Scientific Annals of Computer Science vol. 26 (1), 2016, pp. 27–68
doi: 10.7561/SACS.2016.1.27

Formal Analysis of Android's Permission-Based Security Model [1]

Gustavo BETARTE[2], Juan CAMPO[2], Carlos LUNA[2],
Agustín ROMANO[3]

Abstract

In this work we present a comprehensive formal specification of an idealized formulation of Android's permission model. Permissions in Android are basically tags that developers declare in their applications, more precisely in the so-called application *manifest*, to gain access to sensitive resources. Several analyses have recently been carried out concerning the security of the Android system. Few of them, however, pay attention to the formal aspects of the permission enforcing framework. We provide a complete and uniform formulation of several security properties using the higher order logic of the Calculus of Inductive Constructions and sketch the proofs that have been developed and verified using the Coq proof assistant. We also analyze how the changes introduced in the latest version of Android, that allows to manage permissions at runtime, impact the presented model.

Keywords: Android, security properties, formal verification, Coq.

1 Introduction

Android [39] is an open platform for mobile devices developed by the Open Handset Alliance led by Google, Inc. Android smartphone OS has captured more than 80% of the total market-share, leaving its competitors

[1]Partially funded by project ANII-Clemente Estable FCE_1_2014_1_103803: Mecanismos autónomos de seguridad certificados para sistemas computacionales móviles, Uruguay.

[2]Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Julio Herrera y Reissig 565, Montevideo, Uruguay, E-mail: {gustun, jdcampo, cluna}@fing.edu.uy.

[3]Facultad de Ciencias Exactas, Ingeniería y Agrimensura, Universidad Nacional de Rosario, Avenida Pellegrini 250, Rosario, Argentina, E-mail: agustinr88@gmail.com.

iOS, Windows mobile OS and Blackberry far behind [33]. With increasing capabilities in mobile devices and posterior consumer adoption, these devices have become an integral part of how people perform tasks in their work and personal lives. Unfortunately, the benefits of using mobile devices are sometimes counteracted by security risks.

Concerning security, Android embodies mechanisms at both OS and application level. Being a Linux system, Android behaves as a multi-process system and therefore the security model resembles that of a multi-user server. Access control at the application level is implemented by an Inter-Component Communication reference monitor that enforces mandatory access control (MAC) policies regulating access among applications and components.

Application security is built primarily upon a system of permissions, which specify restrictions on the operations a particular process can perform. Permissions are basically tags that developers declare in their applications, more precisely in the so-called application *manifest*, to gain access to sensitive resources. At installation time the user of the device is requested to grant the permissions required by the application or otherwise the installation of the application is canceled. After a successful installation, an application will be able to access system and application resources depending on the permissions granted by the user.

Several analyses have recently been carried out concerning the security of the Android system. Some of them [26, 38] point out the rigidity of the permission system regarding the installation of new applications in the device. Other studies [22, 31] have shown that many aspects of Android security, like avoiding *privilege escalation*, depend on the correct construction of applications by their developers. Additionally, it has been pointed out [31, 32] that the mechanism of permission delegation offered by the system has characteristics that require further analysis in order to ensure that no new vulnerabilities are added when a permission is delegated. Few work, however, pay attention to the formal aspects of the permission enforcing framework.

Reasoning about implementations provides the ultimate guarantee that deployed mechanisms behave as expected. However, formally proving non-trivial properties of code might be an overwhelming task in terms of the effort required, especially if one is interested in proving security properties rather than functional correctness. In addition, many implementation details are orthogonal to the security properties to be established, and may complicate reasoning without improving the understanding of the essential features for guaranteeing important properties. Complementary approaches are needed where verification is performed on idealized models that abstract away the specifics of any particular implementation, and yet provide a realistic setting in which to explore the security issues that pertain to the realm of those (critical) mechanisms.

Security models play an important role in the design and evaluation of high assurance security systems. Their importance was already pointed out in the Anderson report [1]. The paradigmatic Bell-LaPadula model [18], conceived in 1973, constituted the first big effort on providing a formal setting in which to study and reason on confidentiality properties of data in time-sharing mainframe systems. *State machines*, in turn, are a powerful tool that can be used for modeling many aspects of computing systems. In particular, they can be employed as the building block of a security model. The basic features of a state machine model are the concepts of state and state change. A *state* is a representation of the system under study at a given time, which should capture those aspects of the system that are relevant to the analyzed problem. State changes are modeled by a state transition function that defines the next state based on the current state and input. If one wants to analyze a specific safety property of a system using a state machine model, one must first specify what it means for a state to satisfy the property, and then check if all state transitions *preserve* it. Thus, state machines can be used to model the enforcement of a security policy on a system.

Contribution

The main contribution of the work presented in this paper is the development of a comprehensive formal specification of the Android security model and the machine-assisted verification of several security properties. Most of those properties have been discussed in previous work where they have been presented and analyzed using a variety of formal settings and approaches. In this work we provide a complete and uniform formulation of multiple properties using the higher order logic of the Calculus of Inductive Constructions [27, 40], and the formal verification is carried out using the `Coq` proof assistant [19, 48]. Furthermore, we present and discuss proofs of properties that have not been previously given a formal treatment. The idealized security model formalizes behavior of the security mechanisms of *Kitkat* [2] (as of June 2015 the single most widely used Android version) according to the official documentation and available implementations. We claim that our results also apply to *Lollipop*, the version 5 of Android, and to

Marshallow, the latest version of Android[4]. The formal security model and the proofs of the security properties presented in this work may be obtained from [34].

The `Coq` proof assistant facilitates theoretical confirmation of security properties when the system is updated, when there are new security requirements, or when more constrains are added to the model. This may help to ascertain the security level of the Android permission scheme regarding security certification standards (e.g., Common Criteria), which require formal descriptions for higher-level assurance.

Organization of the paper

This article builds upon and extends a previously published paper [21]. The rest of the paper is organized as follows. Section 2 describes the architecture and basic components of Android, and its security model. Section 3 overviews the formalization of the Android security framework and Section 4 presents and discusses some of the verified security properties. Section 5 extends the analysis of the security model to the latest version of Android that allows to manage permissions at runtime. Section 6 considers related work and finally, Section 7 concludes with a summary of our contributions and directions for future work.

2 Background

Architecture of Android The architecture of Android takes the form of a software stack which comprises an operating system, a run-time environment, middleware, services and libraries, and applications. Figure 1 provides a visual outline of this architecture.

The Linux kernel is positioned at the bottom of the software stack, providing a level of abstraction between the hardware and the upper layers of the software stack.

The multitasking execution environment provided by Linux allows multiple processes to execute concurrently. In fact, each application running on an Android device does so within its own instance of the Dalvik virtual machine (DVM)[5]. The applications running on a DVM are sandboxed, that

[4]Section 5 makes some special considerations about managing permissions on the latest version of Android.

[5]The successor of Dalvik is Android Runtime (ART). This new runtime environment is

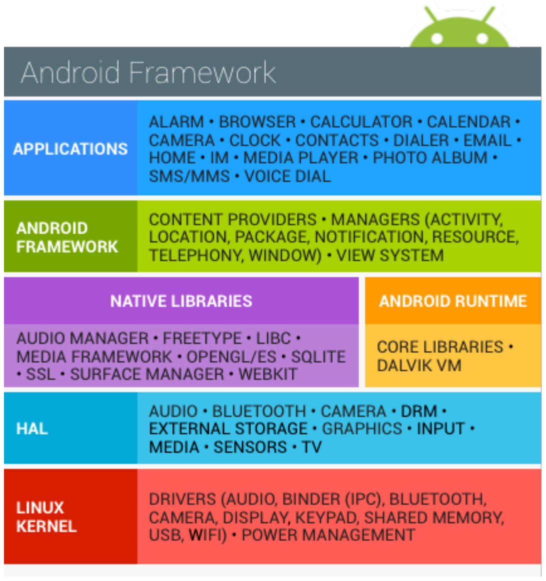
Android Framework
APPLICATIONS
ALARM • BROWSER • CALCULATOR • CALENDAR • CAMERA • CLOCK • CONTACTS • DIALER • EMAIL • HOME • IM • MEDIA PLAYER • PHOTO ALBUM • SMS/MMS • VOICE DIAL
ANDROID FRAMEWORK
CONTENT PROVIDERS • MANAGERS (ACTIVITY, LOCATION, PACKAGE, NOTIFICATION, RESOURCE, TELEPHONY, WINDOW) • VIEW SYSTEM
NATIVE LIBRARIES
AUDIO MANAGER • FREETYPE • LIBC • MEDIA FRAMEWORK • OPENGL/ES • SQLITE • SSL • SURFACE MANAGER • WEBKIT
ANDROID RUNTIME
CORE LIBRARIES • DALVIK VM
HAL
AUDIO • BLUETOOTH • CAMERA • DRM • EXTERNAL STORAGE • GRAPHICS • INPUT • MEDIA • SENSORS • TV
LINUX KERNEL
DRIVERS (AUDIO, BINDER (IPC), BLUETOOTH, CAMERA, DISPLAY, KEYPAD, SHARED MEMORY, USB, WIFI) • POWER MANAGEMENT

Figure 1: Android's architecture [12].

is, they can not interfere with the operating system or other applications, nor can they directly access the device hardware.

