A Formal Specification of the MIDP 2.0 Security Model

Santiago Zanella Béguelin[1], Gustavo Betarte[2], and Carlos Luna[2]

[1] FCEIA, Universidad Nacional de Rosario, Argentina
szanella@fceia.unr.edu.ar
[2] InCo, Facultad de Ingeniería, Universidad de la República, Uruguay
{gustun,cluna}@fing.edu.uy

Abstract. This paper overviews a formal specification, using the Calculus of Inductive Constructions, of the application security model defined by the Mobile Information Device Profile 2.0 for Java 2 Micro Edition. We present an abstract model of the state of the device and security-related events that allows to reason about the security properties of the platform where the model is deployed. We then state and sketch the proof of some desirable properties of this model.

Keywords: Calculus of Inductive Constructions, Coq, formal specification, MIDP 2.0, security.

1 Introduction

Mobile devices, like cell phones, manipulate sensitive personal data (e.g contact lists and call recordings), and are capable of establishing connections with external entities. Users of such devices may, in addition, download and install potentially dangerous applications. Since any security breach may expose sensitive data, prevent the use of the device, or allow applications to perform actions that incur a charge for the user, it is essential to provide an application security model that can be relied upon – the slightest vulnerability may imply millionaire losses due to the scale the technology has been adopted.

Java 2 Micro Edition (J2ME) is a version of the Java platform targeted at resource-constrained devices. J2ME comprises two kinds of components: configurations and profiles. A configuration is composed of a virtual machine and a set of APIs that provide the basic functionality for a particular category of devices. Profiles further specify the target technology by defining a set of higher level APIs built on top of an underlying configuration. This two-level architecture enhances portability and enables developers to deliver applications that run on a range of devices with similar capabilities.

The Connected Limited Device Configuration (CLDC) is a J2ME configuration designed for devices with slow processors, limited memory and intermittent connectivity. CLDC together with the Mobile Information Device Profile (MIDP) provides a complete J2ME runtime environment tailored for devices like mobile phones and personal data assistants. MIDP 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

since the specification was made available. Literally millions of MIDP enabled devices are deployed worldwide and the market acceptance of the specification is expected to continue to grow steadily.

In the original MIDP 1.0 specification [1], 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 (e.g. push activation). 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.

MIDP 2.0 [2] introduces a new security model based on the concept of protection domains. Each sensitive API or function on the device may define permissions in order to prevent it from being used without authorisation. An installed MIDlet suite is bound to a unique protection domain that defines a set of permissions granted either unconditionally or with explicit user authorisation. 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.

Some effort has been put into the evaluation of the security model for MIDP 2.0; [3, 4] analyse the application security model, spot vulnerabilities in various implementations and suggest improvements to the specification. Although these works report on the detection of security holes, they do not intend to prove their absence. The formalisation we overview here, however, provides a formal basis for the verification of the model and the understanding of its intricacies.

We developed our specification using the Coq proof assistant [5, 6], an implementation of the Calculus of Inductive Constructions (CIC). CIC is a dependently typed lambda calculus that extends Coquand and Huet's Calculus of Constructions [7] with inductive definitions as first-class objects. The interested reader may refer to [8, 9] for further background on these calculi.

The rest of the paper is organized as follows, Section 2 defines the notation used, Section 3 describes the formalisation of the MIDP 2.0 security model. Section 4 presents some of its verified properties, and finally Section 5 concludes.

2 The Notation Used

We use standard notation for equality and logical connectives ($\wedge$,$\vee$,$\neg$,$\rightarrow$,$\forall$,$\exists$). Implication and universal quantification may be encoded using CIC dependent product, while equality and the other connectives can be defined inductively. Anonymous predicates are introduced using lambda notation, e.g $(\lambda\, n\,.\, n = 0)$ is a predicate that when applied to n is true iff $n = 0$.

