

$$\frac{\begin{array}{l}\exists i' : Intent, i.idI = i'.idI \wedge i'.cmpName = None \wedge \\ \exists ic : iCmp, (ic, i') \in s.state.sentIntents \wedge \\ \exists c1 : Cmp, s.state.running\ ic = Value\ c1 \wedge \\ \exists c2 : Cmp, inApp\ c2\ a\ s \wedge \\ (\exists iFil : intentFilter, \\ \mathsf{match}\ i'.intType\ \mathsf{with} \\ |intActivity \Rightarrow \exists act : Activity, cmpAct\ act = c2 \wedge iFil \in act.intFilterA \\ |intService \Rightarrow \exists sr : Service, cmpSrv\ sr = c2 \wedge iFil \in sr.intFilterS \\ |intBroadcast \Rightarrow \exists br : BroadReceiver, cmpBR\ br = c2 \wedge iFil \in br.intFilterB \\ \mathsf{end} \wedge \\ actionTest\ i'\ iFil \wedge \\ categoryTest\ i'\ iFil \wedge \\ dataTest\ i'\ iFil) \wedge \\ canStart\ c1\ c2\ s \wedge \\ implicitToExplicitIntent\ i.idI\ a\ s\ s' \wedge \\ s \equiv_{environment} s' \wedge \\ s.state \equiv_{apps,grantedPermGroups,perms,running} s'.state \wedge \\ s.state \equiv_{delPPerms,delTPerms,resCont} s'.state\end{array}}{s \xrightarrow{\texttt{resolveIntent}\ i\ a} s'}$$

Precondición: Debe existir entre los intents enviados uno con el mismo identificador que el *intent* especificado, que aún no haya sido resuelto y haya sido enviado por una instancia en ejecución de un componente, llamémoslo c_1. Además, debe existir un componente c_2 perteneciente a a que pueda ser iniciado por c_1 y cuyo filtro de intents sea respetado por el *intent* que se está resolviendo

Postcondición: Se explicita el *intent* solicitado, asignándole como receptor alguno de los componentes que puedan procesarlo

Acción `receiveIntent` $i\ ic\ a$

Escenario: la aplicación a desea recibir el *intent* i enviado por la instancia en ejecución ic

Regla

$\exists c : Cmp, intentForApp\ i\ a\ c\ ic\ s \wedge \neg isCProvider\ c \wedge$
$\exists c' : Cmp, s.state.running\ ic = Value\ c' \wedge$
$\neg isCProvider\ c' \wedge canStart\ c'\ c\ s \wedge$
$(i.intType = intActivity \implies$
$\forall u : uri, i.data.path = Some\ u \implies$
$(\exists cp : CProvider, existsRes\ cp\ u\ s \wedge$
$canGrant\ cp\ u\ s \wedge$
match $i.intentActionType$ with
$|Read \Rightarrow canRead\ c'\ cp\ s \vee delPerms\ c'\ cp\ u\ Read\ s$
$|Write \Rightarrow canWrite\ c'\ cp\ s \vee delPerms\ c'\ cp\ u\ Write\ s$
$|Both \Rightarrow (canRead\ c'\ cp\ s \vee delPerms\ c'\ cp\ u\ Read\ s) \wedge (canWrite\ c'\ cp\ s \vee$
$delPerms\ c'\ cp\ u\ Write\ s)$
end$))\wedge$
$(i.intType = intBroadcast \wedge i.brperm <> None \implies$
$(\exists p : Perm, i.brperm = Some\ p \wedge$
$appHasPermission\ a\ p\ s)) \wedge$
$(\exists (ic' : iCmp)(c : Cmp), intentForApp\ i\ a\ c\ ic\ s \wedge$
$\neg isCProvider\ c \wedge insNotInState\ ic'\ s \wedge$
$runCmp\ i\ ic'\ c\ s\ s' \wedge$
$(i.intType = intActivity \implies$
$(\exists (u : uri)(cp : CProvider),$
$i.data.path = Some\ u \wedge$
$existsRes\ cp\ u\ s \wedge$
$grantTempPerm\ i.intentActionType\ u\ cp\ ic'\ s\ s') \vee$
$i.data.path = None \wedge s.state.delTPerms = s'.state.delTPerms) \wedge$
$(i.intTypei = intService \implies$
$s.state.delTPerms = s'.state.delTPerms) \wedge$
$(i.intType = intBroadcast \implies$
$s.state.delTPerms = s'.state.delTPerms)) \wedge$
$removeIntent\ i\ ic\ s\ s' \wedge$

