Towards Formal Analysis of the Permission-based Security Model for Android

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

***Abstract*—Since the source code of Android was released to the public, people have concerned about the security of the Android system. Whereas the insecurity of a system can be easily exaggerated even with few minor vulnerabilities, the security is not easily demonstrated. Formal methods have been favorably applied for the purpose of ensuring security in different contexts to attest whether the system meets the security goals or not by relying on mathematical proofs. In order to commence the security analysis of Android, we specify the permission mechanism for the system. We represent the system in terms of a state machine, elucidate the security needs, and show that the specified system is secure over the specified states and transitions. We expect that this work will provide the basis for assuring the security of the Android system. The specification and verification were carried out using the Coq proof assistant.**

***Keywords*-Android; permission; security; formal model; Coq**

I. Introduction

Android is an open mobile platform developed by the Open Handset Alliance (OHA) led by Google, Inc. The Android platform consists of several layers: the Linux kernel, native libraries, the Dalvik virtual machine (VM), and an application framework [1]. The Linux kernel provides basic operating system services and hardware abstraction for the upper software stacks. Native libraries support the miscellaneous functionalities of web browsing, multimedia data processing, database access, and GPS reception optimized for a resource-limited hardware environment. The register-based Dalvik VM runs Java code with low memory demand. At the top of the layers, Android provides a component-based programming framework so that users can easily build their own applications.

An Android application is written with new and reusable application building blocks, such as *activity*, *broadcast intent receiver*, *service*, and *content provider*. After an application is written, it is deployed in a Zip-compatible archive, .apk file, or the Android package file. An Android package file contains codes, resources, and a special XML file called the Android Manifest file. The manifest file contains basic information about an application such as the package name, component descriptions, and permission declarations.

The definitions of Android permissions are found in android. Manifest.permissions class. Each permission is defined as a string and conveys the rights to execute a particular operation. In the manifest file, the permissions are sorted into two categories: permissions required by the application in order to execute and permissions required by others in order to interact with the application components. The permissions that the application uses are listed in the ⟨uses-permission⟩ tag and the permissions that the application imposes are in the ⟨permission⟩ tag (see Code 1).

Code 1 An example of AndroidManifest.xml.

```
...
<manifest xmlns:android="http://schemas.android.
 com/apk/res/android" package="jp.kddilabs.
 AppMarketClient" android:versionCode="1" android:
 versionName="1.0.0">
  <uses-permission xmlns:android="http://
   schemas.android.com/apk/res/android" android:
   name="android.permission.INTERNET"/>
  <uses-permission xmlns:android="http://
   schemas.android.com/apk/res/android" android:
   name="android.permission.READ_OWNER_DATA"/>
  <permission xmlns:android= "http://schemas.
   android.com/apk/res/android" android:label=
   "@+id/textElement" android:name=
   "jp.kddilabs.READ_APPMARKET_DATA"
   android:permissionGroup="@string/app_name"
   android:protectionLevel="normal"/>
  <application android:icon="@drawable/
   icon" android:label="@string/app_name">
    <activity android:name=".AppMarketClientMain"
...
```

Android takes the sandbox approach to prevent an application from exerting a malicious influence on other applications or the system. An Android application basically runs in a secure sandbox; thus, it cannot access other assets unless explicitly authorized to do so. The sandbox mechanism is supported by lower level mechanisms. An application is run within its own Dalvik VM instance and also runs in a separate Linux process with its own Linux user ID. The system assigns a user ID when an application is installed. In order to interact with other applications or the system over the isolation, the application needs permission to do that. The permission authorization occurs at the application installation time as well. The application requests the permissions that it needs as written in the manifest file, and then the Android system asks the user for confirmation if allowing the application to have the requested permissions. After authorization for the application, the system will not ask for more permission again.

The security mechanisms of Android have apparently worked well so far. Several security-related bugs have been reported [2][3], but patched without serious security failures. The

978-0-7695-3750-4/09 $26.00 © 2009 IEEE
DOI 10.1109/ICWMC.2009.21

sandbox mechanism effectively prevents malicious attempts from compromising other parts of the system, as pointed out in [4]. However, it might not be enough to earn customer trust. Moreover, it seems Android faces something bigger than practical threats which is the long-standing mistrust of open source software with regard to security. Contrary to the traditional approach taken by mobile phone service providers and manufactures, OHA released the Android source code to the public and encouraged developers to participate in application development. While the liberation has been applauded by open source communities, end users and service providers have doubted the security of the open mobile platform, although open source does not imply easy-to-break and some people even claims the open source approach is stronger in terms of security [5].

