A formal approach for the verification of the permission-based security model of Android

Gustavo Betarte, Juan Campo, Carlos Luna, Camila Sanz
InCo, Facultad de Ingeniería, Universidad de la República,
Montevideo, Uruguay
{gustun,jdcampo,cluna,csanz}@fing.edu.uy

and

Felipe Gorostiaga
IMDEA Software Institute
Madrid, Spain
felipe.gorostiaga@imdea.org

and

Maximiliano Cristiá
CIFASIS, Universidad Nacional de Rosario,
Rosario, Argentina
cristia@cifasis-conicet.gov.ar

Abstract

This article reports on our experiences in applying formal methods to verify the security mechanisms of Android. We have developed a comprehensive formal specification of Android's permission model, which has been used to state and prove properties that establish expected behavior of the procedures that enforce the defined access control policy. We are also interested in providing guarantees concerning actual implementations of the mechanisms. Therefore we are following a verification approach that combines the use of idealized models, on which fundamental properties are formally verified, with testing of actual implementations using lightweight model-based techniques. We describe the formalized model, present security properties that have been proved using the `Coq` proof assistant and propose the use of a certified algorithm for performing verification activities such as monitoring of actual implementations of the platform and also as a testing oracle. Additionally, we make observations to the latest Android security model.

Keywords: Android, Formal idealized model, Reference monitor, Certified implementation, Model-based security testing.

1 Introduction

Android [1] is a platform for mobile devices that captures more than 85% of the total market-share [2]. 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.

Security models play an important role in the design and evaluation of security mechanisms of systems. Their importance was already pointed out in 1972 in the Anderson report [3], 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 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 must always be invoked (*complete mediation*); ii) the reference validation mechanism must always be tamperproof (*tamperproof*); and iii) the reference validation mechanism 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 security mechanisms of Android. Although not strictly implied by that requirement, we are also interested in determining whether the intended access control policy is correctly specified relative to some goal. Thus, the formal specification of the reference monitor shall be used to establish and prove that the security properties that constitute the policy are satisfied by the modeled behavior of the validation mechanisms.

On the other side, deduction based verification of properties established on a mathematical model do not provide guarantees of the correctness of the code. 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. Yet, we are also interested in providing guarantees concerning actual implementations of the validation mechanisms.

Contribution

We propose a verification approach that consists of: i) using a reasoning framework based on higher-order logic to specify Android reference monitors and to prove properties of these specifications, ii) using the same framework to automatically extract certified functional programs from these specifications, iii) using these functional programs as prototypes on which attacks, discovered from the formal analysis of the specification, can be executed, iv) using lightweight verification techniques, such as model-based testing, to validate that actual implementations of the platform behaves as specified by the abstract reference monitor and be able to check that the discovered attacks can be actually carried out.

In this paper we describe and discuss an extension of the Android security model formalized in [4]. This extension includes, in particular, the modelling of run time requesting/granting of permissions behavior introduced in Android Marshmallow [5] (and further) which was not considered in the previous work. We consider here executions that contemplate error management and a certified monitor of the security model. Additionally, this paper shows that it is posssible use the model to formally state and prove the conditions that must be satisfied to mitigate, or even prevent, the exploitation of vulnerabilities (attacks). We propose the use of a certified algorithm for performing verification activities such as monitoring of actual implementations of the platform and also as a testing oracle. Finally, we make observations to the latest Android security model. This article builds upon and extends a previously published paper [8].

The logical framework we are using is the `Coq` proof assistant [6], which is a software that provides a (dependently typed) functional programming language and a reasoning framework based on higher-order logic to perform proofs of programs. The `Coq` environment supports advanced logical and computational notations, proof search and automation, and modular development of theories and code. It also provides program extraction towards languages like `Ocaml` and `Haskell` for execution of (certified) algorithms [7].

Organization of the paper

The rest of the paper is organized as follows. Section 2 describes the security mechanisms of Android. Section 3 overviews the idealized model of the Android security framework and Section 4 presents some of the formally proved security properties and analyses the impact the change of permission mechanisms has on the security of Android Marshmallow (and further). Section 5 motivates our approach for developing certified security testing and puts forward the use of a certified algorithm for performing verification activities, such as the monitoring of actual (real) implementations of the platform. Section 6 considers related work and finally, Section 7 concludes with a summary of our contributions and directions for future work.

2 Security mechanisms in 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.

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 [10]: i) **activity**, which is essentially a user interface of the application, ii) **service**,

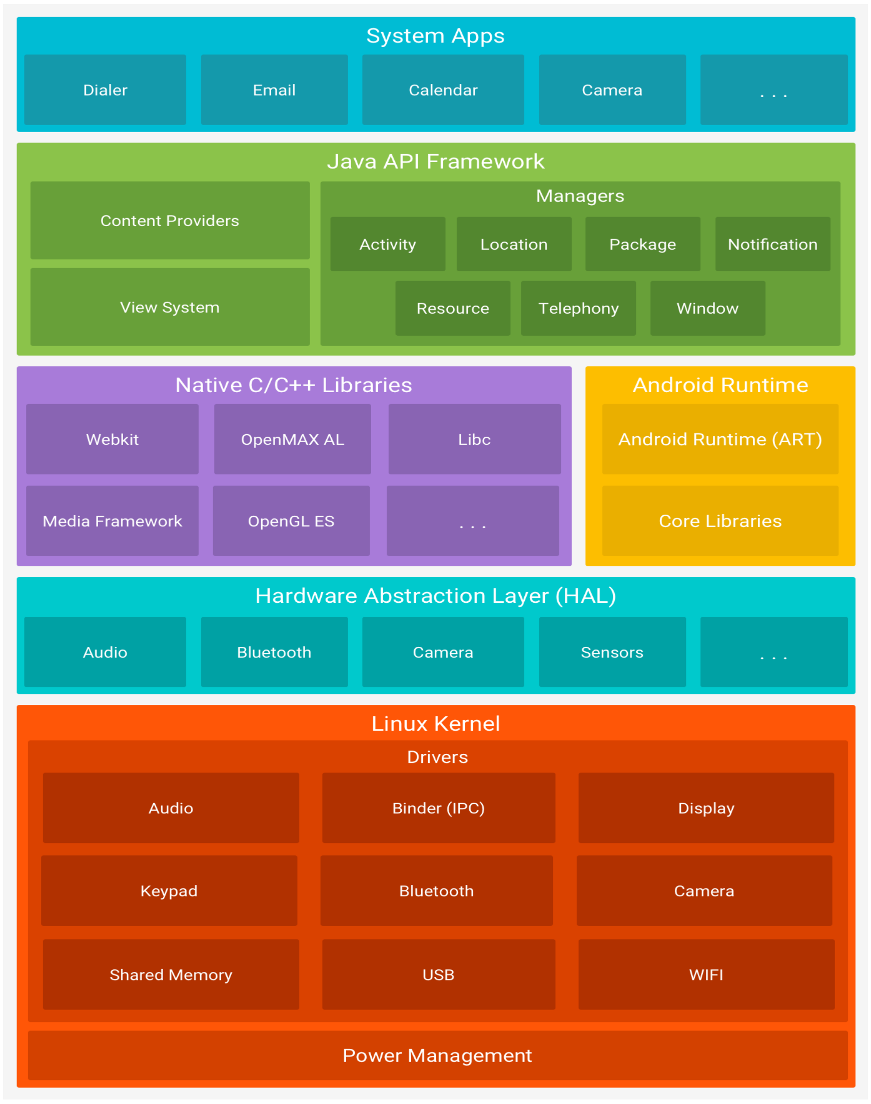


Figure 1: Android's architecture [9].

a component that executes in the background without providing an interface to the user. Any component with the right permissions can start a service or interact with it, iii) **content provider**, a component intended to share information among applications. A component of this type provides an interface through which applications can manage persisted data, 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. 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. 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 [10]. Typically, an intent is used as a broadcast or as a message to interact with activities and services.

2.1 Android's security model

Android implements a least privilege model by ensuring that each application executes in a sandbox. 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 using a user identifier (UID) [11] assigned to each installed application. Interaction among applications is achieved through *Inter Process Communication* (IPC) mechanisms [12]. 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 [13].

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 [14].

2.2 Permissions

Applications usually need to use system resources to execute properly. Since applications run inside sandboxes, this entails the existence of a procedure to grant or not access to those resources (i.e., a reference validation mechanism). Decisions are made by following security policies using a simple notion of permission. Every permission is identified by a name/text, has a protection level and may belong to a permission group. There exist 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. An application declares the set of permissions it needs to acquire further capacities than the default ones. When an action involving permissions is required, the system determines which permissions every application has and either allows or denies its execution.