We extensively use record types; a record type definition

$$R \stackrel{\text{def}}{=} \{field_1 : type_1, \ldots, field_n : type_n\} \tag{1}$$

generates a non-recursive inductive type with just one constructor, mkR, and projections functions $field_i : R \rightarrow type_i$. We often write $\langle a_1, \ldots, a_n \rangle$ instead

of $mkR\ a_1 \ldots a_n$ when the type is obvious from the context. Application of projections functions is abbreviated using dot notation, i.e. $field_i\ r \equiv r.field_i$. For each field $field_i$ in a record type we define a binary relation $\simeq_{field_i}$ over objects of the type as

$$r_1 \simeq_{field_i} r_2 \stackrel{\text{def}}{=} \forall j, j \neq i \rightarrow r_1.field_j = r_2.field_j \ . \tag{2}$$

A parametric inductive type $T : type_1 \rightarrow \ldots \rightarrow type_n \rightarrow sort$ is defined by giving for each constructor c_k an introduction rule of the form

$$\frac{P_1\ a_1\ \ldots\ a_n \quad \cdots \quad P_m\ a_1\ \ldots\ a_n}{T\ a_1\ \ldots\ a_n}\ c_k$$

where $P_j, j = 1, \ldots, m$ are predicates over the parameters of the type. We assume as predefined inductive types, the parametric type *option* T with constructors $None$: *option* T and $Some : T \rightarrow$ *option* T, and the type *seq* T of finite sequences over T. We denote the empty sequence by $[\,]$, and the constructor that appends an element a to a sequence s as $s \frown a$. We represent with $\oplus$ the concatenation operation on sequences.

3 Formalisation of the MIDP 2.0 Security Model

In this section we present the formal specification of the security model.

3.1 Sets and Constants

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 declare the corresponding permissions in its descriptor. MIDlet suites may declare permissions either as required or as optional.

Let $Permission$ be the total set of permissions defined by every protected API or function on the device and $Domain$ the set of all protection domains. Let us introduce, as a way of referring to individual MIDlet suites, the set $SuiteID$ of valid suite identifiers. We will represent an application descriptor as a record composed of two predicates, *required* and *optional*, that identify respectively the set of permissions declared as required and those declared as optional,

$$Descriptor \stackrel{\text{def}}{=} \{required, optional : Permission \rightarrow Prop\} \ . \tag{3}$$

For the sake of conciseness let us introduce a record type to represent installed suites, with fields for its identifier, associated protection domain and descriptor,

$$Suite \stackrel{\text{def}}{=} \{sid : SuiteID, domain : Domain, descriptor : Descriptor\} \ . \tag{4}$$

Permissions may be granted by the user to an active MIDlet suite in either of three modes, only once (oneshot), until it is terminated (session), or until it

is uninstalled (blanket). Let $Mode$ be the enumerated set of user interaction modes – $\{oneshot, session, blanket\}$ – and $\leq_m$ an order relation such that

$$oneshot \leq_m session \leq_m blanket \ . \tag{5}$$

We will assume for the rest of the formalisation that the security policy of the protection domains on the device is an anonymous constant of type

$$\begin{array}{l} Policy \stackrel{\text{def}}{=} \{\ allow : Domain \to Permission \to Prop, \\ \qquad\qquad\ \ user\ \ : Domain \to Permission \to Mode \to Prop\ \} \end{array} \tag{6}$$

such that $allow\ d\ p$ holds when domain d unconditionally grants the permission p and $user\ d\ p\ m$ when domain d grants permission p with explicit user authorisation and maximum allowable mode m (w.r.t $\leq_m$). The permissions effectively granted to a MIDlet suite are the intersection of the permissions requested in its descriptor with the union of the allowed and user granted permissions.

3.2 Device State

To reason about the MIDP 2.0 security model most details of the device state may be abstracted; it is sufficient to specify the set of installed suites, the permissions granted or revoked to them, and the currently active suite in case there is one. The active suite and the permissions granted or revoked to it for the session are grouped into a record structure

$$\begin{array}{l} SessionInfo \stackrel{\text{def}}{=} \{\ sid \qquad\qquad\qquad : SuiteID, \\ \qquad\qquad\qquad\ granted, revoked : Permission \to Prop\ \}\ . \end{array} \tag{7}$$

The abstract device state is described as a record of type

$$\begin{array}{l} State \stackrel{\text{def}}{=} \{\ suite \qquad\quad\ : Suite \to Prop, \\ \qquad\quad session \qquad\quad : option\ SessionInfo, \\ \qquad\quad granted, revoked : SuiteID \to Permission \to Prop\ \}\ . \end{array} \tag{8}$$

Some conditions must hold for an element $s : State$ in order to represent a valid state for a device. These conditions are specified in the rest of this section.

