Universidad Nacional de Rosario

Tesina de grado
para la obtención del grado de
Licenciado en Ciencias de la Computación

Hacia un prototipo certificado del sistema de permisos de Android 10

Alumno:
Guido De Luca

Director:
Dr. Carlos Luna

Co-Director:
Dante Zanarini

11 de noviembre de 2021

Índice general

Capítulo 1

Introducción

Android [33] es uno de los sistemas operativos para celulares más utilizados en el mundo, capturando aproximadamente el 85 % del mercado mundial [1]. Android también posee una tienda oficial, accesible desde cualquier versión del sistema, que permite instalar un sinfín de aplicaciones para facilitar tareas del usuario, muchas de ellas críticas para la privacidad. Por ejemplo, una aplicación para editar fotos probablemente solicite acceso a todas las fotos de la galería para funcionar correctamente, o una aplicación que permita enviar mensajes de texto posiblemente solicite tener acceso a los mensajes históricos del usuario.

Por lo tanto, para ofrecer las garantías de seguridad y privacidad que se necesitan en un sistema de estas características, Android implementa un protocolo de consenso *multi-parte*, donde una acción ocurre sólo si todos los actores involucrados en la tarea están de acuerdo. Supongamos que un usuario quiere editar sus fotos y decide utilizar una aplicación exclusiva para ese propósito. En ese caso, necesitaremos el consenso de las siguientes partes:

- **El usuario:** al elegir qué aplicación utilizará y al otorgar explícitamente los permisos que sean necesarios.
- **Los desarrolladores:** al especificar los permisos que necesitan para que su aplicación pueda acceder a los recursos necesarios para funcionar correctamente.
- **La plataforma:** al actuar como mediador entre las otras dos partes, supervisando que el pacto se cumpla y que solo los recursos protegidos bajo el permiso otorgado sea accesible por la aplicación.

El hecho de que el último actor sea, a fin de cuentas, una pieza de software, lo convierte en un objetivo principal a la hora de aplicar métodos formales para su verficación. Estudiar formalmente estas políticas de seguridad resulta fundamental para lograr un entendimiento preciso sobre qué es lo que se quiere o espera del sistema y las garantías que el mismo está en condiciones de ofrecer.

La importancia de estudiar este tipo de mecanismos de seguridad fue señalada originalmente por el reporte de Anderson [2], donde el concepto de *monitor de referencia* fue introducido por primera vez. Este concepto define los requerimientos para el diseño de un *mecanismo de validación por referencia*, quien es el responsable de imponer las políticas de control de acceso de un sistema. Para su correcto funcionamiento, Anderson definió tres requerimientos de diseño:

- **Mediación completa:** el mecanismo de validación por referencia (MVR) siempre debe ser invocado luego de ejecutar una acción.
- **A prueba de manipulaciones**, o más conocido como *tamper-proof* en inglés: el MVR no debe ser modificable a mano o programáticamente para garantizar la integridad del mismo.
- **Verificable:** el MVR debe ser lo suficientemente pequeño para poder demostrar lógicamente que es completo y que su implementación es correcta.

El trabajo presentado en esta tesina estudia el tercer requerimiento. En particular, se concentra en analizar formalmente y verificar propiedades sobre un modelo idealizado del sistema de permisos de Android, donde no se tienen en cuenta los detalles de implementación. Sin embargo, aún así provee un escenario realista para investigar sobre esta parte crítica del sistema operativo Android.

Esta tesina parte de un trabajo previo realizado por Felipe Gorostiaga [19], en el que se presentó un modelo idealizado para el sistema de permisos de Android 6. En este trabajo, se extiende dicho modelo con nuevas características introducidas en la versiones 7, 8, 9 y 10 de Android. Algunos de los cambios que los desarrolladores de la plataforma introdujeron con estas versiones no implicaron cambios sobre el modelo abstracto. Para esos casos, se incluye un pequeño análisis informal sobre las mejoras de seguridad que implican para la plataforma. Además, extendimos una implementación funcional previa del MVR para que contemple los nuevos cambios en la especificación. Las contribuciones de este trabajo fueron presentadas en TYPES [26].

Organización del informe El capítulo 2 introduce informalmente los conceptos necesarios para entender el funcionamiento del sistema de permisos de Android. También incluye un análisis informal sobre todos los cambios considerados en este trabajo. Los capítulos 3 y 4 presentan la formalización del modelo idealizado y su implementación, respectivamente. Al final de cada uno, se discuten propiedades de relevancia relacionadas a los nuevos cambios. En el capítulo 5 se analizan trabajos similares que también estudian el sistema de permisos de Android. Se incluyeron diferentes enfoques y técnicas, a pesar de que no todos los trabajos citados son de índole formal. Por último, en el capítulo 6 contiene una conclsuión e ideas sobre trabajo futuro. El código Coq con la formalización completa está disponible en `https://github.com/g-deluca/tesina` [25].

Capítulo 2

El modelo de seguridad de Android

2.1. Arquitectura de la plataforma

El sistema operativo Android está compuesto por cinco capas de software, ordenadas en forma de pila (o *stack*), donde cada una de ellas provee un grupo de servicios a la capa inmediatamente superior. De esta forma, se va abstrayendo progresivamente la interacción con el hardware (la base de la pila) hasta llegar al nivel más alto, en el que se ubican las aplicaciones que realizarán las tareas requeridas por los usuarios. A continuación, analizaremos brevemente cada uno de estos niveles.

Núcleo del sistema operativo: Linux

La base de la plataforma Android es el *kernel* de Linux. Desde el punto de vista de la seguridad, utilizar a un núcleo tan estudiado a lo largo de los años como pilar de la arquitectura, ayuda a generar confianza en la misma. Además, le permite a los fabricantes de dispositivos desarrollar controladores de hardware para un sistema ya conocido.

Una de las principales utilidades de Linux de la que Android toma ventaja es del sistema de permisos basado en usuarios[1]. Esta característica permite que cada aplicación sea ejecutada dentro de su propia máquina virtual con un identificador de usuario único (`UUID`) asignado a la misma. Luego, por defecto, estos identificadores se inicializan con permisos de lectura y escritura restringidos, de manera tal que los recursos de cada aplicación queden debidamente aislados y protegidos de potenciales *malwares*. Sin embargo, este tipo de defensa, implica la existencia de un mecanismo de validación de referencia que arbitre las situaciones en las que una aplicación deliberadamente desee compartir recursos con otra. Este mecanismo es implementado en las capas superiores de la arquitectura.

Otras características de Linux utilizadas por Android son: la generación de subprocesos, la administración de memoria de bajo nivel y otras funcionalidades

[1] El mismo no debe confundirse con el sistema de permisos implementado por Android en una de las capas superiores, que es el que se formaliza y estudia en esta tesina

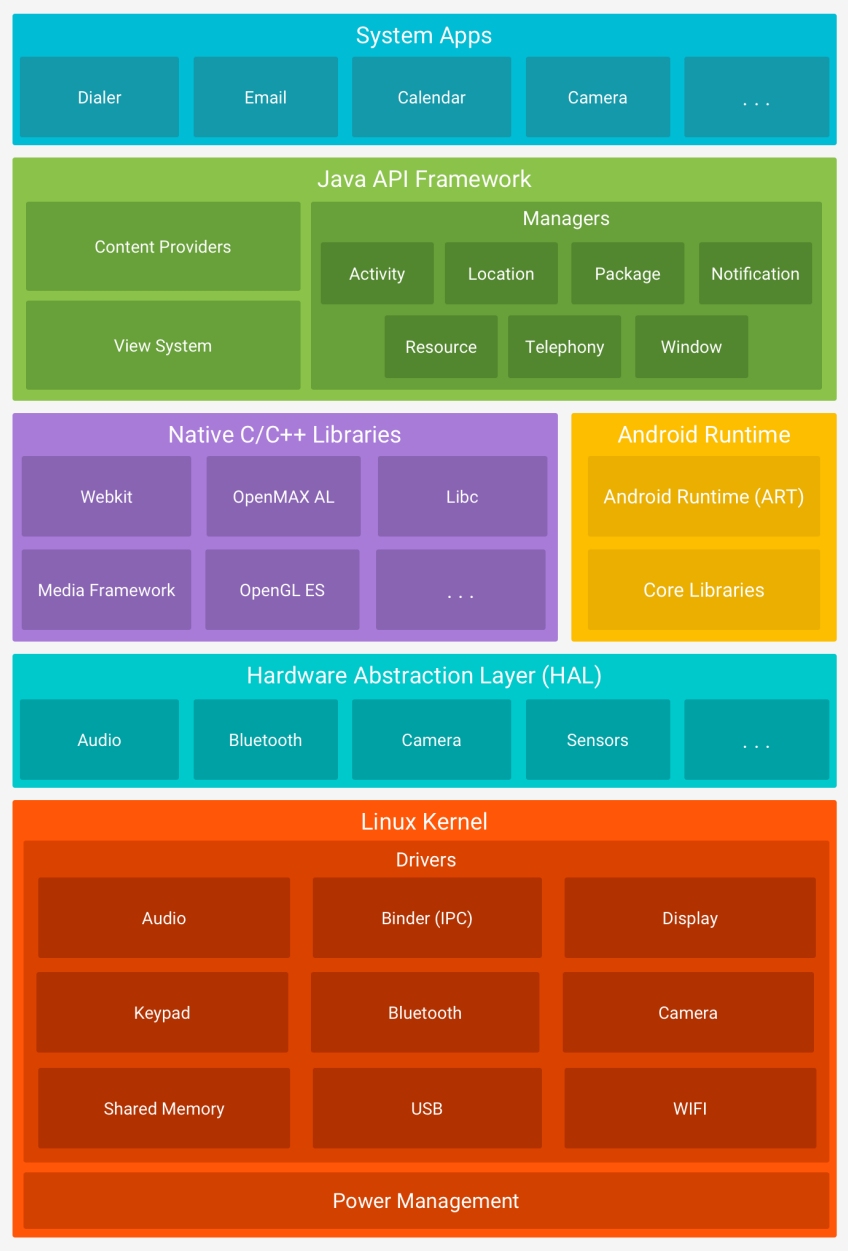

Figura 2.1: Pila de software de Android

encapsuladas dentro de SELinux (***S**ecurity **E**nhanced Linux*) como la política de control de acceso obligatorio (más conocida como *MAC* por sus siglas en inglés) para distinguir entre aplicaciones del sistema y de terceros.

Capa de abstracción de hardware

Esta capa contiene módulos de software encargados de abstraer los diferentes componentes de hardware, como la antena de *Bluetooth* o la cámara. Estas abstracciones deben ser independientes de los *drivers* que se encuentran en el núcleo del sistema, brindando flexibilidad y transparencia. Cada módulo está autocontenido en una librería, que es cargada dinámicamente cuando alguna capa superior lo solicita.

Entorno de *runtime* y bibliotecas nativas

En esta capa encontramos el entorno de *runtime* (o ART [40], por sus siglas en inglés) utilizado por todas las aplicaciones del sistema junto con algunas librerías nativas que la plataforma le ofrece a los desarrolladores.

Marco de trabajo/API de la plataforma

En esta capa se encuentran las interfaces que la plataforma le brinda a los desarrolladores de aplicaciones para poder acceder a todas las funciones del sistema operativo. Es en esta capa donde encontramos los servicios que implementan el mecanismo de validación mencionado previamente.

Aplicaciones del sistema y de terceros

Las aplicaciones representan el escalón final en esta pila de abstracciones, siendo éstas los puntos de entrada para que cualquier usuario de Android interactúe con las funciones del dispositivo. Algunas de estas aplicaciones ya vienen preinstaladas en la plataforma (como la de mensajería SMS o la encargada de brindar la interfaz de configuración del sistema) mientras que otras pueden ser instaladas a través de la tienda oficial o manualmente.

2.2. Conceptos preliminares

El mecanismo que arbitra el acceso a los recursos de las aplicaciones está fuertemente basado en las interacciones entre las mismas. Por eso, en esta sección introduciremos algunos conceptos que serán necesarios para un mejor entendimiento del sistema.

2.2.1. Componentes de una aplicación

Las aplicaciones de Android están conformadas por distintas piezas, que a pesar de ser funcionales por sí mismas, interactúan entre ellas para otorgar al usuario la funcionalidad esperada. Por ejemplo, en una aplicación que reproduce música, existirá un componente que se encargará de presentar la interfaz al usuario mientras que otro, independiente de la interfaz, será el encargado de

reproducir la pista seleccionada. De esta manera, es posible la reproducción de una canción en segundo plano[2].

Existen cuatro tipos de componentes. Cada uno de ellos representa un punto de entrada por el cual el sistema o el usuario pueden comunicarse con la aplicación y tiene un ciclo de vida distinto. A continuación, describiremos brevemente cada uno de ellos.

