Towards a Certified Reference Monitor of the Android 10 Permission System

Guido De Luca ✉
Universidad Nacional de Rosario, Argentina

Carlos Luna ✉ 🏠
InCo, Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay

Abstract

Android is a platform for mobile devices that captures more than 85% of the total market share [14]. Currently, mobile devices allow people to develop multiple tasks in different areas. Regrettably, the benefits of using mobile devices are counteracted by increasing security risks. The important and critical role of these systems makes them a prime target for formal verification. In our previous work [10], we exhibited a formal specification of an idealized formulation of the permission model of version `6` of Android. In this paper we present an enhanced version of the model in the proof assistant `Coq`, including the most relevant changes concerning the permission system introduced in versions `Nougat`, `Oreo`, `Pie` and `10`. The properties that we had proved earlier for the security model have been either revalidated or refuted, and new ones have been formulated and proved. Additionally, we make observations on the security of the most recent versions of Android. Using the programming language of `Coq` we have developed a functional implementation of a reference validation mechanism and certified its correctness. The formal development is about 23k LOC of `Coq`, including proofs.

2012 ACM Subject Classification Security and privacy → Systems security; Theory of computation → Proof theory

Keywords and phrases Android, Permission model, Formal idealized model, Reference monitor, Formal proofs, Certified implementation, Coq

Digital Object Identifier 10.4230/LIPIcs.TYPES.2020.3

Related Version *Preliminary Version*: `https://arxiv.org/abs/2011.00720` [20]

Supplementary Material *Software (Source Code)*:
`https://github.com/g-deluca/android-coq-model`
archived at `swh:1:dir:a190369ad1ad079f2c9acc1470b0de19762520ae`

1 Introduction

Android [24] is the most used mobile OS in the world, capturing approximately 85% of the total market-share [14]. It offers a huge variety of applications in its official store that aim to help people in their daily activities, many of them critical in terms of privacy. In order to guarantee their users the security they expect, Android relies on a *multi-party* consensus system where user, OS and application must be all in favour of performing a task. This security framework is built upon a system of permissions, which are basically tags that developers declare on their applications to gain access to sensitive resources. Whenever an action that requires some of this permissions is executed for the first time, the user will be asked for authorization and if provided, the OS will ensure that only the required access is granted. The important and critical role of this security mechanism makes it a prime target for (formal) verification.

Security models play an important role in the design and evaluation of security mechanisms of systems. Earlier, their importance was already pointed in the Anderson report [1], where the concept of *reference monitor* was first introduced. This concept defines the design requirements for implementing what is called a *reference validation mechanism*, which shall

26th International Conference on Types for Proofs and Programs (TYPES 2020).
Editors: Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch; Article No. 3; pp. 3:1–3:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

be responsible for enforcing the access control policy of a system. For ensuring the correct working of this mechanism three design requirements are specified: i) the reference validation mechanism (RVM) must always be invoked (*complete mediation*); ii) the RVM must always be tamper-proof (*tamper-proof*); and iii) the RVM must be small enough to be subject to analysis and tests, the completeness of which can be assured (*verifiable*).

The work presented here is concerned with the verifiability requirement. In particular we put forward an approach where formal analysis and verification of properties is performed on an idealized model that abstracts away the specifics of any particular implementation, and yet provides a realistic setting in which to explore the issues that pertain to the realm of (critical) security mechanisms of Android. The formal specification of the reference monitor shall be used to establish and prove that the security properties that constitute the intended access control policy are satisfied by the modeled behavior of the validation mechanisms.

Contributions. In our previous work [10] we presented a formal specification of an idealized formulation of the permission model of version 6 of Android. We also developed, using the programming language of `Coq` [27], an executable (functional) specification of the reference validation mechanism and we proved its correctness conforming to the specified model. Lastly, we used the program extraction mechanism provided by `Coq` [18] to derive a certified `Haskell` implementation of the reference validation mechanism. Here we present an enhanced version of the model, including the most relevant changes concerning the permission system introduced in versions `Nougat`, `Oreo`, `Pie` and `10`. Some of these changes don't have a direct impact on our abstract model. In those cases, an informal analysis is included. The executable specification was also updated, and with that, the derived implementation as well. The properties that we had proved for the security model have been either revalidated or refuted, and new ones have been formulated and proved. The formal development is about 23k LOC of `Coq`, including various lemmas and their proofs.

Organization of the paper. Section 2 reviews the security mechanisms of Android and briefly describes the changes introduced in the later versions. Sections 3 and 4 present the formal axiomatic specification and the semantics of the certified implementation, respectively. Both sections discuss relevant properties concerning the new features. Section 5 considers related work and finally, Section 6 concludes with a summary of our contributions and directions for future work. The full formalization is available at `https://github.com/g-deluca/android-coq-model` [19] and can be verified using the `Coq` proof assistant. A preliminary version of this paper is accessible on arXiv [20].

2 Android's security model

2.1 Basic security mechanisms

The Android security model is primarily based on a sandbox and permission mechanism. Each application runs in a private virtual machine with a unique ID assigned to it, which means that one application's code is isolated from the code of the rest. This isolation means that, by default, applications can not interact with each other and have limited access to the OS. For example, if an application tries to do something malicious, like reading the user's contacts without permission, the action will fail due to the lack of privileges. However, these actions could also be started by trusted applications, and therefore, need to be done. Android's permission system is the mechanism in charge of deciding which of these actions should occur and which ones should not, depending on the permissions that each application has.

Every permission is identified by a unique name/text, has a protection level and may belong to a permission group. Furthermore, permissions can be classified into two groups: the ones defined by an application, for the sake of self-protection; and those predefined by Android, which are required to gain access to certain system features, like internet or location. Depending on the protection level of the permission, the system defines the rules to grant that permission. There are three classes of permission levels [4]: i) *normal*, these permissions can be automatically granted since they cover data or resources where there's very little risk to the user's privacy or the operation of other apps; ii) *dangerous*, permissions of this level provide access to data or resources that may be sensitive or could potentially affect the operation of other applications, and explicit user approval is needed to be granted; and iii) *signature*, a permission of this level is granted only if the application that requires it and the application that defined it are both signed with the same certificate. An application must declare –in an XML file called `AndroidManifest`– the set of permissions it needs to acquire further capacities than the default ones. From version `6` of Android, *dangerous* permissions are granted at runtime whereas both *normal* and *signature* are given when the application is installed.