Applying formal methods is a good way to assure users of the security of a system. Formal methods enable us to clearly state the elements and behaviors of the system, clarify desired security properties, and provide the proof that the system satisfies the security properties. Besides convincing users of the security of the system, formal methods provide benefits for developers; during the specification and verification processes, developers can grasp inconsistencies and incompleteness in the requirements and implementation and correct the system.

This paper mainly presents our specifications and the results of the verification of the permission mechanism of the Android system. We represented the system as a state machine aiming to ensure the security of the system in terms of the Basic Security Theorem [6]. The specified state and transition elements of the system and identified security conditions are briefly introduced. We also exemplify some lemmas that are used for security proofs. However, we cannot list all of them in this paper because of space limitations. All specifications and the verification used the Coq proof assistant. Assuming that the reader is already familiar with the basic syntax of Coq and natural deduction, we did not explain Coq syntax in this paper because of space limitations. For further information on Coq, please refer to the Coq Web page [7] and the book by Bertot and Casteran [8].

The rest of the paper proceeds as follows. In Section II, we describe security requirements and specify system elements and operations in terms of state-transition system. Then in Section III, the verification of the security properties are introduced with some exemplified lemmas and proofs. We discuss the limitation and expected contribution of our work in Section IV, and conclude in Section V.

II. Specifications

Although the security of the Android system is enforced through multiple system layers, we only specify the permission mechanism, which is defined at the Android application framework. In other words, we do not specify the sandbox enforcement of the VM or the Linux kernel. The security of the Linux kernel and the VM has been studied separately. Our formal model would be too complicated if we specified the security mechanisms of other layers altogether. Moreover, the notion of permission is to release capabilities, whereas the sandbox is to provide restrictions. Accordingly, we concentrated on the releasing features, assuming that the sandbox mechanism is working properly. We only include security-related features in our specifications in order to keep the specifications to a manageable size.

Modeling the permission mechanism of the Android system goes through several steps, which are similar to the Goguen-Meseguer program described by Cristia [9]:

- Step 1: List the security needs.
- Step 2: Take security-related parts of the system (e.g., data structures, algorithms, input, output, etc.) and describe the system: define states with chosen system elements and define transitions based on the operations that affect the state elements. The specification results are introduced in Section II-A and Section II-C.
- Step 3: Describe security conditions formally in terms of a state. The security conditions are based on the security needs of Step 1.
- Step 4: Prove that security conditions are satisfied over state transitions.

The security needs in the permission-based Android system can be addressed as follows: applications request permissions statically, not dynamically. In other words, permissions that an application requests must be written in its manifest file. The permissions are granted by the user when the application is installed and is maintained for the duration of the life of the application. The grant information is removed when the application is uninstalled from the system. In addition, running components of the application must have only the permissions granted to the application. Thus, it would be possible to carry out the basic idea of the Android security architecture: “no application has permission to perform any operation that would adversely impact other applications, the operating system, or the user” [10]. The needs can be grouped and detailed as follows:

- A user grants an application the permissions that the application requested. The requested permissions ought to be written in the manifest file of the application. No more permissions are given to an application than the application requested.
- An installed application is distinguishable from others by its identifier. An application has authorized permissions only for the duration of its lifetime in the system. No more permissions are allowed than have been authorized. Only installed applications can have permissions in the system.
- A component is distinguishable from others by its identifier. Every component is a child of an application, and they both have each others information. A component is allowed to have the permissions granted to the parent application. At most, only one foreground component exists in a certain state.

A. System state

In this section, we define the primitive elements of the system: permissions, applications, components, authorizations, and states.

An Android permission is the right to execute a specific operation. Since permissions are designed to expand the functionality of a sandboxed process out of the limitation, they are generally related to access to particular data or functions of other applications or the system. In our specifications, we define permissions as a set. The elements of the permission set are distinguishable from each other, and it is sufficient for us to reason about the security of the system. We do not introduce individual definitions of the android.Manifest.permission or user-defined permissions.