A MIDlet suite can be installed and bound to a protection domain only if the set of permissions declared as required in its descriptor are a subset of the permissions the domain offers (with or without user authorization). This compatibility relation between $des : Descriptor$ and $dom : Domain$ can be stated formally as follows,

$$\begin{array}{l} des \wr dom \stackrel{\text{def}}{=} \forall\, p : Permission, \\ \quad des.required\ p \to allow\ dom\ p\ \vee \exists\, m : Mode, user\ dom\ p\ m\ \ . \end{array} \tag{9}$$

For s to be a valid state, every installed suite must be compatible with its associated protection domain,

$$\begin{array}{l} SuiteCompatible \stackrel{\text{def}}{=} \\ \quad \forall\, ms : MIDletSuite, s.suite\ ms \to ms.descriptor \wr ms.domain\ \ . \end{array} \tag{10}$$

Whenever there exists a running session, the suite identifier in $s.session$ must correspond to an installed suite,

$$\begin{aligned} CurrentInstalled \stackrel{\text{def}}{=} \forall\, ses : SessionInfo, s.session = Some\ ses \rightarrow \\ \exists\, ms : MIDletSuite, s.suite\ ms \wedge ms.sid = ses.sid\ . \end{aligned} \tag{11}$$

The *granted* predicate should be valid, i.e. the set of permissions granted for the session must be a subset of the permissions requested in the application descriptor of the active suite, and the associated protection domain policy must allow them to be granted for the session,

$$\begin{aligned} ValidSessionGranted \stackrel{\text{def}}{=} \forall ses : SessionInfo, s.session = Some\ ses \rightarrow \\ \forall\, p : Permission, ses.granted\ p \rightarrow \\ \forall\, ms : MIDletSuite, s.suite\ ms \rightarrow ms.sid = ses.sid \rightarrow \\ (ms.descriptor.required\ p\ \vee ms.descriptor.optional\ p) \wedge \\ (user\ ms.domain\ p\ session\ \vee\ user\ ms.domain\ p\ blanket)\ . \end{aligned} \tag{12}$$

Every installed suite shall have a unique identifier,

$$\begin{aligned} UniqueSuiteID \stackrel{\text{def}}{=} \forall\, ms_1\, ms_2 : MIDletSuite, \\ s.suite\ ms_1 \rightarrow s.suite\ ms_2 \rightarrow ms_1.sid = ms_2.sid \rightarrow ms_1 = ms_2\ . \end{aligned} \tag{13}$$

Additionally, for every installed suite with identifier *sid*, $s.granted\ sid$ should be valid with respect to its descriptor and associated protection domain ($ValidGranted\, s$). It must also be ensured that any granted permission is not being revoked at the same time and viceversa ($ValidGrantedRevoked\, s$). We omit the detailed formalisation of these conditions.

3.3 Events

We define a set *Event* for those events that are meaningful with respect to our abstraction of the device state (Table 1).

Table 1. Events. $UserAnswer \stackrel{\text{def}}{=} \{ua\_allow\ m, ua\_deny\ m \mid m \in Mode\}$

Name	Description	Type
start	Start of session	$SuiteID \rightarrow Event$
terminate	End of session	$Event$
request	Permission request	$Permission \rightarrow option\ UserAnswer \rightarrow Event$
install	MIDlet suite installation	$SuiteID \rightarrow Descriptor \rightarrow Domain \rightarrow Event$
remove	MIDlet suite removal	$SuiteID \rightarrow Event$

The behavior of the events is specified by defining their pre- and postconditions by means of the predicates *Pre* and *Pos* respectively. Preconditions (Table 2) are defined in terms of the device state, while postconditions (Table 3) are defined in terms of the before and after states, and an optional response which is only meaningful for the *request* event and indicates whether the requested permission is given,