Permissions may belong to groups that reunite a device's capabilities. The main purpose of grouping permissions in this way is to handle permission requests at the group level, in order to avoid overwhelming the user with too many questions. For example, the `SMS` group includes the permission needed to read text messages as well as the one needed to receive them (both considered to be *dangerous*). Whenever an application needs one of those for the first time, the user will be asked to authorize the whole group. In Section 2.2, we explain what *authorizing a group* means depending on the platform version.

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 [2]: i) *activity*, which is essentially a user interface of the application; ii) *service*, a component that executes in the background without providing an interface to the user; iii) *content provider*, a component intended to share information among applications; and iv) *broadcast receiver*, a component whose objective is to receive messages, sent either by the system or an application, and trigger the corresponding actions. The communication between components is achieved with the exchange of special messages called *intents*, which can be either i) *explicit*, meaning that the target application is specified; or ii) *implicit*, where only the action to be performed is declared and the system determines which application will run the task (if there is more than one capable application, the user is allowed to choose). In order to be candidates for the resolution of implicit intents, an application must declare on their manifest an *intent filter* that indicates the types of intents it can respond to.

Android provides two mechanisms by which an application can delegate its own permissions to another one. These mechanisms are called *pending intents* and *URI permissions.* An intent may be defined by a developer to perform a particular action. A `PendingIntent` specifies a reference to an action, which might be used by another application to perform the operation with the same permissions and identity of the one that created the intent. The *URI permissions* mechanism can be used by an application that has read/write access to a *content provider* to temporarily delegate those permissions to another application. These permissions are revoked once the receiver activity or service becomes inactive.

2.2 A brief review on the changelog

As we described in our previous work [10], the sixth version of Android introduced an important change to the system, allowing the users to handle permissions at runtime. In this section, we give a short account of the changes introduced between Android `Nougat` (`7`) and Android `10`, that had a significant impact on the permission system.

Filesystem

In order to improve security, the private directory of applications targeting[1] Android `7.0` or higher has restricted access: only the owner is capable of reading, writing or executing files stored in it. This configuration prevents leakage of metadata of private files, like the size or existence. With this change, applications are no longer able to share files simply by changing the file permissions and sharing their private URI; a content provider must be used in order to generate a reference to the file. With this approach, a new kind of URI is generated, which grants a temporary permission that will be available for the receiver activity or service only while they are active/running.

Our previous model already allowed granting temporary permissions to content providers URIs, so no change was required to formalize this new feature.

Grouped permissions

Prior to Android `8`, if an application requested a grouped permission at runtime and the user authorized it, the system also granted the rest of the permissions from the same group that were declared on the manifest. This behaviour was incorrect since it violated the intended least privilege security policy claimed by the designers of the platform. For applications targeting Android `8` or higher, this action was corrected and only the requested permission is granted. However, once the user authorized a group, all subsequent requests for permissions in that group are automatically granted. This change was added to the model.

Normal grouped permissions. According to Android's official documentation, *any permission can belong to a permission group regardless of protection level* [3]. However, it is not specified if normal and dangerous permissions can share a group or, in case that it is possible, how the system should handle this situation. A few questions we have raised are the following: i) Is the authorization to automatically concede permissions from that group granted at installation time together with the normal permissions?; ii) Is the user warned about that decision?; iii) If that is the case, then there's a contradiction with the documentation, since it claims that *a permission's group only affects the user experience if the permission is dangerous*; and iv) If it's not, how does the system avoid that dangerous permissions from the same group are not automatically granted later by the system?

In this work we formalized a worst-case scenario (that still suits the informal specification given by the authors of the platform), where a normal permission enables the automatic granting of dangerous permissions belonging to the same group. We formally discuss this situation in Section 3.4.

[1] Applications can *target* a particular version of the system. Android uses this setting to determine whether to enable any compatibility behaviors or features.

Privacy changes

Android Pie (9) introduced several changes aiming to enhance users' privacy, such as limiting background apps' access to device sensors, restricting information retrieved from Wi-Fi scans, and adding new permission groups and rules to reorganize phone calls and phone state related permissions. Later, the tenth version of the platform continued adding limitations to services: a new permission for accessing the location in the background was added. Furthermore, Android 10 placed restrictions on when a service can start an activity, in order to minimize interruptions for the user and keep the user more in control of what is shown on their screen.

These changes are specific to the implementation, meaning that they have no impact on an abstract representation like ours.

Permission check on legacy apps

Applications that target Android 5.1 or lower are considered to be old[2]. If an *old* application runs on an Android 10 system for the first time, a prompt appears on the screen, giving the user an opportunity to revoke access to permissions that the system previously granted at install time. This feature has been added to our model.

3 Formalization of Android's permission system

In this section we describe the axiomatic semantics of our model of the system, focusing on the features introduced in the later versions. We also discuss some of the verified properties.

Formal language used. Coq is an interactive theorem prover based on higher order logic that allows to write formal specifications and interactively generate machine-checked proofs of theorems. It also provides a (dependently typed) functional programming language that can be used to write executable algorithms. The Coq environment also provides program extraction towards languages like Ocaml and Haskell for execution of (certified) algorithms [17, 18]. In this work, enumerated types and sum types are defined using Haskell-like notation; for example, $option\ T \stackrel{\mathrm{def}}{=} None \mid Some\ (t : T)$. Record types are of the form $\{l_1 : T_1, \ldots, l_n : T_n\}$, whereas their elements are of the form $\{t_1, \ldots, t_n\}$. Field selection is written as $r.l_i$. We also use $\{T\}$ to denote the set of elements of type T. Finally, the symbol × defines tuples, and *nat* is the datatype of natural numbers. We omit Coq code for reasons of clarity; this code is available in [19].

3.1 Model states

The Android security model we have developed has been formalized as an abstract state machine. In this model, states (AndroidST) are modelled as 13-tuples that store dynamic data about the system such as the installed applications and their current permissions, as well as static data like the declared manifest of each installed app. A complete formal definition is given in Figure 1.

The type PermId represents the set of permissions identifiers; PermGroup, the set of permission groups identifiers; Comp, the application components whose code will run on the system; AppId represents the set of application identifiers; iComp is the set of identifiers of running instances of application components; ContProv is a subset of Comp, a special type

[2] We can also refer to them as *legacy* applications.