The Application Framework is a set of services that collectively form the environment in which Android applications run and are managed. This framework implements the concept that Android applications are constructed from reusable, interchangeable and replaceable components. This concept is taken a step further in that an application is also able to publish its capabilities along with any corresponding data so that they can be found and reused by other applications. The Android framework includes several key services, or components, like the Activity Manager which controls all aspects of the application lifecycle and activity stack; and the Content Providers which allow applications to publish and share data with other applications.

Located at the top of the Android software stack are the applications. These comprise both the native applications provided with the particular Android implementation (for example web browser and email applications)

presented as the replacement of Dalvik in the latest version of Android. The permission mechanism analyzed in this work, and the obtained results, also apply for Android platforms running ART.

and the third party applications installed by the user after purchasing the device.

Application components An Android application is built up from *components*. A component is a basic unit that provides a particular functionality and that can be run by any other application with the right permissions. There exist four types of components: Activities, Services, Content Providers and Broadcast Receivers [3]. An **activity** is essentially a user interface of the application. Typically, each application has a principal activity which is the first screen the user sees when the application is started. Even if applications usually have a principal activity, any activity can be started if the initiator has the right permissions. In a same session multiple instances of the same activity can be running concurrently. A **service** is a component that executes in background without providing an interface to the user. Any component with the right permissions can start a service or interact with it [3]. If a component starts a service that is already running no new instance is created, the component just interacts with the running instance of the service [5, 11]. A **content provider** is a component intended to share information among applications. A component of this type provides an interface through which applications can manage persisted data [45]. The information may reside in a SQLite data base, the web or in any other available persistent storage [3], and it can be presented by a content provider in the form of a file or a table. Finally, a **broadcast receiver** is a component whose objective is to receive messages, sent either by the system or an application, and trigger the corresponding actions. Those messages, called *broadcasts*, are transmitted all along the system and the broadcast receivers are the components in charge of dispatching those messages to the targeted applications.

Three out of the four preceding types of components: activities, services and broadcast receivers, are activated by a special kind of message called *intent*. An intent makes it possible for different components, belonging to the same application or not, to interact at runtime [3]. Typically, an intent is used as a broadcast or as a message to interact with activities and services.

Android's security model Android implements a least privilege model by ensuring that each application executes on a sandbox, enforcing then that each application only has unrestricted access to the resources it owns. For an application to access other components of the system it must require, and be granted, the corresponding access permission. The sandbox mechanism is implemented at kernel level and relies on the correct application of a Mandatory Access Control policy which is enforced by a reference monitor using a user identifier (UID) [30] assigned to each installed application. Interaction among applications is achieved through *Inter Process Communication* (IPC) mechanisms [14]. Even if the kernel provides traditional UNIX-like IPC (like sockets and signals), it is recommended that applications use higher level IPC mechanisms provided by Android. One such mechanism is intents, that allow to specify security policies that regulate communication between applications [10].

Every Android application must be digitally signed and be accompanied by the certificate that authenticates its origin. These certificates, however, are not required to be signed by a trusted certification authority. Indeed, current practice indicates that certificates are usually self-signed by the developers. The Android platform uses the certificates to establish that different applications have been developed by the same author. This information is relevant both to assign *signature* permissions (see below) or to authorize applications to share the same UID to allow sharing their resources or even be executed within the same process [6].

Permissions Applications usually need to use system resources to execute properly. Since applications run inside sandboxes, this entails the existence of a decision procedure (a reference monitor) that guarantees the authorized access to those resources. Decisions are made by following security policies using a simple notion of permission. The permission system of Android implements the following procedure: i) an application declares the set of permissions it needs to acquire further capacities than the default ones; ii) at installation time, the required permissions are granted or refused, either automatically by the system depending on the type of permission and the certificate attached to the application, or, as it's more frequently the case, by direct authorization of the user of the device; iii) if a requested permission is refused, the application should not be installed on the device. This is typically the case, but there are ways to install an application with non granted permissions [7].

In the general case, if an application is installed then it may exercise all the permissions it requests. Note that it is not possible to dynamically assign permissions in Android. Every permission is identified by a name/text and has a protection level. There are two principal classes of permissions:

the ones defined by the application, for the sake of self-protection, and those predefined by Android, which are intended to protect access to resources and services of the system. Depending on the protection level of the permission, the system defines the corresponding decision procedure [9]. There are four classes of permission levels: i) *Normal*, assigned to low risk permissions that grant access to isolated characteristics, ii) *Dangerous*, permissions of this level are those that provide access to private data or control over the device, iii) *Signature*, a permission of this level can be granted only if the application that requires it and the application that defined it are both signed with the same certificate, and iv) *Signature/System*, this level is assigned to permissions that regulate the access to critical system resources or services.

In addition, an application can also declare the permissions that are needed to access it. The granularity of the system makes it possible to require different permissions to access different components of the application.

It is also possible for a developer to force the system to execute a verification at runtime. For doing that, Android provides methods that can verify the permissions of an application at runtime. This mechanism might be used, for instance, to force the system to check that an application has specific privileges once a certain internal counter has reached a given value.

Since version *Honeycomb*, a component can access any other component of the same application without being required to have explicitly granted access to that component.

Permission delegation Android provides two mechanisms by which an application can delegate its own permissions to another one. These mechanisms are *pending intents* and *URI permissions*. An intent may be defined by a developer to perform a particular action, for instance to start an activity. A `PendingIntent` is an object which is associated to the action, a reference that might be used by another application to execute that action. The object might be used by authorized applications even if the application that created it, which is the only one that can cancel the reference, is no longer active. The *URI permissions* mechanism can be used by an application that has read/write access to a *content provider* to (partially) delegate those permissions to another application. An application may attach to the result returned by an activity owned by another application an intent with the URIs of resources of a content provider it owns together with an operation identifier. This grants the privileges to perform the operation on the indicated resources to the receiving application, independently of the permissions the application has. The Android specification establishes that only activities may receive an *URI permission* by means of intents. These kinds of permissions may also be explicitly granted using the `grantUriPermission()` method and revoked using the `revokeUriPermission()` method. In any case, for this delegation mechanism to work, an explicit declaration authorizing the access to the resources in question must be added in the application that owns the content provider.

The Android Manifest Every Android application must include an XML file in its root directory called `AndroidManifest`. All the components included in the application, as well as some static attributes of them are declared in that file. Additionally, both the permissions requested at installation time and the ones required to access the application resources are also defined. The authorization to use the mechanism of *URI permissions* explained above is also specified in the manifest file of an application. One of the most important elements of a manifest is <**application**>: it describes the attributes of the application and also the elements that describe the components owned by the application. Each component is declared using one of the following elements: <**activity**>, <**service**>, <**provider**>, and <**receiver**>. Additionally, the body of the manifest includes: i) <**uses-permission**>, that specifies the permissions, defined by the system or an application, which shall be required at installation time; ii) <**permission**>, that defines statically an application level permission and its protection level. There must be one declaration for each defined permission; and iii) <**permission-tree**>, which is used to reserve a name space that can be used to define application level permissions at runtime. It defines prefixes to attach to any permission defined dynamically using the method `addPermission()`. Several declarations of this kind of element are allowed to define different prefixes. Additionally, the element <**application**> has the attribute **android:permission** which is used to specify, if any, the permission required to access any component of the application [4]. As for the elements declared by the components included in the application, there are two common attributes: i) **android:permission**, similar to the one defined for the application, but this one has precedence over it, and ii) **android:exported**, if this attribute is set to `true`, the component shall be available to be accessed from an external application.

3 A Formally Verified Android Security Model

In this section we outline the formalization of the idealized Android security model. We first provide a brief description of the specification setting and the proof-assistant `Coq`, then we describe the model states and provide an axiomatic semantics of successful operations in the Android system. The operations are specified as state transition functions.

3.1 The Proof Setting

The `Coq` proof assistant is a free open source software that provides a (dependently typed) functional programming language and a reasoning framework based on higher order logic to perform proofs of programs. `Coq` allows developing mathematical facts. This includes defining objects (sets, lists, functions, programs); making statements (using basic predicates, logical connectives and quantifiers); and finally writing proofs. The `Coq` environment supports advanced notations, proof search and automation, and modular developments. It also provides program extraction towards languages like Ocaml and Haskell for execution of (certified) algorithms [36, 37]. These features are very useful to formalize and reason about complex specifications and programs.

As examples of its applicability, `Coq` has been used as a framework for formalizing programming environments and designing special platforms for software verification: Leroy and others developed in `Coq` a certified optimizing compiler for a large subset of the C programming language [35]; Barthe and others used `Coq` to develop Certicrypt, an environment of formal proofs for computational cryptography [16]. Also, the Gemalto and Trusted Logic companies obtained the level CC EAL 7 of certification for their formalization, developed in `Coq`, of the security properties of the JavaCard platform [24, 20, 13].

We developed our specification in the Calculus of Inductive Constructions (CIC) using `Coq`. The CIC is a type theory, in brief, a higher order logic in which the individuals are classified into a hierarchy of types. The types work very much as in strongly typed functional programming languages which means that there are basic elementary types, types defined by induction, like sequences and trees, and function types. An inductive type is defined by its constructors and its elements are obtained as finite combinations of these constructors. Data types are called *Sets* in the CIC (in `Coq`). On top of this, a higher-order logic is available which serves to predicate on the various data types. The interpretation of the propositions is constructive, i.e. a proposition is defined by specifying what it means for an object to be a proof of the proposition. A proposition is true if and only if a proof can be constructed.