$s \equiv_{environment} s' \wedge$
$s.state \equiv_{apps,grantedPermGroups,perms} s'.state \wedge$
$s.state \equiv_{delPPerms,resCont} s'.state$

$$s \xleftarrow{\texttt{receiveIntent}\ i\ ic\ a} s'$$

Precondición: el *intent* i debe estar dirigido a alguno de los componentes de a — notémoslo c —, el cual no puede ser un proveedor de contenido. Además, el componente del cual ic es una instancia en ejecución debe poder iniciar a c. Si el *intent* a recibir es de tipo $intActivity$ y pretende delegar un permiso de escritura/lectura sobre un recurso al componente receptor, entonces el propio componente emisor debe ser capaz de efectuar la operación sobre tal recurso. Si en su lugar, el *intent* es de tipo $intBroadcast$ y fue protegido por cierto permiso, entonces el componente receptor debe contar con dicho permiso.

Postcondición: el componente al cual el *intent* fue dirigido es puesto en ejecución. Además, se delega a esta instancia la potestad de ejecutar la operación especi-

ficada asegurando que la función $delTPerms$ del estado del nuevo sistema sea correcta, si es necesario. Si no lo es, los permisos temporalmente delegados no varían. Se quita el *intent* recibido de la lista de intents enviados. El resto de los componentes no varía.

Acción `stop` ic

Escenario: se solicita detener la instancia en ejecución ic

Regla

$$\frac{\begin{array}{l}\exists(c : Cmp), s.state.running\ ic = Value\ c \wedge \\ (\forall(ic' : iCmp)(c' : Cmp), \\ s'.state.running\ ic' = Value\ c' \implies \\ s.state.running\ ic' = Value\ c') \wedge \\ (\forall(ic' : iCmp)(c' : Cmp), \\ s.state.running\ ic' = Value\ c' \implies \\ s'.state.running\ ic' = Value\ c' \vee ic = ic') \wedge \\ map\_correct\ s'.state.running \wedge \\ insNotInState\ ic\ s' \wedge \\ (\forall(ic' : iCmp)(cp : CProvider)(u : uri)(pt : PType), \\ s'.state.delTPerms(ic', cp, u) = Value\ pt \implies \\ s.state.delTPerms\ (ic', cp, u) = Value\ pt) \wedge \\ (\forall(ic' : iCmp)(cp : CProvider)(u : uri)(pt : PType), \\ s.state.delTPerms\ (ic', cp, u) = Value\ pt \implies \\ (s'.state.delTPerms\ (ic', cp, u) = Value\ pt \vee ic' = ic)) \wedge \\ (\forall(cp : CProvider)(u : uri), \neg is\_Value(s'.state.delTPerms\ (ic, cp, u))) \wedge \\ map\_correct\ s'.state.delTPerms \wedge \\ s \equiv_{environment} s' \wedge \\ s.state \equiv_{apps,grantedPermGroups,perms} s'.state \wedge \\ s.state \equiv_{delPPerms,resCont,sentIntents} s'.state\end{array}}{s \xrightarrow{\texttt{stop}\ ic} s'}$$

Precondición: ic es una instancia en ejecución de un componente en el sistema.

Postcondición: la instancia ic ya no se encuentra en ejecución y se revocan todos los permisos temporales que a ella habían sido delegados, asegurando que las funciones $delTPerms$ y $running$ del estado del nuevo sistema es correcta. El resto de los componentes del sistema no varían.

