A Formal Specification of the MIDP 2.0 Security Model

Santiago Zanella Béguelin[1,2], Gustavo Betarte[3], and Carlos Luna[3]

[1] INRIA–Microsoft Research Joint Laboratory, 91893 Orsay Cedex, France
INRIA Sophia Antipolis, 06902 Sophia Antipolis Cedex, France,
`Santiago.Zanella@sophia.inria.fr`

[2] InCo, Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay,
{`gustun,cluna`}`@fing.edu.uy`

Abstract. This paper presents, to the best of our knowledge, the first formal specification of the application security model defined by the Mobile Information Device Profile 2.0 for Java 2 Micro Edition. The specification, which has been formalized in Coq, provides an abstract representation of the state of a device and the security-related events that allows to reason about the security properties of the platform where the model is deployed. We state and sketch the proof of some desirable properties of the security model. Although the abstract specification is not executable, we describe a refinement methodology that leads to an executable prototype.

1 Introduction

Mobile devices (e.g. cell phones, personal digital assistants) often have access to sensitive personal data, are subscribed to paid services and are capable of establishing connections with external entities. Users of such devices may, in addition, download and install applications from untrusted sites at their will. 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 huge losses due to the scale the technology has been deployed.

Java 2 Micro Edition (J2ME) is a version of the Java platform targeted at resource-constrained devices which 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 determine 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 wide range of devices with similar capabilities. This work only concerns the topmost level of the architecture which corresponds to the profile that defines the security model we formalize.

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 cell 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. Nowadays, 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 jeopardizing 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 authorization. Every installed application is bound to a unique protection domain that defines a set of permissions granted either unconditionally or with explicit user authorization. Untrusted applications are bound to a protection domain with permissions equivalent to those in a MIDP 1.0 sandbox. Trusted applications 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; Kolsi and Virtanen [3] and Debbabi et al. [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 formalization 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]. A detailed description of the specification is presented in Spanish in [7]; a shorter, preliminary version of this paper appeared in InCo's technical report series [8]. Both documents, along with the full formalization in Coq may be obtained from `http://www-sop.inria.fr/everest/personnel/Santiago.Zanella/MIDP`.

The rest of the paper is organized as follows, Section 2 describes some of the notation used in this document, Section 3 overviews the formalization of the MIDP 2.0 security model, Section 4 presents some of its verified properties along with outlines of their proofs, Section 5 proposes a methodology to refine the specification and obtain an executable prototype and finally, Section 6 concludes with a summary of our contributions and directions for future work.

2 Notation

We use standard notation for equality and logical connectives ($\wedge$, $\vee$, $\neg$, $\rightarrow$, $\forall$, $\exists$). Implication and universal quantification may be encoded in Coq using 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 is zero.

We extensively use record types; a record type definition

$$R \overset{\text{def}}{=} \{\mathit{field}_1 : A_1, \ldots, \mathit{field}_n : A_n\} \tag{1}$$

generates a non-recursive inductive type with just one constructor, namely mkR, and projections functions $\mathit{field}_i : R \rightarrow A_i$. We write $\langle a_1, \ldots, a_n \rangle$ instead of $\mathit{mkR}\ a_1 \ldots a_n$ when the type R is obvious from the context. Application of projections functions is abbreviated using dot notation (i.e. $\mathit{field}_i\ r = r.\mathit{field}_i$). For each field field_i in a record type we define a binary relation $\equiv_{\mathit{field}_i}$ over objects of the type as

$$r_1 \equiv_{\mathit{field}_i} r_2 \overset{\text{def}}{=} \forall\ j, j \neq i \rightarrow r_1.\mathit{field}_j = r_2.\mathit{field}_j \tag{2}$$

We define an inductive relation I by giving introduction rules of the form

$$\frac{P_1 \cdots\ P_m}{I\ x_1 \ldots x_n}\ \mathit{rule} \tag{3}$$

where free occurrences of variables are implicitly universally quantified.

