A Formal Model to Analyze the Permission Authorization and Enforcement in the Android Framework

Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka
KDDI R&D Laboratories, Inc., Saitama 356-8502, Japan
Email: {wookshin, kiyomoto, ka-fukushima, toshi}@kddilabs.jp

Abstract—This paper proposes a formal model of the Android permission scheme. We describe the scheme specifying entities and relationships, and provide a state-based model which includes the behavior specification of permission authorization and the interactions between application components. We also show how we can logically confirm the security of the specified system. Utilizing a theorem prover, we can verify security with given security requirements based on mechanically checked proofs. The proposed model can be used as a reference model when the scheme is implemented in a different embedded platform, or when we extend the current scheme with additional constraints or elements. We demonstrate the use of the verifiable specification through finding a security vulnerability in the Android system. To our knowledge, this is the first formalization of the permission scheme enforced by the Android framework.

Keywords—permission; formal model; Android

I. INTRODUCTION

Android is the open mobile platform developed by Open Handset Alliance (OHA). The most innovative feature of Android is its openness. This allows anyone to write own applications and freely distribute them. While the openness provides various benefits to developers and users, it also increases security concerns. Due to the lack of control in the application development and distribution processes, it is quite possible for a user to download and install malicious software written by an adversary. Clearly, the consequences can include exposure of private information and damage to the telecommunication network.

Android tackles the security issues by enforcing a permission-based security policy at each device and restricting the capability of installed applications by permissions. Permissions are character strings uniquely distinguishable from each other. When a permission is bound to an operation and a resource object, we must obtain the permission to execute the operation on the object. The Android framework provides a set of default permissions in android.Manifest.permission class, and it also allows us to define new permissions. The new permissions should be declared in an application, and introduced into a system when the application is installed. Authorization of the permission is performed at the time of application installation. An application requests the set of permissions required to complete its task when it is installed on a device.

The list of requested permissions appears on the screen so that the user can review them. Only if the user agrees is the application installed. All of the requested permissions are given to the application. As a result, the system implements the main security principle of Android, namely, applications can perform operations that would affect other parts of the system only when they are permitted to do so.

While the above security countermeasure, which we call the Android permission scheme, has been already implemented and used in Android products, the security of Android has not been rigorously studied. In this paper, we propose a formal specification of the Android permission scheme as a foundation of study. Our specification is verifiable. It enables us to mathematically confirm diverse security-related properties, and verify the security. This may help to ascertain the security level of the Android framework; considering security certification standards (e.g., Common Criteria) requires formal descriptions for higher-level assurance.

Our contributions can be summarized as follows: 1) We propose an entity-relationship model for the Android permission scheme. This model helps us understand the scheme clearly, and can be used as a reference model for those who want to express and evaluate various security properties of the scheme. 2) We construct a state-based formal model and provide behavioral specification. We also explain how security requirements can be obtained and written in logical formulae. For given security requirements, the security can be verified. We illustrate the verification process. 3) We demonstrate the use of the verifiable specification with a security flaw we recently found in the Android system.

This paper is organized as follows: Section II briefly introduces the Android security architecture and explains terminologies. Section III presents a reference model of the Android permission scheme. Section IV describes the state-based specification. The process of security verification using a formal tool is illustrated in Section V. Related work and our findings are discussed in Section VI. We conclude in Section VII.

II. BACKGROUND

Application and components: An Android application is composed of sub-components of four types: activity, service,
content provider, and broadcast receiver. An activity works in the foreground of the handset screen interacting with the user. A service works in the background without a screen UI. A content provider supplies data storage for applications. A broadcast receiver helps application components inter-communicate. Each component can be instantiated and executed separately while interacting with others, and can even be started by other applications as needed. In this paper, we often liken the relationship between an application and its sub-components to the parent-child relationship. We call the former the parent and the latter the children of the former.

