A Framework for the Analysis of Access Control Models for Interactive Mobile Devices

Juan Manuel Crespo[1,2], Gustavo Betarte[3], and Carlos Luna[3]

[1] FCEIA, Universidad Nacional de Rosario, Argentina
[2] IMDEA Software, Madrid, Spain
juanmanuel.crespo@imdea.org
[3] Instituto de Computación, Universidad de la República, Uruguay**
{gustun,cluna}@fing.edu.uy

Abstract. The Java Micro Edition platform (JME), a Java enabled technology, provides the Mobile Information Device Profile (MIDP) standard that facilitates applications development and specifies a security model for the controlled access to sensitive resources of the device. The model builds upon the notion of protection domain, which in turn can be grasped as a set of permissions. An alternative model has been proposed that extends MIDP's by introducing permissions with multiplicities and adding flexibility to the way in which permissions are granted by the user of the device and used by the applications running on it. This paper presents a framework, formalized using the proof-assistant Coq, suitable for defining and comparing the access control policies that can be enforced by (variants of) those security models and to prove desirable properties they should satisfy. The proofs of some of those properties are also stated and discussed in this work.

Keywords: Access control models, mobile devices, formal proofs.

1 Introduction

Devices such as cell phones or personal digital assistants often have access to sensitive personal information and are subscribed to paid services in order to communicate with other entities. In addition to this, users are able to download and install applications from unreliable sources at their will. Java Micro Edition (JME) [10] is a version of the Java platform targeted at resource-constrained devices which comprises two kinds of components: configurations and profiles. The Mobile Information Device Profile (MIDP) [7, 6] defines an application life cycle, a security model and APIs that offer the functionality required by mobile applications, including networking, user interface, push activation and persistent local storage. Many mobile device manufacturers have adopted MIDP

** This work was partially funded by the Project PDT 63/118 STEVE, DINACYT, Uruguay

since the specification was made available. A formal specification of the JME-MIDP 2.0 security model developed using the proof-assistant Coq is presented and described in detail in [12].

In [2], a security model for interactive mobile devices is put forward which can be grasped as an extension of the JME-MIDP model. The work presented in that paper has focused in developing a formal model for studying, in particular, interactive user querying mechanisms for permission granting for application execution on mobile devices. Like in the MIDP case, the notion of permission is central to this model and MIDP is extended by introducing permissions with multiplicities and by adding flexibility to the way in which permissions are granted by the user and used by the applications.

One of the main objectives of the work reported here has been to build a framework which would provide a formal setting to define and analyse the permission models defined by MIDP and the one presented in [2]. This framework, which is formally defined using the Calculus of Inductive Constructions [4, 5], adopts, with variations, most of the security and programming constructions defined in [2]. The principal difference is that most of those constructions are now parameterized by a permission grant policy. In this paper it is shown how the framework can be used to define a type of permission grant policies and to represent the four user permission modes of MIDP and the policies defined in [2] as objects of that type. The paper also presents the definition of an order relation, based on a notion of safe programs, which can be used to perform a comparative analysis of grant policies. In particular, it is described the proof, which has been constructed using the proof-assistant Coq [11], of the theorem that establishes how the grant policies mentioned above are related according to the defined order. The complete definition of the framework as well as the statement and proof of the properties are available in `www.fing.edu.uy/inco/grupos/mf/projects/PermModel/ACM-Coq.zip`.

The structure of the rest of the paper is organized as follows. Section 2 provides a brief account of the permission models that are the object of the analysis presented in this work. Section 3 describes the formal setting and the security concepts that constitute the basis of the access control mechanisms used to define those models. In section 4 the grant policies and the order relation are formally defined. A theorem that establishes the conditions that suffice to prove that two grant policies are in the order relation is also discussed. In section 5 it is presented the proof of the theorem that establishes how the concrete permission grant policies studied in this work are related. Section 6 concludes and describes further work.

2 Security Models for Interactive Mobile Devices

This section provides a brief account of the permission models that are the object of the analysis presented in this work.

2.1 The JME–MIDP Security Model

In MIDP, applications (MIDlets) are packaged and distributed as suites. A MIDlet suite can contain one or more MIDlets and is distributed as two files, an application descriptor file and an archive file that contains the actual classes and resources. A suite that needs access to protected APIs or functions must declaratively request the corresponding permissions in its descriptor. MIDlet suites may request permissions either as required or as optional. In the first version of MIDP [7], any application not installed by the device manufacturer or a service provider runs in a sandbox that prohibits access to security sensitive APIs or functions of the device. Although this sandbox security model effectively prevents any rogue application from jeopardising the security of the device, it is excessively restrictive and does not allow many useful applications to be deployed after issuance of the device.