3.2 Model States

Applications An application, as depicted in Figure 2, is defined by its identifier, the certificate of its public key, the `AndroidManifest`, and the resources that shall be used at run-time.

Application identifiers, which must be unique, correspond to names of applications in Android[6]. Although Android applications do not statically declare the resources they are going to use, we decided to include this declaration in the current version of our model for the sake of simplicity.

Manifest The type Manifest is an abstraction of the `AndroidManifest` file. Manifests are modelled as 6-tuples that respectively declare application components, the set of permissions it needs, the permissions that will be required by the application at runtime and those that are delegated.

An application component (Comp) is either an activity, a service, a broadcast receiver or a content provider. All of them are denoted by a component identifier of type CompId. A content provider (ContProv), in addition, encompasses a mapping to the managed resources (of type Res) from the URIs (of type Uri) assigned to them for external access. We omit the definition types Uri and Res, which are formally defined in the Coq specification. While the components constitute the static building blocks of an application, all runtime operations are initiated by component instances, which are represented in our model as members of the abstract type iComp.

The first element of a manifest (of type Comps) stores the set of application components included in the application. The second element (of type Perms) stores the set of permissions the application needs to be executed properly. A permission (Perm) is defined as a tuple comprised of a permission identifier (PermId) and the permission level (PermLvl) that indicates the security level, which can be either dangerous, normal, signature, or signature/system. The third and fourth elements store the set of permissions that are defined in the application and the application components that

[6]These identifiers must not be confused with the user identifiers (UIDs) mentioned in Section 2.

ContProv	::=	CompId $\times$ Uri $\rightarrow$ Res *Content provider*
Comp	::=	Activity $\mid$ Service $\mid$ BroadReceiv $\mid$ ContProv *Application component*
Comps	::=	{Comp} *Set of components*
PermLvl	::=	*dangerous* $\mid$ *normal* $\mid$ *signature* $\mid$ *signature/system* *Permission level*
Perm	::=	PermId $\times$ PermLvl *Permission*
Perms	::=	{Perm} *Set of permissions*
OptionPerm	::=	$Some(p)$ $\mid$ *none* *A possible empty permission*
CompPerms	::=	Comp $\rightarrow$ Perms *Components permissions*
CPPerms	::=	ContProv $\rightarrow$ Perms *Content providers permissions*
ExtPerms	::=	OptionPerm $\times$ CompPerms $\times$ CPPerms $\times$ CPPerms *External permissions*
UriPerms	::=	ContProv $\rightarrow$ Perms *URI permissions*
DelPerms	::=	ContProvs $\times$ UriPerms *Delegated permissions*
Manifest	::=	Comps $\times$ Perms $\times$ Perms $\times$ Comps $\times$ ExtPerms $\times$ DelPerms *Manifest*
AppRes	::=	{Res} *Application resources*
App	::=	AppId $\times$ Cert $\times$ Manifest $\times$ AppRes *Application*

Figure 2: Formal definition of applications

are exported, respectively. The fifth element (of type ExtPerms) stores the information that is required to access the application, namely the permission required (if any) to access any component of the application, the permission required to access a particular component and the permissions required for performing a read or write operation on a content provider. Finally, the sixth element (of type DelPerms) stores the information concerning the delegation of permissions for accessing content providers and their resources as the result of using the URI permissions mechanism.

States The states of the platform are modelled as 8-tuples that respectively store data about the set of installed applications and their permissions, running components, a registry of temporary and permanent delegated permissions and information about the applications installed in the system image of the platform; the formal definition appears in Figure 3.

The first and second elements of a state record the set of installed applications and the permissions granted to them by the system or the user, respectively. The third stores the permissions defined by each installed application, while the fourth stores the set of running component instances. The fifth and sixth elements keep track of the permanent and temporary permissions delegations, respectively. A permanent delegated permission (of type DelPP) represents that an application has delegated permission to perform either a read, write or read/write operation (of type OpTy) on the resource identified by an URI of the indicated content provider. A temporary delegated permission, in turn, refers to permission that has been delegated to a component instance. The seventh element stores the values of resources of applications. The final element stores the applications installed in the Android system image, information that is relevant when granting permissions of level *signature/system*.

We use some functions and predicates to manipulate and observe the components of the state. Some of these operations, used in this paper, are presented and described in Table 1.

Valid state The model formalizes a notion of valid state that captures several well-formedness conditions. It is formally defined as a predicate *validState* on the elements of type AndroidST. This predicate holds on a state s if the following conditions are met:

- the applications installed in s and their corresponding components have unique identifiers;

InstApps	::=	{App} *Installed applications*
AppPS	::=	{AppId × Perms} *Permissions granted at install time*
AppDefPS	::=	{AppId × Perms} *Permissions defined by each application*
CompInstance	::=	CompId × iComp *Component instance*
CompInsRun	::=	{CompInstance} *Running component instances*
OpTy	::=	*read* | *write* | *rw* *Access type*
DelPP	::=	AppId × ContProv × Uri × OpTy *Delegated permanent permission*
DelPPS	::=	{DelPP} *Delegated permanent URI permissions*
DelTP	::=	iComp × ContProv × Uri × OpTy *Delegated permanent permission*
DelTPS	::=	{DelTP}
ARV	::=	AppId × Res × Val *Value of application resource*
ARVS	::=	{ARV} *Values of applications resources*
ImgApps	::=	{App} *Applications in system image*
AndroidST	::=	InstApps × AppPS × AppDefPS × CompInsRun × DelPPS × DelTPS × ARVS × ImgApps *Android platform state*

Figure 3: The state

$appInstalled(ap, s)$	holds if ap is an installed application in state s.
$compInstalled(c, s)$	holds if component c belongs to an installed application in state s.
$addRes(ap, s)$	returns the applications resources values in state s adding the resources in ap, initialized to a special value.
$addDefPerms(ap, s)$	returns the permissions defined by the applications installed in s as well as the ones being introduced by ap.
$grantPerms(ap, s)$	returns the permissions granted to each application in s, assigning to ap all the permissions it requested in its manifest file.
$isCProvider(c)$	holds if component c is a content provider.
$running(ic, c, s)$	is satisfied if ic is an instance of component c running in state s.
$canStart(c', c, s)$	holds if the application containing component c' (installed in state s) has the required permissions to create a new running instance of component c.
$insNotInState(ic, s)$	requires ic to be a new instance in the state s.
$runComp(ic, c, s)$	returns the running component instances of state s with the addition of the new instance ic of the component c.
$inApp(c, ap)$	holds if component c belongs to application ap.
$inManifest(c, ap)$	holds if component c belongs to the application components that are exported by the application ap in its manifest file.
$existsRes(u, cp, s)$	holds if there exists a resource, pointed to by the URI u, in the content provider cp.
$canOp(c, cp, pt, s)$	is satisfied if the application containing component c has the appropriate permissions to perform the operation pt (of type OpTy) on the content provider cp in the state s.
$delPerms(c, cp, u, pt, s)$	establishes that the component c has been delegated permissions to perform the operation pt on the resource identified by u of content provider cp in the state s.
$canGrant(u, cp, s)$	is satisfied if possible to delegate permissions on the content provider cp for resource identified by u in the state s.
$delPPerms(ap, cp, u, pt, s)$	holds if application ap has permanent delegated permissions to perform the operation pt on the resource identified by u of the content provider cp in the state s.
$delTPerms(ic, cp, u, pt, s)$	is satisfied if the running instance ic has temporary delegated permissions to perform the operation pt on the resource identified by u of the content provider cp in the state s.
$compCanCall(c, sac, s)$	is satisfied if component c can perform the system call sac in the state s.
$grantTPerm(ic, cp, u, pt, s)$	returns the temporary permission delegations of state s, in addition to the new temporary delegated permission corresponding to the running instance ic.

Table 1: Helper functions and predicates

- every component belongs to only one application;
- every user-defined permission is declared in an installed application;
- all the parts involved in active permission delegations are installed in the system;
- if there is a temporary permission delegation taking place, the recipient is running;
- If a component is running, it can not be a content provider;
- all the running instances belong to a unique component, which is part of an installed application; and
- all the resources in the system have a unique value and are owned by an installed application.

All these safety properties have a straightforward interpretation in our model[7] [34]. Valid states are invariant under execution, as will be shown later.

3.3 Platform Semantics

Our formalization considers a representative set of actions to install and uninstall applications, start and stop the execution of component instances, to read and write resources from content providers, to delegate temporary/permanent permissions and revoke them and to perform system application calls; see Table 2. The behavior of an action a (of type Action) is formally described by giving a precondition and a postcondition, which represent the requirements enforced on a system state to enable the execution of a and the effect produced after this execution takes place. We represent the execution of an action with the relation $\hookrightarrow$ (one-step execution):

$$\frac{\begin{array}{c} Pre(s,a) \\ Post(s,a,s') \end{array}}{s \overset{a}{\hookrightarrow} s'}$$

Intuitively, this relation models a system state transition fired by a particular action a. This transition takes place between a state s which fulfills the precondition of the action, and a state s' in which the postcondition holds.