Android Manifest file: The general deployment format of an Android application is a signed archive. Compiled codes of an application and binary resources are archived in a Zip-compatible format, and signed with the author’s certificate. The package also accompanies an XML file, called the Android manifest file. The manifest file includes the following authorization-related information:

- List of application sub-components: The XML element, ⟨application⟩, is composed of a set of XML sub-elements for components such as ⟨activity⟩, ⟨service⟩, ⟨provider⟩, and ⟨receiver⟩.
- List of permission declarations: An application can declare a permission using ⟨permission⟩ element. The permission is added to a system when the application is installed. Let us call the permissions declared-permissions.
- List of permissions expected to be granted: An application lists the permissions needed to accomplish its task, using ⟨uses-permission⟩ element. The permissions are requested at the time of installation, and are listed on the screen. The user either allows the installation or aborts it. Allowing installation means granting all of the requested permissions as well. We use different terms for the requested permissions at the time of installation and for the permissions after being granted: requested-permissions and use-permissions of the application, respectively.
- List of permissions used for protection: The ⟨application⟩ element and the component elements have the android:permission attribute. If a permission name is assigned to the attribute, access to the application or the component requires the permission. If the attribute is set by the application element, all of its sub-components are protected by the permission as well. If a permission is assigned to the attribute in the component element, access to the component requires the permission set by the component, but the application-level permission enforcement setting is ignored. Let us call the permissions used to protect an application or a component, enforce-permissions of the application or the component.

Protection level: The protection level is an attribute of a permission, which determines how the permission is granted. The current security architecture of Android supports four protection levels: normal, dangerous, signature, and signatureOrSystem. The dangerous-level permissions are listed on the screen at the time of application installation, while the normal-level permissions are hidden in a folded menu on the screen. The granting process for the signature-level or signatureOrSystem-level permission requires certificate comparison. The signature-level permission is granted only when the application that requests it and the application that declares it are signed with the same certificate. The signatureOrSystem-level permission is granted in the same manner as the signature-level permission, and is also granted when the requesting application is signed with the same certificate with which the system images are signed.

Formal methods: Formal methods help us confirm whether a certain property holds in a particular system. Coq is an interactive theorem prover built on the theory of the calculus of inductive constructions, which is a formal language that allows us to express mathematical assertions and computer programs. We use Coq Integrated Development Environment (CoqIDE) to represent our state-based model and several security properties formally, and to verify the properties based on rigorous mathematical models.

Notations: We use mathematical symbols such as ∀, ∃, ∧, ∨, ¬, =, →, etc., in the way they are used as quantifiers and connectives in logic. We also use symbols used in set theory such as ⊆, ⊂, ×, ∈, etc. ≜ is used as a definition symbol. We also use a colon to specify the type of an identifier. e.g., id : T means the identifier id has the type T. → is used to annotate the type of a mapping. e.g., f: typeᵢ → typeⱼ means the mapping f maps an object of typeᵢ to an object of typeⱼ. The type of the mapping that has multiple arguments is written as (typeᵢ₁, typeᵢ₂) → typeⱼ or curried type, typeᵢ → typeⱼ → typeₖ.

III. SPECIFICATION OF THE ANDROID PERMISSION SCHEME

This section formally specifies the Android permission scheme by identifying system elements and describing their relationships. We also show how we can impose permission-based restriction on the component interaction in the model.

A. Preliminaries

We represent the Android permission scheme using the entity-relationship model which also has been used to model Role-Based Access Control (RBAC) policy [1] and its variations.

1) Applications, Components, and Permissions: The three major entities of the Android permission scheme are as follows:
- APPS, the set of applications
- COMPS, the set of application components
- PERMS, the set of permissions

Component-based construction of an application and declared-use/enforce-permissions can be represented in the relationships among the entities.
• **COMPOSE ⊆ APPS × COMPS**, an 1:N relationship that expresses the composition of applications.

• **composes**: (cmp: COMPS) → APPS, the mapping of component `cmp` onto its parent application.

• **DECLARE ⊆ APPS × PERMS**, an 1:N relationship that maps an application to a set of permissions declared by the application.

• **declaredBy**: (p:PERMS) → APPS, the mapping of permission `p` to an application that declares the permission.

• **USE ⊆ APPS × PERMS**, a N:M relationship that depicts permissions used by applications.

• **uses**: (app:APPS) → 2^PERMS, the mapping of application `app` to a set of permissions that `app` uses.

• **AENFORCE ⊆ APPS × PERMS**, a N:M relationship that illustrates the permissions that are enforced by applications.

• **aEnforces**: (app:APPS) → 2^PERMS, the mapping of application `app` to a set of permissions that `app` enforces.

• **CENFORCE ⊆ COMPS × PERMS**, a N:M relationship that illustrates the permissions that are enforced by application components.