Version 2.0 of MIDP [6] introduces a new security model based on the concept of protection domain. A protection domain can be grasped as an abstraction of the execution context of an application, and it determines the access rights to the protected functions of the device. Each sensitive API or function on the device may define permissions in order to prevent it from being used without authorisation. A protection domain consists of both a set of permissions which are granted unconditionally, without intervention of the device's user (called **allowed** permissions), and a set of permissions which require authorisation from the user (called **user**). Permissions may be granted by the user to an active MIDlet suite in either of the following three modes:

- **blanket**: the permission is granted for as long as the application remains installed in the device
- **session**: the permission is granted for as long as the application is running
- **one-shot**: the permission is granted for only one use of the function

An installed MIDlet suite is bound to a unique protection domain. Untrusted MIDlet suites are bound to a protection domain with permissions equivalent to those in a MIDP 1.0 sandbox. Trusted MIDlet suites may be identified by means of cryptographic signatures and bound to more permissive protection domains. This security model enables applications developed by trusted third parties to be downloaded and installed after issuance of the device without compromising its security.

The set of permissions effectively granted to a suite is determined from its protection domain, the permissions the suite request in its descriptor and the authorisations granted by the user.

For a more detailed description of the mechanisms defined by the security model the reader is referred to [7, 6]. A formal specification of the MIDP 2.0 security model is presented in [12] and a certified access controller for the enforcement of policies admitted by that model is described in details in [9].

2.2 An Alternative Model

In [2], a security model for interactive mobile devices is put forward which can be grasped as an extension of that of MIDP. The work presented in that paper has focused in developing a formal model for studying, in particular, interactive user querying mechanisms for permission granting for application execution on mobile devices. Like in the MIDP case, the notion of permission is central to this model. A generalisation of the one-shot permission described above is proposed that consists in associating to a permission a multiplicity which states how many times that permission can be used.

The proposed model has two basic constructs for manipulating permissions: **grant** and **consume**. The grant construct models the interactive querying of the user, asking whether he grants a particular permission with a certain multiplicity. The consume construct models the access to a sensitive function which is protected by the security police, and therefore requires (consumes) permissions.

A semantics of the model constructs is proposed as well as a logic for reasoning on properties of the execution flow of programs using those constructs. The basic security property the logic allows to prove is that a program will never attempt to access a resource for which it does not have a permission. The authors also provide a static analysis that makes it possible to verify that a particular combination of the grant-consume constructs does not violate that security property. For developing that kind of analysis the constructs are integrated into a program model based on control-flow graphs. This model has also been used in previous work on modelling access control for Java, see for instance [8, 3].

One of the main objectives of the work that is being reported here, has been to build a framework which would provide a formal setting to define the permission models defined by MIDP and the one presented in [2] (and variants of it) in an uniform way and to perform a formal analysis and comparison of those models. This framework, which is formally defined using the Calculus of Inductive Constructions [4, 5], adopts, with variations, most of the security and programming constructions defined in [2]. In particular it has been modified so as to be parameterized by permission granting policies, while in the original work this relation is fixed.

3 A Framework for Access Control Modeling

This section introduces the formal setting used to define the security concepts that constitute the basis of certain access control mechanisms, to proceed then to described how those mechanisms are used to define the permission granting models which are object of analysis of this work.

3.1 The formal language used

Standard notation is used for equality and logical connectives ($\wedge, \vee, \neg, \rightarrow, \forall, \exists$). Anonymous functions and predicates use standard lambda notation (e.g. $\lambda\ (x :$

$T) \;.\; x, \lambda\ (x : nat) \;.\; x > 10)$. In case there is more than one binder, the standard abbreviation $\lambda\ (x : nat)\ (y : nat) \;.\; x + y$ is used.

An inductive relation I is defined by giving introduction rules of the form:

$$\frac{P_1 \ldots P_m}{I\ x_1 \ldots x_n}$$

where the variables occurring free are implicitly universally quantified. Similarly, inductive types are defined by giving constructors in the following form:

$$\begin{aligned} T \stackrel{def}{=} & \mid C_1 : A_{1,1} \to \ldots A_{1,n_1} \to T \\ & \vdots \\ & \mid C_m : A_{m,1} \to \ldots A_{m,n_m} \to T \end{aligned}$$

where $C_1 \ldots C_n$ are the constructors of T.

A (dependent) record type R is defined as follows:

$$R \stackrel{def}{=} \{field_1 : A_1, \ldots, field_n : A_n\}$$

This definition generates a non-recursive inductive type with a single constructor $mkR : A_1 \to \ldots A_n \to R$ and projection functions $field_i : R \to A_i$. Application of projection functions is abbreviated using dot notation: $field_i\, r = r.field_i$. When the type is clear for the context $\langle x_1, \ldots, x_n \rangle$ is written instead of $mkR\ x_1, \ldots, x_n$.

In the fomalization developed it has been used inductive types that have *valid* and *invalid* cases. In the rest of this paper it is adopted the convention that a type with the same name but prefixed with valid is the type consisting only of the valid cases. Which are the valid constructors is usually clear from the context, otherwise it is specified.

The following parametric inductive types are assumed to be predefined:

- $option\ T$ with constructors $None : option\ T$ and $Some : T \to option\ T$,
- finite lists over T, $list\ T$. The empty list is denoted by [] and the (infix) constructor that inserts an element a at the front of a list s is denoted by $a \triangleright s$. Finite snoc lists over T, $snocList\, T$, that is, lists that are constructed by inserting elements at the back, are also used. [] denotes the empty snoc list and $s \triangleleft\ a$ denotes the insertion of an element a at the back of the snoc list s.