Auxiliary definitions	
OpTy	::= *read* | *write* | *rw*
PermLvl	::= *Dangerous* | *Normal* | *Signature* | *Signature/System*
Perm	::= PermId × *option* PermGroup × PermLvl
CompInstance	::= iComp × Comp
Manifest	::= {Comp} × *option nat* × *option nat* × {PermId} × {Perm} × *option* PermId
State components	
InstApps	::= {AppId}
VerifiedApps	::= {AppId}
AppPS	::= {AppId × PermId}
PermsGr	::= {AppId × PermGroup}
CompInsRun	::= {CompInstance}
DelPPerms	::= {AppId × ContProv × Uri × OpTy}
DelTPerms	::= {iComp × ContProv × Uri × OpTy}
ARVS	::= {AppId × Res × Val}
Intents	::= {iComp × Intent}
Manifests	::= {AppId × Manifest}
Certs	::= {AppId × Cert}
AppDefPS	::= {AppId × Perm}
SysImage	::= {SysImgApp}
AndroidST	::= InstApps × VerifiedApps × PermsGr × AppPS × CompInsRun × DelPPerms × DelTPerms × ARVS × Intents × Manifests × Certs × AppDefPS × SysImage

Figure 1 Android state.

of component that allows sharing resources among different applications; a member of the type Uri is a particular URI (uniform resources identifier); the type Res represents the set of resources an application can have (through its *content providers*, members of ContProv); the type Val is the set of possible values that can be written on resources; an intent –i.e. a member of type Intent– represents the intention of a running component instance to start or communicate with other applications; a member of SysImgApp is a special kind of application which is deployed along with the OS itself and has certain privileges, like being impossible to uninstall.

The first component of the state records the identifiers (AppId) of the applications installed by the user. The second component is a subset of the first one, and represents those applications that are considered to be old but have already been verified, also by the user. The third component keeps track of the permissions granted to every application present in the system, including the ones preinstalled on the system. Similarly, the next component holds the information about what permission groups have already been authorized by the user on each app. The fifth component of the state stores the set of running component instances (CompInstance), while the components DelPPerms and DelTPerms store the information concerning permanent and temporary permissions delegations, respectively[3]. The eighth and ninth components of the state store respectively the values (Val) of resources (Res) of applications and the set of intents (Intent) sent by running instances of components (iComp) not yet processed. The four last components of the state record information that represents the manifests of the applications installed by the user, the certificates (Cert) with which they were signed and the set of permissions they define. The last component of the state stores the set of (native) applications installed in the Android system image, information that is relevant when granting permissions of level *Signature/System*.

A manifest (Manifest) is modelled as a 6-tuple whose members respectively declare application components (set of components, of type Comp, included in the application); optionally, the minimum version of the Android SDK required to run the application;

[3] A permanent delegated permission represents that an app has delegated permission to perform an operation on the resource identified by an URI. A temporary delegated permission refers to a permission that has been delegated to a component instance.

optionally, the version of the Android SDK targeted on development; the set of permissions it may need to run at its maximum capability; the set of permissions it declares; and the permission required to interact with its components, if any. Application components are all denoted by a component identifier. A content provider (ContProv), in addition, encompasses a mapping to the managed resources from the URIs assigned to them for external access. 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 an abstract type.

Valid states. The states defined in this way include some cases that are not relevant with the model we are trying to analyze. For example, we don't want a state where a preinstalled application and one installed by the user the same identifier. In order to prevent this inconsistencies, we define a notion of valid state that captures several well-formedness conditions. It is formally defined as a predicate `valid_state` on the elements of type AndroidST. This predicate holds on a state s if the following conditions are met:

- all the components both in installed applications and in system applications have different identifiers;
- no component belongs to two different applications present in the device;
- no running component is an instance of a content provider;
- every temporally delegated permission has been granted to a currently running component and over a content provider present in the system;
- every running component belongs to an application present in the system;
- every application that sets a value for a resource is present in the system;
- the domains of the partial functions Manifests, Certs and AppDefPS are exactly the identifiers of the user-installed applications;
- the domains of the partial functions AppPS and PermsGr are exactly the identifiers of the applications in the system, both those installed by the users and the system applications;
- every installed application has an identifier different from those of the system applications, whose identifiers differ as well;
- all the permissions defined by applications have different identifiers;
- every partial function is indeed a function, that is, their domains don't have repeated elements;
- every individually granted permission is present in the system; and
- all the sent intents have different identifiers.

All these safety properties have a straightforward interpretation in our model. The full formal definition of the predicate is available in [19].

3.2 Action semantics

We modelled the different functionalities provided by the Android security system as a set of actions (of type Action) that determine how the system is able to transition from one state to another. Table 1 summarises the actions specified in our previous model that remained mostly the same since the new features didn't affect them whereas Table 2 groups those that are new or that suffered a big semantic change.

The behaviour of each action is specified in terms of a precondition (Pre : AndroidST $\rightarrow$ Action $\rightarrow$ $Prop$) and a postcondition ($Post$: AndroidST $\rightarrow$ Action $\rightarrow$ AndroidST $\rightarrow$ $Prop$).

Table 1 Legacy actions.

`install` *app m c lRes*	Install application with id *app*, whose manifest is *m*, is signed with certificate *c* and its resources list is *lRes*.
`uninstall` *app*	Uninstall the application with id *app*.
`read` *ic cp u*	The running comp. *ic* reads the resource corresponding to URI *u* from content provider *cp*.
`write` *ic cp u val*	The running comp. *ic* writes value *val* on the resource corresponding to URI *u* from content provider *cp*.
`startActivity` *i ic*	The running comp. *ic* asks to start an activity specified by the intent *i*.
`startActivityRes` *i n ic*	The running comp. *ic* asks to start an activity specified by the intent *i*, and expects as return a token *n*.
`startService` *i ic*	The running comp. *ic* asks to start a service specified by the intent *i*.
`sendBroadcast` *i ic p*	The running comp. *ic* sends the intent *i* as broadcast, specifying that only those components who have the permission *p* can receive it.
`sendOrdBroadcast` *i ic p*	The running comp. *ic* sends the intent *i* as an ordered broadcast, specifying that only those components who have the permission *p* can receive it.
`sendSBroadcast` *i ic*	The running comp. *ic* sends the intent *i* as a sticky broadcast.
`resolveIntent` *i app*	Application *app* makes the intent *i* explicit.
`stop` *ic*	The running comp. *ic* finishes its execution.
`grantP` *ic cp app u pt*	The running comp. *ic* delegates permanent permissions to application *app*. This delegation enables *app* to perform operation *pt* on the resource assigned to URI *u* from content provider *cp*.
`revokeDel` *ic cp u pt*	The running comp. *ic* revokes delegated permissions on URI *u* from content provider *cp* to perform operation *pt*.
`call` *ic sac*	The running comp. *ic* makes the API call *sac*.