[7]We omit the formal definition of *validState* due to space constraints.

`install` *ap*	Installs application *ap* in the system.
`uninstall` *ap*	Uninstalls application *ap* from the system.
`start` *ic c*	The running component *ic* starts the execution of component *c*.
`stop` *ic*	The running component *ic* finishes its execution.
`read` *ic cp u*	The running component *ic* reads the resource corresponding to URI *u* from content provider *cp*.
`write` *ic cp u val*	The running component *ic* writes value *val* on the resource corresponding to URI *u* from content provider *cp*.
`grantT` *ic cp act u pt*	The running component *ic* delegates temporary permissions to activity *act*. This delegation enables *act* to perform operation *pt* (of type `OpTy`) on the resource assigned to URI *u* from content provider *cp*.
`grantP` *ic cp ap u pt*	The running component *ic* delegates permanent permissions to application *ap*. This delegation enables *ap* to perform operation *pt* on the resource assigned to URI *u* from content provider *cp*.
`revoke` *ic cp u pt*	The running component *ic* revokes delegated permissions on URI *u* from content provider *cp* to perform operation *pt*.
`call` *ic sac*	The running component *ic* makes the API call *sac* (of type $SACall$).

Table 2: Actions

We present the semantics of the following actions: `install` (install an application in the system), `start` (start the execution of a component instance), `read` (a running component reads resources of a content provider), and `grantT` (a running component delegates temporary permissions to an activity). Notice that what is specified is the effect the execution of an action has on the state of the system.

Action `install` *ap*

The application *ap* is installed in the system.

Rule

$$\frac{\begin{array}{c} s = (aps, ps, psD, iCs, delPP, delTP, v, img) \land ap = (id, cert, m, res) \land \\ \forall (ap' : App), ap' \in aps \land ap' = (id', cert', m', res') \rightarrow id \neq id' \land \\ authPerms(ap, s) \land m = (cmps, use, perm, exp, ext, del) \land \\ \forall (c : Comp), c \in cmps \rightarrow \neg compInstalled(c, s) \\ \{ap\} \cup aps = aps' \land addRes(ap, s) = v' \land \\ addDefPerms(ap, s) = psD' \land grantPerms(ap, s) = ps' \land \\ s' = (aps', ps', psD', iCs, delPP, delTP, v', img) \end{array}}{s \xhookrightarrow{\texttt{install}\ ap} s'}$$

Precondition All the permissions requested by application *ap* are granted, either by the system or the user. In addition, *ap*'s id is not assigned to any other application installed in s and none of its components is already present in the current state of the system.

Postcondition Application *ap* is installed in the resulting state, as well as its resources and the permissions defined by it. Moreover, all the granted permissions, which allowed the installation, are registered for future use. Apart from that, both states are equal.

Action `start` *ic c*

The running component *ic* starts the execution of component c.

Rule

$$\frac{\begin{array}{c} compInstalled(c, s) \land \neg isCProvider(c) \land \\ s = (aps, ps, psD, iCs, delPP, delTP, v, img) \land \\ \exists (c' : Comp), running(ic, c', s) \land compInstalled(c', s) \land \\ \neg isCProvider(c') \land canStart(c', c, s) \\ \exists (ic' : iComp), insNotInState(ic', s) \land runComp(ic', c, s) = iCs' \land \\ s' = (aps, ps, psD, iCs', delPP, delTP, v, img) \end{array}}{s \xhookrightarrow{\texttt{start}\ ic\ c} s'}$$

Precondition The component c belongs to an installed application in state s and is not a content provider. Additionally, *ic* is a running instance of a component c' and the application containing this latter component (installed in s) has the required permissions to create a new running instance of component c.

Postcondition The instance ic', which was not running in state s, is a new running instance of the component c in the resulting state and that is the only difference between both states.

Action read *ic cp u*

The running instance *ic* reads resource *u* from content provider *cp*.

Rule

$$\boxed{\dfrac{\begin{array}{c} compInstalled(cp, s) \wedge existsRes(u, cp, s) \wedge \\ \exists(c : Comp), compInstalled(c, s) \wedge running(ic, c, s) \wedge \neg isCProvider(c) \wedge \\ (canOp(c, cp, read, s) \vee delPerms(c, cp, u, read, s)) \end{array}}{s \xhookrightarrow{\mathtt{read}\ ic\ cp\ u} s}}$$

Precondition The content provider *cp* is installed in state *s* and it contains a resource that is pointed to by the *URI u*. The component *ic* is a running instance of the installed component *c*, which is not a content provider. Additionally, the application containing component *c* either has the appropriate permissions to read *cp* in the state *s* or it has been delegated the permissions to perform the operation *read* on the resource identified by *u*. Notice that any component of an application is implicitly granted the permissions that were delegated to a running instance of that application.

Postcondition After the execution of this action, the system state remains unchanged.

Action grantT *ic cp act u pt*

The running component *ic* delegates temporary permissions to activity *act*. This delegation enables *act* to perform operation *pt* on the resource assigned to URI *u* from content provider *cp*.

Rule

$$\frac{\begin{array}{c} \mathit{compInstalled}(cp,s) \wedge \mathit{canGrant}(u,cp,s) \wedge \mathit{existsRes}(u,cp,s) \wedge \\ \mathit{compInstalled}(act,s) \wedge \exists (c:Comp), \mathit{compInstalled}(c,s) \wedge \\ \mathit{running}(ic,c,s) \wedge \mathit{canStart}(act,c,s) \wedge \\ (\mathit{canOp}(c,cp,pt,s) \vee \mathit{delPerms}(c,cp,u,pt,s)) \wedge \\ s=(aps,ps,psD,iCs,delPP,delTP,v,img) \\ \exists\ (ic':iComp), \mathit{insNotInState}(ic',s) \wedge \\ \mathit{runComp}(ic',act,s)=iCs' \wedge \mathit{grantTPerm}(ic',cp,u,pt,s)=delTP' \wedge \\ s'=(aps,ps,psD,iCs',delPP,delTP',v,img) \end{array}}{s \xhookrightarrow{\texttt{grantT}\ ic\ cp\ act\ u\ pt} s'}$$

Precondition The content provider *cp* is installed in state s. Permissions can be delegated on *cp* for resource identified by the URI u. The component *act* is an activity installed in s. The component *ic* is a running instance of a component (c) that belongs to an installed application in s. The application containing component *act* has the required permissions to create a new running instance of component c. Additionally, the application containing component c either has the appropriate permissions to perform the operation *pt* on *cp* or it has has been delegated permissions to perform *pt* on the resource identified by u.

Postcondition A new running component instance of activity *act* is incorporated into the system state. Moreover, a new temporary delegated permission corresponding to the running instance generated is added into the temporary permissions delegations of state s. This delegation enables *act* to perform *pt* on the resource assigned to u from *cp*. Apart from that, both states are equal.

In Appendix A we present the rules of other actions that are relevant to the formulation of the security properties discussed in Section 4.

One-step execution One-step execution preserves valid states, i.e. the state resulting from the execution of an action on a valid state is also valid.

Lemma 1 $\forall (a:\mathsf{Action})(s\ s':\mathsf{AndroidST})$,
$\mathit{validState}(s) \rightarrow s \overset{a}{\hookrightarrow} s' \rightarrow \mathit{validState}(s')$

The property is proved by case analysis on a, for each condition in *validState*, using several auxiliary lemmas.

As an illustrative example, we will depict the proof of one of these lemmas, which claims that the state resulting from the installation of a new application in a valid state partially meets the first condition imposed by *validState*, i.e. the uniqueness of applications ids. For the proof, we will first define predicate *uniqueAppIds*, which is formally defined as the following proposition:

$$\begin{aligned}
&uniqueAppIds(s) = \\
&\quad \forall (ap_1\ ap_2 : \mathsf{App}), appInstalled(ap_1, s) \wedge appInstalled(ap_2, s)\ \wedge \\
&\quad ap_1 = (id, cert_1, m_1, res_1) \wedge ap_2 = (id, cert_2, m_2, res_2) \rightarrow \\
&\quad ap_1 = ap_2
\end{aligned}$$

With this definition, the auxiliary lemma to be proved reads:

Lemma 2 $\forall (ap : \mathsf{App})(s\ s' : \mathsf{AndroidST}), validState\ s \rightarrow s \xhookrightarrow{\texttt{install}\ ap} s' \rightarrow uniqueAppIds(s')$

Proof: Let $ap : \mathsf{App}$, $s = (aps, ps, psD, iCs, delPP, delTP, v, img)$, and $s' = (aps', ps', psD', iCs', delPP', delTP', v', img')$, such that $validState(s)$, and $s \xhookrightarrow{\texttt{install}\ ap} s'$ hold. Our aim is to prove $uniqueAppIds(s')$, i.e.:

$$\begin{gathered}
\forall (ap_1\ ap_2 : \mathsf{App}), \{ap_1, ap_2\} \subseteq aps'\ \wedge \\
ap_1 = (id, cert_1, m_1, res_1) \wedge ap_2 = (id, cert_2, m_2, res_2) \rightarrow \\
ap_1 = ap_2
\end{gathered}$$

Let ap_1 and ap_2 be two applications installed in s' with the same id. We have to prove that ap_1 and ap_2 are in fact the same application.