3.2 Permissions

Every (controlled) resource of the device is given a type. Let $ResType$ be the set of types of resources. If rt is a resource type, $Resources\ rt$ and $Actions\ rt$ define the set of resources of type rt available on the device and the actions that can performed over them, respectively. The permissions of a resource type are defined as follows:

$$
\begin{aligned}
PermRes\ \ (rt : ResType) &\stackrel{def}{=} \\
&\mid valid : list\ (Resources\ rt) \to list\ (Actions\ rt) \to PermRes\ rt \\
&\mid invalid : PermRes\ rt
\end{aligned}
$$

That is, given a resource type rt, an object of type $PermRes\ rt$ is a set (represented by a list) of actions and resources over rt, or the constant *invalid*. A relation $\sqsubseteq_{PermRes}$ is defined by applying set inclusion component-wise. This relation defines a lattice structure where *invalid* is the bottom element $\bot_{PermRes}$ and $\sqcup_{PermRes}$ a lub operator which is obtained applying set union component-wise.

As already mentioned, a notion of multiplicity of granted permission is introduced in [2]. A multiplicity is defined to be either a natural number, a special value ∞ that denotes an irrestricted permission, or an error value $\bot$. A type Mul is defined:

$$
\begin{aligned}
Mul \stackrel{def}{=} &\mid \bot\ \ : Mul \\
&\mid val : nat \to Mul \\
&\mid \infty\ \ : Mul
\end{aligned}
$$

It is straightforward to see that a lattice can be constructed over Mul with $\bot$ and ∞ as the bottom and top elements, respectively. The obvious extensions of functions and predicates defined over naturals to functions and predicates over Mul, such as $\sqsubseteq_{Mul}$, $+_{Mul}$, $-_{Mul}$, $pred_{Mul}$, are also defined.

An accumulated permission for a resource type is comprised of two components: the set of resources and actions allowed and a multiplicity. One such permission (of resource type rt) is then grasped as an object of the following record type:

$$
PermMul\ (rt : ResType) \stackrel{def}{=} \{permRes : PermRes\ rt; mul : Mul\}
$$

The lattice of permissions of a resource type can be obtained by defining the order $\sqsubseteq_{PermMul}$:

$$
\begin{aligned}
pm_1 \sqsubseteq_{PermMul} pm_2 \stackrel{def}{=}\ & pm_1.permRes \sqsubseteq_{PermRes} pm_2.permRes \\
\wedge\ & pm_1.mul \sqsubseteq_{Mul} pm_2.mul
\end{aligned}
$$

where pm_1 and pm_2 are objects of type $PermMul\ rt$. Now, the permission state of the device is defined. One such state is ultimately a mapping that associates a permission to each resource type. Therefore, it is defined as the following dependent function type:

$$
Perm \stackrel{def}{=} \forall (rt : ResType), PermMul\ rt
$$

It is said that two permissions p_1 and p_2 are (extensionally) equal if for every resource type rt it holds that $p_1\ rt = p_2\ rt$.

An order $\sqsubseteq_{Perm}$ can be defined as the product-wise extension of $\sqsubseteq_{PermMul}$ as follows:

$$p_1 \sqsubseteq_{Perm} p_2 \stackrel{def}{=} \forall(rt : ResType), (p_1\ rt) \sqsubseteq_{PermMul} (p_2\ rt)$$

In order to model the operations that affect the state of the permissions an *update* function is introduced:

$$update\ (p : Perm)(rt : ResType)(pres : PermRes\ rt)(m : Mul) : Perm$$

The intended (and formalized) behaviour of this function is that of an usual store updating operator: the permission state remains unchanged for every resource type different from rt, and for rt yields $\langle pres, m\rangle$.

If rt is a resource type and p a permission state, then the following inductive relation *Error* is defined

$$\frac{(p\ rt).permRes = invalid\ rt}{Error\ p} \qquad \frac{(p\ rt).mul = \bot}{Error\ p}$$

The intuition is that an error situation may occur when either there is an attempt to perform an action over a resource of type rt and no valid permission is associated to it (first rule) or when there are no granted permissions for that resource (second rule).

3.3 Programs

A program in, among others, [2, 1] is represented by a control-flow graph that captures the manipulations of permissions and the handling of method calls and returns as well as exceptions.
A control-flow graph is a tuple $G = (NO, EX, KD, TG, CG, EG, n_0)$ where:

- NO is the set of nodes of the graph (one for each instruction),
- EX is the set of exceptions,
- KD is a function of type $KD : NO \rightarrow Instr$ that associates each node to an instruction,
- $TG : NO \rightarrow NO \rightarrow Prop$ is the propositional function that characterizes the set of intra-procedural edges (i.e. $n_1\ TG\ n_2$ if control can be transferred from instruction at node n_1 to instruction at node n_2 within the currect procedure),
- CG is the set of inter-procedural edges (which can be used to capture dynamic method calls),
- $EG : EX \rightarrow NO \rightarrow NO \rightarrow Prop$ are the intra-procedural exception edges,
- $n_0 : NO$ is the graph entry node.