Table 2 New or modified actions.

`grant` *p app*	Grant the permission *p* to the application *app* with user confirmation.
`grantAuto` *p app*	Grant automatically the permission *p* to the application *app* (without user confirmation).
`revoke` *p app*	Remove an ungrouped permission *p* from the application *app*.
`revokePermGroup` *g app*	Remove the every permission of group *g* from the application *app*.
`hasPermission` *p app*	Check if the application *app* has the permission *p*.
`receiveIntent` *i ic app*	Application *app* receives the intent *i*, sent by the running comp. *ic*.
`verifyOldApp` *app*	Application *app* granted permissions are verified by the user

For instance, the axiomatic semantics of the new feature about automatic granting of permissions **grantAuto** is given by:

$$
\begin{aligned}
&Pre(s, \texttt{grantAuto}\ p\ app) \overset{\text{def}}{=}\\
&\quad (\exists m : \mathsf{Manifest},\ m = getManifestForApp(app, s)\\
&\qquad \wedge\ getPermissionId(p) \in (use\ m)) \wedge\\
&\quad (isSystemPerm\ p\ \vee usrDefPerm\ p) \wedge\\
&\quad p\ \notin grantedPerms(app, s) \wedge\\
&\quad permLevel(p)\ = dangerous \wedge\\
&\quad (\exists g : \mathsf{PermGroup},\ getPermissionGroup(p) = Some\ g\\
&\qquad \wedge\ g \in getAuthorizedGroups(app, s))\\
&Post(s, \texttt{grantAuto}\ p\ app, s') \overset{\text{def}}{=}\\
&\quad grantPerm(app, p, s, s') \wedge\\
&\quad sameOtherFieldsOnGrantAuto(s, s')
\end{aligned}
$$

The precondition establishes several conditions that must be fulfilled before this action is able to transition. The first one requires that the permission p is listed on the application's manifest (and this manifest, of course, is required to exist). Regarding the permission, it is also required that it is defined either by the user or the system, that its level is *dangerous* and that it has not been already granted to *app*. Up to this point, the precondition of *grantAuto* is exactly the same as the one of *grant*. The main difference is established by the following condition: the permission at issue should belong to a group g and the system should know that the user had previously authorized that group for automatic granting.

The postcondition of **grantAuto** $p\ a$ requires that for an initial state s and a final state s', the individual permission p is granted to application *app*. This condition is enforced by the **grantPerm** $a\ p\ s\ s'$ predicate which only alters the state in component that maps applications with their current *dangerous* permissions. Every other component of the state remains the same.

3.3 Executions

Whenever the system attempts to execute an action a over a valid state s, there are two possible outcomes. If the precondition holds, the system will transition to another state s' where the postcondition of a is established; but if the precondition is not satisfied on s, then the state remains unchanged and the system answers with an error message determined by the relation $ErrorMsg$[4].

Formally, the possible answers of the system are defined by the type $Response \overset{\text{def}}{=} ok \mid error\ (ec : ErrorCode)$ and the executions can be specified with this operational semantics:

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

One-step execution with error management preserves valid states.

▶ **Lemma 1** (Validity is invariant).

$\forall\ (s\ s' : \mathsf{AndroidST})(a : \mathsf{Action})(r : Response), s \xhookrightarrow{a/r} s' \rightarrow valid\_state(s')$

[4] Given a state s, an action a and an error code ec, $ErrorMsg(s,a,ec)$ holds iff $error\ ec$ is an acceptable response when the execution of a is requested on state s.

The property is proved by case analysis on a, for each condition in *valid_state*, using several auxiliary lemmas [19].

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

3.4 Reasoning over the specified model

In this section we present and discuss some properties about the Android `10` security framework. We focus on safety-related properties about the changes introduced on the later versions of Android (mainly `Oreo` and `10`) rather than on security issues. Nevertheless, we also found potentially dangerous behaviours that may not be considered in the informal documentation of the platform and we formally reasoned about them as well. The full formal definition of these properties can be found in [19], along with the corresponding proofs.

On Table 3 we introduce helper functions and predicates used to define the properties that will follow.

Table 3 Helper functions and predicates.

Function/Predicate	Description
$appHasPermission(app, p, s)$	holds iff *app* is considered to have permission p on state s.
$canGrant(cp, u, s)$	holds iff the content provider *cp* allows the delegation of permissions over the resource at URI u on state s.
$canStart(c', c, s)$	holds if the app containing component c' (installed in s) has the required permissions to create a new running instance of c.
$cmpProtectedByPerm(c)$	returns the permission by which the component c is protected.
$componentIsExported(c)$	holds iff the component c is exported and can be accessed from other applications.
$existsRes(cp, u, s)$	holds iff the URI u belongs to the content provider *cp* on s.
$getAppFromCmp(c, s)$	given a component c on s, returns the app to which it belongs.
$getAppRequestedPerms(m)$	given the manifest m of an app, returns the set of permissions it uses.
$getDefPermsApp(app, s)$	returns the set of permissions defined by *app* on state s.
$getGrantedPermsApp(app, s)$	returns the set of indvidual permissions granted to *app* on s.
$getAuthorizedGroups(app, s)$	returns the set of permission groups that have been authorized for automatic granting for *app* on s.
$getInstalledApps(s)$	returns the set of identifiers of the applications installed on s.
$getManifestForApp(app, s)$	returns the manifest of application *app* on state s.
$getPermissionId(p)$	returns the identifier of permission p.
$getPermissionLevel(p)$	returns the permission level of permission p.
$getPermissionGroup(p)$	returns *Some* g if the permission p is grouped or $None$ if not.
$getRunningComponents(s)$	returns the set of pairs consisting of a running instance id, and its associated component currently running on state s.
$inApp(c, app, s)$	holds iff the component c belongs to application *app* on state s.
$permissionRequiredRead(c)$	returns the permission required for reading the component.
$permSACs(p, sac)$	holds iff permission p is required for performing the system call *sac* (of type $SACall$).
$oldAppNotVerified(a, s)$	holds iff the application a is considered old and the user hasn't verified it in state s.