First, given that $s \xhookrightarrow{\texttt{install}\ ap} s'$ is one of our hypothesis, we can use its definition to claim that $aps' = \{ap\} \cup aps$. Therefore, every application in state s' is either ap or an application already installed in s. In particular, applications ap_1 and ap_2 must fall into these two categories as well, giving us four possible scenarios:

- **Case 1:** ap_1 and ap_2 are already installed in s.
  Since s is a valid state, all the applications installed in it have unique identifiers. Therefore, since ap_1 and ap_2 have the same id, they must be the same application.

- **Case 2:** $ap_1 = ap$ and ap_2 is installed in s.
  Using hypothesis $s \xhookrightarrow{\texttt{install}\ ap_1} s'$, we know by one of its preconditions that, in order to be installed, application ap_1's id cannot be assigned

to any other application in s. Since ap_2 is installed in s, the ids for ap_1 and ap_2 must be different, contradicting the original assumption. Therefore, this case can never happen.

- **Case 3:** ap_1 is installed in s and $ap_2 = ap$.
  Since this case is analogous to the previous one, we can easily use similar arguments to conclude that this scenario is also impossible.

- **Case 4:** $ap_1 = ap$ and $ap_2 = ap$.
  This trivially implies that $ap_1 = ap_2$.

Hence, $ap_1 = ap_2$ for all possible cases. This concludes our proof. □

System state invariants, such as state validity, are useful to analyze other relevant properties of the model. In particular, the results presented in the following section are obtained from valid states of the system.

4 Security Properties

In this section we present and discuss some relevant properties that can be established concerning the Android security framework. Many of these properties have already been analyzed elsewhere. Some of them, however, have not been studied in previous work. All of the properties were successfully stated and proved using our specification, which represents, up to our knowledge, the first comprehensive analysis under the same formal model of multiple safety and security properties of the Android system. We also formally analyze properties (and their consequences) which show weaknesses in the Android security system that could be exploited by attacks.

The corresponding Coq development can be found at [34]. To simplify the presentation that follows we will assume all variables of type AndroidST to be valid states, and variables of type App to be installed applications in a given state, when there is no possibility of confusion. Components will also be assumed to be installed.

4.1 Privileges

One of the most important properties claimed about the Android security model is that it meets the so-called *principle of least privilege*, i.e. that "each application, by default, has access only to the components that it requires to do its work and no more" [3]. Using our specification we have proved several lemmas which were aimed at showing the compliance with this principle when a running instance creates another component instance, reads/writes a content provider or delegates/revokes a permission. In this setting, least privilege means that a running instance will need to have the appropriate permissions to execute the desired action in each of these scenarios. In particular, the following specific properties were proved:

- if components c and c' belong to the same application, then c can start c';
- if a component c' is not exported and the component c belongs to another application, then c cannot start c';
- if components c and c' belong to two different applications ap and ap', and c' requires a permission that ap does not have, then c cannot start c';
- if components c and c' belong to two different applications ap and ap', c' requires no permission, but ap' requires a permission that ap does not have, then c cannot start c';
- if ic can read/write the resource pointed by the URI u in cp, then its associated component belongs to an application that has permission to do so, either from its installation or through a delegation of permissions[8];
- if a content provider cp and a component c belong to the same application, then all running instances of c can read or write cp;
- if ic delegated permissions, temporary or permanent, to read or write a resource pointed by the URI u in cp, then ic can perform this operation;
- if ic revoked permissions to read or write the resource pointed by the URI u in cp, then ic can perform this operation.

All properties have a straightforward representation in our model. For example, the first property above is captured by the following proposition:

$$\begin{aligned}&\forall (s : \mathsf{AndroidST})(ap : \mathsf{App})(c\ c' : \mathsf{Comp}),\\ &inApp(c, ap) \wedge running(ic, c, s) \wedge inApp(c', ap) \wedge \neg isCProvider(c') \rightarrow\\ &Pre(s, \texttt{start}\ ic\ c')\end{aligned}$$

[8]In particular, ic can read/write the resource pointed by u in cp if ic has permission due to a delegation via intents.

where $Pre(s, \texttt{start}\ ic\ c')$ denotes the precondition of the action $\texttt{start}\ ic\ c'$ in state s.

While the fulfillment of the *principle of least privilege* when creating a new instance is widely studied in the literature [32, 44], the analysis of this principle when accessing a content provider or delegating/revoking a permission has not been covered in other publications. Since our model includes these two scenarios, we are able to formally state and prove lemmas like the following:

Lemma 3 $\forall (s : \mathsf{AndroidST})(ap\ ap' : \mathsf{App})(c : \mathsf{Comp})(ic : \mathsf{iComp})$ $(cp : \mathsf{ContProv})(u : \mathsf{Uri}), ap \neq ap' \wedge inApp(c, ap) \wedge running(ic, c, s) \wedge$ $inApp(cp, ap') \wedge \neg inManifest(cp, ap') \wedge existsRes(u, cp, s) \rightarrow$ $Pre(s, \texttt{read}\ ic\ cp\ u) \leftrightarrow delPerms(c, cp, u, read, s))$

Lemma 4 $\forall (s : \mathsf{AndroidST})(ap\ ap' : \mathsf{App})(c : \mathsf{Comp})(ic : \mathsf{iComp})$ $(cp : \mathsf{ContProv})(u : \mathsf{Uri})(v : \mathsf{Val}), ap \neq ap' \wedge inApp(c, ap) \wedge running(ic, c, s) \wedge$ $inApp(cp, ap') \wedge \neg inManifest(cp, ap') \wedge existsRes(u, cp, s) \rightarrow$ $Pre(s, \texttt{write}\ ic\ cp\ u\ val) \leftrightarrow delPerms(c, cp, u, write, s))$

If cp is not exported and c belongs to a different application than a, then cp can be read/written by c if and only if the application corresponding to the latter has delegated permissions to do so.

Lemma 5 $\forall (s : \mathsf{AndroidST})(ap\ ap' : \mathsf{App})(c : \mathsf{Comp})(ic : \mathsf{iComp})$ $(cp : \mathsf{ContProv})(act : \mathsf{Activity})(pt : \mathsf{OpTy})(u : \mathsf{Uri}),$ $inApp(c, ap) \wedge running(ic, c, s) \wedge inApp(cp, ap) \wedge$ $existsRes(u, cp, s) \wedge canGrant(u, cp, s) \rightarrow$ $Pre(s, \texttt{grantT}\ ic\ cp\ act\ u\ pt) \wedge Pre(s, \texttt{grantP}\ ic\ cp\ ap'\ u\ pt)$

If c and cp belong to the same application and cp authorizes the delegation on u, then ic can delegate both temporary and permanent permissions on u.

The above lemmas establish that even if a component of an application is not exported, it can still be accessed from a different application. In particular, Lemmas 3 and 5 show that it is possible for an external application to obtain delegated permissions to access a non-exported content provider. This contradicts the description of exported components given in the official developer's guide [4].

The interested reader is referred to [34] where he can find the `Coq` files with the complete proofs of the lemmas we have just discussed.

4.1.1 Revocation

One of the peculiarities of the Android security model is that the explicit revocation of delegated permissions is relatively coarse-grained, in the sense that it is impossible to only revoke permissions to a particular application.

Although this property was studied in [32], no formal statement or proof is provided in that work. In our formal setting we are able to state and prove the following lemma:

Lemma 6 $\forall (s\ s' : \mathsf{AndroidST})(ic : \mathsf{iComp})(cp : \mathsf{ContProv})(u : \mathsf{Uri})(pt : \mathsf{OpTy}),\ s \xhookrightarrow{\texttt{revoke}\ ic\ cp\ u\ pt} s' \to (\forall (ap : \mathsf{App}), \neg delPPerms(ap, cp, u, pt, s'))$
$\wedge$
$(\forall (ic' : \mathsf{iComp})(c : \mathsf{Comp}), running(ic', c, s') \to \neg delTPerms(ic', cp, u, pt, s')$

If ic revokes the permission to perform operation pt over the resource pointed by u in cp, this revocation will be applied to all the applications in the system.

A direct consequence of this property is that a running component can revoke permissions that were not delegated by itself, which may result in confusing and problematic scenarios [32]. For instance, suppose applications A and B both have the same delegated permission p. In the case that an application C revokes p with the intention that B does not longer use it, A shall also lose that permission without further notice. Application A will just find out when attempting a task that requires p, provoking then a runtime exception.

4.1.2 Privilege Escalation

According to [31], in the Android system a *privilege escalation* problem occurs when "an application with a permission performs a privileged task on behalf of an application without that permission". This privileged task can be, for instance, invoking a system service, or accessing an application. We have proved that a *privilege escalation* scenario involving either task is possible in our model. The proof was divided in two separate lemmas, one for each kind of privileged operation.

Lemma 7 $\forall (s : \mathsf{AndroidST})(ic : \mathsf{iComp})(c : \mathsf{Comp})(sac : \mathsf{SACall}),$
$\neg Pre(s, \texttt{call}\ ic\ sac) \wedge \neg Pre(s', \texttt{call}\ ic\ sac) \wedge compCanCall(c, sac, s') \wedge$
$s \xhookrightarrow{\texttt{start}\ ic\ c} s' \to \exists (ic' : \mathsf{iComp}), running(ic', c, s') \wedge Pre(s', \texttt{call}\ ic'\ sac)$

If ic cannot perform the API call sac but it starts the execution of a component c which is able to do it, then it will be possible to invoke sac through an instance of c.