The application, *AppPackage*, is a record type with the following fields: an identifier, a set of application components, and a permission request. A set of application components is identified by *app_cmpnnts*. The record type *PermRequest* is defined to represent the permission request of an application. Each predicate *permreq_ineed* and *permreq_uneed* maps the listed permissions in the ⟨uses-permission⟩ and ⟨permission⟩ tags, respectively.

```
Record AppPackage: Type := mkAppPackage {
  app_id : AppID;
  app_cmpnnts : CmpnntID → Prop;
  app_permrqst : PermRequest }.
Record PermRequest : Type := mkPermRequest {
  permreq_ineed : Permission → Prop;
  permreq_uneed : Permission → Prop }.
```

Cmpnnt is also a record type for components. It has an identifier *cmpnnt_id* and a parent application *cmpnnt_papp*. The *Cmpnnt* does not have permission information itself, but it will use the permissions of the parent application.

```
Record Cmpnnt : Type := mkCmpnnt {
  cmpnnt_id : CmpnntID;
  cmpnnt_papp : AppPackage }.
```

Before an application is installed, the package installer of the system asks whether the user will grant the set of permissions the application requests. Based on the user response, the system allows the application to have the permission], or not. The allowance is defined as *Authorization* as follows:

```
Record Authorization : Type := mkAuthorization {
  auth_by_user : AppID → Permission → Prop }.
```

Now we can define a system state as a record type:

```
Record State : Type := mkState {
  state_fgcmpnnt : optionT Cmpnnt;
  state_package_tbl : AppPackage → Prop;
  state_perm_tbl : AppID → Permission → Prop }.
```

The *state_fgcmpnnt* is the foreground component running on top of the handset screen. It is the optionT type that is an inductive type with two constructors: *ExT* : *(T:Type)* → *optionT* and *NoneT* : *optionT*. *ExT* is parameterized with a a type; therefore, the optionT can represent the optional existence of an object. For example, when we have *(s:State)* and *(cmp:Cmpnnt)*, *state_fgcmpnnt s* = *ExT cmp* means that there is a foreground component at the state *s*. Conversely, *state_fgcmpnnt s* = *NoneT* means that there is no foreground component at *s*. The definition of optionT is borrowed from another user-contributed Coq library, Icharate.Kernel.lambda_coq. The package table, *state_package_tbl* identifies a set of installed applications at that state. The permission table, *state_permission_tbl* does sets of authorized permissions for applications.

B. Security Property

From the security needs mentioned at the beginning of the Section II, we obtain the security conditions and state those using predicates with respect to the state. The predicates are self-explanatory, but it would be better to clarify the meaning of the naming terms here; an "*installed*" application *app* at state *s* means that the *s* has information about the *app* in its package table, that is, (*state_package_tbl s app*) is true. If an application *app* "*requested*" permission *p*, then the *app* asked for *p* using its permission request *prqst*. It can be stated as (*permreq_ineed prqst p*), when *prqst* = *(app_permrqst app)*. If permission *p* is "*granted*" to an application *app* at state *s*, then *s* has the information in its permission table. Hence, (*state_perm_tbl s appid p*) holds, where the *appid* is the identifier of the *app*. Permission *p* is "*authorized*" for an application *app*, if and only if there was a decision that allows the *app* to have *p*. Thus, a predicate (*auth_by_user auth appid p*) holds for the authorization *auth*. Furthermore, we say permission *p* is "*properly*" given to an application *app* if *p* is "*requested*" by and "*granted*" to the "*installed*" *app*.

1) *SecureAuth* is a predicate that holds, if and only if, any permission given to any application at a state entails that the application requested the permission based on its manifest information.
2) *SecureCmpnnt* is a combination of three predicates, and all of the predicates specify security conditions for the foreground component of a given state; a predicate *fgCmpnntIsAGoodChild* holds when the foreground component has its parent application and both of them, the component and the parent have each other's information. *fgCmpnntIsInstalled* is valid when the parent application is an installed application. *cmpnntAppIsProper* is satisfied when the parent application requested a set of permissions, and the permissions were granted.
3) *SecureApp* is valid when all installed applications in a given state request and are granted the permissions they need.