Acción `revokeDel` $ic\ cp\ u\ pt$

Escenario: se quitan todos los permisos delegados a la instancia en ejecución ic para ejercer la acción pt sobre el recurso u de cp

Regla

$existsRes\ cp\ u\ s \wedge$
$\exists c : Cmp, s.state.running\ ic = Value\ c \wedge$
match pt with
$|Read \Rightarrow canRead\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Read\ s$
$|Write \Rightarrow canWrite\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s$
$|Both\ \Rightarrow\ (canRead\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Read\ s) \wedge (canWrite\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s)$
end
$(\forall (ic' : iCmp)(cp' : CProvider)(u' : uri)(pt' : PType),$
$s'.state.delTPerms\ (ic', cp', u') = Value\ pt' \implies$
$existspt'' : PType, s.state.delTPerms\ (ic', cp', u') = Value\ pt'' \wedge$
$(pt'' = pt' \vee (cp' = cp \wedge u' = u \wedge ptminus\ pt''\ pt = Some\ pt'))) \wedge$
$(\forall (ic' : iCmp)(cp' : CProvider)(u' : uri)(pt' : PType),$
$s.state.delTPerms\ (ic', cp', u') = Value\ pt' \implies$
$(ptminus\ pt'\ pt = None \wedge cp' = cp \wedge u' = u) \vee$
$(\exists pt'' : PType, s'.state.delTPerms\ (ic', cp', u') = Value\ pt'' \wedge$
$(pt'' = pt' \vee (cp' = cp \wedge u' = u \wedge ptminus\ pt'\ pt = Some\ pt'')))) \wedge$
$(\forall ic' : iCmp,$
match $s.state.delTPerms\ (ic', cp, u)$ with
$|Error\ \_ => \neg is\_Value\ (s'.state.delTPerms\ (ic', cp, u))$
$|Value\ pt' =>$ match $ptminus\ pt'\ pt$ with
$|None => \neg is\_Value\ (s'.state.delTPerms\ (ic', cp, u))$
$|Some\ pt'' => s'.state.delTPerms\ (ic', cp, u) = Value\ pt''$
end
end) $\wedge$
$map\_correct\ s'.state.delTPerms \wedge$
$(\forall (a' : idApp)(cp' : CProvider)(u' : uri)(pt' : PType),$
$s'.state.delPPerms\ (a', cp', u') = Value\ pt' \implies$
$\exists pt'' : PType, s.state.delPPerms\ (a', cp', u') = Value\ pt'' \wedge$
$(pt'' = pt' \vee (cp' = cp \wedge u' = u \wedge ptminus\ pt''\ pt = Some\ pt'))) \wedge$
$(\forall (a' : idApp)(cp' : CProvider)(u' : uri)(pt' : PType),$
$s.state.delPPerms\ (a', cp', u') = Value\ pt' \implies$
$(ptminus\ pt'\ pt = None \wedge cp' = cp \wedge u' = u) \vee$
$(\exists pt'' : PType, s'.state.delPPerms\ (a', cp', u') = Value\ pt'' \wedge$
$(pt'' = pt' \vee (cp' = cp \wedge u' = u \wedge ptminus\ pt'\ pt = Some\ pt'')))) \wedge$
$(\forall a' : idApp,$
match $s.state.delPPerms\ (a', cp, u)$ with
$|Error\ \_ => \neg is\_Value\ (s'.state.delPPerms\ (a', cp, u))$
$|Value\ pt' =>$ match $ptminus\ pt'\ pt$ with
$|None => \neg is\_Value\ (s'.state.delPPerms\ (a', cp, u))$
$|Some\ pt'' => s'.state.delPPerms\ (a', cp, u) = Value\ pt''$
end
end) $\wedge$
$map\_correct\ s'.state.delPPerms \wedge$
$s \equiv_{environment} s' \wedge$
$s.state \equiv_{apps, grantedPermGroups, perms} s'.state \wedge$
$s.state \equiv_{running, resCont, sentIntents} s'.state \wedge$

$s \xleftarrow{\texttt{revokeDel}\ ic\ cp\ u\ pt} s'$

Precondición: el recurso apuntado por el uri u debe pertenecer al proveedor de contenido cp y el componente del cual ic es una instancia en ejecución debe ser capaz de efectuar la operación sobre él que desea revocar

Postcondición: se elimina la delegación temporal a ic y la delegación permanente a la aplicación a la cual el componente del que ic es una instancia en ejecución pertenece para efectuar la operación pt sobre el recurso apuntado por u de cp, asegurando que los mapas $delPPerms$ y $delTPerms$ del nuevo estado sean correctos. El resto de los componentes del sistema no cambian.