The instructions are formally defined in the framework by means of the following inductive type:

$$
\begin{aligned}
Instr \stackrel{def}{=} & \\
& \mid\ Grant : \forall (rt : ResType), validPermRes\ rt \to MulValid \to Instr \\
& \mid\ Consume : \forall (rt : ResType), validPermRes\ rt \to Instr \\
& \mid\ Call : Instr \\
& \mid\ Return : Instr \\
& \mid\ Throw : EX \to Instr
\end{aligned}
$$

where $MulValid$ is the type of valid multiplicities, that is, different from the multiplicity $\bot$. The definition of the operational semantics of programs strongly depends on those of the permission granting and consumption mechanisms. They are briefly discussed and described in what follows.

In [2] two variants are discussed concerning the effect of the update operation after a permission has been granted: either the permissions before the update instruction are discarded or they are accumulated. At a first sight these *permission granting policies* have advantages and drawbacks. Furthermore, independently of this particular discussion, it is at this point that the permission model proposed by the authors introduces a generalization with respect to that of MIDP: the multiplicity of a permission. One of the main objectives of the work presented here has been to design a framework that would make it possible to provide a uniform setting where those different permissions models could be formally defined and compared. To that end, the constructions defined to provide semantics to the computational behaviour of the programs as well as to reason over that behaviour have been parameterized by permission granting policies. One such parameter shall be formally represented by an object of the following type:

$$
\begin{aligned}
grantPolicy \stackrel{def}{=} & \forall (rt : ResType), \\
& validPermRes\ rt \to NZMulValid \to Perm \to Perm
\end{aligned}
$$

where an object of type $NZMulValid$ is a valid multiplicity constructed with a non-zero natural.

As to the consumption of permissions, the following is the definition of the consume operation:

$$
\begin{aligned}
consume\ \ & (rt : ResType)(pr : validPermRes\ rt)(p : Perm) : Perm \stackrel{def}{=} \\
& if\ (pr\ \sqsubseteq_{PermRes}\ (p\ rt).permRes) \\
& then\ update\ p\ rt\ (p\ rt).permRes\ (pred_{Mul}\ (p\ rt).mul) \\
& else\ update\ p\ rt\ (invalid\ rt)\ (pred_{Mul}\ (p\ rt).mul)
\end{aligned}
$$

The consume operation is monotonic on permissions. This is stated (and proved) in the following lemma:

Lemma 1.

$$\begin{array}{l} Lemma\ consumeMon: \\ \quad \forall (rt : ResType)(pr : validPermmRes\ rt)(p\ p' : Perm), \\ \quad p \sqsubseteq_{Perm} p' \rightarrow (consume\ rt\ pr\ p) \sqsubseteq_{Perm} (consume\ rt\ pr\ p') \end{array}$$

Following [2] the small-step operational semantics of a control-flow graph has been defined basically as a relation that defines transitions between states consisting of a standard control-flow stack of nodes enriched with the permissions held at that point in the execution. This definition has been extended by making it depend on a permission granting policy g. Formally, it has been defined as an inductive propositional function $\rightsquigarrow_g$ whose rules are depicted in Fig. 1. An im-

$$\frac{KD\ n = Grant\ rt\ pr\ m \quad TG\ n\ n'}{(n \triangleright s)\ None\ p\ \rightsquigarrow_g (n' \triangleright s)\ None\ (g\ rt\ pr\ m\ p)}$$

$$\frac{KD\ n = Consume\ rt\ pr \quad TG\ n\ n'}{(n \triangleright s)\ None\ p\ \rightsquigarrow_g (n' \triangleright s)\ None\ (consume\ rt\ pr\ p)}$$

$$\frac{KD\ n = Call \quad CG\ n\ n'}{(n \triangleright s)\ None\ p\ \rightsquigarrow_g (n' \triangleright n \triangleright s)\ None\ p} \qquad \frac{KD\ r = Return \quad TG\ n\ n'}{(r \triangleright n \triangleright s)\ None\ p\ \rightsquigarrow_g (n' \triangleright s)\ None\ p}$$

$$\frac{KD\ n = Throw\ ex \quad EG\ ex\ n\ h}{(n \triangleright s)\ None\ p\ \rightsquigarrow_g (h \triangleright s)\ None\ p} \qquad \frac{KD\ n = Throw\ ex \quad \forall (h : NO), \neg EG\ ex\ n\ h}{(n \triangleright s)\ None\ p\ \rightsquigarrow_g (n \triangleright s)\ (Some\ ex)\ p}$$

$$\frac{\forall (h : NO), \neg EG\ ex\ n\ h}{(t \triangleright n \triangleright s)\ (Some\ ex)\ p\ \rightsquigarrow_g (n \triangleright s)\ (Some\ ex)\ p} \qquad \frac{EG\ ex\ n\ h}{(t \triangleright n \triangleright s)\ (Some\ ex)\ p\ \rightsquigarrow_g (h \triangleright s)\ None\ p}$$