Combining those conditions, we define the security property. The notion of security is defined by that property. In other words, if the security property holds at a state, we consider the state to be secure. In the verification process, the following property will be tested at every transition between states, in order to confirm whether system operations do not violate the security conditions.

Definition *SecureState (s:State)*: Prop :=
SecureAuth s $\wedge$ *SecureCmpnnt s* $\wedge$ *SecureApp s*.

As an example, let us unfold the *cmpnntAppIsProper* which is a part of the *SecureCmpnnt*. It tests whether the parent application of the foreground component has proper permissions:

Definition *cmpnntAppIsProper* (*s*:*State*) : Prop :=
$\forall$ (*app*:*AppPackage*) (*cmp*:*Cmpnnt*), *state_fgcmpnnt s* = *ExT cmp* $\rightarrow$ *app* = *cmpnnt_papp cmp* $\rightarrow$ *appIsProper s app*.

The *appIsProper* tests whether the given application is proper using *pIsRequested* and *pIsGranted*. The two functions are defined using the previously described primitive predicates, as their names delineate. The following shows the definition of the *pIsGranted*:

Definition *pIsGranted* (*s*:*State*) (*appid*:*AppID*) (*p*:*Permission*) : Prop := *state_perm_tbl s appid p*.

C. Transition

The execution of an operation results in state changes in the system. The operations of our concern are as follows: install application, start a component execution, terminate a component execution, and remove an application. Those operations affect the security of the system by changing the state information: the foreground component, the package table, and the permission table. We specify the four operations in terms of the precondition and the postcondition. A precondition describes the conditions that have to be satisfied at state *s*, and a postcondition explains changes between s and another state *s'*. Let us call those states pre-state and post-state, respectively.

The below example specification below is of the *terminate* operation, which has simpler descriptions than others.

Definition *Pre_terminate* (*s*:*State*) (*cmpid*:*CmpnntID*) : Prop := $\exists$ *cmp*:*Cmpnnt*, *state_fgcmpnnt s* = *ExT cmp* $\wedge$ (*cmpid* = *cmpnnt_id cmp*).

Definition *Post_terminate* (*s s'*:*State*) (*cmpid*:*CmpnntID*): Prop :=
state_package_tbl s = *state_package_tbl s'* $\wedge$
state_perm_tbl s = *state_perm_tbl s'* $\wedge$
state_fgcmpnnt s' = *NoneT*.

Since *terminate* finishes the given foreground component, the operation is executed only when the given component is running in the foreground. The operation *terminate* does not change installed applications and given permissions, but the foreground component will be removed.

The specification of the operation *install* is a little longer:

Definition *Pre_install* (*s*:*State*)
(*newapp*:*AppPackage*) (*auth*:*Authorization*) : Prop :=
($\forall$ (*app*:*AppPackage*), ($\forall$ *p*:*Permission*, *state_perm_tbl s* (*app_id app*) *p* $\rightarrow$ *permreq_ineed* (*app_permrqst app*) *p*)) $\wedge$
($\forall$ *cmp*:*Cmpnnt*, *state_fgcmpnnt s* = *ExT cmp* $\rightarrow$ *cmpnnt_papp cmp* $\neq$ *newapp*) $\wedge$
($\forall$ *p*:*Permission*, *auth_by_user auth* (*app_id newapp*) *p*
$\rightarrow$ *permreq_ineed* (*app_permrqst newapp*) *p* $\rightarrow$
~(*state_perm_tbl s* (*app_id newapp*) *p*)) $\wedge$
($\forall$ *app*:*AppPackage*, *state_package_tbl s app* $\rightarrow$ (*app_id app* $\neq$ *app_id newapp*))