Errores aceptables por acciones

En las tablas 5.2, 5.3 y 5.4 se completan los códigos de error aceptables ante cada falla en las precondiciones de las acciones no presentadas en la sección 2.9

Acción	**Código de error**	**Falla**
$uninstall\ a$	no_such_app	$a \notin s.state.apps$
	$app\_is\_running$	$\neg(\forall(ic : iCmp)(c : Cmp), s.state.running\ ic = Value\ c \implies \neg inApp\ c\ a\ s)$
$hasPermission\ p\ c$	*	$False$
$read\ ic\ cp\ u$	no_such_res	$\neg existsRes\ cp\ u\ s$
	$instance\_not\_running$	$\neg is\_Value\ (s.state.running\ ic)$
	$not\_enough\_permissions$	$\exists c : Cmp, s.state.running\ ic = Value\ c \wedge \neg(canRead\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Read\ s)$
$write\ ic\ cp\ u\ val$	no_such_res	$\neg existsRes\ cp\ u\ s$
	$instance\_not\_running$	$\neg is\_Value\ (s.state.running\ ic)$
	$not\_enough\_permissions$	$\exists c : Cmp, s.state.running\ ic = Value\ c \wedge \neg(canWrite\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s)$

Cuadro 5.2: Tabla de errores

Acción	**Código de error**	**Falla**
$startActivity\ i\ ic$	$incorrect\_intent\_type$	$i.intType <> intActivity$
	$faulty\_intent$	$i.brperm <> None$
	$instance\_not\_running$	$\neg cmpRunning\ ic\ s$
	$intent\_already\_sent$	$\exists(i' : Intent)(ic' : iCmp), (ic', i') \in s.state.sentIntents \wedge i'.idI = i.idI$
$startActivityForResult\ i\ n\ ic$	$incorrect\_intent\_type$	$i.intType <> intActivity$
	$faulty\_intent$	$i.brperm <> None$
	$instance\_not\_running$	$\neg cmpRunning\ ic\ s$
	$intent\_already\_sent$	$\exists(i' : Intent)(ic' : iCmp), (ic', i') \in s.state.sentIntents \wedge i'.idI = i.idI$
$startService\ i\ ic$	$incorrect\_intent\_type$	$i.intType <> intService$
	$faulty\_intent$	$i.brperm <> None$
	$instance\_not\_running$	$\neg cmpRunning\ ic\ s$
	$intent\_already\_sent$	$\exists(i' : Intent)(ic' : iCmp), (ic', i') \in s.state.sentIntents \wedge i'.idI = i.idI$
$sendBroadcast\ i\ ic\ p$	$incorrect\_intent\_type$	$i.intType <> intBroadcast$
	$faulty\_intent$	$i.brperm <> None$
	$instance\_not\_running$	$\neg cmpRunning\ ic\ s$
	$intent\_already\_sent$	$\exists(i' : Intent)(ic' : iCmp), (ic', i') \in s.state.sentIntents \wedge i'.idI = i.idI$
$sendOrderedBroadcast\ i\ ic\ p$	$incorrect\_intent\_type$	$i.intType <> intBroadcast$
	$faulty\_intent$	$i.brperm <> None$
	$instance\_not\_running$	$\neg cmpRunning\ ic\ s$
	$intent\_already\_sent$	$\exists(i' : Intent)(ic' : iCmp), (ic', i') \in s.state.sentIntents \wedge i'.idI = i.idI$
$sendStickyBroadcast\ i\ ic$	$incorrect\_intent\_type$	$i.intType <> intBroadcast$
	$faulty\_intent$	$i.brperm <> None$
	$instance\_not\_running$	$\neg cmpRunning\ ic\ s$
	$intent\_already\_sent$	$\exists(i' : Intent)(ic' : iCmp), (ic', i') \in s.state.sentIntents \wedge i'.idI = i.idI$
$resolveIntent\ i\ a$	no_such_intt	$\neg pre\_resolveIntent\ i\ a\ s$
$stop\ ic$	$instance\_not\_running$	$\neg is\_Value\ (s.state.running\ ic)$

Cuadro 5.3: Tabla de errores (cont.)