Depending on the protection level of the permission, the system defines the corresponding access privileges [15]. 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. From version 6 (Marshmallow) dangerous permissions are not granted at installation time, 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, and iv) *Signature/System*, this level is assigned to permissions that regulate the access to critical system resources or services. Additionally, 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. The user of the device can grant and revoke dangerous permission groups (i.e. permission groups which contain permissions of level *Dangerous*) and dangerous ungrouped permissions (permissions which do not belong to a group) for any application on the system at any time. A running application may ask the user to grant it dangerous permission groups and ungrouped permissions, who in turn can accept or decline this request.

If the execution of an action requires for an application to have certain permission the system will first make sure that this holds by means of the following rules: i) the application must declare the permission as used in its manifest (see Section 2.4), ii) if the permission is of level *Normal*, then the application does have it, iii) if the permission is of level *Dangerous* and belongs to a permission group, such group must have been granted to the application, iv) if the permission is of level *Dangerous* but is ungrouped, then it must have been individually granted to the application, v) if the permission is of level *Signature*, then both the involved application and the one that declares it must have been signed with the same certificate, vi) if the permission is of level *Signature/System*, then the involved application must have been signed with either the same certificate as the one who declares it (just like if it was of level *Signature*) or the certificate of the device manufacturer. Otherwise, an error is thrown and the action is not executed.

2.3 Permission delegation

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, 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.

$$
\begin{array}{lll}
\mathsf{InstApps} & ::= & \{\mathsf{AppId}\} \\
\mathsf{PermsGr} & ::= & \{\mathsf{AppId} \times \{\mathsf{PermGroup}\}\} \\
\mathsf{AppPS} & ::= & \{\mathsf{AppId} \times \{\mathsf{Perm}\}\} \\
\mathsf{CompInsRun} & ::= & \{\mathsf{CompInstance}\} \\
\mathsf{OpTy} & ::= & read \mid write \mid rw \\
\mathsf{DelPPerms} & ::= & \{\mathsf{AppId} \times \mathsf{ContProv} \times \mathsf{Uri} \times \mathsf{OpTy}\} \\
\mathsf{DelTPerms} & ::= & \{\mathsf{iComp} \times \mathsf{ContProv} \times \mathsf{Uri} \times \mathsf{OpTy}\} \\
\mathsf{ARVS} & ::= & \{\mathsf{AppId} \times \mathsf{Res} \times \mathsf{Val}\} \\
\mathsf{Intents} & ::= & \{\mathsf{iComp} \times \mathsf{Intent}\} \\
\mathsf{Manifests} & ::= & \{\mathsf{AppId} \times \mathsf{Manifest}\} \\
\mathsf{Certs} & ::= & \{\mathsf{AppId} \times \mathsf{Cert}\} \\
\mathsf{AppDefPS} & ::= & \{\mathsf{AppId} \times \{\mathsf{Perm}\}\} \\
\mathsf{SysImage} & ::= & \{\mathsf{App}\} \\
 & & \\
\mathsf{AndroidST} & ::= & \mathsf{InstApps} \times \mathsf{PermsGr} \times \mathsf{AppPS} \times \mathsf{CompInsRun} \times \\
 & & \mathsf{DelPPerms} \times \mathsf{DelTPerms} \times \mathsf{ARVS} \times \mathsf{Intents} \times \\
 & & \mathsf{Manifests} \times \mathsf{Certs} \times \mathsf{AppDefPS} \times \mathsf{SysImage}
\end{array}
$$

Figure 2: Android state

2.4 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 of their static attributes 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.

3 Formalization of the permission model

The Android security model we have developed has been formalized as an abstract state machine. In this model, states (AndroidST) are modelled as 12-tuples that respectively store data about the applications installed in the device, their permissions and the running instances of components; the formal definition is depicted in Figure 2, the full definition is available in [16]. Note that we use $\{T\}$ to denote the set of elements of type T.

The first component of a state records the identifiers (AppId) of the applications installed by the user. The second and third components of the state keep track, respectively, of the permission groups (PermGroup) and ungrouped permissions (Perm) granted to each application present in the system, both the ones installed by the user and the system applications. The fourth 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[1]. The seventh and eighth 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 that respectively declare application components (set of components, of type Comp, included in the application), the minimum version of the Android SDK required to run the application, 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.

[1]A permanent delegated permission represents that an application has delegated permission to perform either a read, write or read/write operation on the resource identified by an URI of the indicated content provider (ContProv). A temporary delegated permission, in turn, refers to a permission that has been delegated to a component instance.

`install` $a\ m\ c\ lRes$	Install application with id a, whose manifest is m, is signed with certificate c and its resources list is $lRes$.
`uninstall` a	Uninstall the application with id a.
`grant` $p\ a$	Grant the permission p to the application a.
`revoke` $p\ a$	Remove the permission p from the application a.
`grantPermGroup` $g\ a$	Grant the permission group g to the application a.
`revokePermGroup` $g\ a$	Remove the permission group g from the application a.
`hasPermission` $p\ a$	Check if the application a has the permission p.
`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.
`startActivity` $i\ ic$	The running component ic asks to start an activity specified by the intent i.
`startActivityResult` $i\ n\ ic$	The running component ic asks to start an activity specified by the intent i, and expects as return a token n.
`startService` $i\ ic$	The running component ic asks to start a service specified by the intent i.
`sendBroadcast` $i\ ic\ p$	The running component ic sends the intent i as broadcast, specifying that only those components who have the permission p can receive it.
`sendOrderedBroadcast` $i\ ic\ p$	The running component ic sends the intent i as an ordered broadcast, specifying that only those components who have the permission p can receive it.
`sendStickyBroadcast` $i\ ic$	The running component ic sends the intent i as a sticky broadcast.
`resolveIntent` $i\ a$	Application a makes the intent i explicit.
`receiveIntent` $i\ ic\ a$	Application a receives the intent i, sent by the running component ic.
`stop` ic	The running component ic finishes its execution.
`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.
`revokeDel` $ic\ cp\ u\ pt$	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.

Table 1: Actions

A notion of *valid state*, that captures several well-formedness conditions, is formally defined as a predicate *validState* on the elements of type AndroidST that requires for several properties to be satisfied. For instance, one of those properties states that all the running instances belong to a unique component, which in turn must be part of an installed application. Other properties establish, for instance, the uniqueness of application, component, and resource identifiers. There are also properties that involve permissions on a system state, namely, that all the parts involved in active permission delegations must be installed in the system. The definition of *validState* is provided in Appendix A.

3.1 Specification of the reference monitor

Our model considers a representative set of actions to install and uninstall applications, grant and revoke permissions and permission groups, ask whether a component has certain permission, 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 1. The system access control policy is enforced through the execution of these actions, which provide coverage to the different functionalities of the Android security model[2].

Given a state in our formalism, the execution of an operation in the Android system (e.g., the installation of a new application) is represented as a transition to a new state, along with a response (of type Response) indicating whether the execution was successful or not. The behavior of an action a (of type Action) is formally described by giving a precondition (Pre) and a postcondition ($Post$), which represent the requirements enforced on a system state to enable the execution of a and the effect produced after this execution takes

[2]We consider here twice the number of operations (associated with the security model) contemplated in [16].

$installed(ap, s)$	is satisfied if application ap is installed in state s.
$isNative(ap, s)$	holds if application ap belongs to the set SysImage of state s.
$inApp(c, ap, s)$	holds only when the component c belongs to the installed application ap in state s.
$appCert(ap, c, s)$	is satisfied if application ap has a certificate c in state s.
$isActivity(c)$	holds if component c is an activity.
$isCProvider(c)$	is satisfied if component c is a content provider.
$canBeStarted(c)$	holds if component c can be accessed (if c is a content provider) or started by third parties.
$running(ic, c, s)$	is satisfied if ic is an instance of component c running in state s.
$isIntBReceiver(i)$	holds only when intent i is destined to a broadcast receiver.
$refersTo(i, c, s)$	is satisfied if intent i is directed to component c in state s.
$intentReg(i, ic, s)$	holds if i is an intent, registered in component Intent of state s, sent by running instance ic.
$protected(i, p, s)$	is satisfied if intent i is protected with permission p in state s.

Table 2: Helper predicates

place. Additionally, we define a relation *ErrorMsg* such that 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.