Definition *Post_install_user_success* (*s s'*:*State*)
(*newapp*:*AppPackage*) (*auth*:*Authorization*) : Prop :=
ud = *ok* $\wedge$ *state_fgcmpnnt s* = *state_fgcmpnnt s'* $\wedge$
($\forall$ *app*:*AppPackage*, *app_id app* $\neq$ *app_id newapp*) $\rightarrow$(*state_package_tbl s app* = *state_package_tbl s' app*) $\wedge$
(*state_perm_tbl s* (*app_id app*) = *state_perm_tbl s'* (*app_id app*))) $\wedge$
($\forall$ (*someapp*:*AppPackage*) (*p*:*Permission*), (*app_id newapp* = *app_id someapp*) $\rightarrow$ *state_perm_tbl s'* (*app_id someapp*) *p*) $\wedge$
($\forall$ (*p*:*Permission*), *state_perm_tbl s'* (*app_id newapp*) *p* $\rightarrow$ *auth_by_user auth* (*app_id newapp*) *p*) $\wedge$
($\forall$ (*someapp*:*AppPackage*) (*p*:*Permission*), (*app_id newapp* = *app_id someapp*) $\rightarrow$ *pIsRequested someapp p*) $\wedge$
($\forall$ (*p*:*Permission*), *state_perm_tbl s'* (*app_id newapp*) *p* $\rightarrow$ *auth_by_user auth* (*app_id newapp*) *p* $\rightarrow$ *permreq_ineed* (*app_permrqst newapp*) *p*)$\wedge$
($\forall$ *cmp*:*Cmpnnt*, *state_fgcmpnnt s'* = *ExT cmp* $\rightarrow$ (*app_id* (*cmpnnt_papp cmp*)) $\neq$ (*app_id newapp*)).

The precondition of *intall* states the following conditions: granted permissions to all existing applications are requested beforehand according to the manifest information. If some permissions are authorized for a new application, then the permissions are requested by the application, and there is also no relevant authorization in the current state. The existing applications cannot be the new application. The new application does not exist in the package table of the current state and cannot be the parent of the currently running component.

The postcondition of the *install* are distinctly defined by whether the installation was successful or not. Whereas an unsuccessful *install* execution does not change any information, a successful execution changes the permission table and the package table at post-state *s'*. A new application is installed at *s'* with a set of authorized permissions, and the permissions must have been requested by the application. Other applications stay residing the same as the pre-state and so do their granted permissions. The new application cannot be the parent of the foreground component at *s'*, of course.

Now, we define state transition by composing the specified operations. In a state with a given operation, the transition occurs if the precondition is satisfied and results in yielding a new state as the postcondition describes. If the precondition does not hold, the system stays at the state. In the following definition of the transition, *Operation* is an inductive type composed of the specified operations, so *op* will be pattern-matched.

Inductive *TransFunc* (*s*:*State*) (*op*:*Operation*) (*s'*:*State*): Prop :=
| *transMove* : *PreCondition s op* $\rightarrow$ *PostCondition s s' op* $\rightarrow$ *TransFunc s op s'*
| *transStay* : $\neg$*PreCondition s op* $\rightarrow$ *s* = *s'* $\rightarrow$ *TransFunc s op s'*.

III. Verification

The objective of the verification is to confirm whether all transitions satisfy the security property occurring over states. In order to do that, we have to prove the security conditions hold at poststate *s'* for every operation with the following assumptions:

- The security conditions hold at the prestate *s*

- The precondition and the postcondition are satisfied as well.

Lightened by the stepwise verification strategy in [9], we start with the proofs of basic mathematical theories, and then divide the security theorem into small lemmas. Some small lemmas specify properties that are invariant between *s* to *s'*, and the other type of lemmas describe security conditions that hold at the state *s'*. Having those lemmas proved, we finally verify the security theorem.

We show two small lemmas below. The first one is Lemma (1), which states that the *cmpnntAppIsProper* holds at *s'* after the operation *terminate*. The *terminate*-related lemmas are proved simply and shortly, although short precondition and postcondition do not always promise short proofs.