Table 2. Event preconditions. The *request* event is omitted for reasons of space

$$
\begin{array}{l}
Pre\ s\ (start\ sid) \stackrel{\text{def}}{=} \\
\quad s.session = None \wedge \exists\, ms : MIDletSuite, s.suite\ ms \wedge ms.sid = sid \\
Pre\ s\ terminate \stackrel{\text{def}}{=} s.session \neq None \\
Pre\ s\ (install\ sid\ des\ dom) \stackrel{\text{def}}{=} \\
\quad des \wr dom \wedge \forall\, ms : MIDletSuite, s.suite\ ms \rightarrow ms.sid \neq sid. \\
Pre\ s\ (remove\ sid) \stackrel{\text{def}}{=} \\
\quad (\forall\, ses : SessionInfo, s.session = Some\ ses\ \rightarrow ses.sid \neq sid) \wedge \\
\quad \exists\, ms : MIDletSuite, s.suite\ ms \wedge ms.sid = sid
\end{array}
$$

Table 3. Event postconditions. The *request* event is omitted for reasons of space

$$
\begin{array}{l}
Pos\ s\ s'\ r\ (start\ sid) \stackrel{\text{def}}{=} \\
\quad r = None \wedge s \simeq_{session} s' \wedge s'.session = Some\ \langle sid, (\lambda x.False), (\lambda x.False)\rangle \\
Pos\ s\ s'\ r\ terminate \stackrel{\text{def}}{=} r = None \wedge s \simeq_{session} s' \wedge s'.session = None \\
Pos\ s\ s'\ r\ (install\ sid\ des\ dom) \stackrel{\text{def}}{=} \\
\quad r = None \wedge (\forall\, ms : MIDletSuite, s.suite\ ms \rightarrow s'.suite\ ms) \wedge \\
\quad (\forall\, ms : MIDletSuite, s'.suite\ ms \rightarrow s.suite\ ms \vee ms = \langle sid, dom, des\rangle) \wedge \\
\quad s'.suite\ \langle sid, dom, des\rangle \wedge s'.session = s.session \wedge \\
\quad (\forall\, p : Permission, \neg s'.granted\ sid\ p \wedge \neg s'.revoked\ sid\ p) \wedge \\
\quad (\forall\, sid_1 : SuiteID, sid_1 \neq sid \rightarrow \\
\qquad s'.granted\ sid_1 = s.granted\ sid_1 \wedge s'.revoked\ sid_1 = s.revoked\ sid_1) \\
Pos\ s\ s'\ r\ (remove\ sid) \stackrel{\text{def}}{=} \\
\quad r = None \wedge s \simeq_{suite} s' \wedge \\
\quad (\forall\, ms : MIDletSuite, s.suite\ ms \rightarrow ms.sid \neq sid \rightarrow s'.suite\ ms) \wedge \\
\quad (\forall\, ms : MIDletSuite, s'.suite\ ms \rightarrow s.suite\ ms \wedge ms.sid \neq sid)
\end{array}
$$

$$
\begin{array}{l}
Pre : State \rightarrow Event \rightarrow Prop \\
Pos : State \rightarrow State \rightarrow option\ Response \rightarrow Event \rightarrow Prop\ .
\end{array}
\tag{14}
$$

For example, an event representing a permission request not authorized by the user can only occur when the active suite has declared that permission in its descriptor and is bound to a protection domain that specifies an user interaction mode for the permission (otherwise, the request would be immediately accepted or rejected). Furthermore, the requested permission must not have been revoked or granted for the rest of the session or the MIDlet suite life,

$$
\begin{array}{l}
Pre\ s\ (request\ p\ (Some\ (ua\_deny\ m))) \stackrel{\text{def}}{=} \\
\quad \exists\, ses : SessionInfo, s.session = Some\ ses \wedge \\
\qquad \forall\, ms : MIDletSuite, s.suite\ ms \rightarrow ms.sid = ses.sid \rightarrow \\
\qquad\quad (ms.descriptor.required\ p \vee ms.descriptor.optional\ p) \wedge \\
\qquad\quad (\exists\, m_1 : Mode, user\ ms.domain\ p\ m_1) \wedge \\
\qquad\quad \neg ses.granted\ p \wedge \neg ses.revoked\ p \wedge \\
\qquad\quad \neg s.granted\ ses.sid\ p \wedge \neg s.revoked\ ses.sid\ p\ .
\end{array}
\tag{15}
$$