The first property that we proved establishes a safety condition about the automatic granting of grouped permissions. It states that the system is not able to transition with this action unless the group of the permission involved is already authorized.

▶ **Property 1** (Automatic grant only possible on authorized groups).
$\forall (s, s' : \mathsf{AndroidST}) (p : \mathsf{Perm}) (g : \mathsf{PermGroup}) (app : \mathsf{AppId}),$
$getPermissionLevel(p) = dangerous \land getPermissionGroup(p) = Some\ g \land$
$g \notin getAuthorizedGroups(app, s) \rightarrow\ \neg s \xhookrightarrow{\texttt{grantAuto p app/ok}} s'$
Android's permission system ensures that an automatic granting can only occur on permissions that belong to authorized groups.

However, a few questions arise when trying to formally describe the situations in which *a group is authorized*. For instance, there is at least one valid state where the system can automatically grant a grouped permission to an app even though that the application has no other permission of the same group granted at that moment. This means that an application can have a group authorized for automatic granting via a permission that no longer exists. This is not necessarily a security flaw. It may be a design principle to avoid asking the user to authorize the same group too many times, but the decision is not clear or disambiguated in the official documentation.

▶ **Property 2** (Auto-granting permission without having others of the same group).
$\exists (s : \mathsf{AndroidST}) (p : \mathsf{Perm}) (g : \mathsf{PermGroup}) (app : \mathsf{AppId}),\ valid\_state(s) \land$
$getPermissionLevel(p) = dangerous\ \ \land getPermissionGroup(p) = Some\ g \land$
$\neg(\exists (p' : \mathsf{Perm}), p' \in getGrantedPermsApp(app, s)\ \land$
$getPermissionGroup(p') = Some\ g) \land Pre(s, grantAuto\ p\ a)$
System can automatically grant a permission even though there is currently no other permission of that group granted to the app.

The next property formalizes the situation described in Section 2.2 about normal and dangerous permissions sharing a group. We believe that permissions with different protection levels should not be allowed to share a group, since it could lead to a privilege escalation scenario.

▶ **Property 3** (Dangerous permission automatically granted without explicit consent).
$\forall (s, s' : \mathsf{AndroidST})\ (a : \mathsf{AppId})\ (m : \mathsf{Manifest})\ (c : \mathsf{Cert})\ (resources : list\ \mathsf{Res})$
$(g : \mathsf{PermGroup})\ (pDang\ pNorm : \mathsf{Perm}), s \xhookrightarrow{\texttt{install a m c resources/ok}} s' \rightarrow$
$getPermissionLevel(pDang) = dangerous\ \ \rightarrow getPermissionGroup(pDang) = Some\ g \rightarrow$
$getPermissionLevel(pNorm) = normal\ \ \rightarrow getPermissionGroup(pNorm) = Some\ g \rightarrow$
$\{pDang,\ pNorm\} \subseteq getAppRequestedPerms(m) \rightarrow Pre(s', grantAuto\ pDang\ a)$
An application that uses a normal and a dangerous permission of the same group, can obtain the dangerous one automatically after being installed.

Users are able to revoke permissions at runtime. However, the UI does not allow to revoke grouped permissions individually, the complete group is invalidated instead. We consider this behavior to be expected and desirable, and therefore, we proved that our model is consistent with it.

▶ **Property 4** (Revoking group revokes grouped individual permissions).
$\forall (s, s' : \mathsf{AndroidST})\ (g : \mathsf{PermGroup})\ (app : \mathsf{AppId}),\ s \xhookrightarrow{\texttt{revokePermGroup g app/ok}} s' \rightarrow$
$\neg(\exists (p : \mathsf{Perm}), p \in getGrantedPermsApp(app, s')\ \ \land getPermissionGroup(p) = Some\ g)$
Whenever a user revokes a permission group from an application, every individual permission that belongs to that group is revoked.

The following property reasons about another change mentioned in Section 2.2. It formalizes a good behaviour about the unverified legacy applications.

▶ **Property 5** (Unverified old app cannot receive intents).
$\forall (s, s' : \mathsf{AndroidST})\ (i : \mathsf{Intent})\ (ic : \mathsf{iComp})\ (app : \mathsf{AppId})$,
$oldAppNotVerified(app, s) \to \neg s \xhookrightarrow{\mathtt{receiveIntent}\ i\ ic\ app/ok} s'$
An old application that hasn't been verified by the user yet cannot receive intents, meaning that it can't start activities as well.

Finally, we include here a property that holds since version 6 of Android. Any application that wants to send information through the network must have the permission INTERNET, but since this permission is of level *normal*, the application just needs to declare it as used in its manifest. This will give access to the network in an implicit and irrevocable way. Once again, this has been criticized due to the potential information leakage it allows. The following property formally generalizes this situation and embodies a reasonable argument to roll back this security issue introduced in Android `Marshmallow`.

▶ **Property 6** (Internet access implicitly and irrevocably allowed).
$\forall (s : \mathsf{AndroidST})(sac : SACall)(c : \mathsf{Comp})(ic : \mathsf{iComp})(p : \mathsf{Perm})$,
$valid\_state(s) \to permSAC(p, sac) \to$
$getPermissionLevel(p) = normal \to getPermissionId(p) \in$
$getAppRequestedPerms(getManifestForApp(getAppFromCmp(c, s), s)) \to$
$(ic, c) \in getRunningComponents(s) \to s \xhookrightarrow{\mathtt{call}\ ic\ sac/ok} s$
If the execution of an Android API call only requires permissions of level normal, it is enough for an application to list them as used on its manifest file to be allowed to perform such call.

4 A certified reference validation mechanism

The implementation we developed in our previous model consisted in a set of `Coq` functions such that for every action in our axiomatic specification there exists a function which stands for it. In this work we kept this approach, updating those functions for which its axiomatic counterpart changed and adding new ones for the new actions `verifyOldApp` and `grantAuto`.

Functions that implement actions are basically state transformers. Their definition follows this pattern: first, it is checked whether the precondition of the action is satisfied in state s, and then, if that is the case, another function is called to return a state s' where the postcondition of the action holds. Otherwise, the state s is returned unchanged along with an appropriate response specifying an error code which describes the failure. More formally, the returned value has type $Result \overset{\text{def}}{=} \{resp : Response, st : \mathsf{AndroidST}\}$. In Figure 2 we present, as an example, the function that implements the execution of the `grant` action. The `Coq` code of this function, together with that of the remaining ones, can be found in [19][5]. The function *grant_pre* is defined as the nested validation of each of the properties of the precondition, specifying which error to throw when one of them doesn't hold. In general, every `<action>`*_pre* function is defined this way. The function *grant_post* implements the expected behaviour of the *grant* action: the permission `perm` is prepended to the list[6] of given permissions of the application `app` and, if that permission is grouped, that group is also added to the list of permissions groups authorized by the user on that application.