We assume as predefined inductive types the parametric type *option* T with constructors $\mathit{None} : \mathit{option}\ T$ and $\mathit{Some} : T \rightarrow \mathit{option}\ T$, 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 in infix form as in $s \frown a$. The symbol $\oplus$ stands for the concatenation operator on sequences.

3 Formalization of the MIDP 2.0 Security Model

In this section we present and discuss the formal specification of the security model. We introduce first some types and constants used in the remainder of the formalization, then we define the set of valid device states and security-related events, give a transition semantics for events based on pre- and postconditions and define the concept of a session.

3.1 Sets and Constants

In MIDP, applications (usually called MIDlets) are packaged and distributed as suites. A suite may contain one or more MIDlets and is distributed as two files, an application descriptor file and an archive file that contains the actual Java classes and resources. A suite that uses protected APIs or functions should declare the corresponding permissions in its descriptor either as required for its correct functioning 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 a 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 by the corresponding suite,

$$\mathit{Descriptor} \overset{\text{def}}{=} \{\mathit{required}, \mathit{optional} : \mathit{Permission} \to \mathit{Prop}\} \tag{4}$$

A record type is used to represent an installed suite, with fields for its identifier, associated protection domain and descriptor,

$$\mathit{Suite} \overset{\text{def}}{=} \{\mathit{id} : SuiteID, \mathit{domain} : \mathit{Domain}, \mathit{descriptor} : \mathit{Descriptor}\} \tag{5}$$

Permissions may be granted by the user to an active MIDlet suite in either of three modes, for a single use (oneshot), as long as the suite is running (session), or as long as the suite remains installed (blanket). Let $Mode$ be the enumerated set of user interaction modes $\{\mathit{oneshot}, \mathit{session}, \mathit{blanket}\}$ and $\leq_{\text{m}}$ an order relation such that

$$\mathit{oneshot} \leq_{\text{m}} \mathit{session} \leq_{\text{m}} \mathit{blanket} \tag{6}$$

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

$$\begin{aligned}\mathit{Policy} \overset{\text{def}}{=} \{\ & \mathit{allow} : \mathit{Domain} \to \mathit{Permission} \to \mathit{Prop}, \\ & \mathit{user}\ \ : \mathit{Domain} \to \mathit{Permission} \to Mode \to \mathit{Prop}\ \}\end{aligned} \tag{7}$$

which for each domain specifies at most one mode for a given permission,

$$\begin{aligned}&(\forall\ d\ p, \mathit{allow}\ d\ p \to \forall\ m, \neg \mathit{user}\ d\ p\ m) \land \\ &(\forall\ d\ p\ m, \mathit{user}\ d\ p\ m \to \neg \mathit{allow}\ d\ p\ \land \forall\ m', \mathit{user}\ d\ p\ m' \to m = m')\end{aligned} \tag{8}$$

and such that *allow* $d\ p$ holds when domain d unconditionally grants the permission p and *user* $d\ p\ m$ holds when domain d grants permission p with explicit user authorization and maximum allowable mode m (w.r.t. $\leq_{\text{m}}$). The permissions effectively granted to a MIDlet suite are the intersection of the permissions requested in its descriptor with the union of the permissions given unconditionally by its domain and those given explicitly by the user.

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

$$\begin{aligned} SessionInfo \overset{\text{def}}{=} \{\, & id &&: SuiteID, \\ & granted, revoked &&: Permission \to Prop \,\} \end{aligned} \tag{9}$$

The abstract device state is described as a record of type

$$\begin{aligned} State \overset{\text{def}}{=} \{\, & suite &&: Suite \to Prop, \\ & session &&: option\ SessionInfo, \\ & granted, revoked &&: SuiteID \to Permission \to Prop \,\} \end{aligned} \tag{10}$$

where *suite* is the characteristic predicate of the set of installed suites.