• **cEnforces**: (cmp:COMPS) → 2^PERMS, the mapping of component `cmp` to a set of permissions that `cmp` enforces.

Figure 1 shows the entities and the relations in the form of an Entity-Relationship diagram. Not all functions that appear in the figure are itemized above due to lack of space, yet we believe the types and meanings of those not listed can be easily inferred.

The following is a more detailed explanation of the cardinality constraints in the diagram: (i) An application is composed of one or more components, and the components are introduced into a system as the application is installed on the system. Therefore, in the **COMPOSE** relation, each component **composes** one application and the application is **composedOf** one or more components. (ii) An application declares some permissions or none. Declared permissions by an application are introduced into a system as the application is installed. Therefore, in **DECLARE**, an application optionally **declares** multiple permissions, while each permission has to be **declaredBy** an application. (iii) The relation **USE**, **AENFORCE**, and **CENFORCE** are optional N-to-M relationships. An application can use or enforce some permissions. A component can enforce some permissions, too. Conversely, a permission can be used or enforced by an application, or enforced by a component. None of the use or enforcement relations is mandatory.

2) Permissions, Objects, and Operations: The execution of a privileged operation on a protected resource object may require a permission. Therefore, the notion of permission is related to both operations and objects. The relations determine what kind of permission is required and when the permission needs to be checked. Figure 2 illustrates the relations between permissions, operations, and objects. Note that applications and components correspond to objects in the figure, since we are going to specify the interactions between them. The following entity and relation need to be added:

• **OPS**, the set of operations

• **EXWITH ⊆ OPS × PERMS**, a N:M relationship that illustrates the permissions enforced on operations.

• **executeWith**: (op:OPS) → 2^PERMS, the mapping of operation `op` to a set of permissions.

The definition of a permission includes the relation with a set of operations and the relation with a set of objects. While the latter, **AENFORCE** and **CENFORCE**, can be collected from the manifest information, the former, **EXWITH**, can be obtained from concrete implementation, source codes of the Android framework and applications. For instance, when an activity component sets up its enforce-permission in the manifest file, it does not mean the permission is always checked whichever action is performed on the activity. Yet, when it gets started, the permission checking routines embedded in relevant API calls (e.g., `startActivity()` or `startActivityForResult()`) are triggered. Google's document [2] briefly guides execution of which operation leads to the permission test. The relationship with operations can also be made when an application explicitly invokes check permission functions (e.g., `Context.checkCallingPermission()`) in its code.

The relations in Figure 2 can be refined and redrawn in terms of components, as shown in Figure 3. The new optional N:M relation, **ENFORCE** reflects **CENFORCE** when a component enforces permissions, or **AENFORCE** when a component does not enforce permissions by itself. The mapping **enforce**: (cmp:COMPS) → 2^PERMS maps a component `cmp` to the set of permissions obtained by

![Figure 1: Entities and relations in the Android permission scheme](image1)

![Figure 2: Android permissions](image2)
Every Android application is supposed to
SIGN, DANGEROUS
PERMS
PLEV or, the mapping of
CERTS and →
comps signProtected
app 
COMPS, op 
{
which has the type of
app p SIGN
when the permission
)
}:{
have the
CERTS TRUE, the set of certificates used to sign applications
and
R⊆C δρ
R UR ∈ OPS
is a tuple of (
γ uses
A enforces. For the sake of convenience, we use
op is a set of permissions that
COMPS
A C⊆A×C
∈ cenforces × P
COMPS
2
is a set of child components of
which returns,
scmp 
{

Figure 3: Android permissions (refined)

Figure 4: Overall Diagram

\[
\{ p : \text{PERMS} \mid (cEnforces(cmp) = \phi \Rightarrow p \in cEnforces(cmp)) \lor (cEnforces(cmp) \neq \phi \Rightarrow p \in aEnforces(composes(cmp))) \}
\]

3) Certificates: Every Android application is supposed to be signed with the author’s certificate before the application is distributed. We can define a set CERTS to represent the certificates used to sign applications. Another set PLEV expresses the protection level attribute of the permission. As mentioned in Section II, certificate comparison is required to grant permissions with certain protection levels.