[5] We omit here the formal definition of these functions due to space constraints.
[6] We implement the sets in the model with lists of `Coq`.

```
Definition $grant\_safe(perm, app, s)$ : $Result$ :=
  match $grant\_pre(perm, app, s)$ with
       | $Some\ ec$ ⇒ $\{error(ec), s\}$
       | $None$ ⇒ $\{ok, grant\_post(perm, app, s)\}$
  end.
```

Figure 2 The function that implements the `grant` action.

Step

All of these functions are grouped into a *step* function, which basically acts as an action dispatcher[7]. Figure 3 show the structure of this function.

```
Definition $step(s, a)$ :=
  match $a$ with
       | ... ⇒ ...
       | grant $perm\ app$ ⇒ $grant\_safe(perm, app, s)$
       | ... ⇒ ...
  end.
```

Figure 3 Structure of the `step` function.

Traces

We have modeled the execution of the permission validation mechanism during a session of the system as a function that implements the execution of a list of actions starting in an initial system state. The output of the execution, a trace, is the corresponding sequence of states.

```
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.
```

4.1 Correctness of the implementation

We proceed now to outline the proof that our functional implementation of the security mechanisms of Android correctly implements the axiomatic model. This property has been formally stated as the following correctness theorem which in turn was verified using `Coq` [19].

▶ **Theorem 2** (Correctness of the reference validation mechanism).

$$\forall\ (s : \mathsf{AndroidST})\ (a : \mathsf{Action}),\ valid\_state(s) \to\ s \xhookrightarrow{a/step(s,a).resp} step(s,a).st$$

The proof of this theorem starts by performing a case analysis on the (decidable) predicate $Pre(s, a)$. Then, in case that the predicate holds, we apply Lemma 3; otherwise we continue by applying Lemma 4.

[7] Mechanism to trigger actions, on a state, according to the type of event considered.

▶ **Lemma 3** (Correctness of valid execution).
$\forall\ (s : \mathsf{AndroidST})\ (a : \mathsf{Action}), \mathit{valid\_state}(s) \rightarrow\ \mathit{Pre}(s,a)\ \rightarrow$
$s \xhookrightarrow{a/ok} \mathit{step}(s,a).\mathit{st}\ \wedge\ \mathit{step}(s,a).\mathit{resp} = \mathit{ok}$

▶ **Lemma 4** (Correctness of error execution).
$\forall\ (s : \mathsf{AndroidST})\ (a : \mathsf{Action}), \mathit{valid\_state}(s) \rightarrow\ \neg\mathit{Pre}(s,a) \rightarrow\ \exists\ (ec : \mathit{ErrorCode}),$
$\mathit{step}(s,a).\mathit{st} = s \wedge \mathit{step}(s,a).\mathit{resp} = \mathit{error}(ec) \wedge \mathit{ErrorMsg}(s,a,ec)$

The proof of these lemmas proceeds by applying functional induction on $\mathit{step}(s,a)$. Then, in Lemma 3, the proof continues by applying the corresponding subproof of soundness of the function that implements each action; whereas in Lemma 4, a subproof about the existence of a proper error code is provided.

4.2 Reasoning over the certified reference validation mechanism

In this section we present several security properties we have stated and proved about the function *trace* defined in Section 4.

The first property states that in Android 10, if an application that is considered to be old (as we defined in Section 2.2) is able to run, then it has been verified and validated by the user previously.

▶ **Property 7** (Old applications must be verified).
$\forall(\mathit{initState}, \mathit{lastState} : \mathsf{AndroidST})(\mathit{app} : \mathit{AppId})(l : \mathit{list}\ \mathsf{Action}),\ \mathit{valid\_state}(\mathit{initState}) \rightarrow$
$\mathit{app} \in \mathit{getInstalledApps}(\mathit{initState}) \rightarrow \mathit{oldAppNotVerified}(a, \mathit{initState}) \rightarrow$
$\mathit{canRun}(a, \mathit{lastState}) \rightarrow \mathit{last}(\mathit{trace}(\mathit{initState}, l), \mathit{initState}) = \mathit{lastState} \rightarrow$
$\texttt{uninstall}\ \mathit{app} \notin l \rightarrow \texttt{verifyOldApp}\ \mathit{app} \in l$

The only way for an old application to be able to execute is if the user verified it.

The next property establishes that for an application to have **any** dangerous permission (grouped or ungrouped) it must be explicitly granted to it, either by the user or automatically by the system.

▶ **Property 8** (Dangerous permissions must be explicitly granted).
$\forall(\mathit{initState}, \mathit{lastState} : \mathsf{AndroidST})(\mathit{app} : \mathit{AppId})(p : \mathsf{Perm})(l : \mathit{list}\ \mathsf{Action}),$
$\mathit{valid\_state}(\mathit{initState}) \rightarrow \mathit{app} \in \mathit{getInstalledApps}(\mathit{initState}) \rightarrow$
$\mathit{getPermissionLevel}(p) = \mathit{dangerous} \rightarrow \mathit{appHasPermission}(\mathit{app}, p, \mathit{lastState}) \rightarrow$
$\neg\mathit{appHasPermission}(\mathit{app}, p, \mathit{initState}) \rightarrow \texttt{uninstall}\ \mathit{app} \notin l \rightarrow$
$\mathit{last}(\mathit{trace}(\mathit{initState}, l), \mathit{initState}) = \mathit{lastState} \rightarrow (\texttt{grant}\ p\ \mathit{app} \in l \vee \texttt{grantAuto}\ p\ \mathit{app} \in l)$

The only way for an application to get a permission is if the user authorized it, or if the user authorized a group and the system is able to automatically grant it.

The following property formally states that if an application used to have a permission that was later revoked, only re-granting it will allow the application to have it again.

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

If a permission is revoked from an application, only regranting it will allow the application to have it again.