Lemma *cmpnntAppIsProper_termsec* :
cmpnntAppIsProper s'. (1)

Proof: *cmpnntAppIsProper s'* says that if the foreground component exists, its parent application should properly exists at *s'*, formally (a): *state_fgcmpnnt s' = ExT Hcmp* $\rightarrow$ *Happ = cmpnnt_papp Hcmp* $\rightarrow$ *appIsProper s' Happ* (In (a), we used Hcmp and Happ, instead of $\forall$cmp:Cmpnnt and $\forall$app:AppPackage, by the Universal Introduction rule). A properly installed application entails that the application was granted permissions as requested, as the *appIsProper* characterizes. On the other hand, as the postcondition specifies, the *terminate* results in finishing the foreground component at *s'*, formally (b): *state_fgcmpnnt s' = NoneT*. We replace *state_fgcmpnnt s'* in (a) using (b), and get *NoneT = ExT Hcmp*. By applying Lemma (2) (shown below), the equality yields the Falsity in the hypothesis of (a), so we can get *appIsProper s' Happ* by the Falsity elimination rule. □

Lemma *state_fgcmpnnt_inverse* :
$\forall$ (*cmp*:*Cmpnnt*), *ExT cmp* $\neq$ *NoneT*. (2)

The second small lemma is Lemma (3), which states that, after the operation *install*, the parent application of the foreground component should be an installed one.

Lemma *fgCompnntIsInstalled_installsec_inv_small*:
$\forall$ (*app*:*AppPackage*) (*cmp*:*Cmpnnt*), *state_fgcmpnnt s = ExT cmp* $\rightarrow$ *app = cmpnnt_papp cmp* $\rightarrow$ *state_package_tbl s app = state_package_tbl s' app*. (3)

Proof: First, we introduce Hcmp for $\forall$cmp:Cmpnnt by the Universal Introduction, and then take *state_fgcmpnnt s = ExT Hcmp* as a local assumption by the Implication Introduction. Let us call the local assumption *HcmpEx*. Then, we have to consider two distinct cases where the *install* operation finishes successfully (Case 1) or not (Case 2). Let us consider Case 1 first. We do case analysis again on the application identifiers. Since the equality of the type *AppID* is decidable for all applications, we can have two distinct cases; Case 1-1:(*app_id app* $\neq$ *app_id newapp*) and Case 1-2:(*app_id app = app_id newapp*), for all *app* and *newapp* which is the application to be installed. The Case 1-1 is easily solved. Other than the *newapp*, all the existing applications at *s* remain installed the same at *s'*. This condition is reflected in the postcondition *Post_install_user_success*. By doing the conjunction elimination and the temporal assumption on the postcondition, we obtain *state_package_tbl s app = state_package_tbl s' app*. Consequently, the goal is proved. In the Case 1-2, we already know that (a): *app_id app* $\neq$ *app_id newapp)* $\rightarrow$ *(state_package_tbl s app = state_package_tbl s' app)*. We also know that the *install* operation does not change the foreground component information, formally: *(state_fgcmpnnt s' = state_fgcmpnnt s)*. Using this equality, the *HcmpEx* can be rewritten to *state_fgcmpnnt s' = ExT Hcmp*. On the other hand, the postcondition of the *install* states that $\forall$ *cmp:Cmpnnt, state_fgcmpnnt s' = ExT cmp* $\rightarrow$ *(app_id (cmpnnt_papp cmp))* $\neq$ *(app_id newapp)*. Applying the *HcmpEx*, we obtain the consequent. And then, we get the package table invariability from (a) by the Modus Ponens. Now, let us tackle Case 2. The postcondition *Post_install_user_fail* describes no information changes. By trivial, *state_package_tbl s app = state_package_tbl s'*. □

Based on a number of lemmas we prove, the following theorem *SecureMigration* can be obtained. The theorem states that every operation we specified results in a transition from a secure state to another state, and then the next state is also secure. We omit the proof of the theorem here. It is simply a process of applying all proved lemmas.

Theorem *SecureMigration*:
SecureState s $\rightarrow$ *TransFunc s op s'* $\rightarrow$ *SecureState s'*.

Finally, like as the BST, we can claim that the specified permission-based Android system of this paper is secure in inductively generated states from an initial state, where the initial state s_0 is secure and every action *TransFunc* satisfies the security property. We proved that the *TransFunc* operations preserve the security property, which is the *SecureMigration*.

IV. Discussion

The Coq system has been applied in diverse contexts for formal analysis of security. Among the prior studies, our work is rather inspired by Cristiá's work on a secure UNIX file system formalization [9] and Béguelin's J2ME MIDP Modeling [11] [1]. We took idea of the BST-based state-machine modeling, transition representation, and analysis strategy from Cristiá [9], and chose a set of operations being guided by Béguelin et al [11]. Based on those approaches, we contributed to specify the permission mechanism of the Android system having the abstracted component structure included, specified security-related properties and operations, and verified the security theorem in the context of our specification. This result can be used to evaluate the logical basis of Android security.