Example 1. Consider a MIDlet that periodically connects to a webmail service using HTTPS when possible or HTTP otherwise, and alerts the user whenever they have new mail. The suite containing this MIDlet should declare in its descriptor as required permissions *p_push*, for accessing the PushRegistry (for timer-based activation), and *p_http*, for using the HTTP protocol API. It should also declare as optional the permission *p_https* for accessing the HTTPS protocol API. Suppose that upon installation, the suite (whose identifier is *id*) is recognized as trusted and is thus bound to a protection domain *dom* that allows access to the PushRegistry API unconditionally but by default requests user authorization for opening every HTTP or HTTPS connection. Suppose also that the domain allows the user to grant the MIDlet the permission for opening further connections as long as the suite remains installed. Then, the security policy satisfies:

$$allow\ dom\ p\_push \wedge user\ dom\ p\_http\ blanket \wedge user\ dom\ p\_https\ blanket \tag{11}$$

If st is the state of the device, the suite is represented by some $ms : Suite$ such that $st.suite\ ms$ and $ms.id = id$ hold. Its descriptor $ms.descriptor$ satisfies

$$\begin{aligned} & ms.descriptor.required\ p\_push \wedge ms.descriptor.required\ p\_http \wedge \\ & ms.descriptor.optional\ p\_https \end{aligned} \tag{12}$$

The MIDlet will have unlimited access to the PushRegistry applet, but will have to request user authorization every time it makes a new connection. The user may chose at any time to authorize further connections by granting the corresponding permission in *blanket* mode, thus avoiding being asked for authorization each time the applet communicates with the webmail service.

The remainder of this subsection enumerates the conditions that must hold for an element $s : State$ in order to represent a valid state for a device.

1. 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,

$$\begin{aligned}&des \wr dom \overset{\text{def}}{=} \forall\ p : Permission,\\ &\quad des.required\ p \to allow\ dom\ p\ \ \vee \exists\ m : Mode, user\ dom\ p\ m\end{aligned} \tag{13}$$

   Every installed suite must be compatible with its associated protection domain,

$$\begin{aligned}&SuiteCompatible \overset{\text{def}}{=}\\ &\quad \forall\ ms : Suite, s.suite\ ms \to ms.descriptor \wr ms.domain\end{aligned} \tag{14}$$

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

$$\begin{aligned}&CurrentInstalled \overset{\text{def}}{=} \forall\ ses : SessionInfo, s.session = Some\ ses \to\\ &\quad \exists\ ms : Suite, s.suite\ ms \wedge ms.id = ses.id\end{aligned} \tag{15}$$

3. The set of permissions granted for the session must be a subset of the permissions requested in the application descriptor of the active suite. In addition, the associated protection domain policy must allow those permissions to be granted at least in *session* mode,

$$\begin{aligned}&ValidSessionGranted \overset{\text{def}}{=} \forall\ ses : SessionInfo, s.session = Some\ ses \to\\ &\quad \forall\ p : Permission, ses.granted\ p \to\\ &\qquad \forall\ ms : Suite, s.suite\ ms \to ms.id = ses.id \to\\ &\qquad\quad (ms.descriptor.required\ p\ \ \vee ms.descriptor.optional\ p) \wedge\\ &\qquad\quad (\exists\ m : Mode, user\ ms.domain\ p\ m\ \ \wedge session \leq_{\mathrm{m}} m)\end{aligned} \tag{16}$$

4. Every installed suite shall have a unique identifier,

$$\begin{aligned}&UniqueSuiteID \overset{\text{def}}{=} \forall\ ms_1\ ms_2 : Suite,\\ &\quad s.suite\ ms_1 \to s.suite\ ms_2 \to ms_1.id = ms_2.id \to ms_1 = ms_2\end{aligned} \tag{17}$$

5. For every installed suite with identifier *id*, the predicate *s.granted id* should be valid with respect to its descriptor and associated protection domain ($ValidGranted\ s$). We omit the detailed formalization of this condition.
6. A granted permission shall not be revoked at the same time and viceversa ($ValidGrantedRevoked\ s$). We omit the detailed formalization.