Whenever an application *app* receives a `READ/WRITE` permission *perm*, it also receives the right to delegate this permission to another application, say *app′*, to access that same resource on its behalf. However, if *perm* is later revoked from application *app*, there's a chance that *app′* still has access to that resource, since delegated permissions **are not recursively revoked**. The following property formalizes this situation and is a proof that the current specification allows a behavior which is arguably against the user's will.

▶ **Property 10** (Delegated permissions are not recursively revoked).
$\forall(s : \mathsf{AndroidST})(p : \mathsf{Perm})(app, app' : \mathsf{AppId})(ic, ic' : \mathsf{iComp})(c, c' : \mathsf{Comp})(u : \mathsf{Uri})$
$(cp : CProvider),\ valid\_state(s) \to step(s, \texttt{grant}\ p\ app).resp = ok \to$
$getAppFromCmp(c, s) = app \to getAppFromCmp(c', s) = app' \to$
$(ic, c) \in getRunningComponents(s) \to (ic', c') \in getRunningComponents(s) \to$
$canGrant(cp, u, s) \to existsRes(cp, u, s) \to componentIsExported(cp) \to$
$permissionRequiredRead(cp) = Some\ p \to$
$\mathsf{let}\ opsResult := trace(s, \texttt{[grant}\ p\ app, \texttt{grantP}\ ic\ cp\ app'\ u\ Read,$
$\texttt{revoke}\ p\ app\texttt{]}\ \mathsf{in}\ step(last(opsResult, s), \texttt{read}\ ic'\ cp\ u).resp = ok$

In Android 10, if a permission p is revoked for an application app not necessarily shall it be revoked for the applications to which app delegated p.

The purpose of the following property is to show that with runtime permissions introduced after `Android 6`, certain assertions on which a developer could rely in previous versions do not hold. For example, a running component may have the right of starting another one on a certain state, but may not be able to do so at a later time, even though no involved application was uninstalled. The property still holds on the latest version of Android.

▶ **Property 11** (The right to start an external component is revocable).
$\forall(initState : \mathsf{AndroidST})(l : list\mathsf{Action})(app, app' : \mathsf{AppId})(c : \mathsf{Comp})(act : \mathsf{Activity})$
$(p : \mathsf{Perm}),\ valid\_state(initState) \to$
$getPermissionLevel(p) = dangerous \to permissionIsGrouped(p) = None \to$
$app \neq app' \to p \notin getDefPermsApp(app, initState) \to inApp(c, app, initState) \to$
$inApp(act, app', initState) \to cmpProtectedByPerm(act) = Some\ p \to$
$canStart(c, act, initState) \to \exists(l : list\ \mathsf{Action}), \texttt{uninstall}\ app \notin l\ \wedge$
$\texttt{uninstall}\ app' \notin l \wedge \neg canStart(c, act, last(trace(initState, l), initState))$

A running component may have the right of starting another one on a certain state, but may not be able to do so at a later time.

5 Related work

Several analyses have been carried out concerning the security of the Android permission system. Plenty of them [11, 30, 13, 29, 23, 5] implement a static analysis tool that is capable of detecting overprivileges and unwanted information flow on a set of applications. This pragmatic approach may be helpful for Android users to keep their private information secure, but no properties about the system can be established. Recently, Mayrhofer *et al.* [22] described the Android security platform and documented the complex threat model and ecosystem it needs to operate, but no formal analysis was performed in it.

Few works study the aspects of the permission enforcing framework in a formal way. In particular, Shin *et al.* [25, 26] developed using `Coq` a framework that represents the Android permission system, similarly to what we did. Although, that work does not consider the different types of components, the interaction between a running instance and the system, the R/W operation on a content provider, the semantics of the permission delegation mechanism. Also, their work is based on an older version of the platform and some novel aspects, like the management of runtime permissions or the verification of legacy applications, are not included. Similarly, Bagheri *et al.* [6] formalized Android's permission protocol using `Alloy` [15]. The analysis performed, however, was based on the ability to automatically find counterexamples provided by the Alloy framework, which the authors claim to be tremendously helpful for identifying vulnerabilities. A `Coq`-based approach like ours, requires more human effort to identify a flaw but provides stronger guarantees on security and safety properties. Another formal work on Android is CrashSafe [16], where the authors formalized in `Coq` the inter-component communication mechanism and proved its safety with regard to failures (or *crashes*). This work, similarly to ours, focus on safety properties rather than security ones.

On the other hand, many works have addressed the problem of relating inductively defined relations and executable functions. In particular, Tollitte *et al.* [28] show how to extract a functional implementation from an inductive specification in `Coq`, and [9] exhibits a similar approach for `Isabelle`. Earlier, alternative approaches such as [7, 8] aim to provide reasoning principles for executable specifications. In [12], the verification of properties of imperative programs is performed using techniques based on the specialization of constrained logic programs. In this work we are able to develop independently the specification of the reference monitor and the implementation of the validation mechanism, considering that `Coq` provides a reasoning framework based on higher order logic to perform proofs of specifications and programs and a functional programming language. Other approaches could be considered to develop the formalization. For instance, a logical approach like the one used in [12]. However, a logical approach does not allow us to have the same functionalities in a unified formal environment.

Specifically, in this work we present a model of a reference monitor and demonstrate properties which shall hold for every correct implementation of the model. Then, we have developed a functional implementation in `Coq` of the reference validation mechanism and proved its correctness with respect to the specified reference monitor. Applying the program extraction mechanism provided by `Coq` we have also derived a certified `Haskell` prototype of the reference validation mechanism, which can be used to conduct verification activities on actual real implementations of the platform. The results presented in this paper extend the ones reported in [10, 21]. We have enriched the model presented in [10, 21] so as to consider the changes introduced in Android permission system by version `Nougat`, `Oreo`, `Pie` and `10`.

6 Final remarks

We have enhanced the formal specification considered in our previous work [10] with the new features concerning the permission system that have been added during the later releases of Android. With a conservative approach, we first analyzed the validity of the already formulated properties and then established new ones about the novel changes; summing up a total of 14 valid properties, without including the auxiliary lemmas that have been separated just for modularization. Among these properties we included several that aim to highlight how formal methods help to disambiguate unclear behaviours that may be inferred from an informal specification. For instance, we found a potentially dangerous situation in which an application can gain access to every dangerous permission that shares group with a normal one, without explicit consent of the user (see Property 3). This scenario fits the model (informally) described in the official documentation of the platform.