• CERTS, the set of certificates used to sign applications
• signedWith: (app:APPS) → CERTS, the mapping of application app to the certificate with which the application is signed.
• PLEV = {NORMAL, DANGEROUS, SIGN, SIGNORSYS}, the set of the protection levels given to the permission, where the enumerated elements correspond to the levels.
• signProtected: (p:PERMS) → BOOL, the mapping returns TRUE when the permission p has the protection level of SIGN or SIGNORSYS. Otherwise, returns FALSE.

Finally, Figure 4 shows the overall entities and relations. certs and plev attributes of APPS and PERMS have the type of CERTS and PLEV, and record the certificate used to sign each application and the protection level of each permission, respectively.

B. Specification of Interactions

The interaction between components is composed of operations that one component performs on the other. When we call one performer scmp, the other ocmp, and the sort of operation op, we can then denote the interactive operation as a tuple of (scmp, ocmp, op).

Some of the operations are protected by permissions. Let us refer to the procedure that guards the permission protected operations as checkAccess which has the type of
checkAccess: (scmp, ocmp: COMPS, op : OPS) → BOOL

The checkAccess tests the legitimacy of the interactive operation by calculating if scmp owns all of the permissions that ocmp enforces regarding op. If so, it returns TRUE, otherwise FALSE. The set of permissions that ocmp enforces on op can be calculated by
\{ p : \text{PERMS} \mid p \in \text{enforces(ocmp)} \land p \in \text{executedWith(op)} \}

The set of permissions that scmp uses can be obtained by using compuses() and uses(). Finally, we can define checkAccess() which returns TRUE only if the following condition is satisfied:
∀p: PERMS, p \in \text{enforces(ocmp)} \land p \in \text{executedWith(op)} \Rightarrow p \in \text{uses(compuses(scmp))}

IV. STATE-BASED SPECIFICATION OF THE ANDROID PERMISSION SCHEME

We introduce the notion of state to our specification in order to discuss the legitimacy of interactive operations and other security-related properties at a given moment. The following additional elements and relations are needed to describe the state information.

• RUNS ⊆ COMPS, the set of active components.
• CURXS ⊆ COMPS × COMPS × OPS, the set of on-going interactive operations among components. For a tuple (scmp, ocmp, op) ∈ CURXS, scmp is the component that performs the interactive operation op on ocmp.
• MNFST ⊆ APPS × 2COMPS × 2PERMS × 2PERMS × 2PERMS × 2PERMS, the manifest information of an application. An element of MNFST, m is a tuple of (α, γ, vp, δρ, αenforces, γenforces), and it describes the manifest information of application α ∈ APPS. γ ⊆ COMPS is a set of child components of α, vp ⊆ PERMS is a set of permissions that α wants to use, δρ ⊆ PERMS is a set of permissions that α declares, αenforces, γenforces is a set of permissions that α enforces, and γenforces ⊆ CENFORCES is a set of permissions that each of γ enforces. For the sake of convenience, we use dot operator to project elements of the tuple. That is, m.δρ identifies the set of declared-permissions in the manifest file of m.α.
• UR = \{OK, CANCEL\}, the set of user’s response.

A. States and transitions

For the system that runs the Android permission scheme, STATE is the set of states that the system can have. A system state s ∈ STATE is a tuple of (A, C, P, AC, DP, UP, EP, R, X), where A ⊆ APPS, C ⊆ COMPS, P ⊆ PERMS, AC ⊆ A × C, DP ⊆ A × P, UP ⊆ A × P, EP ⊆ C × P, R ⊆ C, X ⊆ C × C × OPS. Each element of the tuple represents the set of installed applications, application components of the installed applications, permissions, application compositions, declared-permissions by the applications and the system, use-permissions of the installed
applications, enforce-permissions of the components of the installed applications, running components, and on-going interactions, respectively.

We assume that the mappings described in Section III also work for state elements, yet the types and definitions of mappings need to be extended to include the state information. For instance, the mapping composes should be now extended to have the type of \((s : \text{STATE}, \text{cmp : COMP}S) \rightarrow \text{APPS} \) and defined on \(AC\) of the given state \(s\).

Additionally, we define primitive administrative operations that modify the state information and cause a state change from a given state \(s\) to another state \(s'\). The operations are listed in Listing 1. In the listing, we assume the parameters \(s, s' \in \text{STATE}\) are omitted in the ‘...’ symbol, just in order to save space. Dot operator is used again, to project each state element. \(s.A\) is the set of installed applications at \(s\).