Actividades: Una *actividad* es el punto de entrada de interacción con el usuario, el encargado de ubicar los elementos en la pantalla. Una aplicación puede tener muchas actividades, cada una de ellas representando una pantalla distinta. Continuando con el ejemplo de la aplicación para reproducir música, podríamos pensar que la pantalla del reproductor y la del buscador de canciones son dos actividades distintas.

Servicios: Un *servicio* permite ejecutar procesos en segundo plano. En general, es el encargado de la ejecución de procesos largos, que de ser interrumpidos podrían afectar el desempeño de la aplicación. Los servicios son puntos de entrada generales a la aplicación, pueden ser iniciados tanto por un usuario (a través de una actividad), por otro servicio o por el sistema operativo.

Receptores de anuncios: Un *receptor de anuncios* es el componente que se encarga de recibir mensajes del sistema y de otras aplicaciones, incluso cuando la misma no está en ejecución. Por ejemplo, esta es la forma en la que el sistema le avisa a todas las aplicaciones que la carga de la batería es baja. Cada aplicación establece cuáles son los mensajes de su interés, el resto de los mensajes que reciba simplemente serán ignorados.

Proveedores de contenido: El *proveedor de contenido* es el componente encargado de administrar los datos de la aplicación que se encuentran almacenados en medios persistentes. A través de él, cualquier aplicación autorizada puede leer o modificar dichos datos. En otras palabras, un proveedor de contenido actúa como intermediario entre los datos persistidos y las aplicaciones: no solo la dueña de los datos sino también todas aquellas que hayan sido previamente autorizadas. Cada recurso allí presente se identifica con un identificador uniforme de recursos (URI).

2.2.2. Interacción entre componentes

Una característica particular de Android es que cualquier aplicación puede iniciar un componente de otra. Por ejemplo, una aplicación que desee tomar una foto con la cámara del dispositivo, puede hacerlo *activando* el componente correspondiente de la aplicación provista por el sistema, en lugar de implementar una nueva actividad que lo haga. Sin embargo, como los procesos de las aplicaciones ejecutan en procesos aislados y con permisos que limitan el acceso de otras aplicaciones, el sistema debe actuar como intermediario en la activación de un componente. Para eso, la aplicación en cuestión debe enviar un *intent*

[2]Una aplicación se considera en segundo plano cuando se está ejecutando a pesar de no ser la aplicación que actualmente se está mostrando en la pantalla.

que especifique que lo que desea es iniciar una actividad en particular. Luego, el sistema será el encargado de activar ese componente.

Intents

Los *intents* son mensajes asíncronos que permiten la comunicación entre componentes y aplicaciones. Existen tres casos de uso principales: el inicio de una actividad, el inicio de un servicio y la emisión de un evento, como una indicación de que la batería del dispositivo tiene baja carga o comenzó a cargarse. Estos mensajes puede ser *explícitos*, en donde el componente de destino está especificado; o *implícitos*, donde no existe un destinatario específico, pero el mensaje conlleva la información suficiente para que el sistema encuentre algún componente apropiado.

Filtros de intents

Cuando un intent es implícito, Android busca en los *filtros de intents* de cada aplicación para verificar cuales de ellas son aptas para recibir ese mensaje. Si un único filtro de intent es compatible, la aplicación correspondiente será iniciada. Si existe más de uno, el usuario tendrá la posibilidad de elegir qué aplicación se activará. Los filtros de intent se declaran en el *manifiesto* de la aplicación y por lo tanto, son estáticos.

2.2.3. Manifiesto de la aplicación

El archivo de manifiesto de una aplicación es un documento `xml` que contiene información estática sobre la misma. Su tarea principal es declarar cuáles son los componentes que corresponden a la aplicación, para que de esta forma, el sistema pueda reconocerlos como tales. Algunas otras declaraciones que se encuentran en el manifiesto son:

1. El nombre del paquete de la aplicación.
2. Los filtros de intents previamente mencionados.
3. Los aspectos de hardware y software que la aplicación utilizará, como una cámara o los servicios de Bluetooth.
4. Los permisos de usuario que requiere la aplicación

2.3. Mecanismo básico de protección: permisos

Como ya mencionamos en la sección 2.1, existe en Android un mecanismo encargado de arbitrar el acceso a la información sensible de las aplicaciones y a algunos recursos del sistema. Este mecanismo, al que a lo largo de este trabajo llamaremos *sistema de permisos*, cumple un rol fundamental en el modelo de seguridad de Android. Este sistema fue el que se ha estudiado en profunidad y formalizado en esta tesina.

A grandes rasgos, para que una aplicación pueda acceder a un recurso sensible, tanto del sistema como de otra aplicación, debe haber obtenido previamente los permisos necesarios. De ser así, la operación es realizada con éxito. Caso

contrario, la aplicación debe pedirle al sistema operativo que le otorgue el o los permisos en cuestión. Finalmente, el sistema operativo, dependiendo del *nivel de protección* de dicho permiso, puede delegar la decisión al usuario a través de una ventana emergente. Análogamente, si un desarrollador desea proteger un recurso de su aplicación, puede definir un nuevo permiso y establecer que el recurso solo puede ser compartido con aquellas otras aplicaciones que cuenten con el mismo. De esta forma, nos encontramos con una primer clasificación entre permisos: aquellos definidos por las aplicaciones para mantener sus datos protegidos; y los definidos por el sistema, que se necesitan para ganar acceso a los recursos del dispositivo que pueden comprometer la privacidad del usuario (como la cámara o la lista de contactos).

Cada permiso tiene un nombre único y cuenta un nivel de protección. El nivel de protección determina de qué forma procederá el sistema operativo cuando un permiso es solicitado. Existen tres niveles:

1. **Normal:** Estos permisos pueden ser otorgados automáticamente por el sistema ya que se utilizan para información o recursos que tienen un riesgo muy bajo de comprometer la seguridad.

2. **Peligrosos:** Los permisos de este nivel protegen recursos mucho más sensibles y requieren la aprobación explícita del usuario.

3. **De misma firma:** Un permiso de este tipo se otorga solamente si la aplicación que lo está solicitando y la aplicación que lo declara están firmadas con el mismo certificado.

A partir de la versión `6` de Android, todos los permisos peligrosos se otorgan en tiempo de ejecución mientras que los correspondientes a los otros niveles se conceden en el momento en el que la aplicación se instala.

Los permisos que están asociados a una misma característica de la aplicación o dispositivo, suelen estar agrupados. Por ejemplo, tanto el permiso necesario para leer los mensajes de texto, como el que autoriza el envío de los mismos, pertenecen a un mismo *grupo de permisos* llamado `SMS`. El principal objetivo de agrupar permisos es evitar abrumar al usuario con preguntas sobre la concesión de los mismos. De esta forma, si una aplicación requiere un permiso peligroso que está agrupado, el sistema le pedirá que autorice a todo el grupo. En la sección 2.4.2, estudiaremos más en detalle qué significa autorizar a un grupo, dependiendo de la versión de Android que se analice.

Delegación de permisos

Existen dos mecanismos mediante los cuales una aplicación puede delegar sus propios permisos a otra. El primero consiste en dejar un `Intent` pendiente, para que otra aplicación pueda tomarlo y ejecutarlo con los permisos y la identidad de quien lo creó. El segundo método, en cambio, consiste en que una aplicación que cuenta con permisos de escritura/lectura sobre un proveedor de contenido puede delegar *temporalmente* esos permisos a otra aplicación. Los permisos delegados serán revocados una vez que la actividad o servicio que los recibió haya terminado su ejecución.

2.4. Cambios recientes

En esta sección analizamos informal y brevemente los cambios que fueron introducidos entre Android Nougat (7) y Andrid 10 que tuvieron un impacto significativo en el sistema de permisos.

2.4.1. Sistema de archivos

Con intención de aumentar la seguridad de las aplicaciones, el directorio privado de todas aquellas orientadas[3] a una versión posterior a Android 7 tienen permisos restringidos. Solamente la aplicación dueña puede leer, escribir y ejecutar. Esta configuración evita la fuga de metadatos de los archivos privados, como el tamaño o su existencia. De esta forma, las aplicaciones ya no pueden otorgar permisos de lectura/escritura a otras para compartir sus archivos, sino que deben hacerlo otorgando permisos temporales mediante un proveedor de contenido.

El modelo de Android del cual partimos en esta tesina, ya contemplaba el otorgamiento temporal de permisos de lectua/escritura. Además, decidimos **no** modelar el sistema de archivos de Android en este trabajo, dado que nuestro interés se centra en estudiar propiedades más generales relacionadas a los permisos y creemos que el modelado de un sistema de archivos require un gran esfuerzo que no aporta hacia donde queremos llegar. Por lo tanto, no se introdujeron cambios en el modelo con respecto a esta nueva característica.

2.4.2. Comportamiento de permisos agrupados

En versiones previas a Android 8, si una aplicación solicitaba un permiso agrupado y el permiso era otorgado, el sistema también otorgaba el resto de los permisos del mismo grupo que estuvieran declarados en el manifiesto. Este comportamiento se consideraba incorrecto ya que violaba el principio de mínimo privilegio en la plataforma. A partir de la octava versión de Android, este comportamiento ha cambiado. Cuando una aplicación pide un permiso agrupado y lo consigue, solo obtiene el permiso que fue explícitamente solicitado. Sin embargo, si posteriormente desea conseguir otro permiso perteneciente al mismo grupo, el sistema tiene autorización para otorgarlo automáticamente. En otras palabras, cuando una aplicación solicita un permiso **peligroso** agrupado por primera vez, el usuario será quien determine si ese permiso es otorgado o no; pero las solicitudes posteriores de permisos del mismo grupo serán automáticamente aceptadas por el sistema.

Desde un punto de vista conceptual, este cambio es muy favorable en pos de respetar el principio de mínimo privilegio. Sin embargo, cabe preguntarse por qué simplemente no se delega al usuario la autorización de cada uno de los permisos peligrosos que una aplicación solicita, que desde el punto de vista de dicho principio, es una implementación óptima. La respuesta está en la negociación que los autores de la plataforma deben mediar entre políticas muy estrictas de seguridad y una buena experiencia de usuario (invadir al usuario de advertencias y solicitudes no es algo favorable en ese sentido).

[3]Las aplicaciones pueden estar orientadas a distintas versiones del sistema. Android utiliza esta configuración para determinar la compatibilidad de ciertos comportamientos nuevos.

A pesar de lo mencionado en el párrafo anterior, si miramos los cambios introducidos en esta versión desde un lugar de seguridad más concreto, seguimos encontrando situaciones peligrosas como la que describimos a continuación.

Permisos peligrosos y normales en el mismo grupo

De acuerdo a la documentación de Android, "cualquier permiso puede pertenecer a un grupo de permisos sin importar el nivel del protección del mismo" [42]. Sin embargo, en ningún lugar se especifica si permisos de distintos niveles de protección (en particular, uno de nivel normal y otro peligroso) pueden compartir el mismo grupo ni, en caso de que sea posible, cómo debería el sistema manejar el otorgamiento automático de permisos. De aquí se desprenden algunas preguntas interesantes:

1. Dado que los permisos normales se otorgan en tiempo de instalación y sin explícita aprobación del usuario, ¿habilita esto el posterior otorgamiento automático de los permisos peligrosos del mismo grupo? ¿Se informa de esta decisión al usuario?

2. De no ser así, ¿cómo se encarga el sistema de manejar esta situación?

En esta tesina formalizamos el peor de los escenarios, en donde un permiso normal puede habilitar el otorgamiento automático de un permiso peligroso sin que el usuario se entere. Este escenario sigue siendo válido dada la especificación informal de la plataforma. Analizaremos esto formalmente en la sección siguiente.

2.4.3. Cambios en pos de la privacidad del usuario

Tanto en Android 9 como en Android 10 se introdujeron varios cambios que apuntan a mejorar la privacidad de los usuarios. Entre ellos, destacamos:

1. Acceso limitado a los sensores del dispositivo cuando las aplicaciones se ejecutan en segundo plano. Por ejemplo, las aplicaciones ya no pueden acceder ni al micrófono ni a la cámara sin notificar al usuario.

2. Acceso restringido al registro de llamadas y a números telefónicos.

3. Acceso restringido a la información obtenida de un análisis de Wi-Fi.

4. Se agregó un permiso (peligroso) para acceder a la ubicación en segundo plano.

5. Se agregaron restricciones sobre cuándo un servicio puede empezar una actividad. De esta manera, se busca que el usuario tenga más control sobre lo que ve en su pantalla.

Sin embargo, todos estos cambios son específicos a la implementación y no tienen impacto en una representación abstracta como la nuestra.

2.4.4. Chequeo de permisos en aplicaciones *legacy*

A partir de Android 10, cuando una aplicación orientada a una versión previa a Android 6 es ejecutada por primera vez, el usuario será alertado. Además, dado que en las versiones más viejas de la plataforma los permisos peligrosos se otorgaban al momento de la instalación, el usuario ahora tendrá la posibilidad de revisar los mismos antes de que la aplicación se ejecute. Este cambio fue formalizado e incluido en nuestro modelo.