3.3 Events

We define a set *Event* for those events that are relevant to our abstraction of the device state (Table 1). The user may be presented with the choice between accepting or refusing an authorization request, specifying the period of time their choice remains valid. The outcome of a user interaction is represented using the type $UserAnswer$ with constructors

$$ua\_allow, ua\_deny : Mode \rightarrow UserAnswer \tag{18}$$

Table 1. Events

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 behaviour of the events is specified by their pre- and postconditions given by 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 operation is authorized,

$$\begin{array}{l} Pre : State \rightarrow Event \rightarrow Prop \\ Pos : State \rightarrow State \rightarrow option\ Response \rightarrow Event \rightarrow Prop \end{array} \tag{19}$$

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

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

Example 2. Consider an event representing a permission request for which the user denies the authorization. Such an event can only occur when the active

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

$$
\begin{array}{l}
Pos\ s\ s'\ r\ (start\ id) \overset{\text{def}}{=} \\
\quad r = None\ \land s \equiv_{session} s'\ \land\ \exists\ ses', s'.session = Some\ ses'\ \land ses'.id = id \land \\
\quad \forall\ p : Permission, \neg ses'.granted\ p\ \land \neg ses'.revoked\ p \\
Pos\ s\ s'\ r\ terminate \overset{\text{def}}{=} r = None\ \land s \equiv_{session} s'\ \land s'.session = None \\
Pos\ s\ s'\ r\ (install\ id\ des\ dom) \overset{\text{def}}{=} \\
\quad r = None\ \land\ (\forall\ ms : Suite, s.suite\ ms \to s'.suite\ ms) \land \\
\quad (\forall\ ms : Suite, s'.suite\ ms \to s.suite\ ms\ \lor ms = \langle id, dom, des \rangle) \land \\
\quad s'.suite\ \langle id, dom, des \rangle\ \land s'.session = s.session \land \\
\quad (\forall\ p : Permission, \neg s'.granted\ id\ p \land \neg s'.revoked\ id\ p) \land \\
\quad (\forall\ id_1 : SuiteID, id_1 \neq id \to \\
\quad\quad s'.granted\ id_1 = s.granted\ id_1\ \land s'.revoked\ id_1 = s.revoked\ id_1) \\
Pos\ s\ s'\ r\ (remove\ id) \overset{\text{def}}{=} r = None\ \land s \equiv_{suite} s'\ \land \\
\quad (\forall\ ms : Suite, s.suite\ ms \to ms.id \neq id \to s'.suite\ ms) \land \\
\quad (\forall\ ms : Suite, s'.suite\ ms \to s.suite\ ms \land ms.id \neq id)
\end{array}
$$

suite has declared the requested permission in its descriptor and is bound to a protection domain that specifies a user interaction mode for that 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 rest of the suite's life,

$$
\begin{array}{l}
Pre\ s\ (request\ p\ (Some\ (ua\_deny\ m))) \overset{\text{def}}{=} \\
\quad \exists\ ses : SessionInfo, s.session = Some\ ses \land \\
\quad\quad \forall\ ms : Suite, s.suite\ ms \to ms.id = ses.id \to \\
\quad\quad\quad (ms.descriptor.required\ p\ \lor ms.descriptor.optional\ p) \land \\
\quad\quad\quad (\exists\ m_1 : Mode, user\ ms.domain\ p\ m_1) \land \\
\quad\quad\quad \neg ses.granted\ p\ \land\ \neg ses.revoked\ p \land \\
\quad\quad\quad \neg s.granted\ ses.id\ p\ \land \neg s.revoked\ ses.id\ p
\end{array}
\tag{20}
$$

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

$$
\begin{array}{l}
Pos\ s\ s'\ r\ (request\ p\ (Some\ (ua\_deny\ session))) \overset{\text{def}}{=} \\
\quad r = Some\ denied\ \land s \equiv_{session} s'\ \land \\
\quad \forall\ ses : SessionInfo, s.session = Some\ ses \to \\
\quad\quad \exists\ ses' : SessionInfo, \\
\quad\quad\quad s'.session = Some\ ses'\ \land ses' \equiv_{revoked} ses\ \land ses'.revoked\ p \land \\
\quad\quad\quad (\forall\ q : Permission, q \neq p \to ses'.revoked\ q = ses.revoked\ q)
\end{array}
\tag{21}
$$

3.4 One-step Execution

The behavioural specification of the execution of an event is given by the $\hookrightarrow$ relation with the following introduction rules:

$$\frac{\neg Pre\ s\ e}{s \xhookrightarrow{e/None} s}\ npre \qquad \frac{Pre\ s\ e \quad Pos\ s\ s'\ r\ e}{s \xhookrightarrow{e/r} s'}\ pre \tag{22}$$

Whenever 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 notation $s \xhookrightarrow{e/r} s'$ may be read as "the execution of the event e in state s results in a new state s' and produces a response r".