Lemma 8 $\forall (s\ s' : \mathsf{AndroidST})(c\ c'\ c'' : \mathsf{Comp})(ic : \mathsf{iComp})$,
$\neg isCProvider(c'') \wedge \neg canStart(c, c'', s) \wedge \neg canStart(c, c'', s') \wedge$
$canStart(c', c'', s') \wedge s \xhookrightarrow{\texttt{start}\ ic\ c'} s' \rightarrow$
$\exists (ic' : \mathsf{iComp}), running(ic', c', s') \wedge Pre(s', \texttt{start}\ ic'\ c'')$

If ic *cannot access a component* c'' *but it starts the execution of a component* c' *which is able to do it, it will be possible to start* c'' *through an instance of* c'.

Sketch of the proof: The proof of Lemma 8 is fairly straightforward: we need to give a running instance of a component which is always able to access component c'' in state s'. We claim that this witness is the resulting instance of executing the `start` operation in state s (hypothesis $s \xhookrightarrow{\texttt{start}\ ic\ c'} s'$). Calling this instance ic', we have to prove that both $running(ic', c', s')$ and $Pre(s', \texttt{start}\ ic'\ c'')$ are verified. The first predicate is trivially satisfied by the definition of ic'. Next, the precondition of operation `start`, as described in Section 3, requires ic' to be able to start component c'', which must be installed in state s' and be different from a content provider. While the first and third requests are explicitly assumed in the hypotheses of the lemma, we prove that component c'' is installed in state s' beginning by the fact that, by hypothesis, c'' is installed in state s and, since the `start` operation does not change the installed applications, c'' must be in s' as well. The proof of Lemma 7 is analogous to the one just described.

Intuitively, in the above lemmas ic represents the unprivileged running component and c' the component that has the permissions to make a API call (predicate *compCanCall* in Lemma 7) or access another component, respectively. If ic access c' (creating a running instance of c'), then the privileged operation will be available to be executed (by, at least, the running instance just created).

In models that avoid privilege escalation it is not enough to call an instance with the required permissions to perform a privileged operation. In such models, extra controls are implemented in order to prevent the called instance from being used as a deputy of an unprivileged component [22]. The issues just discussed were originally presented in [22, 28, 31] but referred to earlier versions of the Android platform and used different approaches to perform their analysis. Since our formalism fully captures both the interaction between components and the execution of API calls in the Android system we are convinced that the latest versions of the platform are still vulnerable to this kind of privilege escalation problems.

4.2 Permission Redelegation

The last property we want to discuss makes explicit that it is possible to redelegate a permission an unlimited number of times. This particular aspect of the Android security model was also studied by Fragkaki *et al.* [32] and has been successfully represented in our formalism.

Lemma 9 $\forall (s : \mathsf{AndroidST})(ap\ ap' : \mathsf{App})(c : \mathsf{Comp})(ic : \mathsf{iComp})$
$(act : \mathsf{Activity})(cp : \mathsf{ContProv})(u : \mathsf{Uri})(pt : \mathsf{OpTy}),$
$inApp(c, ap) \wedge\ running(ic, c, s) \wedge\ existsRes(u, cp, s)\ \wedge$
$(delTPerms(ic, cp, u, pt, s) \vee delPPerms(ap, cp, u, pt, s)) \rightarrow$
$Pre(s, \texttt{grantT}\ ic\ cp\ act\ u\ pt) \wedge Pre(s, \texttt{grantP}\ ic\ cp\ ap'\ u\ pt)$

If ic or an application ap have a delegated permission, they can redelegate it in a temporary or permanent way.

As a corollary of this property, if a running component receives a temporal permission delegation, then any instance of that component can redelegate the given permission to the application itself in a permanent way. Consequently, a permission that was originally temporarily delegated, ends up being permanently delegated [32]. This behavior means that in practice, the two delegation mechanisms are not substantially different. For example, a running component can receive a permission delegation through an *intent* because the sender wants this permission to get revoked when the recipient finishes execution. However, the receiver could redelegate the given permission to its own application in a permanent way so that it can only be revoked via the method: `revokeUriPermission()`; which would contradict the original purpose of the sender.

5 Requesting (and Granting) Permissions at Run Time

On all versions of Android an application must declare both the normal and the dangerous permissions it needs in its application manifest. However, the effect of that declaration is different depending on the system version and the application's target SDK level [8]. In particular, if a device is running Android 6.0 (Marshmallow) and the application's target SDK is 23 or higher the application has to list the permissions in the manifest, and it must request each dangerous permission it needs while the application is running. The user can grant or deny each permission, and the application can continue to run with limited capabilities even if the user denies a permission request. This modification of the access control and decision process, on the one side, streamlines the application install process, since the user does not need to grant permissions when he/she installs or updates an application. On the other hand, as users can revoke the (previously granted) permissions at any time, the application needs to check whether it has the corresponding privileges every time it attempts to access a resource on the device.

5.1 Impact on the Formal Model

We have already modified the (formal) model presented and discussed in Section 3 so as to consider the run time requesting/granting of permissions behavior introduced in Android Marshmallow. We shall not describe in detail the changes that were carried out and shall limit ourselves to provide some hints concerning the impact the changes had on the essential components of the model:

- The application definition, as depicted in Figure 2, has not changed. In particular, the `AndroidManifest` remains the same
- The state (of type AndroidST) remains unchanged. Also, the validity conditions of state (*validState*) remain invariant.
- Action `install` is simplified; the controls associated with dangerous permissions are removed.
- We have already incorporated actions to model the granting and revocation of a permission at run time.
- The properties formalized in Section 4 remain valid.

5.2 On the Security Policies

In this section we briefly discuss how the management of permissions has been affected by the design changes introduced in Android Marshmallow.

Delegated permissions The official Android documentation does not specify what should happen when permissions acquired and delegated at run time by an application are revoked by the user. The delegation of a permission on a given resource might grant an application access to sensitive information that otherwise it would not have had. A (malicious) application could gain a permission for access sensitive data indefinitely in the case an user revokes a permission and not all delegations and re-delegations of this permission are also revoked.

Permission groups A significant change in the new version of Android is that all dangerous system permissions belong to permission groups. For example, if an application requires permission to read contacts, it must request access to the group *android.permission-group.CONTACTS* that allows the application to read contacts, but also write and access the user's profile. The biggest drawback with this change is that individual permissions remain. Applications developed in the latest version of Android could start applications developed in previous versions, using the individual permissions that, in principle, should not have[9]. This behavior can induce *privilege escalation* and also shows a lack of transparency with the user, who should accept that their applications have permissions to perform operations available on your sensitive data.

Automatic Internet access Another significant change is that permission to access the Internet becomes normal type. This means that all applications can potentially access the Internet without that permission being granted by the user of the device. For example, a flashlight application can access the Internet and the user can not do anything about it. Android developers justify this change by saying that users are still the ones who grant permissions to access sensitive data. However, applications which by their nature can manipulate sensitive data have now also automatically access to the Internet. For example until Android Lollipop, a camera application which had granted access to images but was denied access to the web could be considered a safe application. Users whose devices run Android Marshmallow have no means to know, for instance, if the camera has Internet access, so these applications would potentially no longer be safe.

Missing permissions Android Marshmallow allows users to revoke permissions from any application at any time. A developer should test his application to verify that it behaves properly when it is missing a needed permission. In particular, the user can grant or deny each dangerous permission, and the application can continue to run with limited capabilities even after a user has denied a permission request.

[9]Individual permissions obtained (indirectly) from a permission group.

Social engineering attacks The request of dangerous run time permissions might give rise to the generation of new patterns of attacks based on social engineering. A detailed analysis in this direction has already been identified as future work.

6 Related Work

Existing smartphone security surveys review the state of the art regarding popular mobile OS platforms [46, 41]. In particular, La Polla et al. [41] surveyed smartphone security threats and their solutions for the period 2004–2011, but has very limited coverage of Android. In the survey papers [29, 42, 47] the authors focus on the Android platform, but they only perform an informal analysis of the security model.

Some work have analyzed the limitations and weaknesses of the Android security model. The study of most of the properties that we have formally verified and presented in this paper is scattered in several publications from the literature. The results presented in those publications are formulated using different formal settings in accordance to the type of study and properties in which they are interested.

Felt *et al.* [31] study, although not formally, Android applications to determine whether Android developers follow least privilege policies when declaring applications permission requests. The authors develop in particular an OS mechanism for defending against permission re-delegation (when an application with permissions performs a privileged task for an application without permissions). Our work initiates the development of an exhaustive formal specification of the Android security model from which it is possible to formally reason, for instance, about the property of least privilege for any application.

In Chaudhuri's work [23] a typed language to model the communication between different components of the Android platform is defined. Given an expression defined in this language, if a type can be inferred for it, the operation being modelled is in compliance with some desirable security properties concerning the integrity and confidentiality of the information being exchanged. Analogously, Armando *et al.* [14] and Bugliesi *et al.* [22] present a type and effect system in which they model basic Android components and the semantics of some operations. Although these three publications follow similar approaches, they all define different new languages, which are focused on the features being analyzed in each work. Additionally, no formal guarantee is provided of the correctness of the results obtained.