Capítulo 3

Formalización del sistema de permisos

En este capítulo se describe la extensión realizada sobre una formalización previa [27] del sistema de permisos de la plataforma. El objetivo de la misma es poder modelar los nuevos comportamientos que se introdujeron con las versiones 7, 8, 9 y 10 del sistema operativo Android. Al final del capítulo se presentan las nuevas propiedades que se han probado sobre la formalización actualizada, poniendo el foco en aquellas estrechamente relacionadas con los cambios modelados.

3.1. Lenguaje formal utilizado

La especificación del sistema se realizó dentro del framework de trabajo Coq [43]. Coq es un asistente de pruebas interactivo, que permite el desarrollo de programas consistentes con su especificación. Para lograrlo, provee tres aspectos fundamentales:

1. Un lenguaje de especificación que permite escribir expresiones lógicas de alto orden, algoritmos y teoremas;
2. Un asistente de pruebas que permite desarrollar pruebas matemáticas verificadas;
3. Una herramienta de extracción de programas, que permite sintetizar programas en lenguajes como OCaml [31] o Haskell [20] a partir de las especificaciones formales escritas previamente. Los programas construidos de esta manera suelen llamarse *programas certificados*.

El lenguaje lógico subyacente usado por Coq es el Cálculo de Construcciones Inductivas [34] (también conocido como CIC, por sus siglas en inglés).

3.2. Notación

En las siguientes secciones utilizaremos la misma notación de los trabajos previos [27, 9, 10]. La misma es similar a la sintaxis de Haskell.

3.2.1. Estructuras de datos y tipos generales

Los diccionarios (más conocidos como *records* por su nombre en inglés) tendrán la forma $\{l_1 : T_1, ..., l_n : T_n\}$ y notaremos el acceso a cada uno de sus elementos como $r.l_n$. También usaremos $\{T\}$ para notar a un conjunto que contiene elementos de tipo T.

Para representar tuplas utilizaremos el producto cartesiano. Por ejempo, si A y B son los tipos de los elementos de una tupla de tipo T, lo notaremos como $T = (A * B)$.

Finalmente, para denotar mapeos entre valores (o funciones parciales), decidimos usar la palabra clave utilizada en Coq: *mapping*. En caso de querer relacionar elementos de tipo A con elementos de tipo B, notaremos *mapping A B*

Tipos enumerados

Para definir enumeraciones utilizaremos los tipos inductivos nativos de Coq. Puede parecer algo excesivo utilizar algo tan complejo como la inducción para representar una enumeración de valores, pero es la única forma en la que puede lograrse dicha representación sin utilizar librerías o extensiones del lenguaje. En definitiva, un tipo enumerado es un tipo inductivo cuyos habitantes están definidos por extensión.

Por ejemplo, el nivel de protección de los permisos está definido de la siguiente manera:

$$
\begin{aligned}
&\mathit{Inductive\ permLevel} : \mathit{Set}\ := \\
&\qquad\qquad | \ \mathit{dangerous} \\
&\qquad\qquad | \ \mathit{normal} \\
&\qquad\qquad | \ \mathit{signature} \\
&\qquad\qquad | \ \mathit{signatureOrSys}
\end{aligned}
$$

3.2.2. Tipos básicos de Coq

A continuación detallaremos algunos de los tipos básicos de Coq más utilizados en esta tesina.

Tipo *option*

$$
\begin{aligned}
&\mathit{Inductive\ option}\ T\ := \\
&\qquad\qquad | \ \mathit{Some}\ :\ T \rightarrow\ \mathit{option}\ T \\
&\qquad\qquad | \ \mathit{None}\ :\ \mathit{option}\ T
\end{aligned}
$$

El tipo `option`, análogo a la mónada `Maybe` de Haskell [28], sirve para representar la posible ausencia de valores.

Tipo *list*

$$\begin{aligned} &\mathit{Inductive\ list\ T}\ :=\\ &\qquad\qquad \mid (::)\ :\ T \rightarrow\ \mathit{list\ T}\ \rightarrow\ \mathit{list\ T}\\ &\qquad\qquad \mid \mathit{nil}\ :\ \mathit{list\ T} \end{aligned}$$

El tipo utilizado para denotar una lista de elementos. Está definido de manera inductiva, siendo :: un operador de punto fijo.

Tipo *nat*

Utilizado para denotar a los números naturales.

Tipo *Prop*

Este tipo representa uno de los dos "universos" donde viven las proposiciones lógicas en Coq. El tipo *Prop* es impredicativo y por lo tanto, una proposición de este tipo no contiene valor computacional [35] y puede ser descartada a la hora de extraer un programa. Utilizaremos este tipo para escribir teoremas que nos permitan razonar sobre el modelo. El otro universo que contiene proposiciones lógicas se llama *Set* y las proposiciones de este tipo sí tienen valor computacional y deben ser preservadas en los programas extraídos.

3.3. Estado del sistema

3.3.1. Formalización de los componentes básicos

En esta sección presentaremos las estructuras de datos utilizadas para formalizar los distintos componentes presentados en la sección 2.2. El orden en el que introduciremos dichas abstracciones será incremental con respecto a la complejidad de las mismas y por lo tanto, diferirá del orden en el que los componentes fueron presentados anteriormente.

Tipos atómicos

En la tabla 3.1 se enumeran las tipos de datos atómicos. Entendemos por tipo de dato atómico a aquellos que representan entidades básicas o bien, a aquellos cuya definición queda sujeta a cuestiones de implementación. En el asistente de pruebas Coq, los tipos atómicos fueron definidos como parámetros de tipo *Set*.

Permisos

Los permisos descriptos en la sección 2.3 fueron formalizados como un registro de tres campos: el nombre o identificador del permiso en cuestión, el nombre o identificador del grupo al que pertenece el permiso (en caso de que no pertenezca a ningún, lo representaremos mediante el valor *None*) y el nivel de protección del mismo. Formalmente,

$$\mathit{Perm}\ :=\ \{\mathit{idP} : \mathit{idPerm};\ \mathit{maybeGrp} : \mathit{option\ idGrp};\ \mathit{pl} : \mathit{permLevel}\}$$

Nombre del tipo	**Descripción**
idApp	Identificadores/nombres de las aplicaciones.
idPerm	Identificadores/nombres de los permisos.
idGrp	Identificadores/nombres de los grupos de permisos.
idCmp	Identificadores/nombres de los componentes de una aplicación.
iCmp	Identificadores/nombres de las instancias en ejecución de los componentes de una aplicación.
uri	Identificadores de los recursos guardados en los proveedores de contenido.
res	Recursos definidos por los proveedores de contenido.
mimeType	Parámetro utilizado para representar los distintos tipos de MIME [14]. Este tipo de datos es utilizado por los *intents* y los filtros de *intents* para saber cómo decodificar la información allí almacenada.
Category	Categoría a la que puede pertenecer un *intent*.
Cert	Certificados con los que se firman las aplicaciones.
Extra	Abstracción usada para representar el campo de información "extra" que puede enviarse a través de los *intents*.
Flag	Banderas que pueden ser activadas por el emisor de un *intent* para indicar cómo debe leerse/manejarse ese mensaje.
SACall	Llamadas a la API de Android que interfieren con el modelo de seguridad (por ejemplo, para tener acceso a internet).
Val	Tipo genérico para representar valores. Usado para representar los valores de los recursos (*res*) de una aplicación.
vulnerableSdk	SDK a partir del cual las aplicaciones empiezan a considerarse como *legacy* y requieren una verificación del usuario antes de ser ejecutadas por primera vez.

Cuadro 3.1: Tipos atómicos

El tipo de datos *permLevel* es un tipo de datos enumerado cuyos valores posibles son: *normal*, *dangerous* y *signature*. Cada uno de estos valores representa uno de los niveles de protección descriptos en la sección 2.3.

Intents

De manera análoga a los permisos (y a la mayoría de los componentes que mencionaremos), también utilizamos registros para formalizar a los *intents*. En este caso, los registros poseen una mayor cantidad de campos, algunos de ellos representados con estructuras complejas como un nuevo registro. Los nueve campos que definen a un *intent* son:

- *idI*, el nombre o identificador del intent.
- *cmpName*, en caso de que el intent esté dirigido a un componente en

particular, el identificador del mismo.

- *intType*, define el caso de uso del intent. Puede ser: iniciar una actividad, iniciar un servicio o transmitir un evento por *broadcast* a cualquier aplicación interesada.
- *action*, la acción que ejecutará el intent. Los valores posibles de este campo son genéricos; por ejemplo, tenemos la acción de "ver" (representada por la constante `ACTION_VIEW`) o la acción de "enviar" (representada por `ACTION_SEND`). Para definir qué dato puede verse o enviarse y para definir qué componente será el encargado de hacerlo, deben utilizarse otros datos presentes en el *intent* (*data* y *category*, respectivamente).
- *data*, contiene la información necesaria para identificar los datos que son necesarios para ejecutar la acción en cuestión.
- *category*, contiene información adicional sobre la acción y el componente que puede llevarla a cabo. Es utilizado principalmente en los intents implícitos.
- *extra*, contiene información adicional de cualquier tipo.
- *flags*, banderas que contienen información sobre cómo manejar el *intent*.
- *brperm*, en caso de que el componente que recibe el intent necesite un permiso para ejecutarlo, el mismo deberá estar listado aquí.

Formalmente, un *intent* está definido con la siguiente estructura:

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

La estructura *Data* está definida a su vez como un record que contiene un URI al recurso que obtendrá quien reciba el intent, el tipo de dato al que se intentará acceder y, opcionalmente, el tipo de dato en formato MIME[1].

$$Data \ := \ \{path : option \ uri; \ type : dataType; \ mime : option \ mimeType\}$$

La estructura *dataType* es una enumeración que contiene los siguientes valores: *content*, *file* y *other*.

[1] Conocido de esta manera por sus siglas en inglés (*Multipurpose Internet Mail Extensions*). Es un estándar que se utiliza para indicar la naturaleza y el formato de un documento.

Filtros de intents

Los filtros de intents están definidos como registros de tres valores. Cada uno de ellos actúa como un filtro sobre algún campo del intent en cuestión. Formalmente:

$$
\begin{aligned}
IntentFilter \ := \ \{ & \\
& actFilter : list\ intentAction; \\
& dataFilter : list\ Data \\
& catFitler : list\ Category \\
\} &
\end{aligned}
$$

Coloquialmente, al construir un elemento de tipo `IntentFilter` construiremos un filtro que establecerá qué tipos de acciones pueden ejecutarse sobre qué datos. El "tipo de acción" se define en conjunto entre la acción y la categoría. Por ejemplo, si una aplicación define un filtro de *intents* donde la categoría es `CATEGORY_BROWSABLE` y la acción es `ACTION_VIEW`, entonces estará en condiciones de ser elegida para abrir un *link* que ha sido clickeado desde un navegador. En una implementación real, los filtros de intents pueden construirse con más campos. Sin embargo, para este modelo elegimos estos tres ya que los mismos representan las características definitorias de un intent implícito [41].

Componentes de una aplicación

Como mencionamos previamente en la sección 2.2.1, existen cuatro tipos de componentes que pueden conformar a una aplicación de Android: actividades, servicios, receptores de anuncios y proveedores de contenido. Los tres primeros fueron definidos análogamente. Todos están representados por un registro de cuatro campos que contiene: un identificador, un indicador sobre si el componente está exportado o no (es decir, si componentes de otras aplicaciones pueden interactuar con el mismo), el permiso (en caso de existir) que protege al componente y una lista de los filtros de *intents* definidos por el mismo. A modo de ejemplo, mostraremos la definición formal de una actividad:

$$
\begin{aligned}
Activity \ := \ \{ & \\
& id : idCmp; \\
& exp : bool \\
& perm : option\ Perm \\
& intFilter : list\ IntentFilter \\
\} &
\end{aligned}
$$

Para el caso de los proveedores de contenidos, el registro se extiende con los siguientes valores:

- *readPerm* y *writePerm*, permitiendo una mayor granularidad en los permisos necesarios para leer o escribir del mismo. En caso de que estos valores existan, tienen prioridad por encima de *perm*.

- *grantU* (nombrado así para acortar el nombre *grantUriPermissions*), un campo de valor *booleano* encargado de indicar si el componente tiene la capacidad de delegar permisos de escritura/lectura sobre sus recursos.

- *map_res*, un mapa que asocia URIs con recursos.

- *uri*, una lista con todos los URIs pertenecientes al proveedor de contenido. El valor de este campo coincide con el listado de las claves del mapa definido anteriormente. Sin embargo, decidimos agregarlo al componente para simplificar algunas demostraciones.