We also enriched our previous functional implementation of the reference validation mechanism with these new characteristics and updated its correctness proof. As consequence, the derived `Haskell` prototype obtained using the program extraction mechanism provided by the proof assistant, has been updated as well. The full certified code is available in [19] and is about 23k LOC of `Coq`, including proofs.

One important goal of our work is to keep our formalization up to date with the later versions of Android in order to constitute a reliable framework for reasoning about its permission system. We aim to help to increase the confidence on Android's security mechanisms by providing certified guarantees about the enforcement of this measures. The use of idealized models and certified prototypes is a good step forward but no doubt the definitive step is to be able to provide similar guarantees concerning actual implementations of the platform. We plan to use the certified extracted algorithm as a testing oracle and also to conduct verification activities on actual implementations of the platform, following the methodology proposed in [21]. In particular, we are investigating the use of that algorithm to compare the results of executing an action on a real Android platform and executing that same action on the correct program. This would allow us to monitor the actions performed in a real system and assessing whether the intended security policy is actually enforced by the particular implementation of the platform.

On September 8th 2020, Android `11` was released. This update includes features that continue increasing the security of the device, such as auto-resetting permissions from unused applications or one-time permissions for the most sensitive resources, like the microphone or camera. In future work, we intend to add this features to our model.

References

1 J. 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. *Application Fundamentals*. Available at: `http://developer.android.com/guide/components/fundamentals.html`. Last access: Feb. 2021.

3 Android Developers. *Permissions*. Available at: `http://developer.android.com/guide/topics/security/permissions.html`. Last access: Feb. 2021.

4 Android Developers. *Protection levels*. Available at: `https://developer.android.com/guide/topics/permissions/overview#normal-dangerous`. Last access: Feb. 2021.

5 H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. Covert: Compositional analysis of android inter-app permission leakage. *IEEE Transactions on Software Engineering*, 41(9):866–886, September 2015. `doi:10.1109/TSE.2015.2419611`.

6 Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. A formal approach for detection of security flaws in the android permission system. *Formal Aspects of Computing*, 30, November 2017. `doi:10.1007/s00165-017-0445-z`.

7 A. Balaa and Y. Bertot. Fix-point equations for well-founded recursion in type theory. In M. Aagaard and J. Harrison, editors, *TPHOLs*, volume 1869 of *LNCS*, pages 1–16. Springer, 2000. `doi:10.1007/3-540-44659-1_1`.

8 G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive functions: A practical tool for the coq proof assistant. In M. Hagiya and P. Wadler, editors, *FLOPS*, volume 3945 of *LNCS*, pages 114–129. Springer, 2006. `doi:10.1007/11737414_9`.

9 Stefan Berghofer, Lukas Bulwahn, and Florian Haftmann. Turning inductive into equational specifications. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *LNCS*, pages 131–146. Springer, 2009. `doi:10.1007/978-3-642-03359-9_11`.

10 Gustavo Betarte, Juan Campo, Felipe Gorostiaga, and Carlos Luna. *A Certified Reference Validation Mechanism for the Permission Model of Android: 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers*. Springer, July 2018. `doi:10.1007/978-3-319-94460-9_16`.

11 P. Chester, C. Jones, M. Wiem Mkaouer, and D. E. Krutz. M-perm: A lightweight detector for android permission gaps. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 217–218, May 2017. `doi:10.1109/MOBILESoft.2017.23`.

12 E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Program verification via iterated specialization. *Sci. Comput. Program.*, 95:149–175, 2014. `doi:10.1016/j.scico.2014.05.017`.

13 Michael Gordon, Kim deokhwan, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of android applications in droidsafe. In *NDSS Symposium 2015*, January 2015. `doi:10.14722/ndss.2015.23089`.

14 International Data Corporation (IDC). Smartphone market share. Technical report, International Data Corporation (IDC), 2020.

15 Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2012.

16 Wilayat Khan, Habib Ullah, Aakash Ahmad, Khalid Sultan, Abdullah Alzahrani, Sultan Khan, Mohammad Alhumaid, and Sultan Abdulaziz. Crashsafe: a formal model for proving crash-safety of android applications. *Human-centric Computing and Information Sciences*, 8, December 2018. `doi:10.1186/s13673-018-0144-7`.

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

18 Pierre Letouzey. A New Extraction for Coq. In *Proceedings of TYPES'02*, volume 2646 of *LNCS*, 2003.

19 Guido De Luca and Carlos Luna. Formal verification of the security model of Android 10: Coq code. Available at: `https://github.com/g-deluca/android-coq-model`. Last access: Feb. 2021.

20 Guido De Luca and Carlos Luna. Towards a certified reference monitor of the android 10 permission system. *CoRR*, abs/2011.00720, 2020. `arXiv:2011.00720`.

21 Carlos Luna, Gustavo Betarte, Juan Diego Campo, Camila Sanz, Maximiliano Cristiá, and Felipe Gorostiaga. A formal approach for the verification of the permission-based security model of android. *CLEI Electron. J.*, 21(2), 2018. `doi:10.19153/cleiej.21.2.3`.

22 René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The android platform security model. *CoRR*, abs/1904.05572, 2019. `arXiv:1904.05572`.

23 Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 77–88, Piscataway, NJ, USA, 2015. IEEE Press. URL: `http://dl.acm.org/citation.cfm?id=2818754.2818767`.

24 Open Handset Alliance. *Android project.* Available at: `//source.android.com/`. Last access: Feb. 2021.

25 W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka. A formal model to analyze the permission authorization and enforcement in the android framework. In *SocialCom'10*, pages 944–951, Washington, DC, USA, 2010. IEEE Computer Society. `doi:10.1109/SocialCom.2010.140`.

26 W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka. A frst step towards automated permission-enforcement analysis of the android framework. In *SAM 2010*, pages 323–329. CSREA Press, 2010.

27 The Coq Team. *The Coq Proof Assistant Reference Manual – Version V8.12.0*, 2020. URL: `http://coq.inria.fr`.

28 P.-N. Tollitte, D. Delahaye, and C. Dubois. Producing certified functional code from inductive specifications. In Chris Hawblitzel and Dale Miller, editors, *CPP*, volume 7679 of *LNCS*, pages 76–91. Springer, 2012. `doi:10.1007/978-3-642-35308-6_9`.

29 Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security*, 21:1–32, April 2018. `doi:10.1145/3183575`.

30 S. Wu and J. Liu. Overprivileged permission detection for android applications. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2019. `doi:10.1109/ICC.2019.8761572`.