Acción	**Código de error**	**Falla**
$receiveIntent\ i\ ic\ a$	no_such_intt	$\neg(\exists c : Cmp, intentForApp\ i\ a\ c\ ic\ s)$
	$cmp\_is\_CProvider$	$\exists c : Cmp, (intentForApp\ i\ a\ c\ ic\ s \vee s.state.running\ ic = Value\ c) \wedge isCProvider\ c$
	$instance\_not\_running$	$\neg is\_Value\ (s.state.running\ ic)$
	$a\_cant\_start\_b$	$\exists(c, c' : Cmp), intentForApp\ i\ a\ c\ ic\ s \wedge s.state.running\ ic = Value\ c' \wedge \neg canStart\ c'\ c\ s$
	$not\_enough\_permissions$	$i.intType = intBroadcast \wedge i.brperm <> None \implies (\exists p : Perm, i.brperm = Some\ p \wedge \neg appHasPermission\ a\ p\ s)$
	$no\_CProvider\_fits$	$\exists c' : Cmp, s.state.running\ ic = Value\ c' \wedge (i.intType = intActivity \implies \forall u : uri, i.data.path = Some\ u \implies \neg(\exists cp : CProvider, existsRes\ cp\ u\ s \wedge canGrant\ cp\ u\ s \wedge$ match $i.intentActionType$ with $\mid Read \Rightarrow canRead\ c'\ cp\ s \vee delPerms\ c'\ cp\ u\ Read\ s \mid Write \Rightarrow canWrite\ c'\ cp\ s \vee delPerms\ c'\ cp\ u\ Write\ s \mid Both \Rightarrow (canRead\ c'\ cp\ s \vee delPerms\ c'\ cp\ u\ Read\ s) \wedge (canWrite\ c'\ cp\ s \vee delPerms\ c'\ cp\ u\ Write\ s)$ end))
$grantP\ ic\ cp\ a\ u\ pt$	$CProvider\_not\_grantable$	$\neg canGrant\ cp\ u\ s$
	no_such_res	$\neg existsRes\ cp\ u\ s$
	no_such_app	$\neg isAppInstalled\ a\ s$
	$instance\_not\_running$	$\neg is\_Value\ (s.state.running\ ic)$
	$not\_enough\_permissions$	$\exists(c : Cmp), s.state.running\ ic = Value\ iCmpc \wedge \neg($match pt with $\mid Read \Rightarrow canRead\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Read\ s \mid Write \Rightarrow canWrite\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s \mid Both \Rightarrow (canRead\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Read\ s) \wedge (canWrite\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s)$ end)
$revokeDel\ ic\ cp\ u\ pt$	no_such_res	$\neg existsRes\ cp\ u\ s$
	$instance\_not\_running$	$\neg is\_Value\ (s.state.running\ ic)$
	$not\_enough\_permissions$	$\exists c : Cmp, s.state.running\ ic = Value\ c \wedge \neg($match pt with $\mid Read \Rightarrow canRead\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Read\ s \mid Write \Rightarrow canWrite\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s \mid Both \Rightarrow (canRead\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Read\ s) \wedge (canWrite\ c\ cp\ s \vee delPerms\ c\ cp\ u\ Write\ s)$ end)
$call\ ic\ sac$	$instance\_not\_running$	$\neg is\_Value\ (s.state.running\ ic)$
	$not\_enough\_permissions$	$\exists c : Cmp, s.state.running\ ic = Value\ c \wedge \neg(\forall(a : idApp)(p : Perm)(H : isSystemPerm\ p), inApp\ c\ a\ s \implies permSAC\ p\ H\ sac \implies appHasPermission\ a\ p\ s)$

Cuadro 5.4: Tabla de errores (cont.)