3.2 One-step execution

We represent the execution of an action with the relation $\hookrightarrow$ (one-step execution), defined by the following two rules:

$$Exec_{ok}\,\frac{validState(s) \qquad Pre(s,a) \qquad Post(s,a,s')}{s \overset{a/ok}{\hookrightarrow} s'}$$

$$Exec_{err}\,\frac{validState(s) \qquad ErrorMsg(s,a,ec)}{s \overset{a/error\ ec}{\hookrightarrow} s}$$

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 *For any* a : Action*,* $s\ s'$: AndroidST *and* r : Response*, if* $s \overset{a/r}{\hookrightarrow} s'$ *holds, then* $validState(s')$ *also holds.*

The property is proved by case analysis on a, for each condition in *validState*, using several auxiliary lemmas [16].

System state invariants, such as state validity, are useful to analyze other relevant properties of the model. In particular, the results presented in [4] and validated in this work to hold for Android Marshmallow were obtained from valid states of the system.

The full formalization of the idealized permission model of Android, which extends the formal specification presented in [4], may be obtained from [16] and verified using the Coq proof assistant.

4 Analysis of security policies

We have stated and proved several security properties that formally establish that the specified reference monitor provides protection against unauthorized access to sensitive resources of a device running Android. 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" [10]. Using our specification we have proved several lemmas, as Lemma 2 below, which were aimed at showing the compliance with this principle when a running instance of an application component creates another component instance, reads/writes a content provider or delegates/revokes a permission.

The predicates used to define the lemmas discussed in this section are presented and described in Table 2. The full formal definition of the lemmas can be found in [16], along with their proofs.

Lemma 2 *For any* s : $\mathsf{AndroidST}$*,* ap : AppId*,* $c\ c'$: Comp*,* ic : iComp*,* i : Intent*, if* $validState(s)$ *and* $inApp(c, ap, s)$ *and* $running(ic, c, s)$ *and* $isActivity(c')$ *and* $inApp(c', ap, s)$ *and* $refersTo(i, c', s)$*, then* $Pre(s, \texttt{startActivity}\ i\ ic)$ *holds.*

If component c *and activity* c' *belongs to the same application, then* c *can start* c' *through intent* i *directed to this activity.*

However, an implementation of the Android system that respects the access control policy is nevertheless vulnerable to a kind of attack that exploits the mechanisms of `Intents` (see [17, 18] for further details). In particular, it has been shown that the system is vulnerable to unauthorized monitoring of information (*eavesdropping*), unintended inter-application communication (*intent spoofing*) and privilege escalation (*permision collusion*) through the (deceived) installation of collaborating malware by the device user. The common idea behind these attacks is the abuse of the *intent mechanism* to obtain unauthorized access to private information.

The fact is that those attacks can be prevented if certain additional controls are considered, and enforced. Using our model, and as a complementary contribution, we precisely state the conditions that would help preventing the exploitation on the identified vulnerabilities and prove that under those hypotheses the attacks cannot be carried out. We illustrate below our approach for two attacks, one implementing *eavesdropping* (section 4.1) and the another one *intent spoofing* (section 4.2). In section 4.3 we briefly discuss how the management of permissions has been affected by the design changes introduced in Android Marshmallow (and further).

4.1 Eavesdropping

If there is a malicious application running on an Android device, each time sensitive information is sent using a public broadcast intent, the device becomes vulnerable to the **eavesdropping attack**. However, if all broadcast intents were protected with a *Signature* or *Signature/System* permission, it can be ensured that only applications signed with the transmitter application's certificate will be candidates for the intent reception [18]. In such conditions the eavesdropping attack can be prevented.

We establish that in a scenario where an intent protected with a *Signature* or *Signature/System* permission is sent using the `sendBroadcast` or `sendOrderedBroadcast` operation, no application with a certificate different than the one from the transmitter application will be able to receive it. As a consequence, under these conditions we can ensure the absence of the eavesdropping attack.

Lemma 3 *For any* s : $\mathsf{AndroidST}$*,* ic : iComp*,* i : Intent*,* c : Comp*,* $ap\ ap'$: AppId*,* $cert$: Cert*, if* $validState(s)$ *and* $running(ic, c, s)$ *and* $\neg isCProvider(c)$ *and* $inApp(c, ap, s)$ *and* $intentReg(i, ic, s)$ *and* $isIntBReceiver(i)$ *and* $(protected(i, \mathsf{signature}, s)$ *or* $protected(i, \mathsf{signature/system}, s))$ *and* $installed(ap', s)$ *and* $\neg isNative(ap', s)$ *and* $appCert(ap, cert, s)$ *and* $\neg appCert(ap', cert, s)$*, then* $\neg Pre(s, \texttt{receiveIntent}(i, ic, ap'))$ *holds.*

If a component c *that belongs to an application* ap *sends a broadcast intent protected with a Signature or Signature/System permission, then if a non native application* ap' *does not have the same certificate as* ap *it will not be able to receive it.*

4.2 Intent spoofing

In Android, whether an application can be started by third parties depends on the `exported` attribute and the existence of `<intent-filter>` elements in its manifest. Application misconfiguration generally happens when the `exported` attribute is not present, pretending that no external invocations are allowed, but if `<intent-filter>` elements are used, the default value of the `exported` attribute is true. In case an application was not intended to be initiated by other applications but was misconfigured, it could be victim of the **intent spoofing attack** explained in [17].

A simple way of checking if an application is not vulnerable to an intent spoofing attack is statically verifying its manifest. Thus, if the `exported` attribute is false or if it is absent and no `<intent-filter>` element is declared, as stated in Lemma 4, the application cannot be started by external applications. Therefore the intent spoofing attack can be avoided.

Lemma 4 *For any* s : $\mathsf{AndroidST}, ap\ ap'$: $\mathsf{AppId}, c\ c'$: Comp, ic : iComp, i : Intent, *if* $validState(s)$ *and* $inApp(c, ap, s)$ *and* $inApp(c', ap', s)$ *and* $ap \neq ap'$ *and* $\neg canBeStarted(c)$ *and* $running(ic, c', s)$ *and* $\neg isCProvider(c')$ *and* $refersTo(i, c, s)$*, then* $\neg Pre(s, \texttt{receiveIntent}(i, ic, ap'))$ *holds.*

If a component c cannot be started by other applications, then the application that contains it cannot receive an intent directed to c.

In Lemmas 3 and 4 we have also shown that in the presence of vulnerabilities we can use the model to formally state and prove the conditions that must be satisfied to mitigate, or even prevent, the exploitation of those vulnerabilities.

4.3 Observations to the security of Android Marshmallow (and higher)

On all versions of Android an application must declare both the normal and the dangerous permissions it needs in its manifest. However, the effect of that declaration is different depending on the system version and the application's target SDK level [5]. In particular, if a device is running Android Marshmallow (or higher) 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.

No formal treatment of the issues we proceed to discuss below has been carried out yet.

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. There are basically two ways of delegating a permission: through a pending intent and through an URI permission.

If the permission was delegated using a pending intent, the following situation may happen: an application sends a pending intent that enables an action protected by a particular permission, later on that permission is revoked but the application that has received the pending intent can still use the delegated permission. For example, imagine an application where the user can schedule the sending time for a text message and this is performed by sending a pending intent to one of its components. When the user sets a time for a text message, a pending intent is created. If the application has permissions to send a text message so will the pending intent. Even if the user revokes that permission before the established time the text message will be sent because the pending intent has a permission to do so.

When the permission is granted through an URI permission the problem is that if the application that has granted the permission no longer has it, then it will not be able to revoke it. As a consequence, the receiving application could acquire the permission permanently. For example, suppose a device that has an email application with no access to the image gallery and an application for image editing with no access to the email but that allows the user to send the images by email using the operation `grantUriPermission`. When the user wants to send one or more pictures through his/her email, the email application gains permission to access the gallery. This permission can be used until the execution of the `revokeUriPermission` operation by some application. If the user revokes the permission to access the gallery to the image editing application, then it will not be able to execute the `revokeUriPermission` operation for the delegated permission. As a consequence, the email application may gain access to the gallery for an indefinite period of time.

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 to write and access to 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, they should not have[3]. This behavior can induce *privilege escalation* and also shows a lack of transparency with the users, who should accept that their applications have permissions to perform operations available on their sensitive data.

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

In Android, all dangerous system permissions belong to permission groups. The way that the system handles this type of permission depends on the version of Android and the SDK version running on the device.

On a device running Android Marshmallow (or higher) where the SDK version is 23 (or higher) when a permission is needed the system checks if any other permission of the same permission group was granted previously [19]. In that case, the permission is granted automatically. If no permission belonging to the permission group was granted previously, then the system ask the user to grant permission to the permission group. 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 to write and access to the user's profile.