3.5 Sessions

A session is the period of time spanning from a successful *start* event to a *terminate* event, in which a single suite remains active. A session for a suite with identifier *id* (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,

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

$$\begin{tikzcd} s_0 \arrow[r, hook, "start\ id/r_1"] & s_1 \arrow[r, hook, "e_2/r_2"] & s_2 \arrow[r, hook, "e_3/r_3"] & \cdots \arrow[r, hook, "e_{n-1}/r_{n-1}"] & s_{n-1} \arrow[r, hook, "terminate/r_n"] & s_n \end{tikzcd}$$

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

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

$$\frac{Pre\ s_0\ (start\ id) \quad s_0 \xhookrightarrow{start\ id/r_1} s_1}{PSession\ s_0\ ([\,] \frown \langle start\ id, s_1, r_1 \rangle)}\ psession\_start \tag{23}$$

$$\frac{PSession\ s_0\ (ss \frown last) \quad e \neq terminate \quad last.s \xhookrightarrow{e/r} s'}{PSession\ s_0\ (ss \frown last \frown \langle e, s', r \rangle)}\ psession\_app \tag{24}$$

Now, sessions can be easily defined as follows,

$$\frac{PSession\ s_0\ (ss \frown last) \quad last.s \xhookrightarrow{terminate/r} s'}{Session\ s_0\ (ss \frown last \frown \langle terminate, s', r \rangle)}\ session\_terminate \tag{25}$$

4 Verification of Security Properties

This section is devoted to establishing relevant security properties of the model. Due to space constraints, proofs are merely outlined; however, all proofs have been formalized in Coq and are available as part of the full specification.

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 show next that the validity of the device state, as defined in Section 3.2, is 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 $s \xhookrightarrow{e/r} s'$ hold, then $Valid\ s'$ also holds.*

Proof. By case analysis on $s \xhookrightarrow{e/r} s'$. 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 (21) entails that $s \equiv_{session} s'$, that the session remains active, and that $ses' \equiv_{revoked} ses$. Therefore, the set of installed suites remains unchanged ($s'.suite = s.suite$), the set of permissions granted for the session does not change ($ses'.granted = ses.granted$) and neither does the set of permissions granted or revoked in *blanket* mode ($s'.granted = s.granted$, $s'.revoked = s.revoked$). From these equalities, every validity condition of the state s' except $ValidGrantedRevoked\ s'$ follows immediately from the validity of s. We next prove $ValidGrantedRevoked\ s'$.

We know from the postcondition of the even that

$$\forall\ q, q \neq p \rightarrow ses'.revoked\ q = ses.revoked\ q \tag{26}$$

Let q be any permission. If $q \neq p$, then from (26) follows $ses'.revoked\ q = ses.revoked\ q$ and because q was not granted and revoked simultaneously before the event, neither it is afterwards. If $q = p$, then we know from the precondition (20) that p were not granted before and thus it is not granted afterwards. 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 over $seq\ T$ by the following rules:

$$\frac{}{all\ P\ [\,]}\ all\_nil \qquad \frac{all\ P\ ss \quad P\ s}{all\ P\ (ss \frown s)}\ all\_snoc \tag{27}$$

Theorem 2. *Let s_0 be a valid state and ss a partial session starting from s_0, then every state in ss is valid,*

$$all\ (\lambda\ step\ .\ Valid\ step.s)\ ss \tag{28}$$

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

– When constructed using *psession_start*, ss has the form $[\,]^\frown\langle start\ id, s_1,\ r_1\rangle$ and $s_0 \xhookrightarrow{start\ id/r_1} s_1$ holds. We must prove

$$all\ (\lambda\ step\ .\ Valid\ step.s)\ ([\,]^\frown \langle start\ id, s_1,\ r_1\rangle) \tag{29}$$