Listing 2 displays abstract operations of the system, which are composed of the state elements, the mappings, and the primitive operations we have defined. The \(install\_ok\) operation adds an application to the system, whereas \(install\_nok\) does nothing. \(install\_ok\) is for the case where the user agrees to the installation at the installation screen, while \(install\_nok\) is for the case where the user cancels the installation. \(uninstall\) removes an application if no component of the application is currently running. \(start\) and \(stop\) start and stop an application, respectively. \(access\) begins working on a permission-protected object, and \(release\) finishes it.

In Listing 2, we assume that state information of \(st\) remain the same as that of \(s\) unless explicitly altered by the operations. As an example, \(install\_ok\) may make changes to the state information of \(s\), except the running components (\(R\)) and on-going accesses (\(X\)). We could have added two invariance equations of \((st.R = s.R)\) and \((st.X = s.X)\), but omitted in order to save space. As another example,
in \texttt{install\_nok}, we can define equalities between every state information of \textit{s} and that of \textit{st}. Let us refer to these implicit equivalence relations as implicit invariances.

\subsection*{B. State Validity}

We can obtain some sentences that mention security requirements of the permission scheme from the document, “Security and Permissions”, written by Google [2]. Although the requirements are not explicitly itemized, it is possible to identify them from sentences, and translate them into logical representations written in terms of our state-based specification elements.

For example, the first sentence of the “Security Architecture” section of the document includes: “no application has permission to perform any operation that would adversely impact another application”. This can be rewritten in an entailment-statement; for instance, the fact that an application performs a security sensitive operation on another application implies the application has the relevant permission (to perform the operation). Again, the antecedent of the entailment can be expressed by the access tuple, \((\texttt{scpmp,ocnmp,op})\) if we map the components to the application. The consequent also can be written using the function \texttt{uses()} and \texttt{executeWith()}. As a result, the statement can be converted into:

\begin{equation}
\text{accessValidity}(s;STATE) \overset{\text{def}}{=} \forall \text{subj}, \text{obj} : \text{COMPS}, \text{op} : \text{OPS}, \langle \text{subj}, \text{obj}, \text{op} \rangle \in s.X \Rightarrow \\
\forall p : \text{PERMS}, p \in \text{executeWith}(s, op) \Rightarrow \\
p \in \text{enforces}(s, obj) \Rightarrow \\
\exists \text{app} : \text{APPS}, \text{app} = \text{compose}(s, \text{subj}) \land p \in \text{uses}(s, \text{app}).
\end{equation}

As another example, “... permissions requested by the application are granted to it... based on checks against the signatures of the applications declaring those permissions...” is found in the first sentence of the third paragraph in the “Using Permissions” section. We can rephrase into two statements: \textit{permissions used by an application means the ones requested by the application (in the manifest file)} and if the signature-level permission is used by an application, then signatures need to be compared between the requester and declarer of the permission. The statements can be converted into two security conditions:

\begin{equation}
\text{requestedAndGranted}(s;STATE) \overset{\text{def}}{=} \\
\forall \text{subj}, \text{obj} : \text{COMPS}, \text{op} : \text{OPS}, \langle \text{subj}, \text{obj}, \text{op} \rangle \in s.X \Rightarrow \\
\exists m : \text{MNFST}, \text{app} = m.\text{APP} \Rightarrow \\
p \in m.\text{vp}.
\end{equation}

\begin{equation}
\text{signPermAuth}(s;STATE) \overset{\text{def}}{=} \\
\forall \text{subj}, \text{obj} : \text{COMPS}, \text{op} : \text{OPS}, \langle \text{subj}, \text{obj}, \text{op} \rangle \in s.X \Rightarrow \\
\text{signProtected}(s, \text{p}) = \text{TRUE} \Rightarrow \\
\exists \text{app} : \text{APPS}, \text{app} = \text{declaredBy}(s, \text{p}) \Rightarrow \\
\exists \text{cert} : \text{CERTS}, \text{cert} = \text{signedWith}(s, \text{app}) \land \\
\text{cert} = \text{signedWith}(s, \text{app}).
\end{equation}