Fig. 1. Semantics of instructions

portant property of this semantics is that it is non-intrusive, that is to say, the permission state does not interfere with execution. In other words, a transition will not be blocked by the absence of permissions. This is formally stated, and proved, in the following lemma:

Lemma 2.

$$\begin{array}{l} Lemma\ nonIntrusive: \\ \quad \forall (g : grantPolicy)(s\ s' : list\ NO)(ex\ ex' : option\ EX)(p\ p' : Perm), \\ \quad s\ ex\ p\ \rightsquigarrow_g s' ex' p' \rightarrow \forall (p : Perm), (\exists (p' : Perm), s\ ex\ p\ \rightsquigarrow_g\ s'\ ex'\ p') \end{array}$$

3.4 Traces

In [2] global results on the execution of programs are expressed on traces, which in turn are defined in terms of the operational semantics described above (instantiated for a particular grant policy) as follows: *a partial trace of a control-flow graph is a sequence (of type snocList $(NO, option\ EX)$) of nodes* $[] \triangleleft \langle n_0, None \rangle \triangleleft$

$\langle n_1, e_1\rangle \triangleleft \cdots \triangleleft \langle n_k, e_k\rangle$ *such that for all* $0 \leq\ i < k$ *there exists* $\rho, \rho' \epsilon\ Perm$, $s, s' \epsilon\ (list\ NO)$ *and verifying* $n_i \triangleright s, e_i, \rho \rightsquigarrow n_{i+1} \triangleright s', e_{i+1}, \rho'$.

The stacks s and s' in the above definition are existentially quantified because they are not defined to be components of the elements of a trace. This quantification however induces a loss of information w.r.t. the operational semantics. An example[4] should clarify this situation. Consider the control-flow graph:

$$NO = \{A, B, C, D\},\ TG = \{(B, C), (C, D)\},\ EX = CG = EG = \{\},\ n_0 = A$$
$$KD = \{(A, Return), (B, x), (C, Consume\ rt\ y), (D, Return)\}$$

where $x : Instr$, $rt : ResType$, $y : validPermRes\, rt$, and with initial permission $p_{init} = \lambda\ (rt : ResType)\ .\ \langle (valid\ rt\ []\ []), (val\ 0)\rangle$. Fig. 2 depicts the control-flow graph in question. From this definition it can be noticed that

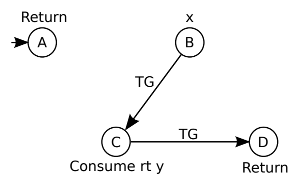

Fig. 2. Control-flow graph example

$[] \triangleleft \langle A, None\rangle$ is the only admissible trace yielding a valid permission state. According to the definition of partial trace stated above, the object $([] \triangleleft \langle A, None\rangle \triangleleft \langle C, None\rangle \triangleleft \langle D, None\rangle)$ is admitted as a partial trace of the defined control-flow graph. This trace can be built using the transition rules for the *Consume* and *Return* instructions (see Fig. 1). However, this latter trace yields an error situation, because the transition from node C to node D attempts to consume a not available permission.

The definition of program execution traces that are proposed in the framework presented here remedies the situation described above by including the node stack as a component of the elements of the trace. This is formally represented by the following type: $Trace \stackrel{def}{=} snocList\ \{noT : NO, stT : list\ NO, exT : option\ EX\}$.

The notion of parameterized partial trace is then inductively defined over elements of type $Trace$ as follows:

$$\frac{}{PTrace_g\ []} \qquad \frac{}{PTrace_g([] \triangleleft \langle n_0, [], None\rangle)}$$

$$\frac{PTrace_g\ (tr \triangleleft \langle n, s, ex\rangle) \quad \exists (p\ p' : Perm), n \triangleright s\ ex\ p \rightsquigarrow_g n' \triangleright s'\ ex'\ p'}{PTrace_g\ (tr \triangleleft \langle n, s, ex\rangle \triangleleft \langle n', s', ex'\rangle)}$$

Let tr be a trace and g be a grant policy, if $PTrace_g\ tr$ holds then it shall be said that tr is a valid trace according to g.