Note that our specifications do not cover every feature of

[1]Specification and verification codes of Cristiá's work can be found at `http://coq.inria.fr/contribs-eng.html`. Béguelin's work is supposed to be found at the link in the paper, but it seems the link is not available for the present (lastly checked on 03.20.2009).

the Android system. As an example, we do not reflect the application signing concept. Before granting permissions, the Android system checks the certificate of an application as well as the user response. Signature-based permissions enable an application to expose its code and data to the applications with specific certificates. Moreover, if two or more applications are owned by the same person and signed with the same certificate, they share the user ID. We do not have detailed specifications about the scheme yet. We simply specified the *Authorization* and represented what is allowed to whom, instead of covering why it is allowed. Due to the gap between the abstracted representation and the concrete implementation, when a security failure is found in our formal model, the model cannot say which exact implemented mechanism causes the failure. That is, it is not always easy to figure out why the security failure happens. However, it is still clear that the formal model reveals logical errors and shows fundamental vulnerabilities that must be handled in the concrete implementation.

The current specification can be utilized in various ways. The permission mechanism can be compared with other security system formalizations, like the J2ME MIDP formalization [11] that has own notion of sandbox, permission, and protection domain. As a result, similarities between the Android framework and the MIDP could be formally investigated.

The security of external codes can also be tested when the codes accompany annotated pre-/postconditions, similar to the JML tools [12]. Furthermore, if we provide verified APIs and specifications as templates, executables and their specifications could be generated at once, like Bhargavan's approach [13].

V. Conclusion and Future Work

For the purpose of establishing a logical foundation for security analysis, we specified the permission mechanism of the Android system. Using the state machine-based approach, we specified the system elements, characterized security conditions in terms of authorization, components and applications, and described a set of operations and exemplified the *terminate* and the *install*. We also verified the specified system operates preserving the security property. We expect the security theorem of this paper can be used to assure the security of the Android system. In our following work, the coverage of the specification would be improved, by describing other features of the Android and by deepening the specification.

References

[1] Google, Inc., "What is Android?" [Online]. Available: http://code.google.com/android/what-is-android.html 03.20.2009.

[2] "xda-developers." [Online]. Available: http://forum.xda-developers.com/showthread.php?t=442480 03.20.2009.

[3] C. Miller, "Pulling a John Connor: Defeating Android." [Online]. Available: http://www.shmoocon.org/presentations-all.html#johnconnor 03.20.2009.

[4] A. Greenberg, "More security angst for Android," Forbes, Feb 2009. [Online]. Available: http://www.forbes.com/2009/02/05/google-android-security-technology-security_0205_android.html 03.20.2009.

[5] D. Danchev, "Open source software security improving," ZDNet, May 2008. [Online]. Available: http://blogs.zdnet.com/security/?p=1182 03.20.2009.

[6] D. E. Bell and L. J. LaPadula, "Secure computer systems: mathematical foundations," MITRE Technical Report 2547, Volume 1, Mar 1973.

[7] TypiCal Project, "The Coq proof assistant." [Online]. Available: http://coq.inria.fr/ 03.20.2009.

[8] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[9] M. Cristiá, "Formal verification of an extension of a secure, compatible UNIX file system," in *Master's thesis, Instituto de Computacin, Universidad de la República*, 2002.

[10] Google, Inc., "Security and permissions." [Online]. Available: http://code.google.com/android/devel/security.html 03.20.2009.

[11] S. Z. Béguelin, G. Betarte, and C. Luna, "A formal specification of the MIDP 2.0 security model," in *The fourth International Workshop of Formal Aspects in Security and Trust*, 2006, pp. 220–234.

[12] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 3, pp. 212–232, 2005.

[13] K. Bhargavan, C. Fournet, and A. D. Gordon, "Verified reference implementations of WS-Security protocols," in *Web Services and Formal Methods*, ser. LNCS, vol. 4184. Springer, 2006, pp. 88–106.