Manifiesto

Como mencionamos previamente en 2.2.3, el manifiesto de una aplicación contiene información estática sobre la misma. Nuestra representación de un manifiesto está acotada a los datos relevantes para las propiedades demostradas. Fue definido a través de un registro de seis valores, conformado por:

- *cmp*, aquí se listan los componentes que conforman la aplicación.

- *targetSdk*, la versión de Android para la cual fue pensada/diseñada la aplicación.

- *minSdk*, la versión mínima de Android necesaria para poder ejecutar la aplicación. Aquellos dispositivos que cumplan con este valor pero no con el anterior, podrán ejecutar la aplicación con capacidades reducidas.

- *use*, aquí se declaran los permisos que la aplicación necesitará para funcionar correctamente. Los permisos que no estén aquí declarados no podrán ser otorgados a la aplicación bajo ninguna circunstancia.

- *usrP*, aquí se listan los permisos declarados por la aplicación.

- *appE*, aquí se declara, si corresponde, un permiso de seguridad que se puede utilizar para limitar el acceso a funciones o componentes específicos de esta aplicación.

Formalmente,

$$
\begin{aligned}
Manifest \;:=\; \{ \\
&cmp : list\ Cmp; \\
&minSdk : option\ nat \\
&targetSdk : option\ nat \\
&use : list\ idPerm \\
&usrP : list\ Perm \\
&appE : option\ Perm \\
\}
\end{aligned}
$$

Aplicaciones

A diferencia de los componentes que definimos hasta ahora, las aplicaciones instaladas por el usuario no poseen una estructura delimitada por un registro. La representación de las mismas es más abstracta: consta de un identificador (de tipo *idApp*) y de distintos *mapeos*, guardados en el estado del sistema, hacia las distintas partes que la conforman. Por ejemplo, el estado del sistema mantendrá una relación entre los identificadores de las aplicaciones (*idApp*) y su manifiesto (de tipo $Manifest$). A continuación, cuando definamos el estado, se explicitarán todos los datos asociados a las aplicaciones.

Distinto es lo que sucede con las aplicaciones pre-instaladas en el sistema. Las mismas sí cuentan con una estructura definida por un registro, con la siguiente información: un identificador, el certificado con el que fue firmada, el manifiesto y un listado de los recursos y permisos que la aplicación define. Como estas aplicaciones no pueden ser desinstaladas ni modificadas, esta información se almacenará en forma de lista en la parte estática del estado del sistema.

3.3.2. Definición de estado

Nuestra formalización del sistema de permisos de Android puede pensarse como una máquina de estado abstracta. En este modelo, los estados del sistema están conformados por dos componentes: uno que almacena la información dinámica del sistema, como las aplicaciones instaladas y los permisos otorgados a las mismas; y otro que contiene información estática como el manifiesto de cada aplicación. Formalmente:

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

El componente *State* es el que contiene la información dinámica y está conformado por los siguientes elementos:

- Una lista con los identificadores de las aplicaciones instaladas.
- Una lista con información de las aplicaciones *legacy* que ya han sido verificadas por el usuario. Es un subconjunto de las aplicaciones instaladas.
- Un mapeo entre las aplicaciones instaladas y los permisos otorgados a cada una de ellas.
- Un mapeo entre las aplicaciones instaladas y los grupos de permisos para los cuales el usuario ha permitido el otorgamiento automático de permisos individuales.
- Un mapeo entre los componentes que define una aplicación y las instancias en ejecución de ellos.
- Un mapeo entre los recursos de un proveedor de contenidos para una aplicación y los permisos permantentes otorgados sobre el mismo.
- Un mapeo entre los recursos de un proveedor de contenidos para una aplicación y los permisos temporales otorgados sobre el mismo.
- Un mapeo indicando el valor de cada uno de los recursos de una aplicación.

- Una lista de los intents que han sido enviados, junto con su emisor.

Por otro lado, el componente *Environment* contiene la siguiente información estática:

- Un mapeo que asocia a cada aplicación con su archivo de manifiesto.
- Un mapeo que asocia a cada aplicación con el certificado que se utilizó para firmarla.
- Un mapeo que asocia a las aplicaciones con los permisos definidos por ellas.
- Una lista de las aplicaciones pre-instaladas del sistema.

A continuación daremos la definición formal de ambos componentes. El orden en el que se definen los campos de cada componente es el mismo que el de las numeraciones previas.

$$
\begin{aligned}
State \;:=\; \{ \\
&apps : \; list\ idApp; \\
&alreadyVerified : \; list\ idApp; \\
&grantedPermGroups : \; mapping\ idApp\ (list\ idGrp); \\
&perms : \; mapping\ idApp\ (list\ Perm); \\
&running : \; mapping\ iCmp\ Cmp; \\
&delPPerms : \; mapping\ (idApp * CProvider * uri)\ PType; \\
&delTPerms : \; mapping\ (iCmp * CProvider * uri)\ PType; \\
&resCont : \; mapping\ (idApp * res)\ Val; \\
&sentIntents : \; list\ (iCmp * Intent) \\
\}
\end{aligned}
$$

$$
\begin{aligned}
Environment \;:=\; \{ \\
&manifest : \; mapping\ idApp\ Manifest; \\
&cert : \; mapping\ idApp\ Cert; \\
&defPerms : \; mapping\ idApp\ (list\ Perm); \\
&systemImage : \; list\ SysImgApp; \\
\}
\end{aligned}
$$

De aquí en adelante, al hablar del estado del sistema, nos estaremos refiriendo al componente *System*. En caso de que querramos referirnos a alguno de sus sub-componentes seremos explícitos con su nombre en inglés.

Estados válidos del sistema

No todos los elementos que habitan el conjunto definido anteriormente son relevantes al sistema que queremos estudiar. Por ejemplo, no queremos trabajar sobre un estado en el que una aplicación pre-instalada del sistema y una aplicación instalada por el usuario puedan tener el mismo identificador.

Inicialmente, a la hora de definir los componentes, fue necesario pensar qué condiciones debían cumplir estos estados para representar estados de Android que tengan sentido. En consecuencia, definimos una noción de **estado válido** para restringir el universo de estados a aquellos que cumplen ciertas condiciones que nos garantizarán que nuestros estados del modelo tienen sentido al compararlos con estados reales del sistema. Vale aclarar, que nuestra definición de estado válido no es completa y que, de alguna manera, está focalizada en las propiedades que probaremos luego.

Se definió formalmente un predicado *valid_state* que se satisface cuandos se cumplen las siguientes condiciones:

- Todos los componentes, ya sea que pertenezcan a una aplicación instalada por el usuario o a una aplicación pre-instalada tienen identificadores diferentes.
- Ningún componente pertenece a más de una aplicación.
- Ningún componente en ejecución es una instancia de un proveedor de contenido (los mismos no se *ejecutan*).
- Todo permiso temporalmente otorgado ha sido otorgado a un componente en ejecución y es sobre un recurso de un proveedor de contenido existente.
- Todo componente en ejecución pertenece a una aplicación instalada en el sistema.
- Toda aplicación que establece un valor a un recurso está instalada en el sistema.
- El dominio de las funciones parciales que definen el $manifest$, $cert$ y $defPerms$ es el conjunto de todas las aplicaciones instaladas por el usuario.
- El dominio de las funciones parciales que definen $grantedPermGroups$ y $perm$ es el conjunto de todas las aplicaciones del sistema, tanto instaladas por el usuario como pre-instaladas.
- Todas las aplicaciones del sistema tienen identificadores diferentes.
- Todos los permisos definidos por las aplicaciones tienen identificadores diferentes.
- Todos los permisos otorgados existen en el sistema.
- Todos los intents que han sido enviados tienen identificadores diferentes.

3.4. Acciones

Modelamos las operaciones de Android que nos interesan estudiar como un conjunto de acciones (definidas a través del tipo `Action`), donde cada una de ellas determina la manera en la que nuestro sistema puede transicionar. Las tablas siguientes resumen todas las acciones disponibles.

Acción	Descripción
`install` *app m c res*	Instala la aplicación con identificador *app*, cuyo manifiesto es m, su certificado es c y la lista de recursos es *res*.
`uninstall` *app*	Desinstala la aplicación con identificador *app*.
`read` *ic cp u*	El componente en ejecución *ic* lee el recurso correspondiente al identificador URI u del proveedor de contenido *cp*.
`write` *ic cp u val*	El componente en ejecución *ic* escribe el valor *val* en el recurso correspondiente al identificador u del proveedor de contenido *cp*.
`startActivity` *i ic*	El componente en ejecución *ic* solicita comenzar la actividad especificada por el intent i.
`startActivityRes` *i n ic*	El componente en ejecución *ic* solicita comenzar la actividad especificada por el intent i y espera como respuesta un token n.
`startService` *i ic*	El componente en ejecución *ic* solicita comenzar el servicio especificado por el intent i.
`sendBroadcast` *i ic p*	El componente en ejecución *ic* envía el intent i en modo *broadcast*, especificando que solo los componentes que posean el permiso p pueden recibirlo.
`sendOrdBroadcast` *i ic p*	El componente en ejecución *ic* envía el intent i en modo *broadcast* ordenado, especificando que solo los componentes que posean el permiso p pueden recibirlo.
`sendSBroadcast` *i ic*	El componente en ejecución *ic* envía el intent i en modo *sticky broadcast*.
`resolveIntent` *i app*	La aplicación *app* vuelve al intent i explícito.
`stop` *ic*	El componente en ejecución *ic* termina su ejecución.
`grantP` *ic cp app u op*	El componente en ejecución *ic* delega permisos permantentes a la aplicación *app*. Esta delegación autoriza a *app* a realizar la operación *op* en el recurso asignado al URI u del proveedor de contenido *cp*.
`revokeDel` *ic cp u op*	El componente en ejecución *ic* revoca los permisos otorgados al recurso u del proveedor de contenidos *cp* para realizar la operación *op*.
`call` *ic sac*	El componente en ejecución *ic* realiza el llamado a una función del sistema denominada *sac*.

Cuadro 3.2: Acciones del sistema. Parte 1.

Acción	**Descripción**
`grant` *p app*	Otorga el permiso *p* a la aplicación *app* con la confirmación del usuario.
`grantAuto` *p app*	Otorga automáticamente el permiso *p* a la aplicación *app* (sin requerir confirmación del usuario).
`revoke` *p app*	Revoca un permiso no agrupado *p* de la aplicación *app*.
`revokePermGroup` *g app*	Revoca todos los permisos pertenecientes al grupo *g* de la aplicación *app*.
`hasPermission` *p app*	Chequea si la aplicación *app* posee el permiso *p*.
`receiveIntent` *i ic app*	La aplicación *app* recibe el intent *i*, enviado por el componente en ejecución *ic*.
`verifyOldApp` *app*	El usuario verifica los permisos que han sido otorgados a la aplicación *app*. Solo se utiliza para aquellas aplicaciones que fueron instaladas previamente a la versión 6 de Android.

Cuadro 3.3: Acciones del sistema. Parte 2.

La semántica de las mismas está dada en términos de pre-condición y post-condición. Para ello, definimos los predicados Pre y $Post$ de manera tal que para que una acción a pueda transicionar el sistema desde un estado s hacia otro estado s', deberán cumplirse $Pre\ s\ a$ y $Post\ s\ s'\ a$. Notaremos la transición de un estado a otro de la siguiente manera: $s \xrightarrow{a} s'$.

A continuación y a modo de ejemplo, describiremos informalmente las acciones que han sido introducidas o modificadas con las últimas actualizaciones del sistema.

Semántica de `grant`

Esta operación es la encargada de otorgar un permiso p a una aplicación a. La misma ya estaba presente en la formalización de la que se partió en esta tesina. Sin embargo, su semántica ha sido modificada a raíz de las actualizaciones de la plataforma mencionadas en la sección 2.4.2. En particular, ahora el sistema podrá transicionar con esta operación solo si el permiso p no pertenece a un grupo o, en caso de que pertenezca, es el primero del grupo en ser otorgado a la aplicación. El resto de las precondiciones necesarias para transicionar se mantuvieron: el permiso p debe existir (es decir, debe estar definido o bien por el sistema o por alguna aplicación), debe estar declarado como usado en el manifiesto de la aplicación a, debe ser un permiso peligroso y no debe haber sido otorgado a la aplicación previamente.

Si la precondición se cumple, el sistema transicionará hacia un estado en dónde el permiso se agrega a los permisos otorgados a la aplicación, es decir, se agrega p al conjunto $perms\ a$; y en caso de corresponder, ocurre lo análogo con el grupo de p y $grantedPermGroups\ a$. El resto de los componentes del sistema no se verán modificados.