[4] This example is due to Santiago Zanella

Given a trace tr and a grant policy g, the function $PermsOf_g : Perm \rightarrow Trace \rightarrow Perm$ computes the permission state resulting from the execution of the program that tr represents:

$$
\begin{array}{l}
PermsOf_g(p_{init} : Perm)(tr : Trace) : Perm \stackrel{def}{=} \\
match\ tr\ with \\
\quad |[] \Rightarrow p_{init} \\
\quad |tr' \triangleleft e \Rightarrow match\ KD\ e.noT\ with \\
\qquad |Consume\ rt\ pr \Rightarrow consume\ rt\ pr\ (PermsOf_g\ p_{init}\ tr') \\
\qquad |Grant\ rt\ pr\ m \Rightarrow g\ rt\ pr\ m\ (PermsOf_g\ p_{init}\ tr') \\
\qquad |\_ \Rightarrow PermsOf_g\ p_{init}\ tr' \\
\quad end \\
end
\end{array}
$$

Finally, given a grant policy g, a trace is said to be safe if none of its prefixes yields a faulty permission state:

$$
\begin{array}{l}
Safe_g(tr : Trace)(p_{init} : Perm) \stackrel{def}{=} \\
\qquad \forall tr' : Trace, (prefix\ tr'\ tr) \rightarrow \neg Error(PermsOf_g\ p_{init}\ tr')
\end{array}
$$

4 Permission Grant Policies

Two kinds of grant policies are analysed in [2]: given a resource type rt, one of the policies establishes that when a new permission is granted to resources of rt, all previous granted permissions are overwritten. This policy is called here $grant_{ow}$. The another policy, called here $grant_{ac}$, establishes that new granted permissions for rt are accumulated with the ones previously obtained for that resource type. These policies are formally defined as follows:

$$
\begin{array}{l}
grant_{ow} : grantPolicy \stackrel{def}{=} \\
\qquad \lambda\ (p : Perm)\ (rt : ResType)\ (pr : PermRes\ rt)\ (m : Mul)\ . \\
\qquad update\ p\ rt\ pr\ m \\
grant_{ac} : grantPolicy \stackrel{def}{=} \\
\qquad \lambda\ (p : Perm)\ (rt : ResType)\ (pr : PermRes\ rt)\ (m : Mul)\ . \\
\qquad update\ p\ rt\ (pr \sqcup_{PermRes} (p\ rt).permRes)\ (m +_{mul} (p\ rt).mul)
\end{array}
$$

The permission modes defined by MIDP are also defined below as grant policies. The $grant_{bk}$ term represents the blanket permission mode, which specifies unrestricted access to a given resource type. The one-shot permission mode, which specifies a single access to a given resource type, is represented by the term $grant_{os}$.

$$
\begin{array}{l}
grant_{bk} : grantPolicy \stackrel{def}{=} \\
\qquad \lambda\ (p : Perm)\ (rt : ResType)\ (pr : PermRes\ rt)\ (m : Mul)\ . \\
\qquad update\ p\ rt\ (pr \sqcup_{PermRes} (p\ rt).permRes)\ \infty \\
grant_{os} : grantPolicy \stackrel{def}{=} \\
\qquad \lambda\ (p : Perm)\ (rt : ResType)\ (pr : PermRes\ rt)\ (m : Mul)\ . \\
\qquad update\ p\ rt\ pr\ 1
\end{array}
$$

It should be noticed that both the allowed mode and the session permission mode specified by MIDP 2.0 can be modeled as a blanket grant policy. In the first case, the granted permission would hold for the rest of the life cycle of the application to which is granted the permission and, in the second case, the scope would be that of a session during which that application is active.

In order to perform a comparative analysis of grant policies of the kind of the ones just defined, the following relation is defined:

$$\begin{aligned} g_1 \sqsubseteq_g g_2 \stackrel{def}{=} & \forall (tr : Trace)(p : Perm), \\ & Ptrace_{g_1}\ tr \rightarrow Safe_{g_1}\ p\ tr \rightarrow Safe_{g_2}\ p\ tr \end{aligned}$$

This order establishes that given a control-flow graph, for every valid trace of the graph according to g_1 and every initial set of permissions it holds that if the trace is safe by granting the permissions using g_1 as policy, then it must also be safe if the permissions are granted using the policy g_2. Intuitevely, g_1 yields a more restrictive permission model.

The following lemma states that the order relation between permission states preserves error situations. It can also be proved that $\sqsubseteq_g$ is a partial order (reflexive, transitive and antisymmetric). These results shall be of help when relating the grant policies described so far.

Lemma 3.

$$Lemma\ lePermError : \forall (p_1\ p_2 : Perm), Error\ p_1 \rightarrow p_1 \sqsubseteq_{Perm} p_2 \rightarrow Error\ p_2$$

The relation $\sqsubseteq_g$ defines a lattice structure with the policies $grant_{os}$ and $grant_{bk}$ as the bottom and top elements respectively.

The following theorem states a sufficient condition (a criterion) to prove that two permission granting policies, g_1 and g_2 say, are in the order relation ($g_1 \sqsubseteq_g g_2$):

1. the error situations that arise using g_2 as a policy are also error situations if g_1 is used, and
2. if a grant policy g_1 is applied, then every permission available at the end of a trace is also available if g_2 is used instead of g_1.

This theorem is important in order to compare different security policies.

