Towards formal model-based analysis and testing of Android's security mechanisms

Gustavo Betarte*, Juan Campo*, Maximiliano Cristiá†, Felipe Gorostiaga‡, Carlos Luna* and Camila Sanz*
*Fac. de Ingeniería, Universidad de la República, Montevideo, Uruguay
Email: {gustun,jdcampo,cluna,csanz}@fing.edu.uy
†CIFASIS, Universidad Nacional de Rosario, Rosario, Argentina
Email: cristia@cifasis-conicet.gov.ar
‡FCEIA, Universidad Nacional de Rosario, Rosario, Argentina
Email: feligorostiaga@gmail.com

***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 verified using the `Coq` proof assistant and discuss a testing technique that relies on the use of certified algorithms.**

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

978-1-5386-3057-0/17/$31.00

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 gain confidence in whether actual implementations of the platform conform with the abstract reference monitor, in particular on whether the discovered attacks can be actually performed or not.

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] which was not considered in the previous work.

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 and discusses some of the verified security properties. Section 5 motivates our approach for developing certified security testing. 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 [9]: 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. 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

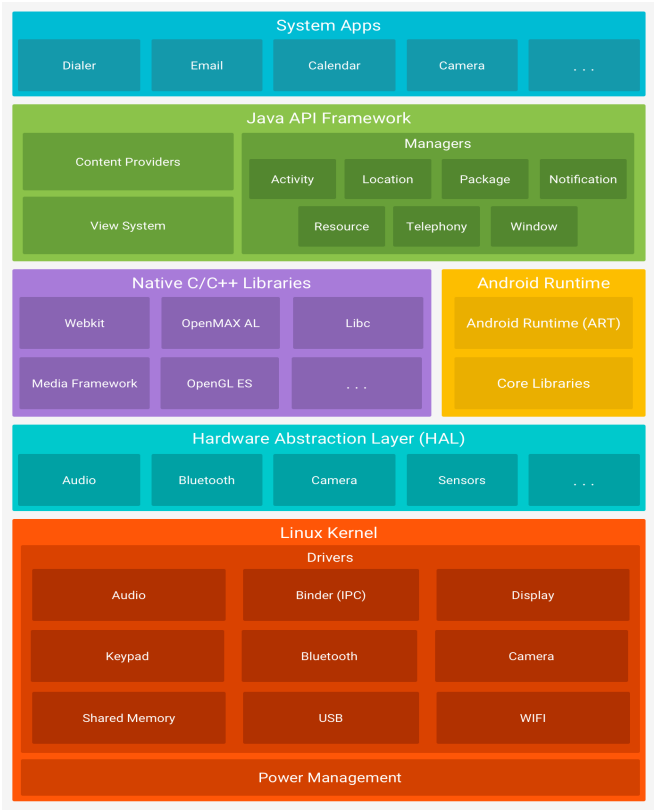

Figure 1. Android's architecture [8].

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 [9]. 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) [10] assigned to each installed application.

Every Android application must be digitally signed and be accompanied by the certificate that authenticates

its origin. 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 [11].

2.2. Permissions

Applications usually need to use system resources to execute properly. Since applications run inside sandboxes, this entails the existence of a decision procedure (a reference validation mechanism) that guarantees the authorized access to those resources. 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 decision procedure [12]. 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; 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) lastly, 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.

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 static attributes of theirs 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

InstApps ::= {AppId}
PermsGr ::= {AppId × {PermGroup}}
AppPS ::= {AppId × {Perm}}
CompInsRun ::= {CompInstance}
OpTy ::= $read \mid write \mid rw$
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 ::= {App}

AndroidST ::= InstApps × PermsGr × AppPS × CompInsRun × DelPPerms × DelTPerms × ARVS × Intents × Manifests × Certs × AppDefPS × SysImage

Figure 2. Android state

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 [13]. 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 eight 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.

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 the property 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 full (formal) definition of the predicate is available in [13].

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

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.

2. We consider here twice the number of operations (associated with the security model) contemplated in [13].

`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

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 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 [13].

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 [13] 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 the system 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" [9]. 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 [13], along with their proofs.

Lemma 2. For any s : 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 [14], [15] 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 on. We illustrate below our approach for two attacks, one implementing *eavesdropping* (section 4.1) and the another one *intent spoofing* (section 4.2).

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 [15]. In such conditions the eavesdropping attack can be prevented.

We establish, see Lemma 3, and prove, 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 : 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 $(protected(i, \text{signature}, s)$ or $protected(i, \text{signature/system}, s))$ 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 [14].

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's 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 : AndroidST, ap ap' : 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.

5. Certified security testing

Using the programming language of `Coq` we have developed an executable (functional) specification of the

$installed(ap, s)$	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

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`[3].

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.

Model-based testing (MBT) is a lively research area whose main goal is to use models of implementations as test case and oracle sources [16]. One of the possible high-level MBT processes is depicted in Figure 3. That is, MBT methods 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 our case, though, in the last step (i.e. verification w.r.t. to model) we use `CertAndroidSec` instead of the `Coq` axiomatic specification. In effect, as `CertAndroidSec` is a certified prototype of the `Coq` specification it behaves as prescribed by the specification. Furthermore, as `CertAndroidSec` is a program it can be easily run on the abstract test cases, thus greatly simplifying this step.

In our case we are working in adapting and applying a MBT method known as the Test Template Framework (TTF) [17], which is based on a systematic analysis of a formal specification. The TTF was originally thought as a MBT method for Z specifications [18] 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. Abstract test cases generated by applying the TTF, 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 (at the `Coq` level). Essentially, such a test case sets the initial state and input values from which the state transition to be tested must be executed. In this way, abstract test cases are expressible in `Coq` as the specification from where they were generated.