Semántica de `grantAuto`

Para modelar en su totalidad los cambios mencionados en la sección 2.4.2, además de los cambios en la semántica a la operación `grant`, se introdujo una nueva acción `grantAuto`. Con esta nueva operación se busca representar al otorgamiento automático de un permiso por parte del sistema operativo. Su semántica difiere de la anterior en que el sistema solo podrá transicionar con `grantAuto` si el permiso que se intenta otorgar pertenece a un grupo que el usuario ya ha autorizado. En caso de que esto se cumpla, el modelo mutará hacia un estado en donde la aplicación en cuestión obtuvo el permiso solicitado.

Decidimos representar el otorgamiento con consentimiento del usuario y el otorgamiento automático de permisos con dos acciones distintas para obtener una mayor granularidad en las trazas de ejecución. Esta decisión, en consecuencia, facilita la demostración de las propiedades que involucran otorgamiento de permisos.

Semántica de `revoke` y `revokePermGroup`

De manera similar a lo ocurrido con `grant` y `grantAuto`, la semántica de las operaciones `revoke` y `revokeGroup` también se modificaron con los cambios de las últimas versiones. Decidimos modelar estas operaciones de manera tal que se mantenga una relación con la experiencia de usuario al revocar permsisos. De esta manera, la operación `revoke` es la encargada de revocar permisos individuales **no** agrupados mientras que `revokePermGroup` quita todos los permisos pertenecientes al grupo deseado. En otras palabras, los permisos pertenecientes a algún grupo no pueden ser revocados de manera individual, el grupo entero debe ser invalidado.

Si el sistema transiciona con `revoke`, entonces dado un permiso p y una aplicación a, obtendremos un estado en donde la aplicación a ya no tendrá acceso a los recursos protegidos por p. Análogamente, dado un permiso g y una aplicación a', al transicionar con `revokePermGroup`, la aplicación a ya no tendrá ningún permiso perteneciente al grupo g. Además, el sistema ya no estará autorizado a otorgar a la aplicación a permisos del grupo g de manera automática.

Semántica de `verifyOldApp`

Esta operación se agregó al modelo para razonar sobre el nuevo el comportamiento mencionado en la sección 2.4.4. Dada una aplicación a, para poder transicionar el sistema utilizando la operación $verifyOldApp\ a$, deben cumplirse las siguientes condiciones: a debe ser una aplicación instalada en el sistema, aún no debe haber sido ejecutada y debe estar orientada a una versión previa a la sexta versión de Android.

En la implementación real de la plataforma, al momento de verificar una aplicación vieja se muestra al usuario un menú con los permisos otorgados a la misma en el momento en que fue instalada, junto con la posibilidad de elegir cuales de ellos se desea mantener y cuales revocar. Nuestra operación $verifyOldApp\ a$ simplemente transiciona hacia un estado en donde a la aplicación a se le han revocado todos sus permisos y se ha marcado como verificada. Para modelar la acción en la que el usuario selecciona los permisos que desea mantener mediante la interfaz ofrecida por Android, deberemos dar una sucesión de acciones, don-

de primero se verifica la aplicación y luego se otorgan los permisos que se eligió mantener.

Semántica de `receiveIntent`

A raíz del cambio mencionado previamente y en la sección 2.4.4, agregamos una nueva condición que deberá cumplirse para que una aplicación pueda recibir un *intent*: para que una aplicación a pueda recibir el intent i, entonces la misma no debe ser una aplicación *legacy* o, en caso de serlo, debe haber sido previamente verificada por el usuario. De esta manera, las aplicaciones *legacy* que queden en el sistema podrán ser visibles a la hora de resolver un *intent* (es decir, un usuario podrá observarla entre las aplicaciones disponibles para realizar determinada acción); pero en caso de elegirla, el usuario primero deberá verificar los permisos antes de que la aplicación pueda ejecutar la acción.

3.5. Ejecuciones

Cuando el sistema intenta ejecutar una acción a en un estado válido s, hay dos posibles resultados. Si la precondición de la acción se cumple, el sistema transicionará hacia otro estado s' donde la postcondición de a también se satisface. Sin embargo, si la precondición no se cumple, el sistema permanecerá en el mismo estado en el que se encontraba al intentar la ejecución de a y responderá con un mensaje de error determinado por la relación $ErrorMsg$, definida a continuación. Dados el estado s y la aplicación a mencionados previamente, y un código de error ec, la relación $ErrorMsgsaec$ se satisface si y sólo si el código ec es una respuesta aceptable cuando el sistema falla al ejecutar a en el estado s.

Formalmente, las posibles respuestas de sistema se definen a través de la siguiente semántica operacional:

$$\frac{valid\_state(s) \;\; Pre(s,a) \;\; Post(s,a,s')}{s \xrightarrow{a/ok} s'} \qquad \frac{valid\_state(s) \;\; ErrorMsg(s,a,ec)}{s \xrightarrow{a/error(ec)} s}$$

El siguiente teorema garantiza que toda ejecución mantiene la validez del estado del sistema. Su demostración se realizó utilizando el asistente de pruebas `Coq` y puede encontrarse en el archivo `ValidityInvariance.v`. Consiste, fundamentalmente, en un análisis por casos en la acción a ejecutar ya que previamente se han demostrado por separado que cada acción preserva la validez del estado.

Teorema 1 (Las ejecuciones preservan la validez del estado)**.**
$\forall\ (s\ s' : \mathsf{AndroidST})(a : \mathsf{Action})(r : Response), s \xrightarrow{a/r} s' \rightarrow valid\_state(s')$

Demostrar este tipo de invariantes facilita el razonamiento cuando se estudian otros comportamientos más específicos del sistema. En particular, todas las propiedades que mencionamos a continuación fueron probadas sobre estados válidos. De esta manera, infinidad de estados en los que la propiedad no hubiese sido verdadera fueron automáticamente descartados, dado que no se trataban de escenarios válidos.

3.6. Propiedades del modelo

En esta sección presentaremos y discutiremos las propiedades que establecimos y demostramos sobre nuestra formalización de Android. Todas las propiedades han sido demostradas utilizando el asistente de pruebas `Coq`. Nos enfocamos en propiedades de *safety*[2] aunque también formalizamos algunos comportamientos potencialmente peligrosos que no han sido considerados en la especificación informal de la plataforma.

En la tabla 3.4 introducimos algunas funciones y predicados auxiliares que nos ayudarán a definir los teoremas presentados.

Función/Predicado	**Descripción**
$getAppFromCmp(c, s)$	Dado un componente c y un estado s, devuelve la aplicación a la cual pertenece dicho componente.
$getAppRequestedPerms(m)$	Dado un manifiesto m de una aplicación, devuelve los permisos listados como usados.
$getGrantedPermsApp(app, s)$	Devuelve los permisos con los que cuenta la aplicación app en el estado s.
$getAuthorizedGroups(app, s)$	Dada una aplicación app y un estado s, devuelve los grupos de permisos que se encuentran autorizados para otorgar permisos automáticamente a dicha aplicación.
$getManifestForApp(app, s)$	Devuelve el manifiesto de la aplicación app. El estado es necesario como argumento porque el manifiesto se encuentra guardado en la parte estática del mismo.
$getPermissionId(p)$	Devuelve el identificador del permiso p.
$getPermissionLevel(p)$	Devuelve el nivel de protección del permiso p.
$getPermissionGroup(p)$	Devuelve $Some\ g$ si el permiso p pertenece al grupo g, o $None$ en caso contrario.
$getRunningComponents(s)$	Devuelve un conjunto de pares conformados por el ID de una instancia en ejecución con el componente asociado a la misma.
$oldAppNotVerified(app, s)$	Válido si y solo si la aplicación app es considerada *legacy* y el usuario aún no la ha verificado en el estado s.

Cuadro 3.4: Funciones auxiliares y predicados

A raíz del cambio mencionado en la sección 2.4.2, la primer propiedad que formulamos establece una condición necesaria para que nuestra formalización del sistema sea consecuente con la documentación de la plataforma: solo los permisos que pertenecen a un grupo ya autorizado por el usuario pueden ser otorgados de manera automática por el sistema.

[2]Utilizamos el término en inglés para diferenciarlo de *security*, dado que en la traducción al español mantener esa diferencia es más complejo.

Propiedad 1.

$\forall(s, s' : \mathsf{AndroidST})(p : \mathsf{Perm})(g : \mathsf{PermGroup})(app : \mathsf{AppId}),$
$getPermissionLevel(p) = dangerous \wedge getPermissionGroup(p) = Some\ g \wedge$
$g \notin getAuthorizedGroups(app, s) \rightarrow\ \neg s \xrightarrow{grantAuto\ p\ app/ok} s'$

El sistema de permisos de Android garantiza que un otorgamiento automático de permisos peligrosos puede ocurrir solamente para aquellos permisos que pertenecen a grupos autorizados por el usuario.

Sin embargo, algunas incógnitas surgen al intentar formalizar en qué situaciones un grupo de permisos se encuentra autorizado. Por ejemplo, observamos que existen estados válidos del sistema en los que un permiso puede ser otorgado automáticamente a una aplicación, a pesar de que **actualmente** no existan otros permisos de ese mismo grupo ya otorgados a la misma. Esta situación podría alcanzarse con la siguiente secuencia de acciones:

1. Una aplicación A declara el permiso P agrupado en el grupo G
2. Una aplicación B solicita el permiso P
3. El usuario otorga el permiso P a la aplicación B
4. La aplicación A se desinstala (y por lo tanto los permisos declarados por ella son eliminados)
5. La aplicación B puede otorgar automáticamente los permisos pertenecientes al grupo G (a pesar de que ya no cuenta con el permiso P)

Es importante mencionar que este escenario no implica que existe un falla de seguridad en el sistema. Podría ser una decisión tomada al diseñar la plataforma con la intención de evitar abrumar al usuario con advertencias y cuadros de diálogos solicitando acciones. Sin embargo, esta decisión no está clara y no es desambiguada en ningún lugar de la documentación. A continuación formalizamos el escenario descripto:

Propiedad 2.

$\exists(s : \mathsf{AndroidST})(p : \mathsf{Perm})(g : \mathsf{PermGroup})(app : \mathsf{AppId}),\ valid\_state(s) \wedge$
$getPermissionLevel(p) = dangerous \wedge getPermissionGroup(p) = Some\ g \wedge$
$\neg(\exists(p' : \mathsf{Perm}), p' \in getGrantedPermsApp(app, s)\ \wedge$
$getPermissionGroup(p') = Some\ g) \wedge Pre(s, grantAuto\ p\ a)$

El sistema puede otorgar automáticamente un permiso a pesar de que no hay otro permiso del mismo grupo actualmente otorgado a la aplicación.

La siguiente propiedad formaliza el escenario mencionado en la sección 2.4.2 sobre los permisos normales y peligrosos compartiendo el mismo grupo. Como mencionamos previamente, formalizamos el peor escenario posible que satisface

la especificación informal de la plataforma. Sin embargo, nuestra postura es que no debería permitirse que permisos de distintos niveles de protección compartan grupo ya que podría facilitar un ataque por escalamiento de privilegios. Por ejemplo, si un permiso A con nivel de protección normal comparte grupo con un permiso peligroso B, una aplicación podría obtener autorización para que B sea otorgado automáticamente en el momento en el que el sistema concede a A (es decir, en tiempo de instalación). Independientemente de si en una implementación real del sistema de permisos el usuario es notificado de esta decisión, creemos que es una situación no deseada dado que de alguna manera rompe con la idea de que los permisos peligrosos son otorgados en tiempo de ejecución.

Propiedad 3.

$\forall(s, s' : \mathsf{AndroidST})\ (a : \mathsf{AppId})\ (m : \mathsf{Manifest})\ (c : \mathsf{Cert})\ (resources : list\ \mathsf{Res})\ (g : \mathsf{PermGroup})$
$(pDang\ pNorm : \mathsf{Perm}), s \xrightarrow{install\ a\ m\ c\ resources/ok} s' \rightarrow$
$getPermissionLevel(pDang) = dangerous\ \rightarrow$
$getPermissionGroup(pDang) = Some\ g \rightarrow$
$getPermissionLevel(pNorm) = normal\ \rightarrow$
$getPermissionGroup(pNorm) = Some\ g \rightarrow$
$\{pDang,\ pNorm\} \subseteq getAppRequestedPerms(m) \rightarrow$
$Pre(s', grantAuto\ pDang\ a)$

Una aplicación que usa un permiso normal y uno peligroso del mismo grupo de permisos, puede obtener el segundo automáticamente luego de ser instalada.

Como mencionamos previamente, los usuarios tienen la posibilidad de revocar cualquier permiso previamente otorgado en cualquier momento. Sin embargo, en el caso de los permisos agrupados, no es posible hacerlo de manera granular. El grupo entero debe ser invalidado. Creemos que esta situación es deseable, considerando que al otorgar un permiso agrupado el sistema concede cierto privilegio sobre el grupo entero (en lugar del permiso solicitado en sí). Por lo tanto, probamos que nuestro sistema es consistente con este comportamiento.

Propiedad 4.