When $m = session$, the user revokes the permission for the rest of the session, therefore, the response denies the permission, and the state is updated accordingly,

$$\begin{array}{l} Pos\ s\ s'\ r\ (request\ p\ (Some\ (ua\_deny\ session))) \stackrel{\text{def}}{=} \\ \quad r = Some\ denied \wedge s \simeq_{session} s' \wedge \\ \quad \forall\, ses : SessionInfo, s.session = Some\ ses \rightarrow \\ \qquad \exists\, ses' : SessionInfo, \\ \qquad\quad s'.session = Some\ ses' \wedge ses' \simeq_{revoked} ses \wedge ses'.revoked\ p \wedge \\ \qquad\quad (\forall\, q : Permission, q \neq p \rightarrow ses'.revoked\ q = ses.revoked\ q)\ . \end{array} \tag{16}$$

3.4 One-step Execution

When an event occurs for which the precondition does not hold, the state must remain unchanged. Otherwise, the state may change in such a way that the event postcondition is established. The behavioral specification of the execution of an event is given by the $\hookrightarrow$ relation with the following introduction rules:

$$\frac{\neg Pre\ s\ e}{s \xhookrightarrow{e/None} s}\ npre \qquad \frac{Pre\ s\ e \quad Pos\ s\ s'\ r\ e}{s \xhookrightarrow{e/r} s'}\ pre\ . \tag{17}$$

3.5 Sessions

A session is a period of time spanning from the start of a session to its termination. A session for a suite with identifier *sid* (Fig. 1) is determined by an initial state s_0 and a sequence of steps $\langle e_i, s_i, r_i \rangle$ $(i = 1, \ldots, n)$ such that the following conditions hold,

- $e_1 = start\ sid$;
- $Pre\ s_0\ e_1$;
- $\forall\, i \in \{2, \ldots, n-1\}, e_i \neq terminate$;
- $e_n = terminate$;
- $\forall\, i \in \{1, \ldots, n\}, s_{i-1} \xhookrightarrow{e_i/r_i} s_i$.

$$s_0 \xhookrightarrow{start\ sid/r_1} s_1 \xhookrightarrow{e_2/r_2} s_2 \xhookrightarrow{e_3/r_3} \cdots \xhookrightarrow{e_{n-1}/r_{n-1}} s_{n-1} \xhookrightarrow{terminate/r_n} s_n$$

Fig. 1. A session for a suite with identifier *sid*

To define the session concept it is better to introduce before the concept of partial session. A partial session is a *session* for which the *terminate* event has not yet been elicited; it may be defined inductively by the following rules,

$$\frac{Pre\ s_0\ (start\ sid) \quad s_0 \xhookrightarrow{start\ sid/r_1} s_1}{PSession\ s_0\ ([\,] \frown \langle start\ sid, s_1, r_1 \rangle)}\ psession\_start \tag{18}$$

$$\frac{PSession\ s_0\ (ss \frown last) \quad e \neq terminate \quad last.s \xrightarrow{e/r} s'}{PSession\ s_0\ (ss \frown last \frown \langle e, s', r\rangle)}\ psession\_app \quad . \tag{19}$$

Now, sessions can be easily introduced as follows,

$$\frac{PSession\ s_0\ (ss \frown last) \quad last.s \xrightarrow{terminate/r} s'}{Session\ s_0\ (ss \frown last \frown \langle terminate, s', r\rangle)}\ session\_terminate \quad . \tag{20}$$

4 Verification of (Security) Properties

In what follows we state some propositions about the specification. Although detailed proofs of these propositions have been constructed and checked using Coq, we only give here an outline of the proofs.

4.1 An Invariant of One-step Execution

We call one-step invariant a property that remains true after the *execution* of every event if it is true before. We expect the validity of the device state to be a one-step invariant of our specification.