If a device is running Android Lollipop (or lower) or if the SDK version is 22 (or lower), the system asks the user to grant all the dangerous permissions at installation time [19]. At this time the user is informed if the application requires access to a permission group.

The main difference between Android Marshmallow and previous versions takes place when the user wants to check which permissions were granted to a particular application. For example, *Facebook* requires permission to access to the text messages. Figures 3 and 4 show what the user sees if (s)he checks the permissions that were granted in Android Marshmallow and Android Lollipop respectively. Clearly in Figure 3, the user can see that the application only has permission to read the text messages, whereas in Figure 4 the user will not know which specific permission was granted to the application.

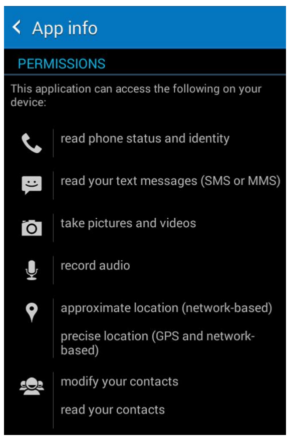

App info
PERMISSIONS
This application can access the following on your device:
read phone status and identity
read your text messages (SMS or MMS)
take pictures and videos
record audio
approximate location (network-based)
precise location (GPS and network-based)
modify your contacts
read your contacts

Figure 3: Android Lollipop Messages access

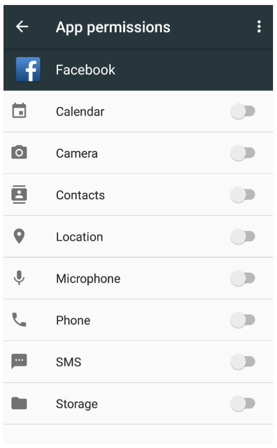


Figure 4: Android Marshmallow Messages access

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 the user granting that permission. For example, a flashlight application can access the Internet and the user can not prevent 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 that has been granted access to images but had no 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.

Even if automatic Internet access was introduced in Android Marshmallow, it affects all previous versions. The main drawback of this change is that the user can not know whether an application is using Internet access. For example, Figure 5 shows a flashlight application available in Google Play with 10 millions downloads approximately. In Android Kit Kat the only permission requested is access to the camera, considering that flashlight application uses the camera flash, this permission is reasonable. However, Figure 6 shows the application manifest where Internet access is listed as necessary. Thus, when this application runs on Marshmallow it will have access to the Internet and the user will not be aware of that.

Adaptation of malicious applications The goal of changing the permission system is to make the user more conscious of the permissions that an application is requesting. As the user has to grant each permission while using the application, (s)he can reflect about the reason why that permission is being requested.

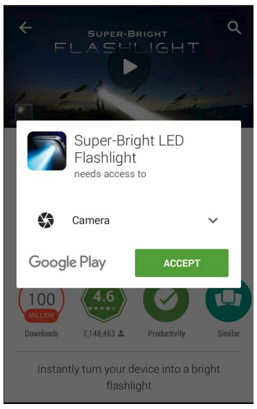
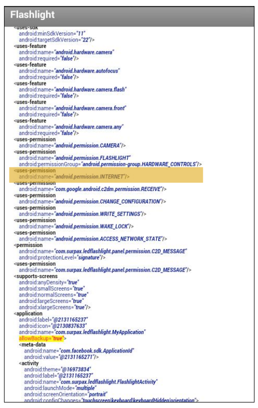


Figure 5: Flashlight application Permissions



```
Flashlight
  <uses-sdk
    android:minSdkVersion="11"
    android:targetSdkVersion="22"/>
  <uses-feature
    android:name="android.hardware.camera"
    android:required="false"/>
  <uses-feature
    android:name="android.hardware.autofocus"
    android:required="false"/>
  <uses-feature
    android:name="android.hardware.camera.flash"
    android:required="false"/>
  <uses-feature
    android:name="android.hardware.camera.front"
    android:required="false"/>
  <uses-feature
    android:name="android.hardware.camera.any"
    android:required="false"/>
  <uses-permission
    android:name="android.permission.CAMERA"/>
  <uses-permission
    android:name="android.permission.FLASHLIGHT"
    android:permissionGroup="android.permission-group.HARDWARE_CONTROLS"/>
  <uses-permission
    android:name="android.permission.INTERNET"/>
  <uses-permission
    android:name="com.google.android.c2dm.permission.RECEIVE"/>
  <uses-permission
    android:name="android.permission.CHANGE_CONFIGURATION"/>
  <uses-permission
    android:name="android.permission.WRITE_SETTINGS"/>
  <uses-permission
    android:name="android.permission.WAKE_LOCK"/>
  <uses-permission
    android:name="android.permission.ACCESS_NETWORK_STATE"/>
  <permission
    android:name="com.surpax.ledflashlight.panel.permission.C2D_MESSAGE"
    android:protectionLevel="signature"/>
  <uses-permission
    android:name="com.surpax.ledflashlight.panel.permission.C2D_MESSAGE"/>
  <supports-screens
    android:anyDensity="true"
    android:smallScreens="true"
    android:normalScreens="true"
    android:largeScreens="true"
    android:xlargeScreens="true"/>
  <application
    android:label="@2131165237"
    android:icon="@2130837633"
    android:name="com.surpax.ledflashlight.MyApplication"
    allowBackup="true">
    <meta-data
      android:name="com.facebook.sdk.ApplicationId"
      android:value="@2131165271"/>
    <activity
      android:theme="@16973834"
      android:label="@2131165237"
      android:name="com.surpax.ledflashlight.FlashlightActivity"
      android:launchMode="multiple"
      android:screenOrientation="portrait"
      android:configChanges="touchscreen|keyboard|keyboardHidden|orientation">
```

Figure 6: Flashlight application Manifest

However, if the device is running Android Lollipop (or lower) or if the SDK version is 22 (or lower), the permissions are still requested at installation time.

Even if the developer can not change the version of Android running in the device, (s)he can set the manifest's element `<uses-sdk>` and ensure that when the device is running Android Lollipop (or lower) permissions will be requested at installation time. With this, malicious applications can adapt to Android Marshmallow easily. One example of a malicious application that has adapted to Android's changes is *Bankosy* [20], which is a financial Trojan horse that makes calls, reads and deletes text messages, deletes data and enables silent mode. The application steals users' fingerprints and uses them to make transactions in the bank accounts of the victims.

Missing permissions Android Marshmallow makes it possible for users to revoke permissions from any application at any time. A developer should test his/her application to verify that it behaves properly when a needed permission is lacking. 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.

5 Certified security testing

Using the programming language of `Coq` we have developed an executable (functional) specification of the reference validation mechanism. This ultimately amounts to the definition of the functions that implement the execution of the actions specified in the reference monitor. We have proved that those functions conform to the axiomatic specification of action execution as specified in the model. Additionally, and using the program extraction mechanism provided by `Coq`, we have derived a certified `Haskell` prototype of the reference validation mechanism that we call `CertAndroidSec`. In Appendix B, we provide listings of part of the `Haskell` code that has been automatically generated using `Coq`. The prototype and its proof of correctness are available in [16].

Thus, in this setting the control access policy specified by the permission model of Android is enforced by the combined execution of the actions. The behavior of the security mechanisms during the execution of a session of the device is represented by the sequence of system states (the trace of execution) obtained from executing the sequence of actions starting in an (initial) system state.

We plan to use `CertAndroidSec` for performing verification activities such as monitoring of actual implementations of the platform and also as a testing oracle. We briefly comment and motivate these techniques in what follows.

5.1 Model-Based Security Testing

One important goal of our work is to help increase the level of reliability on the security of the Android platform by providing certified guarantees that the specified security mechanisms effectively allow to enforce the expected security policies. 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. In what follows we discuss the techniques we have begun experimenting with in order to test the security mechanisms of Android by generating test cases from the `Coq` axiomatic specification.