$\forall(s, s' : \mathsf{AndroidST})\ (g : \mathsf{PermGroup})\ (app : \mathsf{AppId}),$
$s \xrightarrow{revokePermGroup\ g\ app/ok} s' \rightarrow$
$\neg(\exists(p : \mathsf{Perm}), p \in getGrantedPermsApp(app, s')$
$\wedge\ getPermissionGroup(p) = Some\ g)$

Cuando un usuario revoca el acceso a un grupo de permisos para determinada aplicación, todos los permisos individuales son revocados también.

El último cambio incorporado al modelo fue el que mencionamos en la sección 2.4.4. El mismo agrega restricciones a las acciones que pueden ser ejecutadas por

aplicaciones orientadas a versiones viejas de la plataforma, dado que las mismas consiguieron todos los permisos en tiempo de instalación. La siguiente propiedad establece que ninguna aplicación *legacy* que **no ha sido verificada aún** puede recibir *intents*. De esta manera, ninguna de esas aplicaciones estará en condiciones de iniciar nuevas actividades o servicios maliciosos con los permisos adquiridos en la instalación.

Propiedad 5.

$\forall(s, s' : \mathsf{AndroidST})\ (i : \mathsf{Intent})\ (ic : \mathsf{iComp})\ (app : \mathsf{AppId}),$
$oldAppNotVerified(app, s) \rightarrow$
$\neg s \xrightarrow{receiveIntent\ i\ ic\ app/ok} s'$

Una aplicación vieja que no ha sido verificada por el usuario no está autorizada a recibir intents.

Finalmente, incluimos una propiedad que ha estado vigente en el modelo desde que fue actualizado para incluir los cambios de la versión 6 de Android. Esta propiedad sigue siendo válida luego de la actualización del modelo. Cualquier aplicación que desee enviar información por internet deberá contar con un permiso llamado `INTERNET`. Sin embargo, como el nivel de protección del mismo es "normal", basta con listarlo en la sección de permisos usados del manifiesto para conseguirlo. Una vez más, criticamos esto dado que facilita escenarios de fuga de información. La propiedad siguiente formaliza este comportamiento, presentando un argumento razonable para volver atrás este cambio introducido en Android Marshmallow.

Propiedad 6.

$\forall(s : \mathsf{AndroidST})(sac : SACall)(c : \mathsf{Comp})(ic : \mathsf{iComp})(p : \mathsf{Perm}),$
$valid\_state(s) \rightarrow permSAC(p, sac) \rightarrow$
$getPermissionLevel(p) = normal \rightarrow getPermissionId(p) \in$
$getAppRequestedPerms(getManifestForApp(getAppFromCmp(c, s), s))$
$\rightarrow (ic, c) \in getRunningComponents(s) \rightarrow s \xrightarrow{call\ ic\ sac/ok} s$

Si la ejecución de un llamado a la API de Android solo requiere permisos con nivel de protección normal, basta con listar dicho permiso en el manifiesto para estar habilitado a realizar el llamado.

Más propiedades del modelo pueden encontrarse en la especificación completa [25]. No fueron incluidas en este informe dado que no están estrictamente relacionadas a los cambios introducidos por las últimas versiones de la plataforma. Sin embargo, todas las propiedades aún presentes en la especificación han sido ratificadas, demostrando de esta manera que dichos comportamientos se mantuvieron durante las sucesivas evoluciones de Android.

Capítulo 4

Implementación de un prototipo certificado

Formalizar sistemas delicados en términos de seguridad, como un sistema de permisos de una plataforma usada masivamente, es en sí una herramienta teórica interesante. Permite establecer propiedades sobre el sistema en cuestión y razonar formalmente sobre ellas. Mantener esa formalización actualizada, por su parte, otorga la posibilidad de estudiar las características más recientes de la plataforma, y en muchas situaciones, detectar comporatmientos erróneos o inseguros antes de que sean aprovechados maliciosamente.

Por otro lado, contar con una especificación formal de un sistema permite desarrollar implementaciones *certificadas*. Una implementación certificada es una implementación que cumple correctamente con una especificación y que dicha corrección ha sido demostrada matemáticamente. Este trabajo además de extender la especificación del sistema de permisos de Android, extiende la implementación de la plataforma desarrollada en los trabajos previos [9, 27]. Claro está, que al partir de un modelo tan abstracto como el nuestro, la implementación obtenida no estaría en condiciones de sustituir a la implementación real de Android. Sin embargo, sí podría utilizarse como un monitor de referencia [2].

A continuación, describiremos las principales características de nuestra implementación del sistema de permisos de Android y algunas propiedades demostradas sobre la misma.

4.1. Características principales

Nuestra implementación de la plataforma consiste de un conjunto de funciones definidas en Coq de manera tal que por cada acción dada en la especificación existe una función encargada de llevarla a cabo. Estas funciones son, básicamente, transformadores de estado. Si bien sus definiciones son distintas entre sí, pues tienen semánticas muy diferentes, todas respetan el mismo patrón. Primero, en un estado inicial, se evalúa si una expresión *booleana* equivalente a la precondición de la acción se satisface. En caso de que la misma se cumpla, se ejecuta una función auxiliar encargada de mutar el estado de manera tal que el nuevo estado cumpla con la postcondición establecida por la acción del modelo. En caso de que la precondición no se cumpla, el estado no sufre cambios y se devuelve un

mensaje de error.

A modo de ejemplo, explicaremos con un poco más de detalle la implementación de la acción *grant*. En la figura 4.1 puede observarse su definición formal. La definición formal completa de esta y el resto de las operaciones puede encontrarse en el código en GitHub [25].

$$
\begin{array}{l}
\textbf{Definition}\ grant\_safe(perm, app, s)\ : Result\ := \\
\quad \textbf{match}\ grant\_pre(perm, app, s)\ \textbf{with} \\
\qquad \mid Some\ ec\ \Rightarrow\ \{error(ec), s\} \\
\qquad \mid None\ \Rightarrow\ \{ok, grant\_post(perm, app, s)\} \\
\quad \textbf{end}.
\end{array}
$$

Figura 4.1: Definición de la función $grant\_safe$, encargada de ejecutar la acción *grant*.

La función *grant_pre* se define como la conjunción de todas las condiciones establecidas por la precondición de *grant*. En ella también se especifica el error que debe mostrarse cuando alguna de ellas no se cumple. Por otro lado, la función *grant_post* implementa la modificación esperada en el estado: el permiso *perm* pasa a formar parte de los permisos otorgados a la aplicación *app* y, si el mismo estuviera agrupado, su grupo también se agregaría al listado de grupos autorizados por el usuario para el otorgamiento automático de permisos a la aplicación correspondiente.

Modelamos el resultado de la operación `grant_safe` con el siguiente tipo de datos:

$$Response\ :=\ ok \mid error\ ec$$

donde *ec* es un código de error definido de manera enumerada por `ErrorCode`

$$Result\ =\ \{response:\ Response;\ st:\ System\}$$

Luego, tanto a $grant\_safe$ como al resto de las operaciones definidas de manera análoga, se las agrupa en una función llamada *step*, que actúa principalmente como un despachador de acciones. En la figura 4.2 se muestra la estructura de la misma.

$$
\begin{array}{l}
\textbf{Definition}\ step(s, a)\ := \\
\quad \textbf{match}\ a\ \textbf{with} \\
\qquad \mid \ldots \Rightarrow\ \ldots \\
\qquad \mid \texttt{grant}\ perm\ app\ \Rightarrow\ grant\_safe(perm, app, s) \\
\qquad \mid \texttt{grantAuto}\ perm\ app\ \Rightarrow\ grantAuto\_safe(perm, app, s) \\
\qquad \mid \ldots \Rightarrow\ \ldots \\
\quad \textbf{end}.
\end{array}
$$

Figura 4.2: Estructura de la función *step*

Finalmente, definimos una función *trace* que captura la idea de ejecutar acciones sucesivas en el sistema. Dado un estado inicial y una lista de acciones,

esta función será la encargada de ir uniéndolas de manera tal que el estado obtenido como resultado de una acción sea utilizado como estado inicial de la siguiente. Mostramos su definición a continuación.

Function *trace* (*s* : AndroidST) (*actions* : *list* Action) : *list* AndroidST :=
  match *actions* **with**
    | *nil* ⇒ *nil*
    | *action* :: *rest* ⇒ **let** *s*′ := (*step s action*).*st* **in** *s*′ :: *trace s*′ *rest*
  end.

4.2. Corrección de la implementación

Como mencionamos previamente, contar con una formalización de un sistema permite desarrollar sistemas correctos, confiables, en el sentido de tener garantías de que el comportamiento de los mismos será adecuado según lo dictamine la especificación de cada uno. En este caso, demostramos un teorema en Coq que garantiza que cualquier acción ejecutada por nuestra implementación, partiendo de un estado válido, es un ejecución correcta del sistema.

Teorema 2 (Corrección del monitor de referencia)**.**
∀ (*s* : AndroidST) (*a* : Action),
  $valid\_state(s) \rightarrow s \xrightarrow{a/step(s,a).resp} step(s,a).st$

Comenzamos la demostración de este teorema haciendo inducción en la acción a ejecutar. De esta manera, podemos subdividir la prueba en lemas más pequeños, y demostrar la corrección de cada una de las operaciones por separado. Luego, cada uno de estos sub-lemas posee la misma estructura: dada una acción, demostraremos por un lado que la ejecución es correcta cuando se cumple la pre-condición; y por otro, demostraremos que la ejecución también lo es cuando la pre-condición no se satisface. A modo de ejemplo, presentamos el caso de la operación *grant*.

Lema 1 (Corrección de una ejecución válida de *grant*)**.**
∀ (*s* : AndroidST) (*app* : AppId) (*p* : Perm),
  $valid\_state(s) \rightarrow Pre(s, grant\ p\ a) \rightarrow$
  $Post(grant\_safe(p, app, s).st,\ grant\ p\ a)$

Lema 2 (Corrección de una ejecución errónea de *grant*)**.**
∀ (*s* : AndroidST) (*app* : AppId) (*p* : Perm),
  $valid\_state(s) \rightarrow \neg Pre(s, grant\ p\ a) \rightarrow$
  $grant\_safe(p, app, s).st = s \wedge$
    $(\exists\ (ec : \mathsf{ErrorCode}),\ grant\_safe(p,\ app,\ s).response = error\ ec$
      $\wedge\ ErrorMsg\ s\ (grant\ p\ a)\ ec)$

Con esos dos lemas demostrados, obtener una prueba de la corrección de la operación *grant* se reduce, a grandes rasgos, a diferenciar los casos en que la precondición se cumple de lo que no, para luego aplicar el sub-lema correspondiente.

Lema 3 (Corrección de la operación *grant*)**.**
$\forall$ (s : AndroidST) (app : AppId)(p : Perm),
$valid\_state(s) \rightarrow$
$s \xrightarrow{grant\ p\ a\ /\ step(s,\ grant\ p\ a).response} step(s, grant\ p\ a).st$

La demostración completa del lema 3 puede encontrarse en el módulo *GrantIsSound.v* en el código de la formalización [25]. Análogamente, la demostración para cada una de las otras acciones puede encontrarse en el módulo *ActionIsSound.v* correspondiente. La demostración del teorema 2 se encuentra en el módulo *Soundness.v*.

4.3. Propiedades sobre la implementación

De la misma manera en la que en la sección 3.6 describimos las propiedades demostradas sobre la especificación, a continuación presentaremos algunas propiedades que fueron demostradas sobre nuestra implementación. Las mismas, en general, establecen condiciones de *safety* sobre ciertas trazas de ejecución que consideramos de interés.

La primer propiedad establece que si nuestra implementación permite que una aplicación *legacy* sea ejecutada normalmente, entonces podemos asegurar que el usuario la ha verificado previamente. Para generalizar esta situación, definimos una traza t de longitud l, de la que solo sabemos que:

- En el primer estado de la traza la aplicación **no** está en condiciones de ser ejecutada
- En el último estado de la traza la aplicación **sí** puede ser ejecutada

Además de estas condiciones necesarias y relevantes para la propiedad en cuestión, requerimos que durante la ejecución de la traza t la aplicación no sea desinstalada. Esto es necesario porque, dado el nivel de abstracción del modelo, si desinstalamos la aplicación e instalamos una nueva, ambas aplicaciones podrían recibir el mismo identificador. De esta manera, al evaluar una condición en el estado inicial de la traza estaríamos referenciando a una aplicación; mientras que al evaluar la misma condición en el estado final, estaríamos refiriéndonos a una aplicación distinta.

Propiedad 7.
$\forall(initState, lastState$: AndroidST$)(app : AppId)(l : list$ Action$)$,
$valid\_state(initState) \rightarrow app \in getInstalledApps(initState) \rightarrow$
$oldAppNotVerified(a, initState) \rightarrow canRun(a, lastState) \rightarrow$
$last(trace(initState, l), initState) = lastState \rightarrow$
$uninstall\ app \notin l \rightarrow verifyOldApp\ app \in l$

Si una aplicación vieja puede ser ejecutada entonces debe haber sido autorizada por el usuario previamente.