Theorem 1.

$$\begin{aligned} & Theorem\ lePolicyCrit : \\ & \quad \forall (g_1\ g_2 : grantPolicy) \\ & \quad (H_{errors} : \forall (rt : ResType)\ (pr : validPermRes\ rt)\ (m : NZMulValid) \\ & \qquad\qquad (p : Perm), Error(g_2\ rt\ pr\ m\ p) \rightarrow Error(g_1\ rt\ pr\ m\ p)) \\ & \quad (H_{perms} : \forall (p : Perm)\ (tr : Trace), \\ & \qquad\qquad (PermsOf_{g_1}\ p\ tr) \sqsubseteq_{Perm} (PermsOf_{g_2}\ p\ tr)), \\ & \quad g_1 \sqsubseteq_g g_2 \end{aligned}$$

Proof. The proof proceeds by induction over $(PTrace_{g_1}\ tr)$, which is obtained after unfolding $g_1 \sqsubseteq_g g_2$. If the trace tr is empty, then the theorem holds trivially. In the case the trace is a singleton node, the proof uses hypothesis H_{errors} and proceeds by doing case analysis on the instruction type associated with that node; the interesting case corresponds to the *Grant* instruction, since *consume* is monotonic w.r.t. $\sqsubseteq_{Perm}$ and the rest of the instructions do not affect the permission state.

The inductive step follows basically from the lemma *lePermError*, the hypothesis H_{perms}, and the induction hypothesis. □

5 Relating Permission Grant Policies

Using the formal setting defined so far it is now possible to state and prove a theorem that establishes how the four policies described in the previous section are related according to the order relation $\sqsubseteq_g$.

Theorem 2.

$$Theorem\ grantPolicyRel : grant_{os} \sqsubseteq_g grant_{ow} \sqsubseteq_g grant_{ac} \sqsubseteq_g grant_{bk}$$

Proof. The proof of this theorem proceeds first by proving the three inequalities $grant_{os} \sqsubseteq_g grant_{ow}$, $grant_{ow} \sqsubseteq_g grant_{ac}$ and $grant_{ac} \sqsubseteq_g grant_{bk}$, and then applying the transitivy of the order $\sqsubseteq_g$. Each inequality is proved applying the theorem that establishes the sufficient conditions to prove that two grant policies are in the order relation (theorem *lePolicyCrit*), and following a similar strategy. Here it shall be presented in detail the proof of the first inequality, indications on how to proceed for the remainding two cases shall also be provided.

The application of the lemma *lePolicyCrit* to prove $grant_{os} \sqsubseteq_g grant_{ow}$ generates in turn the following proof obligations:

1. $\forall (rt : ResType)\ (pr : validPermRes\ rt)\ (m : NZMulValid)\ (p : Perm),$ $Error(g_2\ rt\ pr\ m\ p) \rightarrow Error(g_1\ rt\ pr\ m\ p))$
2. $\forall (p : Perm)\ (tr : Trace), (PermsOf_{g_1}\ p\ tr) \sqsubseteq_{Perm} (PermsOf_{g_2}\ p\ tr))$

The proof of (1) proceeds by first applying the lemma *lePermError*. This leads to have to prove that $(grant_{os}\ rt\ pr\ m\ p) \sqsubseteq_{Perm} (grant_{ow}\ rt\ pr\ m\ p)$. Unfolding the definition of $grant_{ow}$ and $grant_{os}$, and applying the lemmas that characterize the function *update*, we have to prove $\langle pr, 1\rangle \sqsubseteq_{PermMul} \langle pr, m\rangle$ and $(p\ rt) \sqsubseteq_{PermMul} (p\ rt)$. The latter follows directly because $\sqsubseteq_{PermMul}$ is reflexive. As to the former, as $m : NZMulValid$ so the least number it can be is 1, in which case, since $\sqsubseteq_{PermMul}$ is reflexive, the obligation is discharged.

For (2), the proof poceeds by induction on tr:

- $tr = []$, the inequality simplifies to $p \sqsubseteq_{Perm} p$ and since $\sqsubseteq_{Perm}$ is reflexive, this obligation is discharged.

– $tr = tr' \triangleleft \langle n, st, ex\rangle$, the proof proceeds by case analysis on $KD\ n$. The relevant cases are *Grant* and *Consume*, since the rest of the instructions do not affect the permission state. The *Consume* case is straightforward since the function *consume* is monotonic, and by induction hypothesis it is known that $(PermsOf_{grant_{os}}\ p\ tr') \sqsubseteq_{Perm} (PermsOf_{grant_{ow}}\ p\ tr')$. The *Grant* case is proved using transitivity of $\sqsubseteq_{Perm}$, the induction hypothesis and the following two lemmas:
  • $\forall(rt : ResType)(pr : validPermRes\ rt)(m : NZMulValid)(p : Perm)$, $(grant_{os}\ rt\ pr\ m\ p) \sqsubseteq_{Perm} (grant_{ow}\ rt\ pr\ m\ p)$
  • $\forall(rt : ResType)(pr : validPermRes\ rt)(m : NZMulValid)(p\ p' : Perm), p \sqsubseteq_{Perm} p' \to (grant_{ow}\ rt\ pr\ m\ p) \sqsubseteq_{Perm} (grant_{ow}\ rt\ pr\ m\ p')$.

  The proofs of these lemmas are omitted due to space restrictions.