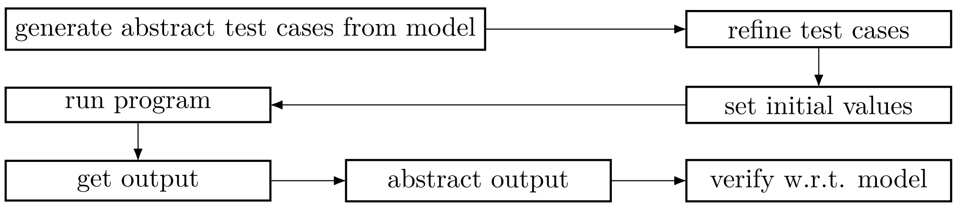


Figure 7: Proposed MBT process

Model-based testing (MBT) is a lively research area whose main goal is to use models of implementations as test case and oracle sources [21]. One of the possible high-level MBT processes is depicted in Figure 7. MBT methods based on such a process generate *abstract test cases* by performing different static analyzes of a (formal) model (of a given program). This abstract test cases are later refined to the level of the program; the program is run on these refined test cases; the outputs are collected and abstracted away to the level of the model; and, finally, the model, the abstract test cases and the abstracted outputs are used to decide whether the program has errors or not.

In particular, for the generation of abstract test cases, we are working in adapting and applying a MBT method known as the Test Template Framework (TTF) [22]. The TTF is a method for the first step of Figure 7, that is, abstract test case generation. The TTF was originally thought as a MBT method for `Z` specifications [23] but it can be easily adapted to our `Coq` axiomatic specification. In effect, our model is a state machine featuring complex state and input variables and state transitions given by their pre- and post-conditions, much as `Z` specifications. Furthermore, pre- and post-conditions of our `Coq` axiomatic specification make use of non-trivial library operations on lists, similarly as `Z` specifications rest on library operations on sets. Then, we will give some details of how we adapted the TTF to our `Coq` specification. For the remaining steps of Figure 7 we will use the OVAL technology as explained below.

The Test Template Framework In the TTF the specification of the system under test (SUT) takes the form of a state machine (not necessarily a finite one). The TTF defines a methodology for generating abstract test cases for each state transition of the state machine. Then, let $A(\vec{i}, \vec{s}, \vec{s}', \vec{o})$ be such an state transition depending on a vectors $\vec{i}$, $\vec{s}$, $\vec{s}'$, and $\vec{o}$ of, respectively, input, before-state, after-state and output variables[4]. Such an state transition is specified by its pre- and post-conditions: $A(i, s, s', o) \equiv Pre_A(i, s) \implies Post_A(i, s, s', o)$, where Pre_A and $Post_A$ are two predicates.

The first step of the test case generation method of the TTF is to define the *valid input space* of each state transition, which is the set of all possible values for i and s satisfying the precondition. Formally, $A^{VIS} \equiv \{(i, s) \mid Pre_A(i, s)\}$.

As the second step, the TTF states that the VIS must be partitioned by a technique called *recursive domain partition* (RDP). RDP is the core of the TTF. RDP starts by applying a *testing tactic* to the VIS thus generating a partition of the VIS in the form of a collection of *test specifications*. Each testing tactic describes how a test specification can be partitioned in order to obtain new and more demanding test specifications (see an example in Section 5.2). A test specification is simply a set of test cases described by a predicate depending of the input and before-state variables. These predicates are called *characteristic predicates*. Then, each testing tactic is defined by the collection of characteristic predicates that it generates.

Hence, if testing tactic T_a applied to A^{VIS} generates the (characteristic) predicates $P^a_1, \dots, P^a_{n_a}$, the TTF states that A has to be tested with test cases taken from the following test specifications:

$$\begin{gathered}
A^{T_a}_1 \equiv \{A^{VIS} \mid P^a_1(i, s)\} \\
A^{T_a}_2 \equiv \{A^{VIS} \mid P^a_2(i, s)\} \\
\dots \\
A^{T_a}_{n_a} \equiv \{A^{VIS} \mid P^a_{n_a}(i, s)\}
\end{gathered}$$

RDP continues by picking one or more of the $A^{T_a}_i$ and applying to them one or more testing tactics. For

[4]For the sake of simplicity, from now on, we will avoid the vector notation.

example, testing tactic T_b can be applied to $A_2^{T_a}$ thus generating the following test specifications:

$$\begin{gathered}
A_1^{T_b} \equiv \{A_2^{T_a} \mid P_1^b(i,s)\} \\
A_2^{T_b} \equiv \{A_2^{T_a} \mid P_2^b(i,s)\} \\
\dots \\
A_{n_b}^{T_b} \equiv \{A_2^{T_a} \mid P_{n_b}^b(i,s)\}
\end{gathered}$$

This process continues until engineers think they have a good coverage of the specification. The test specifications generated in this way can be accommodated in a so-called *testing tree*. The testing tree has the VIS as its root node, the test specifications generated after applying the first testing tactic in the first level; the test specifications generated after applying more testing tactics to the first level, in the second level; and so forth. Note that the more testing tactics are applied, the more precise are the leaves of the testing tree. In effect, observe that the characteristic predicates of each testing tactic are conjoined to each other thus turning the characteristic predicates of leaves into conjunctions of several constraints. For example, if $A_2^{T_a}$ is expanded inside $A_2^{T_b}$ we have:

$$A_2^{T_b} \equiv \{A^{VIS} \mid P_2^a(i,s) \land P_2^b(i,s)\}$$

The third step in the TTF is to prune the unsatisfiable leaves off the testing tree. If the TTF is mechanically applied it tends to generate many unsatisfiable leaves as contradictory constraints can be conjoined in the same test specification (see an example in Section 5.2). Depending on the theory used for the specification of the SUT, pruning can be an intractable problem but in general there exist algorithms that perform well in practice [24].

The fourth and final step in the TTF is to generate an abstract test case from each surviving leaf of the testing tree. In this context, abstract test cases take the form of closed propositional sentences binding state and input variables to concrete values. More precisely, an abstract test case is a conjunction of equalities between variables and constant terms. For example, if $A_3^{T_b}$ is a satisfiable test specification, its corresponding test case is described as follows:

$$A_3^{TC} \equiv \{A_3^{T_b} \mid i = c_i \land s = c_s\}$$

where c_i and c_s are two constant (vector) terms (see an example in Section 5.2). Observe that such a test case must verify all the characteristic predicates all the way up in the testing tree to the root node. Then, essentially, such a test case sets the initial state and input values from which the state transition to be tested must be executed. In the TTF, test case generation is the dual problem of leaf pruning as both amount to solve the same satisfiability problem (i.e. determining whether test specifications are satisfiable or not) [25].

One of the advantages of the TTF is that it encourages that all the artifacts involved in the testing process (e.g. state transitions and test specifications) can be expressed in the same notation.

Using the OVAL technology for test case refinement, execution and abstraction According to Figure 7, abstract test cases must refined into the implementation platform so they can be executed on the SUT. In our case, the implementation platform is the Android operating system and more specifically its security mechanism. For test case refinement, execution and output abstraction we propose to use the OVAL technology [26]. OVAL is an information security community effort to standardize how to assess and report upon the machine state of computer systems. OVAL is an XML-based language that allows to express specific machine states such as vulnerabilities, configuration settings and patch states. Real analysis is performed by OVAL interpreters such as Ovaldi [27], XOvaldi [28] and Xovaldi4Android [29, 30].

One of the key components in the OVAL framework is the OVAL language. This language allows for the description of three phases of the security assessment process: representing configuration information of the SUT; checking the SUT for the presence of the specified vulnerability, configuration, patch state, etc. (in the form of machine states); and reporting the results of the assessment. In our context we will use the OVAL language for testing whether the Android platform correctly implements the `Coq` axiomatic specification, where the test cases are generated according to the TTF. Note the link between the TTF abstract test cases and the OVAL language: TTF test cases define values for state variables describing the state of Android's permission system whereas the OVAL language lets users to set Android's configuration states. Hence, all that needs to be done is to represent `Coq` states in the OVAL language. OVAL can also be used to lift the results of executing a test case on the platform to the `Coq` level. In effect, the OVAL language includes a report language providing detailed information about the assertions and states that have been analyzed, and detailed results of the evaluation. Hence, the results of executing a test case on the Android platform can be mapped back to the `Coq` specification by means of the OVAL's report language.

Using the prototype for conformance checking At this point we have the abstract test cases and the result of mapping back the outputs collected by the OVAL report language. The final step in Figure 7 is to check whether each abstract output conforms with the corresponding abstract test case. For this purpose we use the `CertAndroidSec` prototype. Recall that `CertAndroidSec` is a certified prototype of the `Coq` specification as it has been automatically extracted from the `Coq` axiomatic specification. Furthermore, `CertAndroidSec` operates at the same abstraction level of the abstract test cases thus they become inputs to the prototype.