In the case of the work by Fragkaki *et al.* [32], instead of defining a typed language, the authors generalize the Android permission scheme and propose an abstract formal framework to represent the systems that meet these general characteristics. This model is used to enunciate security properties that, according to the authors, any instantiation should obey. In this way, the Android platform is represented as a particular instance of the proposed abstract model and its formal analysis consists of checking if the enunciated security properties actually hold on it. As pointed out in Section 4, many of the properties studied in [32] were selected to be proved in our own model. The success in doing so shows that our formal framework is expressive enough to state and prove the properties in question, offering the support of a widely used tool, such as `Coq`, in all the stages of the proof development process.

Finally, Shin *et al.* [44] adapt the approach followed by Zanella *et al.* [17] to build a formal framework that represents the Android permission system, which is based on the Calculus of Inductive Constructions and it is developed in `Coq`, as we do. However, that formalization does not consider several aspects of the platform covered in our model, namely, the different types of components, the interaction between a running instance and the system, the reading/writing operation on a content provider and the semantics of the permission delegation mechanism. These last two aspects of our model allow us to formulate and prove security properties which cannot be formally studied in Shin's model, such as Lemmas 6, 8, and 9, addressed in Section 4. Furthermore, there are important differences between the two models regarding the way of representing, for example, the applications and its components, the state of the platform, the `AndroidManifest` file and the operations execution. We claim that the results we have obtained constitute a quite complete, expressive and extensible model of the Android permission system. Moreover, we understand our contribution as an alternative adaptation of the work presented in [17] rather than an extension of the one proposed by Shin.

7 Conclusion and Future Work

This work constitutes an essential step towards the development of an exhaustive formal specification of the Android security model that includes elements and properties that have been partially analyzed in previous work.

Formal model	1k
Valid state invariance	3k
Security properties	1.5k
Total	5.5k

Figure 4: LOC of `Coq` development

The formal model considers the latest version of the security mechanisms of the platform. Furthermore, we present the proof of security properties concerning the Android permission mechanisms that have not previously been formally verified. Thus, this specification represents, up to our knowledge, the first comprehensive analysis under the same formal model of several security properties of Android. The formal development is about 5.5kLOC of `Coq` (see Figure 4), including proofs, and constitutes a suitable basis for reasoning about Android's security model.

There are several directions for future work. We are already working in enriching the model so as to include the actions of sending and receiving broadcasts and implicit intents, application update, and management of signatures and certificates. We are also interested in exploring the formal analysis of the novel mechanisms of Android that make it possible to grant permissions at run time. In particular, we would like to explore properties of permission delegation in this context.

We are also elaborating a formal analysis of a variety of attacks that target specifically the `Intents` for malicious purposes, such as indicated in [43, 25]: i) *Intent spoofing*: Applications communicate to other applications even thought they are not meant to. This attack exploits the lack of necessary security configurations, allowing the invocation of internal Activities by external applications. ii) *Permission Collusion*: An application that only has access to a restricted set of permissions executes a permission escalation, or augmentation attack by invoking another collaborating application through Intents. To perform an attack of this type, malevolent developers could deceive users to install another collaborating malware application that is used to compromise the privacy of the data. The common idea behind these attacks is the abuse of the Intent mechanism to obtain unauthorised access to private information. We are working on proving lemmas to ensure the absence of such attacks under certain conditions expressible on the state of the system.

We are also interested in producing an alternative definition of the model which would be better suited for the verification of a toy implementation of the control access decision procedure. Using the program extraction mechanisms provided by `Coq`, we expect to derive a certified `Haskell` prototype of the reference monitor following the approach used in [15]. This work would proceed as follows: i) An executable specification of the control access decision procedure is written using the `Coq` proof assistant. This ultimately amounts to the definition of the functions that implement the execution of the specified actions. Those functions must be defined so as to conform to the axiomatic specification of action execution as specified in the model. ii) The next step is the construction of the proof that the executable specification of the control access decision procedure correctly implements the axiomatic model. It shall be formally stated as a soundness theorem and verified using the `Coq` proof assistant. iii) Finally, using the extraction mechanism of `Coq` we shall be able to derive the corresponding `Haskell` code from the verified Coq code of the decision procedure.

References

[1] James P. Anderson. Computer Security technology planning study. Technical report, Deputy for Command and Management System, USA, 1972. URL: http://csrc.nist.gov/publications/history/ande72.pdf.

[2] Android Developers. *Android KitKat*. Available at: https://developer.android.com/about/versions/kitkat.html. Last access: July 2016.

[3] Android Developers. *Application Fundamentals*. Available at: http://developer.android.com/guide/components/fundamentals.html. Last access: July 2016.

[4] Android Developers. *Application Manifest*. Available at: //developer.android.com/guide/topics/manifest/manifest-intro.html. Last access: July 2016.

[5] Android Developers. *Context*. Available at: http://developer.android.com/reference/android/content/Context.html. Last access: July 2016.

[6] Android Developers. *<manifest>*. Available at: http://developer.android.com/guide/topics/manifest/manifest-element.html#uid. Last access: July 2016.

[7] Android Developers. *Permissions*. Available at: http://developer.android.com/guide/topics/security/permissions.html. Last access: July 2016.

[8] Android Developers. *Requesting Permissions at Run Time*. Available at: https://developer.android.com/intl/es/training/permissions/requesting.html. Last access: July 2016.

[9] Android Developers. *R.styleable*. Available at: http://developer.android.com/reference/android/R.styleable.html. Last access: July 2016.

[10] Android Developers. *Security Tips*. Available at: http://developer.android.com/training/articles/security-tips.html. Last access: July 2016.

[11] Android Developers. *Services*. Available at: http://developer.android.com/guide/components/services.html. Last access: July 2016.

[12] Android Open Source Project. *Android Security Overview*. http://source.android.com/devices/tech/security/index.html. Last access: July 2016.

[13] June Andronick. *Modélisation et Vérification Formelles de Systèmes Embarqués dans les Cartes à Microprocesseur – Plate-Forme Java Card et Système d'Exploitation*. PhD thesis, Université Paris-Sud, 2006.

[14] Alessandro Armando, Gabriele Costa, and Alessio Merlo. Formal modeling and reasoning about the Android security framework. In Catuscia Palamidessi and Mark Dermot Ryan, editors, *Trustworthy Global Computing - 7th International Symposium, TGC 2012, Newcastle upon Tyne, UK, September 7-8, 2012, Revised Selected Papers*, volume 8191 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2012. doi:10.1007/978-3-642-41157-1_5.

[15] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Jesús Mauricio Chimento, and Carlos Luna. Formally verified implementation of an

idealized model of virtualization. In Ralph Matthes and Aleksy Schubert, editors, *19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France*, volume 26 of *LIPIcs*, pages 45–63. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013. `doi:10.4230/LIPIcs.TYPES.2013.45`.

[16] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. *SIGPLAN Not.*, 44(1):90–101, January 2009. `doi:10.1145/1594834.1480894`.

[17] Santiago Zanella Béguelin, Gustavo Betarte, and Carlos Luna. A formal specification of the MIDP 2.0 security model. In Theodosis Dimitrakos, Fabio Martinelli, Peter Y. A. Ryan, and Steve A. Schneider, editors, *Formal Aspects in Security and Trust, Fourth International Workshop, FAST 2006, Hamilton, Ontario, Canada, August 26-27, 2006, Revised Selected Papers*, volume 4691 of *LNCS*, pages 220–234. Springer, 2006. `doi:10.1007/978-3-540-75227-1_15`.

[18] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[19] Yves Bertot, Pierre Castéran, Gérard (informaticien) Huet, and Christine Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004. Données complémentaires http://coq.inria.fr. URL: `http://opac.inria.fr/record=b1101046`.

[20] G. Betarte, E. Giménez, C. Loiseaux, and B. Chetali. FORMAVIE: Formal Modeling and Verification of the Java Card 2.1.1 Security Architecture. In *Proceedings of eSmart'02*, 2002.

[21] Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and Agustín Romano. Verifying Android's permission model. In Martin Leucker, Camilo Rueda, and Frank D. Valencia, editors, *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, volume 9399 of *Lecture Notes in Computer Science*, pages 485–504. Springer, 2015. `doi:10.1007/978-3-319-25150-9_28`.

[22] Michele Bugliesi, Stefano Calzavara, and Alvise Spanò. Lintent: Towards security type-checking of Android applications. In *Proceedings of Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference*, FMOODS/FORTE 2013, pages 289–304, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-38592-6_20.

[23] Avik Chaudhuri. Language-based security on Android. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 1–7, New York, NY, USA, 2009. ACM. doi:10.1145/1554339.1554341.

[24] Boutheina Chetali and Quang-Huy Nguyen. About the world-first smart card certificate with EAL7 formal assurances. Slides 9th ICCC, Jeju, Korea, September 2008. Available at: www.commoncriteriaportal.org/iccc/9iccc/pdf/B2404.pdf.

[25] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM. doi:10.1145/1999995.2000018.

[26] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. CRePE: context-related policy enforcement for Android. In *Proceedings of the 13th international conference on Information security*, ISC'10, pages 331–345, Berlin, Heidelberg, 2011. Springer-Verlag. doi:10.1007/978-3-642-18178-8_29.