Furthermore, we can add some constraints based on our insight and observation; such as, if \textit{there is an ongoing access, its subject component must be in the running state and every running component is a part of an installed application}, which can also be formally stated as follows:

\begin{equation}
\text{accessByActiveOnes}(s;STATE) \overset{\text{def}}{=} \\
\forall \text{subj}, \text{obj} : \text{COMPS}, \text{op} : \text{OPS}, \langle \text{subj}, \text{obj}, \text{op} \rangle \in s.X \Rightarrow \\
\exists \text{app} : \text{APPS}, \text{app} = \text{compose}(s, \text{subj}) \land p \in \text{uses}(s, \text{app}).
\end{equation}

\begin{equation}
\text{accessByInstalledOnes}(s;STATE) \overset{\text{def}}{=} \\
\forall \text{subj}, \text{obj} : \text{COMPS}, \text{op} : \text{OPS}, \langle \text{subj}, \text{obj}, \text{op} \rangle \in s.X \Rightarrow \\
\exists \text{app} : \text{APPS}, \text{app} = \text{compose}(s, \text{subj}) \land \text{app} \in s.A.
\end{equation}

We have identified ten security conditions up to now, and specified them as logical predicates. By simply combining the predicates, we define a predicate, \textit{ValidState}(s;STATE) which tests the validity of a state with regard to security. It is worth noting that the valid state can be defined with different conditions, depending on our interests. We do not list up here all of the predicates we have specified. We do not claim that those security conditions are complete or that they will satisfy everyone’s needs. Neither is this our main concern. Instead, we aim at building a formal foundation to analyze the Android permission scheme with given security requirements.

Utilizing a formal tool, we can confirm if the permission scheme satisfies the security requirements. The requirements could be added or the current permission scheme could be updated as needed. Still, the security can be evaluated through the verification process. The verification process and benefits from it are discussed in the following sections.

\section*{V. Verification}

The purpose of the verification is to prove that all transition operations alter system states satisfying the security requirements. The proposed Android permission model is verifiable: We can prove the security for given security requirements, and a theorem prover can be utilized for the verification task.

We use the \texttt{Coq} interactive theorem prover. The primitive elements, the abstract states, and the abstract operations already mentioned can be translated into the specification language of \texttt{Coq}. The abstract operations are specified by

\begin{verbatim}
Listing 3: Encoded examples in Coq

accessByActiveOnes(s;STATE) \overset{\text{def}}{=} \\
\forall \text{subj}, \text{obj} : \text{COMPS}, \text{op} : \text{OPS}, \langle \text{subj}, \text{obj}, \text{op} \rangle \in s.X \Rightarrow \\
\exists \text{app} : \text{APPS}, \text{app} = \text{compose}(s, \text{subj}) \land p \in \text{uses}(s, \text{app}).
\end{verbatim}

\begin{verbatim}
accessByInstalledOnes(s;STATE) \overset{\text{def}}{=} \\
\forall \text{subj}, \text{obj} : \text{COMPS}, \text{op} : \text{OPS}, \langle \text{subj}, \text{obj}, \text{op} \rangle \in s.X \Rightarrow \\
\exists \text{app} : \text{APPS}, \text{app} = \text{compose}(s, \text{subj}) \land \text{app} \in s.A.
\end{verbatim}

\begin{verbatim}
...
By the postcondition (which includes implicit invariances), we have a set of equivalence equations between state information of $s$ and $st$. After unfolding the accessValidity $s$, and replacing occurrences of $s$ with $st$ using the equivalence equations, we can obtain accessValidity $st$.

Case 2: In the case where the user agrees to the installation, we have pre-/post-conditions of the install-ok. Again, we need to perform case analysis for existing applications and for the new application. Case 2-1: In the case of existing applications, by the implicit invariances, their permission use ($UP$) and enforcement ($EP$) do not change since the post-condition of the install-ok only updates the permission information of the new application. Therefore, for the existing applications, we can derive some equivalence equations between $s$ and $st$, and consequently, rewrite accessValidity $s$ with accessValidity $st$ using the equations. Case 2-2: In the case of a new application, the new application cannot participate in any kind of access after the installation due to the pre-/post-condition of install-ok. We can show accessValidity holds trivially.