The structure of the proof of the two remaining inequalities are quite similar to the one just described above. In both cases the bulk of the proof reduces to prove auxiliary lemmas similar to the ones of the proof obligation (2) for the involved grant policies. □

This theorem and its proof provide a formal evidence that, in the first place, of the four policies, MIDP's one-shot is the most restrictive policy and MIDP's blanket is the most permissive one. In addition to that, these two policies have been formally related with the permission grant policies defined in [2]. Furthermore, the theorem also formally relates these two latter granting policies, showing that the accumulative one is more permissive than the overwriting one.

The difference between accumulating permissions and overwriting permissions is subtle. The problem with accumulating permissions is that at any program point to approximate the permissions available for a given resource type it has to be considered all the consumptions and all the permissions granted for that resource type. Whereas in the overwriting grant policy it is enough to consider the last grant operation and the subsequent consume operations. This suggest that a static permission analysis might be simpler using the overwriting grant policy.

6 Conclusion and Further Work

This paper reports work concerning the formal specification and analysis of access control models for interactive mobile devices.

Here it has been presented an unprecedented framework, formalized using the proof-assistant Coq, that provides a uniform setting to define and analyse access control models which incorporate interactive permission requesting/granting mechanisms. In particular, the work presented here has focused on two distinguished permission models: the one defined by version 2.0 of MIDP and the one defined by Besson et al. in [2]. A drawback of MIDP permission model is that the user is forced to decide between tedious continuous interruption in interactive programs in order to grant a (one-shot) permission or otherwise to trust applications and concede almost irrestricted permission for it to access sensible resources. The model proposed in [2] is more flexible than MIDP's, allowing additional possibilities in the way permissions are granted. A characterization of both models in terms of a formal definition of grant policy has also been provided.

Another kind of permission policies can also be expressed in the framework. In particular, it can be adapted to introduce a notion of permission revocation, a permission mode not considered in MIDP. A revoke can be modeled in the permission overwriting approach, for instance, by assigning a zero multiplicity to a resource type. In the accumulative approach, revocation might be modeled using negative multiplicities. To introduce revocations, in turn, enables, without further changes to the framework, to model a notion of permission scope. One such scope would be grasped as the session interval delimited by an activation and a revocation of that permission.

An order relation $\sqsubseteq_g$ on grant policies has also been presented in this work. Two theorems have been established and their proofs discussed: one that states a sufficient condition to prove that two permission granting policies are related by that order, and another one that establishes a precise comparison of permission granting policies defined by the models. In particular it is formally proved that the accumulative grant policy is more permissive than the overwriting one. Furthermore, it has been shown that $\sqsubseteq_g$ defines a lattice structure with the policies $grant_{os}$ and $grant_{bk}$ as the bottom and top elements respectively, providing then a formal algebraic setting in which grant policies can be precisely related and compared.

Further work is the study and specification, using the formal setting provided by the framework, of algorithms for enforcing the security policies derived from different sort of permission models to control the access to sensitive resources of the devices. Moreover, one main objective is to extend the framework so as to be able to construct certified prototypes from the formal definitions of those algorithms.

References

[1] M. Bartoletti, P. Degano, and G-L. Ferrari. Static analysis for stack inspection. In *Electronic Notes in Computer Science*, volume 54, 2001.

[2] F. Besson, G. Dufay, and T. Jensen. A formal model of access control for mobile interactive devices. In *11th European Symposium on Research in Computer Security (ESORICS'06), LNCS 4189*, pages 110–126, 2006.

[3] F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model ckecking security properties of control flow graphs. *Journal of Computer Security*, 9:217–250, 2001.

[4] T. Coquand and G. Huet. The Calculus of Constructions. In *Information and Computation*, volume 76, pages 95–120. Academic Press, February/March 1988.

[5] T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1990.

[6] JSR 118 Expert Group. Mobile information device profile for java 2 micro edition. version 2.0. Technical report, Sun Microsystems, Inc. and Motorola, Inc., 2002.

[7] JSR 37 Expert Group. Mobile information device profile for java 2 micro edition. version 1.0. Technical report, Sun Microsystems, Inc., 2000.

[8] T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security properties. In *Proc. of the 20th IEEE Symp. on Security and Privacy*, pages 89–103. New York: IEEE Computer Society, may 1999.

[9] R. Roushani, G. Betarte, and C. Luna. A Certified Access Controller for JME-MIDP 2.0 enabled Mobile Devices. In *I Chilean Workshop on Formal Methods, Punta Arenas, Chile*. IEEE Computer Society, 2008. To be published.

[10] Sun Microsystems, Inc. *Java Platform Micro Edition.*, Last accessed: Oct. 2008. http://java.sun.com/javame/index.jsp.

[11] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1*, 2006.

[12] S. Zanella Béguelin, G. Betarte, and C. Luna. A formal specification of the MIDP 2.0 security model. In *Formal Aspects in Security and Trust, Fourth International Workshop, FAST 2006, Hamilton, Ontario, Canada, August 26-27, 2006, Revised Selected Papers*, volume 4691 of *LNCS*, pages 220–234. Springer-Verlag, 2006.