Abstract test cases can be refined into the actual Android platform using the OVAL technology [19], which 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, patch states. Real analysis is performed by OVAL interpreters such as

3. The prototype and its proof of correction are available in [13].

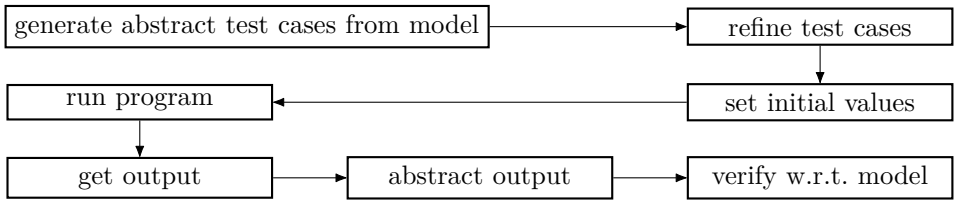

Figure 3. Proposed MBT process

Ovaldi [20], XOvaldi [21] and Xovaldi4Android [22]. Note the link and similarities between abstract test cases and OVAL conditions: our test cases define values for state variables whereas the OVAL language lets users to set platform states. Hence, all that needs to be done is to map `Coq` states into OVAL states. OVAL can also be used to lift the results of executing a test case on the platform to the `Coq` level, as it provides also a language for reporting the results from the evaluated systems.

So, given a test case t, let O_a denote the result of abstracting the output produced by running the OVAL counterpart of t on the platform. Moreover, let O_p denote the output produced by executing `CertAndroidSec` on t. At this point we would be in condition of performing 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.

The following example shows how test cases can be generated from the `Coq` axiomatic specification. Due to space restrictions we cannot show test case refinement nor output abstraction via the OVAL technology.

5.2. Example

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 *input space* of the operation, noted *IS*. The input space of the operation is the set of input values from which the operation can be executed. It includes the values of all before state and input variables declared in the operation. Then, in this case we have:

$$IS = \{s : \mathsf{AndroidST}, ap : \mathsf{AppId}\}$$

The second step in test case generation is to *recursively partition* the input space of the operation. Each partition is generated by applying a *testing tactic*. Then, a first testing tactic, say T^1, is applied to *IS* thus generating a (finite) family of subsets, $T^1_1, \ldots, T^1_{n_1}$, called *test conditions*. Each test condition is defined by a *characteristic predicate* depending on the variables of *IS*. This process can continue until the developer considers that a good partition has been generated.

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 conditions as terms in the DNF. In this case, each test condition 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:

$$DNF_1 = \{IS \mid validState(s) \wedge Pre(s, \texttt{install})\}$$
$$DNF_2 = \{IS \mid validState(s) \wedge ErrorMsg(s, \texttt{install}, ec)\}$$

As a second tactic we will apply Enumerated Types. This tactic applies to *IS* variables of some enumerated type. Hence, if v is a variable whose type is $E = \{c_1, \ldots, c_k\}$, the EC tactic generates test conditions characterized by the following predicates: $v = c_1, \ldots, v = c_k$. In DNF_2 we have that *ec* is of type *ErrorCode* (listing all the 28 possible errors of the system [13]). Therefore, we can apply EC to DNF_2 resulting in:

$$EC_1 = \{DNF_2 \mid ec = app\_already\_installed\}$$
$$\ldots\ldots\ldots$$
$$EC_{28} = \{DNF_2 \mid ec = CProvider\_not\_grantable\}$$

Observe how the successive test conditions are linked together by conjunction thus producing increasingly demanding test conditions. In this way, for instance EC_1, 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 EC_i will be unsatisfiable for `install` as many of these errors will not be raised by this operation. Unsatisfiable test conditions must be ignored and not further considered for partitioning.

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

$$TC_1 = \{EC_1 \mid s.apps = [Chrome] \wedge ap = Chrome\}$$

6. Related work

Several analyses have recently been carried out concerning the security of the Android system. Some of them [23], [24] point out the rigidity of the permission system regarding the installation of new applications in the device. Other studies [25], [26], [27] 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 [26], [28] 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.* [29], [30] 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. Also, we consider here executions that contemplate error management and a certified monitor of the security model. Finally, 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).

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. 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 [31]. 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. [32] have applied QuickChick to a problem of enumerative combinatorics. Very recently Becker et al. [33] reported the application of a MBT method based on random testing to a `Coq` model of the NOVA micro hypervisor. Tuerk [34], 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.

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.

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 [35]. 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 [36] where a decision procedure for quantifier-free set and relational formulas is applied [37]. We believe it should be feasible to implement a similar framework using `Coq` technology.

Acknowledgments

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

References

[1] Open Handset Alliance, "*Android project*," Available at: http://source.android.com/, Last access: March 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: March 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: March 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] Android Open Source Project, "*Platform Architecture*," Available at: //developer.android.com/guide/platform/index.html, Last access: March 2017.

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

[10] 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

[11] Android Developers, "<*manifest*>," Available at: http://developer.android.com/guide/topics/manifest/manifest-element.html#uid, Last access: March 2017.

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

[13] 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: March 2017.

[14] 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

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

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

[17] 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.

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

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

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

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

[22] 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.

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

[24] 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.

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

[26] 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

[27] 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

[28] 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.

[29] 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

[30] ——, "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.

[31] 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

[32] 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

[33] 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.

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

[35] 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

[36] 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

[37] *CAV 2016*, ser. Lecture Notes in Computer Science, vol. 9779, 2016.