So, given an abstract test case t, let t_{oval} be the representation of t in the OVAL language and let O_a denote the result of abstracting the output produced by running t_{oval} on the Android platform. Moreover, let O_p denote the output produced by executing `CertAndroidSec` on t. At this point we can perform validation/verification procedures based in the similarities and/or differences between O_a and O_p. Note that these procedures could also be extracted as certified algorithms. More formally, Android shows the presence of a security issue if and only if $O_a \not\sim O_p$, where $\sim$ is an equivalence relation between Android states and the prototype states. Note that a simple check such as $O_a \neq O_p$ is not enough because Android is free to implement the model in different ways. For example, when installing a new application, Android may add it at the end of the list of installed applications whereas the prototype may add it at the front. In this case $O_a \neq O_p$ will hold but it does not mean Android is flawed. One of the advantages of using `Coq` is that the equivalence relation ($\sim$) can also be implemented by a certified `Coq` function. That is, the equivalence relation can be specified in `Coq`, proved to be correct in `Coq`, and a certified `Haskell` prototype can be generated in `Coq`.

The fact that $O_a \neq O_p$ cannot be used as the conformance criterion is one of the reasons to not include expected outputs in the test cases generated from the `Coq` axiomatic specification.

5.2 Example of the Application of the TTF

Consider the `install` operation of the `Coq` axiomatic specification (see Table 1). The first step in generating test cases from the specification consists in defining the VIS of the operation. Then, in this case we have:

$$\texttt{install}^{VIS} = \{s : \mathsf{AndroidST}, ap : \mathsf{AppId}\}$$

The second step is to apply a testing tactic to the VIS. One typical testing tactic is Disjunctive Normal Form (DNF). This tactic writes the specification of the operation into DNF and then generates a partition with as many test specifications as terms in the DNF. In this case, each test specification is characterized by the precondition of each term of the DNF. In order to simplify the presentation we will consider that the DNF of the operation is the disjunction between $Exec_{ok}$ and $Exec_{err}$ (Section 3) applied to `install`. Then, we have:

$$\texttt{install}_1^{DNF} = \{\texttt{install}^{VIS} \mid \mathit{validState}(s) \land \mathit{Pre}(s, \texttt{install})\}$$
$$\texttt{install}_2^{DNF} = \{\texttt{install}^{VIS} \mid \mathit{validState}(s) \land \mathit{ErrorMsg}(s, \texttt{install}, ec)\}$$

As a second tactic we will apply Enumerated Types (ET). This tactic applies to VIS variables of some enumerated type. Hence, if v is a variable whose type is $E = \{c_1, \dots, c_k\}$, the ET tactic generates test specifications characterized by the following predicates: $v = c_1, \dots, v = c_k$. In $\texttt{install}_2^{DNF}$ we have that ec is of type *ErrorCode* (listing all the 28 possible errors of the system [16]). Therefore, we can apply ET to $\texttt{install}_2^{DNF}$ resulting in:

$$\texttt{install}_1^{ET} = \{\texttt{install}_2^{DNF} \mid ec = \mathit{app\_already\_installed}\}$$
$$\dots\dots\dots$$
$$\texttt{install}_{28}^{ET} = \{\texttt{install}_2^{DNF} \mid ec = \mathit{CProvider\_not\_grantable}\}$$

In this way, for instance $\texttt{install}_1^{ET}$, will test the application from a valid state where the application that is going to be installed is already installed. Also note that many of the $\texttt{install}_i^{ET}$ will be unsatisfiable for `install` as many of these errors will not be raised by this operation. Unsatisfiable test conditions must be pruned from the testing tree and not further considered for partitioning.

Once the developer is done with partitioning and all unsatisfiable test conditions has been pruned from the testing tree, (s)he must produce an element from each of the surviving test specifications. These elements are the abstract test cases. For instance, the following is a test case for EC_1:

$$\texttt{install}_1^{TC} = \{\texttt{install}_1^{ET} \mid s.apps = [\mathit{Chrome}] \land ap = \mathit{Chrome}\}$$

5.3 Monitor error detection

As suggested above, `CertAndroidSec` might be used to compare the results of executing an action on an analyzed Android platform and executing that same action on the certified validation mechanism. This would give us the possibility of monitoring the actions performed in a real system and evaluating if certain properties hold on it. The monitor should be able at any given time to interpret the system's state and constructing an equivalent state in the formal model. In addition to that, the monitor must be aware of the internal representation of the error codes at implementation level and it must be able to intercept the system calls able to mutate the state of the certified implementation and replicate them.

The activity of the monitor is depicted in Figure 8, and consists of, when it detects the call to a function that represents the execution of a modeled action (2), extracting the state prior to its execution (1), replicating the performed action on it (3), extracting the state and the error code reached after the operation in the real platform (4), and finally compare them with the results obtained by the execution of the action in the extracted state (5).

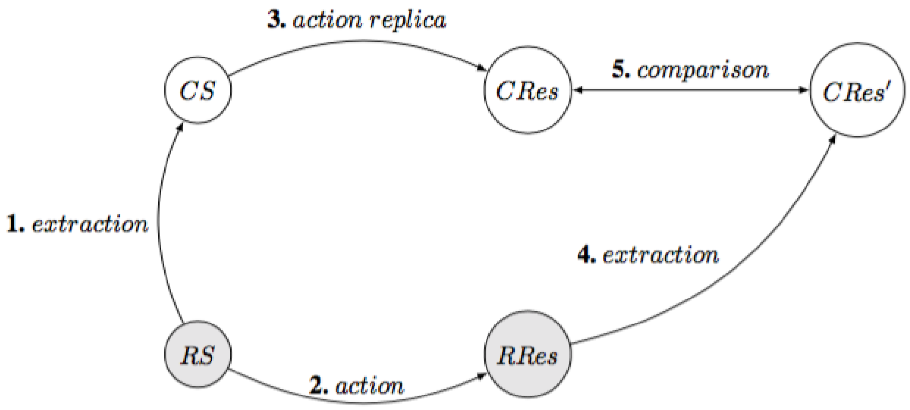


Figure 8: Monitor's activty.

These steps can be executed successively along the system's lifecycle, resulting in a program that monitors the truth of interesting properties in real time. Which properties to check and what to do when they don't hold is up to the monitor's configurator. For example, given a function $\mathit{validState\_bool} : \mathsf{AndroidST} \to \mathit{bool}$ such that $\forall s, \mathit{validState}(s) \iff \mathit{validState\_bool}(s) = \mathit{true}$, it is feasible to develop a monitor that checks on runtime if the successive states a device goes through are valid; and act consequently in case they are not.

Nevertheless, the certified implementation developed is just one of all the possible correct implementations: a different result on a real platform doesn't necessarily imply a specification violation. Such a situation can be witnessed, for instance, in the `install` *app m c lRes* action; its specification requires —among other things— that if its precondition is met, then *app* is part of the list of installed applications in the reached state s'. The certified implementation prepends *app* to the list of installed applications, but a system that chooses to append it to the end instead, is equally correct. To bypass this problem, the monitor's user may define an equivalence function between two systems and results[5] gotten after performing the same action on the same initial state: $\mathit{eq\_res} : \mathsf{Action} \to \mathsf{AndroidST} \to \mathit{Result} \to \mathit{Result} \to \mathit{bool}$ which will be noted $\sim$, and we will say that given two results r_1 and r_2, an action a and a state s, $r_1 \underset{a,s}{\sim} r_2 \iff \mathit{eq\_res}\ a\ s\ r1\ r2 = \mathit{true}$. The idea behind the equivalence function is to allow the reuse of the proof of *step*'s soundness [16], by adjusting the technical differences between the certified implementation and the real platform that arise from the specification's non-determinism. Naturally, the more these two diverge, the less can be reused from the original proof, and the effort invested in coding the equivalence function will be greater.

6 Related work

Existing smartphone security surveys review the state of the art regarding popular mobile OS platforms [31, 32]. In particular, La Polla et al. [32] surveyed smartphone security threats and their solutions, but has very limited coverage of Android. In the survey papers [33, 34, 35] the authors focus on the Android platform, but they only perform an informal analysis of the security model. Also, several other analyses have recently been carried out concerning the security of the Android system. Some of them [36, 37] point

[5]The type *Result* includes a state and information on an execution that allows to distinguish if it was successful or produced an error.

out the rigidity of the permission system regarding the installation of new applications in the device. Other studies [38, 39, 40] 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 [39, 41] 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 works, however, pay attention to the formal aspects of the permission enforcing framework. In particular, Shin *et al.* [42, 43] 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, for instance, 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. They also do not consider novel aspects of the Android security model, such as managing runtime permissions.