[27] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, February 1988. doi:10.1016/0890-5401(88)90005-3.

[28] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *Proceedings of the 13th international conference on Information security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag. doi:10.1007/978-3-642-18178-8_30.

[29] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Android security: A survey of issues, malware penetration, and defenses.

IEEE Communications Surveys and Tutorials, 17(2):998–1022, 2015. doi:10.1109/COMST.2014.2386139.

[30] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM. doi:10.1145/2046707.2046779.

[31] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*. USENIX Association, 2011. URL: http://static.usenix.org/events/sec11/tech/full_papers/Felt.pdf.

[32] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. Modeling and enhancing Android's permission system. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Proceedings of the 17th European Symposium on Research in Computer Security, ESORICS 2012*, volume 7459 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2012. doi:10.1007/978-3-642-33167-1_1.

[33] Gartner. Gartner says worldwide smartphone sales grew 9.7 percent in fourth quarter of 2015. Technical report, Gartner, Inc., 2016. Available at: www.gartner.com/newsroom/id/3215217. Last access: July 2016.

[34] GSI. Formal verification of the security model of Android: Coq code. Available at: http://www.fing.edu.uy/inco/grupos/gsi/documentos/proyectos/modelo.tar.gz. Last access: July 2016.

[35] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52:107–115, July 2009. doi:10.1145/1538788.1538814.

[36] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

[37] Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *LNCS*. Springer-Verlag, 2003. doi:10.1007/3-540-39185-1_12.

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

[39] Open Handset Alliance. *Android project.* Available at: //source.android.com/. Last access: July 2016.

[40] C. Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In M. Bezem and J. F. Groote, editors, *1st Int. Conf. on Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 328–345. Springer-Verlag, 1993. doi:10.1007/BFb0037116.

[41] Mariantonietta La Polla, Fabio Martinelli, and Daniele Sgandurra. A survey on security for mobile devices. *IEEE Communications Surveys and Tutorials*, 15(1):446–471, 2013. doi:10.1109/SURV.2012.013012.00028.

[42] Bahman Rashidi and Carol Fung. A survey of android security threats and defenses. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 6(3):3–35, September 2015. URL: http://isyou.info/jowua/papers/jowua-v6n3-1.pdf.

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

[44] Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka. A formal model to analyze the permission authorization and enforcement in the Android framework. In *Proceedings of the 2010 IEEE Second International Conference on Social Computing*, pages 944–951, Washington, DC, USA, 2010. IEEE Computer Society. doi:10.1109/SocialCom.2010.140.

[45] Jeff Six. *Application Security for the Android Platform.* O'Reilly Media, 2011.

[46] Guillermo Suarez-Tangil, Juan E. Tapiador, Pedro Peris-Lopez, and Arturo Ribagorda. Evolution, detection and analysis of malware for

smart devices. *IEEE Communications Surveys and Tutorials*, 16(2):961–987, 2014. doi:10.1109/SURV.2013.101613.00077.

[47] Sufatrio, Darell J. J. Tan, Tong-Wei Chua, and Vrizlynn L. L. Thing. Securing Android: A survey, taxonomy, and challenges. *ACM Comput. Surv.*, 47(4):58:1–58:45, May 2015. doi:10.1145/2733306.

[48] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.5*, 2016. URL: http://coq.inria.fr.

A Semantics of Actions `call`, `grantP`, `revoke`, and `stop`

The formulation of the propositions that establish the pre- and postcondition of each action make use of auxiliary predicates and functions, which are describe in Table 3.

The semantics of the actions are depicted in Figures 5, 6, 7, and 8.

$grantPPerm(ap, cp, u, pt, s)$	returns the permanent permissions delegations of state s with the addition of the permanent delegated permission to application ap to perform the operation pt on the resource identified by u of the content provider cp.
$revokePPerm(cp, u, pt, s)$	returns the permanent permissions delegations of state s without the permanent delegated permissions on the resource identified by u, of the content provider cp, to perform the operation pt.
$revokeTPerm(cp, u, pt, s)$	returns the temporary permissions delegations of state s without the temporary delegated permissions on the resource identified by u, of the content provider cp, to perform the operation pt.
$isSystemPerm(p)$	holds if p is a predefined permission in the system.
$permlSAC(p, sac)$	holds if p is required to call sac.
$stopIns(ic, s)$	returns the running component instances of state s without the instance ic whose execution is being stopped.
$revokeTPermsIns(ic, s)$	returns the temporary permissions delegations of state s without the temporary delegated permissions to the instance ic.

Table 3: Helper functions and predicates

Action `call` *ic sac*

The running component *ic* makes the API call *sac*.

Rule

$$\boxed{\dfrac{\begin{array}{c}\exists (c : Comp), compInstalled(c, s) \wedge running(ic, c, s) \wedge \\ s = (aps, ps, psD, iCs, delPP, delTP, v, img) \wedge \forall (a : App)(p : Perm), \\ isSystemPerm(p) \wedge a \in aps \wedge inApp(c, a) \ \wedge permSAC(p, sac) \rightarrow (a, p) \in ps\end{array}}{s \xhookrightarrow{\texttt{call}\ ic\ sac} s}}$$

Precondition The component *ic* is a running instance of the installed component *c* in state *s* and the corresponding application has the necessary permissions to make the API call *sac*.
Postcondition After the execution of this action, the system state remains unchanged.

Figure 5: Action `call`

Action `grantP` *ic cp a u pt*

The running component *ic* delegates permanent permissions to application *a*. This delegation enables *a* to perform operation *pt* on the resource assigned to URI *u* from content provider *cp*.

Rule

$$\boxed{\dfrac{\begin{array}{c}compInstalled(cp, s) \wedge s = (aps, ps, psD, iCs, delPP, delTP, v, img) \wedge \\ canGrant(u, cp, s) \wedge existsRes(u, cp, s) \wedge inApp(c, a) \wedge \\ \exists (c : Comp), compInstalled(c, s) \wedge running(ic, c, s) \wedge \\ (canOp(c, cp, pt, s) \vee delPerms(c, cp, u, pt, s)) \\ grantPPerm(a, cp, u, pt, s) = delPP' \wedge \\ s' = (aps, ps, psD, iCs, delPP', delTP, v, img)\end{array}}{s \xhookrightarrow{\texttt{grantP}\ ic\ cp\ a\ u\ pt} s'}}$$

Precondition The component *cp* is a content provider installed in state *s*. Permissions can be delegated on *cp* for resource identified by the URI *u*. The application *ap* is installed in *s*. The component *ic* is a running instance of a component (*c*) that belongs to an installed application in *s*. Also, the application containing component *c* either has the appropriate permissions to perform the operation *pt* on *cp* or it has delegated permissions to perform *pt* on *u*.
Postcondition A permanent delegated permission to application *ap* to perform the operation *pt* on the resource identified by *u*, of the content provider *cp*, is added to the permanent permissions delegations of state *s*, and that is the only difference between both states.

Figure 6: Action `grantP`

Action `revoke` *ic cp u pt*

The running component *ic* revokes delegated permissions on URI *u* from content provider *cp* to perform operation *pt*.

Rule

$$\frac{\begin{array}{c} \mathit{compInstalled}(cp, s) \wedge s = (aps, ps, psD, iCs, delPP, delTP, v, img) \wedge \\ \mathit{existsRes}(u, cp, s) \wedge \exists\ (c : Comp), \mathit{compInstalled}(c, s) \wedge \\ \mathit{running}(ic, c, s) \wedge \mathit{canOp}(c, cp, pt, s) \\ \mathit{revokeTPerm}(cp, u, pt, s) = delTP' \wedge \\ \mathit{revokePPerm}(cp, u, pt, s) = delPP' \wedge \\ s' = (aps, ps, psD, iCs, delPP', delTP', v, img) \end{array}}{s \xhookrightarrow{\texttt{revoke}\ ic\ cp\ u\ pt} s'}$$

Precondition The content provider *cp* is installed in state *s* and it contains a resource that is pointed to by the *URI u*. The component *ic* is a running instance of the installed component *c*. Additionally, the application containing component *c* has the appropriate permissions to perform the operation *pt* on *cp*.
Postcondition The temporary and permanent delegated permissions on the resource identified by *u*, of the content provider *cp*, to perform the operation *pt* are removed from the temporary and permanent permissions delegations of state *s*. Apart from that, both states are equal.

Figure 7: Action `revoke`

Action `stop` *ic*

The execution of the running instance *ic* is stopped.

Rule

$$\frac{\begin{array}{c} s = (aps, ps, psD, iCs, delPP, delTP, v, img) \ \wedge\ \exists(c : Comp), \\ \mathit{compInstalled}(c, s) \wedge \mathit{running}(ic, c, s) \\ \mathit{stopIns}(ic, s) = iCs' \wedge \mathit{revokeTPermsIns}(ic, s) = delTP' \wedge \\ s' = (aps, ps, psD, iCs', delPP, delTP', v, img) \end{array}}{s \xhookrightarrow{\texttt{stop}\ ic} s'}$$

Precondition The component *ic* is a running instance of a component that belongs to an installed application in state *s*.
Postcondition The instance whose execution is being stopped is not present in the resulting state and all the permissions delegated temporarily to that instance are revoked. Apart from that, both states are equal.

Figure 8: Action `stop`