Theorem 1. *Let $Valid$ be a predicate over $State$ defined as the conjunction of the validity conditions in Sect. 3.2. For any $s\ s' : State$, $r : option\ Response$ and $e : Event$, if $Valid\ s$ and $s \xrightarrow{e/r} s'$ hold, then $Valid\ s'$ also holds.*

Proof. By case analysis on $s \xrightarrow{e/r} s'$. When $Pre\ s\ e$ does not hold, $s = s'$ and s' is valid because s is valid. Otherwise, $Pos\ s\ s'\ r\ e$ must hold and we proceed by case analysis on e. We will only show the case $request\ p\ (Some\ (ua\_deny\ session))$, obtained after further case analysis on a when $e = request\ p\ (Some\ a)$.

The postcondition (16) entails that $s'.granted = s.granted$, $s'.revoked = s.revoked$, $s'.suite = s.suite$, and that there exists ses' such that $s'.session = Some\ ses'$, $ses'.sid = ses.sid$, $ses'.granted = ses.granted$ and

$$\forall\, q, q \neq p \rightarrow ses'.revoked\ q = ses.revoked\ q \quad . \tag{21}$$

Rewriting with these equalities, we immediately prove $SuiteCompatible\ s'$, $CurrentInstalled\ s'$, $UniqueSuiteID\ s'$ and $ValidGranted\ s'$ from the validity of s. It remains to prove $ValidGrantedRevoked\ s'$.

Let q be any permission. If $q \neq p$, then from (21) $ses'.revoked\ q = ses.revoked\ q$, and because q was not granted and revoked simultaneously before the event, neither it is after. If $q = p$, then we know from the precondition (15) that p was not granted before, and so it is not granted after. This proves $ValidGrantedRevoked\ s'$, and together with the previous results, $Valid\ s'$. □

4.2 Session Invariants

We call session invariant a property of a step that holds for the rest of a session once it is established in any step. Let P be a predicate over T, we define $all\ P$ as an inductive predicate defined over $seq\ T$ by

$$\frac{}{all\ P\ []}\ all\_nil \qquad \frac{all\ P\ ss \quad P\ s}{all\ P\ (ss \frown s)}\ all\_app \tag{22}$$

Theorem 2. *If s_0 is a valid state, and $PSession\ s_0\ ss$, then*

$$all\ (\lambda\, step\,.\, Valid\ step.s)\ ss\ . \tag{23}$$

Proof. By induction on the structure of $PSession\ s_0\ ss$.

- When it is constructed using *psession_start*, ss is like $[] \frown \langle start\ sid, s_1,\ r_1\rangle$ and $s_0 \xleftarrow{start\ sid/r_1} s_1$ holds. We must prove

$$all\ (\lambda\, step\,.\, Valid\ step.s)\ ([] \frown \langle start\ sid, s_1,\ r_1\rangle)\ . \tag{24}$$

  By applying *all_app* and then *all_nil* the goal is simplified to $Valid\ s_1$, and is proved from $s_0 \xleftarrow{start\ sid/r_1} s_1$ and $Valid\ s_0$ by applying Theorem 1.
- When it is constructed using *psession_app*, ss has the form $ss_1 \frown last \frown \langle e, s', r\rangle$ and $last.s \xleftarrow{e/r} s'$ holds. The induction hypothesis is

$$all\ (\lambda\, step\,.\, Valid\ step.s)\ (ss_1 \frown last) \tag{25}$$

  and we must prove $all\ (\lambda\, step\,.\, Valid\ step.s)\ (ss_1 \frown last \frown \langle e, s', r\rangle)$. By applying *all_app* and then (25) the goal is simplified to $Valid\ s'$. From (25) we know that $last.s$ is a valid state. The goal is proved from $last.s \xleftarrow{e/r} s'$ and $Valid\ last.s$ by applying Theorem 1. □

The above theorem may be easily extended from partial sessions to sessions by applying Theorem 1. State validity is just a particular property that is true for a partial session once it is established, the result can be generalised for other properties as in the following lemma.