By applying *all_app* and then *all_nil* the goal is simplified to $Valid\ s_1$ and is proved from $s_0 \xhookrightarrow{start\ id/r_1} s_1$ 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 $last.s \xhookrightarrow{e/r} s'$ holds. The induction hypothesis is

$$all\ (\lambda\ step\ .\ Valid\ step.s)\ (ss_1 {}^\frown last) \tag{30}$$

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 (30) the goal is simplified to $Valid\ s'$. From (30) we know that $last.s$ is a valid state. The goal is proved from $last.s \xhookrightarrow{e/r} s'$ and $Valid\ last.s$ by applying Theorem 1. □

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

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

$$\begin{aligned}&\forall\ (s\ s' : State)(r\ r' : option\ Response)(e\ e' : Event),\\ &\quad e' \neq terminate \to s \xhookrightarrow{e'/r'} s' \to P\ \langle e, s, r\rangle \to P\ \langle e', s', r'\rangle\ ,\end{aligned} \tag{31}$$

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 a guarantee of the proper enforcement of revocation. We prove 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 refused.

Lemma 2. *The following property satisfies* (31),

$$(\lambda\ step\ .\ \exists\ ses, step.s.session = Some\ ses \wedge ses.revoked\ p) \tag{32}$$

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*

$$all\ (\lambda\ step\ .\ \forall\ o, step.e = request\ p\ o \to step.r \neq Some\ allowed)\ ss_1 \tag{33}$$

Proof. Since $Pos\ step.s\ step_1.s\ step_1.r\ step_1.e$ must hold, 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 $step_2$ in ss_1. We know that p is revoked for the session in the state before $step_2.s$. If the precondition for e does not hold in the state before[3], then $step_2.r = None$. Otherwise, e must be $request\ p\ None$ and its postcondition entails $step_2.r = Some\ denied$. □

5 Refinement

In the formalization described in the previous sections we have specified the behaviour of events implicitly as a binary relation on states instead of explicitly as a state transformer. Moreover, the described formalization is higher-order because, for instance, predicates are used to represent part of the device state and the transition semantics of events is given as a relation on states. The most evident consequence of this choice is that the resulting specification is not executable. What is more, the program extraction mechanism provided by Coq to extract programs from specifications cannot be used in this case. However, had we constructed a more concrete specification at first, we would have had to take arbitrary design decisions from the beginning, unnecessarily restricting the allowable implementations and complicating the verification of properties of the security model.

We will show in the rest of this section that it is feasible to obtain an executable specification from our abstract specification. The methodology we propose produces also a proof that the former is a refinement of the latter, thus guaranteeing soundness of the entire process. The methodology is inspired by the work of Spivey [9] on operation and data refinement, and the more comprehensive works of Back and von Wright [10] and Morgan [11] on refinement calculus.

5.1 Executable Specification

In order to construct an executable specification it is first necessary to choose a concrete representation for every object in the original specification not directly implementable in a functional language. In particular, the transition relation that defines the behaviour of events implicitly by means of their pre- and postconditions must be refined to a function that deterministically computes the outcome of an event. At this point, it is unavoidable to take some arbitrary decisions about the exact representation to use. For example, a decidable predicate P on a finite set A might be represented as a function from A to a type isomorphic to *bool*, as a exhaustive list of the elements of A that satisfies the predicate, or in some other equally expressive way. For every type T in the abstract specification, we will denote its concrete model as $\overline{T}$. Let $a : A$ and $\overline{a} : \overline{A}$, we will indicate that $\overline{a}$ is a refinement of a as $a \sqsubseteq \overline{a}$.