The results presented in this paper update and extend the ones reported in [4]. We have already extended the (formal) model presented in [4] so as to consider, in particular, the run time requesting/granting of permissions behavior introduced in Android Marshmallow (and further). We consider here executions that contemplate error management and a certified monitor of the security model. Additionally, this paper shows that it is posssible use the model to formally state and prove the conditions that must be satisfied to mitigate, or even prevent, the exploitation of vulnerabilities (attacks). We propose here the use of a certified algorithm for performing verification activities such as monitoring of actual implementations of the platform and also as a testing oracle. Finally, we make observations to the latest Android security model. This article is an extended version of the one published in [8].

MBT is about guiding the testing process from a model of the system under test (SUT). In particular, test case generation in MBT is performed by analyzing a model of the SUT. As a test case generarion method the TTF has been successfully applied when the SUT is specified in the `Z` notation [25]. Despite the fact that test case generation in the context of MBT is a very active research area, not so much activity has been seen with `Coq` specifications. Many MBT proposals for `Coq` specifications are based on random test case generation. Maybe the most used tool combining MBT with `Coq` is `QuickChick` [44]. This tool is a randomized property-based testing plugin for `Coq`. If an engineer writes some testing code to test the SUT (s)he can verify that this code is testing the intended property by using QuickChick. For instance, Dubois et al. [45] have applied QuickChick to a problem of enumerative combinatorics. Very recently Becker et al. [46] reported the application of a MBT method based on random testing to a `Coq` model of the NOVA micro hypervisor. Tuerk [47], like us, uses the `Coq` code extraction feature to apply MBT in a security context, but we could not find a detailed description of the results presented in that work. Although random test has its merits, we are more inclined towards a more formal test case generation strategy in line with the work performed over `Isabelle/HOL` [48]. On the other end, MBT has been applied to the Android platform. Jing et al. [49] use the Alloy modeling language for testing Android applications. In [50] the authors make a general evaluation of the applicability of MBT methods in the context of mobile applications developed for the Android platform.

7 Conclusions and future work

In this paper we have described the challenges we are facing when attempting to apply formal methods to perform analysis and verification of the security mechanisms defined by Android to enforce permission-based access control policies. The idealized model we have developed allows us to perform machine-assisted reasoning to provide, on the one side, certified guarantees that the claimed access control policy is effectively enforced by those mechanisms. On the other side, we have also shown that in the presence of vulnerabilities we can use the model to formally state and prove the conditions that must be satisfied to mitigate, or even prevent, the exploitation of those vulnerabilities. We have also motivated the use of certified extracted algorithms to implement lightweight model-based testing and for performing verification activities such as monitoring of actual implementations of the platform. Finally, we have formulated observations to the current security model of Andoroid.

We are currently working on refining the certified testing strategy we have briefly motivated in the paper. Although we find random testing quite appealing, we are inclined towards a more formal test case generation strategy in line with the work performed over `Isabelle/HOL` [48]. Furthermore, we think that for testing the security of the Android platform, most of the testing code can be automatically generated, thus making the verification of the testing code not really necessary. In effect, as we have exemplified in Section 5, we envision a `Coq`-based testing framework in which test conditions are automatically generated from the specification and the abstract test cases are automatically generated by suitable decision procedures. Similar features have already been implemented for `Z` specifications [25] where a decision procedure for quantifier-free set and relational formulas is applied [51]. We believe it should be feasible to implement a similar framework using `Coq` technology.

Acknowledgments

The work of G. Betarte, J. Campo and C. Luna was partially funded by project ANII-Clemente Estable FCE_1_2014_1_103803: Mecanismos autónomos de seguridad certificados para sistemas computacionales móviles, Uruguay. The work of M. Cristiá was partially funded by ANPCyT's PICT 2014-2200.

References

[1] Open Handset Alliance, "*Android project*," Available at: http://source.android.com/, Last access: December 2017.

[2] Gartner, "Gartner says worldwide sales of smartphones grew 7 percent in the fourth quarter of 2016," Gartner, Inc., Tech. Rep., 2017, available at: http://www.gartner.com/newsroom/id/3609817. Last access: December 2017.

[3] J. P. Anderson, "Computer Security technology planning study," urlhttp://csrc.nist.gov/publications/history/ande72.pdf, Deputy for Command and Management System, USA, Tech. Rep., 1972. [Online]. Available: http://csrc.nist.gov/publications/history/ande72.pdf

[4] G. Betarte, J. D. Campo, C. Luna, and A. Romano, "Formal analysis of android's permission-based security model„" *Sci. Ann. Comp. Sci.*, vol. 26, no. 1, pp. 27–68, 2016. [Online]. Available: http://dx.doi.org/10.7561/SACS.2016.1.27

[5] Android Developers, "*Requesting Permissions at Run Time*," Available at: https://developer.android.com/intl/es/training/permissions/requesting.html, Last access: December 2017.

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

[7] P. Letouzey, "A New Extraction for Coq," in *Proceedings of TYPES'02*, ser. LNCS, vol. 2646, 2003.

[8] G. Betarte, J. Campo, M. Cristiá, F. Gorostiaga, C. Luna, and C. Sanz, "Towards formal model-based analysis and testing of android's security mechanisms," in *Proceedings of CLEI-SLISW '17*, 2017. [Online]. Available: http://www.clei2017-46jaiio.sadio.org.ar/sites/default/files/Mem/SLISW/slisw-02.pdf

[9] Android Open Source Project, "*Platform Architecture*," Available at: //developer.android.com/guide/platform/index.html, Last access: December 2017.

[10] Android Developers, "*Application Fundamentals*," Available at: http://developer.android.com/guide/components/fundamentals.html, Last access: December 2017.

[11] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of CCS '11*, 2011. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046779

[12] A. Armando, G. Costa, and A. Merlo, "Formal modeling and reasoning about the android security framework," in *TGC 2012*, 2012.

[13] Android Developers, "*Security Tips*," Available at: http://developer.android.com/training/articles/security-tips.html, Last access: December 2017.

[14] ——, "*<manifest>*," Available at: http://developer.android.com/guide/topics/manifest/manifest-element.html#uid, Last access: December 2017.

[15] ——, "*R.styleable*," Available at: http://developer.android.com/reference/android/R.styleable.html, Last access: December 2017.

[16] GSI, "Formal verification of the security model of Android: Coq code," Available at: http://www.fing.edu.uy/inco/grupos/gsi/documentos/proyectos/Android6-Coq-model.tar.gz, Last access: December 2017.

[17] 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.*, vol. 57, no. 6, pp. 2:10–2:10, Nov. 2013. [Online]. Available: http://dx.doi.org/10.1147/JRD.2013.2284403

[18] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of MobiSys '11*, 2011.

[19] Android Developers, "*Android NDK*," Available at: https://developer.android.com/guide/topics/security/permissions.html, Last access: December 2017.

[20] SecurityWeek News, "*Bankosy*," Disponible en: http://www.securityweek.com/android-trojans-exploit-marshmallows-permission-model, Last access: December 2017.

[21] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[22] P. Stocks and D. Carrington, "A Framework for Specification-Based Testing," *IEEE Transactions on Software Engineering*, vol. 22, no. 11, pp. 777–793, Nov. 1996.

[23] J. M. Spivey, *The Z Notation: A Reference Manual.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[24] M. Cristiá, G. Rossi, and C. S. Frydman, "{log} as a test case generator for the Test Template Framework," in *SEFM*, ser. Lecture Notes in Computer Science, R. M. Hierons, M. G. Merayo, and M. Bravetti, Eds., vol. 8137. Springer, 2013, pp. 229–243.

[25] M. Cristiá, P. Albertengo, C. S. Frydman, B. Plüss, and P. R. Monetti, "Tool support for the test template framework," *Softw. Test., Verif. Reliab.*, vol. 24, no. 1, pp. 3–37, 2014. [Online]. Available: http://dx.doi.org/10.1002/stvr.1477

[26] "OVAL Language," http://oval.mitre.org/, Last access: December 2017.

[27] "Ovaldi, the OVAL Interpreter reference implementation," http://oval.mitre.org/language/interpreter.html, MITRE Corporation, Last access: December 2017.

[28] M. Barrère, G. Betarte, and M. Rodríguez, "Towards Machine-assisted Formal Procedures for the Collection of Digital Evidence," in *Proceedings of PST'11*, 2011.

[29] M. Barrère, H. G., R. Badonnel, and O. Festor, "Increasing Android Security using a Lightweight OVAL-based Vulnerability Assessment Framework," in *Proceedings of IEEE SafeConfig'12*, 2012.