We only draw a sketch of the proof above. The actual verification process is carried out using the Coq interactive theorem prover. For the proof of each lemma, inference rules are applied and construct each step of the proof. CoqIDE shows the result of every rule application until we complete the proof. We can obtain complete formal proofs from the process in the sense that every step of the proof is justified by the rule which also convinces the machine. Our Coq specification is written with 3919 lines and includes 123 lemmas that we have proved. Combining the lemmas, we finally prove the following theorem:

**Theorem SecureMigration:** Let ValidState be a predicate over STATE, defined by the conjunction of the given security conditions, as mentioned in Section IV-B. For any states $s$, $st$ and an event $e$, if ValidState $s$ and $e \Rightarrow$ hold, then ValidState $st$.

Based on the theorem SecureMigration, we can inductively assure that the system would remain secure if it starts from a secure state and provided it executes the verified transition operations. A possible initial state is the one where no application has been installed and therefore the validity predicate trivially holds.

VI. DISCUSSION: RELATED WORK; APPLICATION AREA

There have been several studies done on the topic of security of the Android system, but few of them focus on the formal aspect of the permission enforcing framework. Enck et al. developed an application evaluation tool, Kirin [3]. They represent the Android security policy with the notion of access matrix and test security policy invariants of an application at the time of installation. Chaudhuri proposed a typed language [4] to specify applications and reason about data flow. The type checking result of an application code makes it clear whether the application can preserve
the secrecy and integrity of local data or not. Enck et al.’s automated tool and Chaudhuri’s language-based approach differ from ours in that they evaluate applications in order to exclude malicious applications.

We have focused on the behavioral aspect of the framework rather than specification of application logic, and have attempted to confirm if it correctly authorizes permissions in accordance with the given requirements. To achieve this object, we aim at building a formal model of the Android permission scheme. To our knowledge, this is the first formalization of the permission scheme which is enforced by the Android framework. The specification covers most of the features that are supported by the current Android system. The part of our study that deals with the specification of the Android permission scheme was inspired by the standard Role-Based Access Control model [1]. The formalization of the abstract state has been guided by Z specification examples [5] and collecting abstract operations was inspired by the specification of J2ME MIDP [6]. Due to the difference between J2ME MIDP and Android, our abstract operation set does not include permission request/revoke operations, whereas the MIDP specification does. Instead, they are represented in the primitives and performed when applications are installed/uninstalled. Our specification also includes the notion of code signing. The permission authorization process involves the comparison of signing certificates. Moreover, the notion of access is introduced to model the permission-protected interactions between application components.

Our specification is verifiable using an interactive theorem prover. We provide an example using Coq, but other formal tools can be used depending on the purpose. The tools facilitate theoretical confirmation of security properties when the system is updated or when there are new security requirements. Our recent experience of finding a security vulnerability in the Android system [7] demonstrates the use of our model: The definition of uninstall did not include the removeAllPermUse primitive in our earlier version of the specification, since we had not been able to find it in the documentations and sources. Absence of the primitive could lead to dangling permissions after an application is uninstalled, which means a permission is granted without being declared anywhere. In consequence, the meaning of a granted permission can be replaced with another, and can result in a permission being spoofed. We logically confirmed the existence of the vulnerability after we added a new security requirement, which states that every permission can be used only if the declaration of the permission exists in the system, and also confirmed it actually existed in the concrete implementation by writing and running exploit examples. We reported the vulnerability to Google, and they wrote the fix. The patch includes a routine that cleans up use-permissions when the application declares that those permissions have been removed, which corresponds to the removeAllPermUse primitive.

Another potential advantage of using Coq is the feature of code extraction. We can obtain executable codes written in functional languages like OCaml or Haskell from Coq specifications. The extracted codes can guide us when we update the concrete implementation.

VII. Conclusion

We presented a formal model of the permission scheme which is currently implemented in the Android system. We proposed a specification which enables us to express authorization and permission-protected interactions among application components. We showed how we can confirm if the system that enforces the permission scheme meets given security requirements. The security of the system is mathematically verifiable with the help of a formal tool.

Our verifiable specification can be used when the permission scheme is implemented on another platform or when we want to extend the current scheme by adding more features or constraints. We exemplified the use of the specification through a security vulnerability of Android we had found. Handset manufacturers and mobile network operators can benefit from our work, especially when they plan to customize the Android framework and build their own editions. Due to growing concerns about protecting both customer privacy and the telecommunication network, the security properties need to be evaluated before the products are released.

References