Similarmente, la siguiente propiedad establece que si al comenzar una traza de ejecución una aplicación no posee un permiso peligroso y en el estado final sí,

entonces en algún momento de la ejecución de esa traza el permiso fue explícitamente otorgado. Notar que "explícitamente" significa que el permiso puede haber sido otorgado por el usuario o automáticamente por el sistema. Lo que se busca resaltar es que hay una entidad responsable de ese otorgamiento. Esta propiedad es importante para marcar la diferencia con lo que ocurría en versions previas de Android, explicado en la sección 2.4.2.

Propiedad 8.
$\forall(initState, lastState : \mathsf{AndroidST})(app : AppId)(p : \mathsf{Perm})(l : list\ \mathsf{Action}),$
$valid\_state(initState) \rightarrow app \in getInstalledApps(initState) \rightarrow$
$getPermissionLevel(p) = dangerous \rightarrow$
$appHasPermission(app, p, lastState) \rightarrow$
$\neg appHasPermission(app, p, initState) \rightarrow \texttt{uninstall}\ app \notin l \rightarrow$
$last(trace(initState, l), initState) = lastState \rightarrow (\texttt{grant}\ p\ app \in l \vee \texttt{grantAuto}\ p\ app \in l)$

La única forma de que una aplicación consiga un permiso es si el usuario lo autoriza, o si el usuario había autorizado un grupo previamente y el sistema puede otorgar el permiso automáticamente.

La última propiedad que presentaremos en esta tesina demuestra que la única forma de recuperar un permiso peligroso luego de que el mismo fue revocado es si el mismo se otorga nuevamente mediante las operaciones conocidas.

Propiedad 9.
$\forall(initState, sndState, lastState : \mathsf{AndroidST})(app : \mathsf{AppId})(p : \mathsf{Perm})(l : list\ \mathsf{Action}),$
$valid\_state(initState) \rightarrow getPermissionLevel(p) = dangerous \rightarrow$
$p \notin getDefPermsForApp(app, initState) \rightarrow$
$step(initState, \texttt{revoke}\ p\ app).st = sndState \rightarrow$
$step(initState, \texttt{revoke}\ p\ app)).resp = ok \rightarrow$
$\texttt{uninstall}\ app \notin l \rightarrow \texttt{grant}\ p\ app \notin l \rightarrow \texttt{grantAuto}\ p\ app \notin l \rightarrow$
$last(trace(sndState, l), sndState) = lastState \rightarrow$
$\neg appHasPermission(app, p, lastState)$

Si un permiso fue revocado, solo volver a otorgarlo con las acciones establecidas permitirá que una aplicación vuelva a tenerlo.

Más propiedades sobre nuestra implementación del modelo pueden encontrarse en el repositorio que contiene el código en `Coq` [25]. Sin embargo, decidimos no agregarlas a este informe ya que no están estrictamente relacionadas con los cambios introducidos en esta tesina. Sin embargo, dichas propiedades aún son válidas en la versión 10 de Android.

Capítulo 5

Trabajos relacionados

En esta sección se presentan algunos trabajos académicos relacionados con el modelo de seguridad de Android. Se organizará de la siguiente manera: primero, se describen trabajos informales[1] que son de interés por abarcar características nuevas, por la información que condensan o por ser pioneros en el área. Luego, se mencionan varias herramientas (o aplicaciones) que utilizan técnicas de análisis estático de código para detectar potenciales vulnerabilidades, todas desarrolladas dentro de un marco académico. A pesar de que este enfoque es distinto al que se propone en esta tesina, se tomó la decisión de incluirlo en este capítulo ya que conforma gran parte de la investigación actual y de los últimos años. Por último, se analizan aquellos trabajos que utilizan métodos formales para dar una especificación de la plataforma. Estos trabajos se describen con más detalle que los anteriores e incluso se contemplan artículos que no son recientes pero presentan un enfoque novedoso o relevante a este trabajo.

Dentro del primer grupo, se encuentran artículos como el de William Enck *et al.* [15], uno de los primeros trabajos académicos en describir el modelo de seguridad de Android. Los autores buscaban *desenmascarar* la complejidad a la que debían enfrentarse los desarrolladores cuando se proponían construir aplicaciones seguras. A pesar de la antigüedad de este artículo, hoy en día sigue siendo relevante por la explicación concreta y concisa sobre las bases del sistema de permisos de Android. También se encuentra el trabajo de Wang y Wu [44], quienes resumen las vulnerabilidades que existen en el sistema de comunicación entre componentes, para luego discutir sobre el estado del arte en la detección y prevención de las mismas.

Recientemente, René Mayrhofer *et al.* [29] realizaron un trabajo similar para la versión 9.0 de Android. En él, se definen los principios de seguridad del sistema y un modelo de amenazas *por capas*, a partir cual se derivan amenazas concretas que dependen de la cercanía entre atacante y dispositivo móvil. Además, este artículo analiza los cambios que se fueron introduciendo en el sistema operativo para mitigar estas amenazas.

Este tipo de trabajos constituyen un complemento importante a la documentación oficial de Android, brindándole referencias más claras a los desarrolladores y nuevas herramientas que permiten resguardar sus aplicaciones. Un ejemplo de esto fue el trabajo de Felt *et al.* [16], quienes estudiaron un grupo de apli-

[1] Entendemos por trabajo informal a aquellos que describen coloquialmente el sistema, por más rigurosa que sea la descripción

caciones disponibles para la versión 2.2 de Android y detectaron que muchas de ellas pedían más permisos de los que realmente necesitaban. Los autores investigaron las causas de sobreprivilegio de estas aplicaciones y encontraron que, muchas veces, los desarrolladores intentaban otorgar la menor cantidad de privilegios necesarios, pero en reiteradas ocasiones fallaban por falta de una documentación precisa. En consecuencia, el grupo desarrolló Stowaway, una de las primeras herramientas dedicadas a la detección de permisos innecesarios.

Actualmente existe una gran cantidad de herramientas de análisis estático que ayudan a detectar sobreprivilegios o un flujo de información indebido, siendo las más recientes: M-Perm [13], IC3 [32], Droidtector [46], Covert [4] y Separ [6]. Las dos últimas combinan análisis estático con métodos formales para inferir automáticamente propiedades sobre un conjunto de aplicaciones y luego derivar políticas de seguridad. La gran oferta que existe de este tipo de aplicaciones da lugar a trabajos como el Lina Qiu *et al.* [36], en el que se analizan y comparan las herramientas, que según los autores las más destacadas son: Flowdroid combinada con IccTA [24], Amandroid [45] y DroidSafe [18].

Abordar la seguridad de Android con este enfoque presenta una diferencia fundamental con respecto a la que se propone en esta tesina: estas herramientas se focalizan en proteger una aplicación en particular (o en algunos casos, un conjunto de aplicaciones); mientras que una especificación formal de la plataforma subyacente permite extraer propiedades relevantes a **todas** las aplicaciones y al sistema en general. Por otro lado, este tipo de aplicaciones son un recurso más accesible para los usuarios que desean resguardar información sensible sin ser especialistas en el área.

Entrando en el terreno de los métodos formales, nos encontramos en primer lugar con el trabajo de Chaudhuri [12]. En el mismo, se desarrolló un lenguaje que permite describir un subconjunto de aplicaciones de Android y razonar sobre ellas. Adicionalmente, se presentó un sistema de tipos para este lenguaje y se demostró un teorema que garantiza que las aplicaciones bien tipadas preservan la confidencialidad de los datos que manejan. Parcialmente inspirado en este trabajo, Bugliesi *et al.* desarrollaron π-Perm [11], un sistema de tipos y efectos que tiene como finalidad detectar problemas de *privilege escalation*. De manera análoga al trabajo de Chaudhuri, una expresión bien tipada en π-Perm garantiza que la aplicación real a la cual está representando no es vulnerable al ataque mencionado. Similarmente, Armando *et al.* definen un lenguaje [3], acompañado por su semántica operacional, que permite describir interacciones entre aplicaciones. Al igual que en los trabajos anteriores, se define un sistema de tipos y efectos; pero este está basado en un formalismo del estilo del álgebra de procesos, conocido como *history expressions* [8]. Puesto en términos simples, una *history expression* sirve para representar los efectos laterales vinculados a la seguridad del dispositivo, que se producen al realizar una computación. Finalmente, los autores prueban que cualquier comportamiento que la plataforma pueda tener en tiempo de ejecución está contenido en este modelo; y por lo tanto, puede analizarse estáticamente.

Recientemente, Wilayat Khan *et al.* [21] retomaron el trabajo de Chaudhuri y modelaron el lenguaje en él definido dentro del *framework* lógico-matemático Coq [43]. De esta forma, pudieron no solo estudiar la corrección y seguridad de las aplicaciones de manera mecánica y rigurosa, sino que también utilizaron este asistente para probar la corrección -o *soundness*- del lenguaje en sí. En otro trabajo actual en el que participó Khan [22], se definió en Coq un modelo para

estudiar el sistema de comunicación entre componentes. El principal objetivo de este trabajo es analizar la robustez de la plataforma cuando una aplicación detiene su ejecución a causa de un fallo en la resolución de un *intent*. A diferencia del resto de los trabajos citados, éste se concentra en estudiar propiedades de *safety* y no de *security*, a pesar de que los *intents* pueden ser explotados para filtrar información sensible de los usuarios [24].

Por otra parte, Sadeghi *et al.* [37] presentan una formalización de la plataforma escrita en TLA+, un lenguaje de especificación basado en la lógica lineal temporal [23]. Al incoporar el aspecto temporal al modelo, los autores buscan definir propiedades cuya veracidad dependa del momento en el que se la evalúe y de esta forma, modelar el comportamiento del sistema a medida que evoluciona en el tiempo. Luego, proponen un monitor de seguridad que otorga permisos temporales a las aplicaciones siempre y cuando se cumplan todas las propiedades (o reglas) de seguridad previamente definidas. Este permiso "prestado" es automáticamente revocado si en algún momento el sistema se encuentra en un estado que compromete alguna de las reglas.

Similarmente, Bagheri *et al.* [5] proponen una formalización del sistema de permisos de Android escrita en Alloy [30]. Alloy es un lenguaje basado en la lógica relacional de primer orden, que incorpora una herramienta capaz de realizar análisis de satisfacibilidad automáticos sobre los modelos en él descriptos. Con la ayuda de esta formalización, los autores identificaron distintos tipos de vulnerabilidades que permiten esquivar el chequeo de permisos. Particularmente, estudiaron la vulnerabilidad de permisos personalizados, mediante la cual una aplicación maliciosa puede acceder a todos los recursos de otra que estén protegidos por permisos personalizados. Esta falla surge de que el sistema no impone restricciones con respecto al nombre de los nuevos permisos que definen y, como consecuencia, dos permisos distintos podrían tener el mismo nombre. Este trabajo luego se extendió para una nueva versión de Android [7]. La falla por permisos personalizados había sido reportada previamente por Shin *et al.* [39]. Una diferencia fundamental entre este enfoque y el de esta tesina es el tipo de análisis que se realizó. A pesar de que Alloy es capaz de producir contraejemplos de manera automática, algo realmente útil a la hora de buscar potenciales fallas; no es posible demostrar propiedades de una manera rigurosa y formal.

En trabajos previos encabezados por Gustavo Betarte y Carlos Luna [9, 10, 27], se utilizó el asistente de pruebas Coq para modelar un sistema de transición de estados que representa, principalmente, los distintos estados que atraviesa la plataforma cuando se realizan operaciones sobre ella (por ejemplo, al instalar o desinstalar una aplicación). A partir de esta especificación, no solo se probaron propiedades relevantes a la seguridad del modelo, sino que también se extrajo una implementación certificada del mismo. Los autores explican cómo esta implementación puede utilizarse para generar casos de pruebas abstractos dentro del *testing* basado en modelos, o bien, cómo puede usarse para monitorear las acciones realizadas en un sistema real y evaluar si las propiedades deseadas efectivamente se cumplen. Estos trabajos presentan el modelo que fue actualizado y extendido en esta tesina.

Un enfoque similar a este, es el de Wook Shin *et al.*, quienes también utilizaron Coq para modelar el sistema de permisos de Android [38]. Sin embargo, esta formalización no considera aspectos de la plataforma que sí son considerados por los trabajos anteriores (y por ende, por esta tesina); como por ejemplo, los dis-

tintos tipos de componentes, la interacción entre instancias de aplicaciones en ejecución y el sistema, la operación de escritura/lectura en un *content provider* y la semántica del sistema de delegación de permisos. Al mismo tiempo, cuando Android incorporó los permisos otorgados en tiempo de ejecución, el modelo de Shin *et al.* no fue actualizado. El trabajo de Fragkaki *et al.* tambień presenta un modelo formal basado en transiciones de estado [17], pero el mismo no está desarrolado dentro de un *framework* que permita realizar pruebas asisistidas por computadora. Además, el modelo se corresponde con una de las primeras versiones de Android, por lo que tampoco contempla los cambios más recientes en el sistema de permisos.