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

In our case, every predicate to be refined is decidable and is satisfied only by a finite subset of elements in its domain (they are all characteristic predicates of finite sets). Let P be one of such predicates on a set A and let l be a list of elements of $\overline{A}$, we will say that l refines P whenever

$$\begin{array}{l}(\forall\ a, P\ a \to \exists\ \overline{a}, \overline{a} \in l\ \wedge\ a \sqsubseteq \overline{a})\ \wedge \\ (\forall\ \overline{a}, \overline{a} \in l \to \exists\ a, P\ a\ \wedge\ a \sqsubseteq \overline{a})\end{array} \tag{34}$$

where $x \in l$ means that there exists at least one occurrence of x in l. When A and $\overline{A}$ coincide, $\sqsubseteq$ is the equality relation on A, and the condition (34) simplifies to $\forall\ a, P\ a \leftrightarrow a \in l$. Let $a : A$ and $\overline{a} : \overline{A}$ be such that $a \sqsubseteq \overline{a}$, we define

$$\begin{array}{ll}(None : option\ A) & \sqsubseteq (None : option\ \overline{A}) \\ Some\ a & \sqsubseteq Some\ \overline{a}\end{array} \tag{35}$$

The above concrete representations can be used to obtain a concrete model for the device state and the security-related events:

$$\begin{array}{lll}\overline{State} := \{ & suite & : list\ \overline{Suite}, \\ & session & : option\ \overline{SessionInfo}, \\ & granted, revoked & : SuiteID \to list\ Permission\ \}\end{array} \tag{36}$$

$$\begin{array}{ll}\overline{start} & : SuiteID \to \overline{Event} \\ \overline{terminate} & : \overline{Event} \\ \overline{request} & : Permission \to option\ UserAnswer \to \overline{Event} \\ \overline{install} & : SuiteID \to \overline{Descriptor} \to Domain \to \overline{Event} \\ \overline{remove} & : SuiteID \to \overline{Event}\end{array} \tag{37}$$

The refinement relation $\sqsubseteq$ can be naturally extended to states, events and the rest of the types used in the formalization.

5.2 Soundness

Having chosen a concrete representation for the objects in the specification, everything is set for specifying the behaviour of events as a function

$$interp : \overline{State} \to \overline{Event} \to \overline{State} \times (option\ Response) \tag{38}$$

The soundness of the $interp$ function w.r.t. the transition relation $\hookrightarrow$ is given by the following simulation condition, illustrated in Fig. 2.

$$\begin{array}{l}\forall\ (s : State)\ \left(\overline{s} : \overline{State}\right)\ (e : Event)\ \left(\overline{e} : \overline{Event}\right)\ (r : option\ Response), \\ \quad s \sqsubseteq \overline{s} \to e \sqsubseteq \overline{e} \to \\ \quad \text{let } (\overline{s}', r) := interp\ \overline{s}\ \overline{e} \text{ in } \exists\ s' : State, s' \sqsubseteq \overline{s}'\ \wedge\ s \xhookrightarrow{e/r} s'\end{array} \tag{39}$$

It can be shown that the refinement relation $\sqsubseteq$ on states satisfies

$$\forall\ \overline{s} : \overline{State}, \exists\ s : State, s \sqsubseteq \overline{s} \tag{40}$$

$$\begin{tikzcd}
State & s \arrow[r, dashed, "\hookrightarrow"] \arrow[d, "\sqsubseteq"'] & s' \arrow[d, "\sqsubseteq"] \\
\overline{State} & \overline{s} \arrow[r, "interp"] & \overline{s}'
\end{tikzcd}$$

Fig. 2. Simulation relation between *interp* and the relation $\hookrightarrow$. Given sates s, $\overline{s}$ and events e, $\overline{e}$ such that $s \sqsubseteq \overline{s}$ y $e \sqsubseteq \overline{e}$, for every state $\overline{s}'$ and response r computed by *interp* there must exists a corresponding abstract state s' refined by $\overline{s}'$ reachable from s by the $\hookrightarrow$ relation with the same response