[30] M. Barrare, G. Hurel, R. Badonnel, and O. Festor, "Ovaldroid: An oval-based vulnerability assessment framework for android," in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, May 2013, pp. 1074–1075.

[31] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda, "Evolution, detection and analysis of malware for smart devices," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 2, pp. 961–987, 2014. [Online]. Available: http://dx.doi.org/10.1109/SURV.2013.101613.00077

[32] M. L. Polla, F. Martinelli, and D. Sgandurra, "A survey on security for mobile devices," *IEEE Communications Surveys and Tutorials*, vol. 15, no. 1, pp. 446–471, 2013. [Online]. Available: http://dx.doi.org/10.1109/SURV.2012.013012.00028

[33] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Communications Surveys and Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015. [Online]. Available: http://dx.doi.org/10.1109/COMST.2014.2386139

[34] B. Rashidi and C. Fung, "A survey of android security threats and defenses," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, vol. 6, no. 3, pp. 3–35, September 2015.

[35] Sufatrio, D. J. J. Tan, T.-W. Chua, and V. L. L. Thing, "Securing android: A survey, taxonomy, and challenges," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 58:1–58:45, May 2015. [Online]. Available: http://doi.acm.org/10.1145/2733306

[36] M. Conti, V. T. N. Nguyen, and B. Crispo, "Crepe: context-related policy enforcement for android," in *Proceedings of ISC'10*, 2011.

[37] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of ASIACCS '10*, 2011.

[38] M. Bugliesi, S. Calzavara, and A. Spanò, "Lintent: Towards security type-checking of android applications," in *Proceedings of FMOODS/FORTE 2013*, 2013.

[39] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses." in *USENIX Security Symposium.* USENIX Association, 2011. [Online]. Available: http://dblp.uni-trier.de/db/conf/uss/uss2011.html#FeltWMHC11

[40] A. Armando, R. Carbone, G. Costa, and A. Merlo, "Android permissions unleashed," in *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, C. Fournet, M. W. Hicks, and L. Viganò, Eds. IEEE Computer Society, 2015, pp. 320–333. [Online]. Available: https://doi.org/10.1109/CSF.2015.29

[41] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey, "Modeling and enhancing android's permission system." in *Proceedings of ESORICS'12*, S. Foresti, M. Yung, and F. Martinelli, Eds., 2012.

[42] W. Shin, S. Kiyomoto, K. Fukushima, and T. 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.* Washington, DC, USA: IEEE Computer Society, 2010, pp. 944–951. [Online]. Available: http://dx.doi.org/10.1109/SocialCom.2010.140

[43] ——, "A first step towards automated permission-enforcement analysis of the android framework," in *Proceedings of the 2010 International Conference on Security & Management, SAM 2010, July 12-15, 2010, Las Vegas Nevada, USA, 2 Volumes*, H. R. Arabnia, K. Daimi, M. R. Grimaila, G. Markowsky, S. Aissi, V. A. Clincy, L. Deligiannidis, D. Gabrielyan, G. Margarov, A. M. G. Solo, C. Valli, and P. A. H. Williams, Eds. CSREA Press, 2010, pp. 323–329.

[44] Z. Paraskevopoulou, C. Hritcu, M. Dénès, L. Lampropoulos, and B. C. Pierce, "Foundational property-based testing," in *Proceedings of ITP 2015*, 2015, pp. 325–343. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-22102-1_22

[45] C. Dubois, A. Giorgetti, and R. Genestier, "Tests and proofs for enumerative combinatorics," in *Proceedings of TAP 2016*, ser. LNCS, vol. 9762, 2016. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-41135-4_4

[46] H. Becker, J. M. Crespo, J. Galowicz, U. Hensel, Y. Hirai, C. Kunz, K. Nakata, J. L. Sacchini, H. Tews, and T. Tuerk, "Combining mechanized proofs and model-based testing in the formal analysis of a hypervisor," in *Proceedings of FM 2016*, 2016.

[47] T. Tuerk, "Efficiently executable sets used by fireeye," in *The 8th Coq Workshop Nancy, France, August 26, 2016*, 2016.

[48] A. D. Brucker and B. Wolff, "On theorem prover-based testing," *Formal Asp. Comput.*, vol. 25, no. 5, pp. 683–721, 2013. [Online]. Available: http://dx.doi.org/10.1007/s00165-012-0222-y

[49] Y. Jing, G. Ahn, and H. Hu, "Model-based conformance testing for android," in *Advances in Information and Computer Security - 7th International Workshop on Security, IWSEC 2012, Fukuoka, Japan, November 7-9, 2012. Proceedings*, ser. Lecture Notes in Computer Science, G. Hanaoka and T. Yamauchi, Eds., vol. 7631. Springer, 2012, pp. 1–18. [Online]. Available: https://doi.org/10.1007/978-3-642-34117-5_1

[50] G. de Cleva Farto and A. T. Endo, "Evaluating the model-based testing approach in the context of mobile applications," *Electr. Notes Theor. Comput. Sci.*, vol. 314, pp. 3–21, 2015. [Online]. Available: https://doi.org/10.1016/j.entcs.2015.05.002

[51] M. Cristiá and G. Rossi, "A decision procedure for sets, binary relations and partial functions," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, ser. Lecture Notes in Computer Science, vol. 9779, 2016, pp. 179–198. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-41528-4_10

A 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:

- 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 exatcly 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 to 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 [16].

B Generated code

Just for the sake of illustration, in what follows we provide listings of part of the Haskell code that has been automatically generated using the Coq extraction mechanism. The code is annotated with inline comments and manually indented to fit on the page width.

We have included the definition of the system, as a datatype, and the code of the dispatcher (*step* function), which implements the execution of an action in a given state. The full Coq code is available in [16].

Listing 1: The System

```
{- The System is represented as a datatype
 - comprising a State and an Environment }

data System =
    Sys State Environment

-- The Environment datatype
data Environment =
    Env
    -- The manifest and certificate of installed user applications
      (Mapping IdApp Manifest)
      (Mapping IdApp Cert)
    -- The permissions defined by the applications
      (Mapping IdApp (([]) Perm0))
    -- System applications
      (([]) SysImgApp)

-- The datatype State
data State =
    St
    -- The installed user applications
      (([]) IdApp)
    -- Granted group and individual permissions for each application
      (Mapping IdApp (([]) IdGrp))
      (Mapping IdApp (([]) Perm0))
    -- Running components and their instances
      (Mapping ICmp Cmp)
    -- Permanent and temporary permission delegations
      (Mapping ((,) ((,) IdApp CProvider) Uri) PType)
      (Mapping ((,) ((,) ICmp CProvider) Uri) PType)
    -- Values of resources
      (Mapping ((,) IdApp Res) Val)
    -- Sent intents
      (([]) ((,) ICmp Intent0))
```

Listing 2: Dispatcher

```
{- The function step is just a dispatcher which
 - performs pattern matching on the action to be
 - executed and calls the corresponding function
 - (for example, install_safe) }

step :: System -> Action -> Result0
step s a =
   case a of {
    Install app0 m c lRes -> install_safe app0 m c lRes s;
    Uninstall app0 -> uninstall_safe app0 s;
    Grant p app0 -> grant_safe p app0 s;
    Revoke p app0 -> revoke_safe p app0 s;
    GrantPermGroup grp app0 -> grantgroup_safe grp app0 s;
    RevokePermGroup grp app0 -> revokegroup_safe grp app0 s;
    HasPermission a0 p -> Result Ok s;
    Read0 ic cp u -> read_safe ic cp u s;
    Write0 ic cp u v -> write_safe ic cp u v s;
    StartActivity intt ic -> startActivity_safe intt ic s;
    StartActivityForResult intt n ic -> startActivity_safe intt ic s;
    StartService intt ic -> startService_safe intt ic s;
    SendBroadcast intt ic p -> sendBroadcast_safe intt ic p s;
    SendOrderedBroadcast intt ic p -> sendBroadcast_safe intt ic p s;
    SendStickyBroadcast intt ic -> sendStickyBroadcast_safe intt ic s;
    ResolveIntent intt a0 -> resolveIntent_safe intt a0 s;
    ReceiveIntent intt ic a0 -> receiveIntent_safe intt ic a0 s;
    Stop ic -> stop_safe ic s;
    GrantP ic cp a0 u pt -> grantP_safe ic cp a0 u pt s;
    RevokeDel ic cp u pt -> revokeDel_safe ic cp u pt s;
    Call ic sac -> call_safe ic sac s }
```