Lemma 1. *For any property P of a step satisfying*

$$e \neq terminate \wedge s \xleftarrow{e'/r'} s' \wedge P\ \langle e, s, r\rangle \rightarrow P\ \langle e', s', r'\rangle\ , \tag{26}$$

if $PSession\ s_0\ (ss \frown step \oplus ss_1)$ and $P\ step$ then $all\ P\ ss_1$ holds.

Perhaps a more interesting property is that once a permission is revoked by the user for the rest of a session, any further request for the same permission in the same session is rejected.

Lemma 2. *The following property satisfies* (26)*,*

$$(\lambda\, step\,.\, \exists\, ses, step.s.session = Some\ ses \wedge ses.revoked\ p)\ . \tag{27}$$

Theorem 3. *For any permission p, if $PSession\ s_0\ (ss \frown step \frown step_1 \oplus ss_1)$, $step_1.e = request\ p\ (Some\ (ua\_deny\ session))$ and $Pre\ step.s\ step_1.e$, then*

$$all\ (\lambda\, step\,.\, \forall\, o, step.e = request\ p\ o \rightarrow step.r \neq Some\ allowed)\ ss_1\ . \tag{28}$$

Proof. Since $Pos\ step.s\ step_1.s\ step_1.r\ step_1.e$ holds, p is revoked for the session in $step_1.s$. From Lemmas 1 and 2, p remains revoked for the rest of the session. Let $e = request\ p\ o$ be an event in a step $step$ in ss_1. We know that p is revoked for the session in the state before $step.s$. If the precondition for e does not hold in the state before[1], then $step.r = None$. Otherwise, the postcondition for e is true and in any case it entails $step.r = Some\ denied \neq Some\ allowed$. □

[1] Actually, it holds only when $o = None$.

5 Conclusions

The informal specification in [2] puts forward an application security model that any implementation of the profile must satisfy. Although testing implementations of that model might help discover vulnerabilities, it is not an exhaustive technique and does not compare to the assurance level that gives a formal verification of the model.

We have produced an unprecedented verifiable formalisation of the MIDP 2.0 security model and have also constructed the proofs of several important properties that should be satisfied by any implementation that fulfills the specification of the security model. It is shown in this paper that the formal specification is precise and detailed enough to study, for instance, the interference between the security rules that control access to the device resources and mechanisms such as application installation. Two simplifying assumptions have been made: 1) the security policy is static; 2) up to one suite may be active at a time. Most implementations actually enforce these assumptions, however, it would be interesting to further explore the consequences of relaxing them.

We strongly believe the specification we have produced can be formally refined down to an executable specification. As a future work, we plan to complete such a refinement and obtain executable functional code using the Coq extraction mechanism.

References

1. JSR 37 Expert Group. *Mobile Information Device Profile for Java 2 Micro Edition. Version 1.0.* Sun Microsystems, Inc. (2000).
2. JSR 118 Expert Group. *Mobile Information Device Profile for Java 2 Micro Edition. Version 2.0.* Sun Microsystems, Inc. and Motorola, Inc. (2002).
3. O. Kolsi and T. Virtanen. MIDP 2.0 Security Enhancements. In: *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)* (IEEE Computer Society, 2004), p. 90287.3.
4. M. Debbabi, M. Saleh, C. Talhi, and S. Zhioua. Security Analysis of Wireless Java. In: *Proceedings of the 3rd Annual Conference on Privacy, Security and Trust* (2005), pp. 1–11.
5. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0* (2004).
6. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science (Springer-Verlag, 2004).
7. T. Coquand and G. Huet. The Calculus of Constructions. In: *Information and Computation*, vol. 76 (Academic Press, 1988), pp. 95–120.
8. C. Paulin-Mohring. Inductive Definitions in the system Coq - Rules and Properties. In: *First Int. Conf. on Typed Lambda Calculi and Applications*, edited by M. Bezem and J. F. Groote, *LNCS*, vol. 664 (Springer-Verlag, 1993), pp. 328–345.
9. B. Werner. *Une Théorie des Constructions Inductives.* Phd Thesis, Université Paris 7, France (1994).