Thus, the existential quantifier in the above condition may be replaced by a universal quantifier to obtain the stronger (but sometimes easier to prove) condition:

$$\begin{aligned}
&\forall\ (s\ s' : State)\ \left(\overline{s} : \overline{State}\right)\ (e : Event)\ \left(\overline{e} : \overline{Event}\right)\ (r : option\ Response), \\
&\quad s \sqsubseteq \overline{s} \rightarrow e \sqsubseteq \overline{e} \rightarrow \\
&\quad \text{let } \left(\overline{s}', r\right) := interp\ \overline{s}\ \overline{e} \text{ in } s' \sqsubseteq \overline{s}' \ \rightarrow s \xhookrightarrow{e/r} s'
\end{aligned} \tag{41}$$

With a function *interp* satisfying either (39) or (41) and a concrete initial state $\overline{s_0}$ that refines an initial abstract state s_0, the Coq program extraction mechanism can be used to produce an executable prototype of the MIDP 2.0 security model in a functional language such as OCaml, Haskell or Scheme.

6 Conclusions

The informal specification in [2] puts forward an application security model that any MIDP 2.0 enabled device must satisfy. Although analyses of particular implementations have been proved useful for discovering vulnerabilities, so far the problem of the verification of the security model has not been addressed. We believe our contribution constitutes an excellent starting point for a thorough verification of the model, which would give a higher assurance level than the techniques applied so far.

We have produced, to the best of our knowledge, an unprecedented verifiable formalization 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 fulfils its specification. It is unclear from the MIDP 2.0 specification exactly how other mechanisms interact with the security model. Our formalization 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. This issues have not been treated anywhere else.

The specification has been completed in 4 man-months and comprises around 2200 lines, about 1000 of which are dedicated to proofs. In its construction, two simplifying assumptions have been made:

1. the security policy is static;
2. up to one suite may be active at a time.

Actually, most implementations (if not all) enforce these assumptions. However, the MIDP 2.0 specification does not and therefore, it would be interesting to explore the consequences of relaxing them by extending the formalization. An orthogonal direction is to divert from the MIDP 2.0 specification, enriching the model to allow more expressive policies. We imagine two different possibilities:

1. abandon the unstructured model of permissions in favour of a hierarchical one;
2. generalize user interaction modes.

The former would allow group of permissions to be revoked or granted according to a tree structure, perhaps exploiting the already hierarchical naming of permissions in MIDP. The latter would allow richer policies such as granting a permission for a given number of uses or sessions. In a recent, as yet unpublished work Besson, Dufay and Jensen [12] describe a generalized model of access control for mobile devices that follows these directions to some extent.

We have also proposed a refinement methodology that might be used to obtain a sound executable prototype of the security model. Although we do not show it here, we have followed this methodology with a restricted set of events. Judging by the success of this experience, we strongly believe it is feasible to obtain a prototype for the whole set of events.

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. Kolsi, O., Virtanen, T.: MIDP 2.0 security enhancements. In: Proceedings of the 37th Annual Hawaii International Conference on System Sciences, Washington, DC, USA, IEEE Computer Society (2004) 90287.3
4. Debbabi, M., Saleh, M., Talhi, C., Zhioua, S.: Security analysis of wireless Java. In: Proceedings of the 3rd Annual Conference on Privacy, Security and Trust. (2005)
5. The Coq Development Team: The Coq Proof Assistant Reference Manual – Version V8.0. (2004)
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer-Verlag (2004)

7. Zanella Béguelin, S.: Especificación formal del modelo de seguridad de MIDP 2.0 en el Cálculo de Construcciones Inductivas. Master's thesis, Universidad Nacional de Rosario (2006)
8. Zanella Béguelin, S., Betarte, G., Luna, C.: A formal specification of the MIDP 2.0 security model. Technical Report 06-09, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay (2006)
9. Spivey, J.M.: The Z Notation: A Reference Manual. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK (1989)
10. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer-Verlag (1998) Graduate Texts in Computer Science.
11. Morgan, C.: Programming from specifications. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1990)
12. Besson, F., Dufay, G., Jensen, T.: A formal model of access control for mobile interactive devices. In: Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS). Lecture Notes in Computer Science, Springer-Verlag (2006) To appear.