Capítulo 6

Conclusiones y trabajo futuro

En esta tesina hemos actualizado una especificación formal previa del modelo de permisos de Android [27, 10], agregando funcionalidades nuevas y modificando la semántica de algunas operaciones ya existentes de acuerdo a los cambios introducidos en las versiones 7, 8, 9 y 10 de la plataforma. Con un acercamiento conservador, primero revisamos las propiedades ya existentes del modelo para verificar si seguían siendo válidas. Algunas de ellas fueron mínimamente modificadas por cambios en el modelo que no las afectaban directamente y otras han cambiado radicalmente para adaptarse a las últimas modificaciones introducidas. También se han agregado propiedades completamente nuevas. Entre las propiedades agregadas, se han incluido varias cuyo principal objetivo es resaltar cómo los métodos formales pueden ayudar a desambiguar comportamientos poco claros que pueden ser inferidos a partir de una especificación informal. En particular, la propiedad 3 presenta un escenario en donde una aplicación podría conseguir acceso a los permisos peligrosos de una aplicación sin consentimiento explícito del usuario. Este escenario es posible dentro de la especificación informal descripta en la documentación oficial de Android.

Además, como consecuencia de haber actualizado la formalización, hemos actualizado la implementación funcional del monitor de referencia con las nuevas características y hemos actualizado su prueba de corrección. El código Coq en su totalidad puede encontrarse en Github [25] y contiene alrededor de 23000 líneas de código incluyendo las pruebas.

El objetivo más importante de esta tesina es tratar de mantener al día esta formalización del sistema de permisos de Android para formar un *framework* confiable que nos permita razonar y establecer propiedades sobre el mismo. Utilizar modelos idealizados y prototipos certificados es algo positivo *per se*, que permite generar más transparencia en los mecanismos de seguridad de Android y da lugar a un entendimiento más claro sobre el funcionamiento de la plataforma. Sin embargo, no hay duda que el paso definitivo sería que el framework que construimos nos permita establecer garantías sobre la implementación real de la plataforma. En pos de ese objetivo, hay mucho trabajo por delante aún. Por ejemplo, es posible utilizar el programa extraído en *Haskell* para comparar los resultados producidos por las acciones reales de la plataforma con las acciones

del monitor de referencia. Esto permitiría monitorear las acciones que se llevan a cabo en una implementación real de Android y evaluar, en tiempo real, si las políticas de seguridad se cumplen o no. La metodología para lograr ese objetivo ya fue descripta en el trabajo a partir del cual esta tesina comenzó [27].

En Septiembre del 2020, la versión 11 de Android fue lanzada oficialmente. Esta versión incluye algunos cambios que siguen apostando a mejorar la seguridad del sistema de permisos, como por ejemplo permisos que se eliminan automáticamente cuando las aplicaciones entrán en desuso por mucho tiempo; o permisos que pueden utilizarse una única vez en los recursos más sensibles, como la cámara o el micrófono. Al momento de redactar esta tesina, la versión 12 de Android se encuentra en *beta*. En ella, se expande el comportamiento del restablecimiento automático de permisos que se introdujo en Android 11 con el concepto "el estado de hibernación". Fundamentalmente, una aplicación entra en estado de hibernación cuando ha estado en desuso por algunos meses. En ese estado, la misma no podrá ejecutar servicios ni alertas en segundo plano. Estos cambios podrían ser formalizados y agregados al modelo en el futuro.

Bibliografía

[1] International Data Corporation (IDC). *Smartphone Market Share*. 2021.

[2] J. P. Anderson. *Computer Security technology planning study*. Inf. téc. Deputy for Command y Management System, USA, 1972. URL: http://csrc.nist.gov/publications/history/ande72.pdf.

[3] Alessandro Armando, Gabriele Costa y Alessio Merlo. "Formal Modeling and Reasoning about the Android Security Framework". En: *Trustworthy Global Computing*. Ed. por Catuscia Palamidessi y Mark D. Ryan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, págs. 64-81. ISBN: 978-3-642-41157-1.

[4] H. Bagheri y col. "COVERT: Compositional Analysis of Android Inter-App Permission Leakage". En: *IEEE Transactions on Software Engineering* 41.9 (sep. de 2015), págs. 866-886. ISSN: 0098-5589. DOI: 10.1109/TSE.2015.2419611.

[5] H. Bagheri y col. "Detection of design flaws in the Android permission protocol through bounded verification". En: *Proceedings of the 2015 International Symposium on Formal Methods* volume 9019 of Lecture Notes in Computer Science (2015), págs. 73-89.

[6] H. Bagheri y col. "Practical, Formal Synthesis and Automatic Enforcement of Security Policies for Android". En: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Jun. de 2016, págs. 514-525. DOI: 10.1109/DSN.2016.53.

[7] Hamid Bagheri y col. "A Formal Approach for Detection of Security Flaws in the Android Permission System". En: *Springer Journal on Formal Aspects of Computing* (2017). DOI: 10.1007/s00165-017-0445-z.

[8] Massimo Bartoletti y col. "Types and Effects for Resource Usage Analysis". En: *Proceedings of the 10th International Conference on Foundations of Software Science and Computational Structures*. FOSSACS'07. Braga, Portugal: Springer-Verlag, 2007, págs. 32-47. URL: http://dl.acm.org/citation.cfm?id=1760037.1760043.

[9] Gustavo Betarte y col. "A certified reference validation mechanism for the permission model of Android". En: *CoRR* abs/1709.03652 (2017). arXiv: 1709.03652. URL: http://arxiv.org/abs/1709.03652.

[10] Gustavo Betarte y col. "Formal Analysis of Android's Permission-Based Security Model". En: *Scientific Annals of Computer Science* 26 (jun. de 2016), págs. 27-68. DOI: 10.7561/SACS.2016.1.27.

[11] Michele Bugliesi, Stefano Calzavara y Alvise Spanò. "Lintent: Towards Security Type-Checking of Android Applications". En: *Formal Techniques for Distributed Systems*. Ed. por Dirk Beyer y Michele Boreale. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, págs. 289-304. ISBN: 978-3-642-38592-6.

[12] Avik Chaudhuri. "Language-based Security on Android". En: *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. PLAS '09. Dublin, Ireland: ACM, 2009, págs. 1-7. ISBN: 978-1-60558-645-8. DOI: 10.1145/1554339.1554341. URL: http://doi.acm.org/10.1145/1554339.1554341.

[13] P. Chester y col. "M-Perm: A Lightweight Detector for Android Permission Gaps". En: *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. Mayo de 2017, págs. 217-218. DOI: 10.1109/MOBILESoft.2017.23.

[14] MDN Web Docs. *MIME types*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types.

[15] W. Enck, M. Ongtang y P. McDaniel. "Understanding Android Security". En: *IEEE Security Privacy* 7.1 (ene. de 2009), págs. 50-57. ISSN: 1540-7993. DOI: 10.1109/MSP.2009.26.

[16] Adrienne Porter Felt y col. "Android Permissions Demystified". En: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. Chicago, Illinois, USA: ACM, 2011, págs. 627-638. ISBN: 978-1-4503-0948-6. DOI: 10.1145/2046707.2046779. URL: http://doi.acm.org/10.1145/2046707.2046779.

[17] Elli Fragkaki y col. "Modeling and Enhancing Android's Permission System". En: *Computer Security – ESORICS 2012*. Ed. por Sara Foresti, Moti Yung y Fabio Martinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, págs. 1-18. ISBN: 978-3-642-33167-1.

[18] Michael Gordon y col. "Information-Flow Analysis of Android Applications in DroidSafe". En: ene. de 2015. DOI: 10.14722/ndss.2015.23089.

[19] F. Gorostiaga. *Especificación e implementación de un prototipo certificado del sistema de permisos de Android*. Tesina de grado, Licenciatura en ciencias de la computación, Universidad Nacional de Rosario, Argentina, 2016.

[20] *Haskell Language*. https://haskell.org/.

[21] W. Khan y col. "Formal Analysis of Language-Based Android Security Using Theorem Proving Approach". En: *IEEE Access* 7 (2019), págs. 16550-16560. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2895261.

[22] Wilayat Khan y col. "CrashSafe: A Formal Model for Proving Crash-safety of Android Applications". En: *Hum.-centric Comput. Inf. Sci.* 8.1 (dic. de 2018), 144:1-144:24. ISSN: 2192-1962. DOI: 10.1186/s13673-018-0144-7. URL: https://doi.org/10.1186/s13673-018-0144-7.

[23] Leslie Lamport. *TLA+*. URL: https://lamport.azurewebsites.net/tla/tla.html.

[24] L. Li y col. "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps". En: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. Mayo de 2015, págs. 280-291. DOI: 10.1109/ICSE.2015.48.

[25] Guido De Luca. *Coq code of Android formalization*. URL: https://github.com/g-deluca/android-coq-model.

[26] Guido De Luca y Carlos Luna. "Towards a Certified Reference Monitor of the Android 10 Permission System". En: *26th International Conference on Types for Proofs and Programs, TYPES 2020, March 2-5, 2020, University of Turin, Italy*. Ed. por Ugo de'Liguoro, Stefano Berardi y Thorsten Altenkirch. Vol. 188. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 3:1-3:18. DOI: 10.4230/LIPIcs.TYPES.2020.3. URL: https://doi.org/10.4230/LIPIcs.TYPES.2020.3.

[27] Carlos Luna y col. "A formal approach for the verification of the permission-based security model of Android". En: *CLEI Electronic Journal* 21 (ago. de 2018), 3:1-3:22. DOI: 10.19153/cleiej.21.2.3.

[28] *Maybe Monad*. https://wiki.haskell.org/Maybe.

[29] R. Mayrhofer y col. *The Android Platform Security Model*. 2019. URL: https://arxiv.org/abs/1904.05572.

[30] Software Design Group at MIT. *Alloy*. URL: http://alloytools.org/.

[31] *OCaml Language*. https://ocaml.org/.

[32] Damien Octeau y col. "Composite Constant Propagation: Application to Android Inter-component Communication Analysis". En: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 2015, págs. 77-88. ISBN: 978-1-4799-1934-5. URL: http://dl.acm.org/citation.cfm?id=2818754.2818767.

[33] Open Handset Alliance. *Android project*. Available at: //source.android.com/.

[34] Christine Paulin-Mohring. "Introduction to the Calculus of Inductive Constructions". En: (nov. de 2014).

[35] *Proof Irrelevance*. https://coq.inria.fr/library/Coq.Logic.ProofIrrelevance.html.

[36] Lina Qiu, Yingying Wang y Julia Rubin. "Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe". En: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: ACM, 2018, págs. 176-186. ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213873. URL: http://doi.acm.org/10.1145/3213846.3213873.

[37] Alireza Sadeghi y col. "A Temporal Permission Analysis and Enforcement Framework for Android". En: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: ACM, 2018, págs. 846-857. ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180172. URL: http://doi.acm.org/10.1145/3180155.3180172.

[38] W. Shin y col. "A Formal Model to Analyze the Permission Authorization and Enforcement in the Android Framework". En: *2010 IEEE Second International Conference on Social Computing*. Ago. de 2010, págs. 944-951. DOI: 10.1109/SocialCom.2010.140.

[39] W. Shin y col. "A Small But Non-negligible Flaw in the Android Permission Scheme". En: *2010 IEEE International Symposium on Policies for Distributed Systems and Networks*. Jul. de 2010, págs. 107-110. DOI: 10.1109/POLICY.2010.11.

[40] Android development team. *Android Runtime (ART)*. URL: https://source.android.com/devices/tech/dalvik/index.html.

[41] Android development team. *Intents*. URL: https://developer.android.com/guide/components/intents-filters.

[42] Android development team. *Permisos*. URL: https://developer.android.com/guide/topics/permissions/overview.

[43] *The Coq Proof Assistant*. https://coq.inria.fr/.

[44] Jice Wang y Hongqi Wu. "Android Inter-App Communication Threats, Solutions, and Challenges". En: *CoRR* abs/1803.05039 (2018). arXiv: 1803.05039. URL: http://arxiv.org/abs/1803.05039.

[45] Fengguo Wei y col. "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps". En: *ACM Transactions on Privacy and Security* 21 (abr. de 2018), págs. 1-32. DOI: 10.1145/3183575.

[46] S. Wu y J. Liu. "Overprivileged Permission Detection for Android Applications". En: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. Mayo de 2019, págs. 1-6. DOI: 10.1109/ICC.2019.8761572.