A Formal Specification of the MIDP 2.0 Security Model

Santiago Zanella Béguelin\textsuperscript{1}  
Gustavo Betarte\textsuperscript{2}  
Carlos Luna\textsuperscript{2}

\textsuperscript{1}Everest Project, INRIA Sophia Antipolis  
INRIA–Microsoft Research Joint Laboratory

\textsuperscript{2}InCo, Universidad de la República, Uruguay

Workshop on Formal Aspects in Security and Trust, 2006
What is a Mobile Device?

<table>
<thead>
<tr>
<th>Defining characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>portable</td>
</tr>
<tr>
<td>scarce resources (compared with other platforms)</td>
</tr>
<tr>
<td>communicated</td>
</tr>
<tr>
<td>stores personal information</td>
</tr>
<tr>
<td>subscribed to pay-per-use services</td>
</tr>
</tbody>
</table>
Some Examples

Cell Phones  Personal Digital Assistants

A Formal Specification of the MIDP 2.0 Security Model
The Problem

What a secure mobile device should enforce:

- Data confidentiality and integrity
- Cost control
- Availability

...even in the presence of malicious applications
The Problem

What a **secure** mobile device should enforce:
- Data confidentiality and integrity
- Cost control
- Availability

...even in the presence of malicious applications
If the device supports loading of executable code after issuance...

A

B

Zanella Béguelin, Betarte, Luna

A Formal Specification of the MIDP 2.0 Security Model
If the device supports loading of executable code after issuance...

A

B
If the device supports loading of executable code after issuance...

A

B
If the device supports loading of executable code after issuance...
If the device supports loading of executable code after issuance...
If the device supports loading of executable code after issuance...
First Solution
Removing the cause

Either

- Don’t allow users to download code
  but they love to do so
  (and it’s a big market opportunity)
- Don’t allow downloaded code to access sensitive APIs
  but many useful applications must do so
  (e.g. synchronization, news push)

Roughly, MIDP 1.0 used this last solution (a sandbox model)
A security policy is a mapping from a set of properties that characterize code to a set of access permissions granted to that code.
A security policy is a mapping from a set of properties that characterize code to a set of access permissions granted to that code.
Layered J2ME - MIDP architecture

- Users may only download MIDP applications
- MIDP applications access resources through restricted interface
In MIDP 1.0, sandbox-like model
In MIDP 2.0, model based on protection domains

Protection Domain
- It’s an abstraction of the context of execution of a piece of code
- Restricts access to sensitive functions
- In MIDP 2.0, each application belongs to a suite and each suite is bound to a unique Protection Domain
In MIDP 1.0, sandbox-like model
In MIDP 2.0, model based on protection domains

Protection Domain

- It’s an abstraction of the context of execution of a piece of code
- Restricts access to sensitive functions
- In MIDP 2.0, each application belongs to a suite and each suite is bound to a unique Protection Domain
Protection Domains in Practice

A Formal Specification of the MIDP 2.0 Security Model
Protection Domains in Practice

Policy

Motivation  Specification  Verification  Refinement

Zanella Béguelin, Betarte, Luna

A Formal Specification of the MIDP 2.0 Security Model
Protected function \(\rightarrow\) Permission

A Protection Domain determines:

- A set of permissions granted unconditionally
- A set of permissions that could be granted with explicit user authorization, together with a mode that specifies its validity
  - \texttt{blanket} until the removal of the suite
  - \texttt{session} for the current session
  - \texttt{oneshot} for a single use

\[
\text{oneshot} \leq_m \text{session} \leq_m \text{blanket}.
\]

The specified mode is an upper bound
Permissions Acquired by a Suite

A suite declares at installation time the permissions it requires

- Permissions required by the suite
- Permissions granted unconditionally
- Permissions granted by an explicit user authorization
A suite declares at installation time the permissions it requires

\[ \text{Acquired} = \text{Requested} \cap (\text{Unconditionally granted} \cup \text{Granted by user authorization}) \]
New Problems

Issues

- Does the security model enforce the security policy?
- Do implementations conform to the model?
- How do other operations interfere with the model?
New Problems

Issues

- Does the security model enforce the security policy?
- Do implementations conform to the model?
- How do other operations interfere with the model?
- What is exactly the security model?
Outline

1. Motivation
2. Specification
3. Verification
4. Refinement
Remarks

- Formalized in the Calculus of Inductive Constructions
- Developed with the Coq proof assistant
- Abstract higher-order specification
Motivation

Specification

Verification

Refinement

The Calculus of Inductive Constructions

CIC

CIC is an extension of the simple-typed lambda calculus with:

- Polymorphic types \([(\lambda x \cdot x) : A \rightarrow A]\)
- Higher-order types \([A \rightarrow A : * : □]\)
- Dependent types \([(\lambda a : A \cdot f a) : (\forall a : A . B_a)]\)

- Implemented in Coq
  Type checker + Proof assistant
- Can encode higher-order predicate logic
- Inductive definitions
- Curry-Howard isomorphism
  types ↔ propositions
  terms ↔ proofs

Zanella Béguelin, Betarte, Luna
A Formal Specification of the MIDP 2.0 Security Model
Formalizing the state of the device

State components relevant to the security model:

- installed suites
- current session (if it exists)
  - current suite
  - permissions granted or revoked in session mode
- permissions granted or revoked for the session in blanket mode

State := {
  suite : Suite → Prop,
  session : option SessionInfo,
  granted, revoked : SuiteID → Permission → Prop
}

Higher-order specification (notice predicates in the state)
Formalizing the state of the device

State components relevant to the security model:

- installed suites
- current session (if it exists)
  - current suite
  - permissions granted or revoked in session mode
- permissions granted or revoked for the session in blanket mode

$$\text{State} := \{ \text{suite} : \text{Suite} \rightarrow \text{Prop}, \text{session} : \text{option SessionInfo}, \text{granted, revoked} : \text{SuiteID} \rightarrow \text{Permission} \rightarrow \text{Prop} \}$$

Higher-order specification (notice predicates in the state)
Events

- Session start *(start)*;
- Session end *(terminate)*;
- Authorization request by the current suite *(request)*;
- Suite installation *(install)*;
- Suite removal *(remove)*.

Their behavior is specified by means of pre- and postconditions.

**Example (Session start)**

\[
\text{Pre } s \text{ (start id)} = \\
\text{s.session} = \text{None} \land \exists \ ms : \text{Suite}, \ s.\text{suite } ms \land ms.\text{id} = id
\]

\[
\text{Pos } s s' r \text{ (start id)} = r = \text{None} \land s \equiv_{\text{session}} s' \land \\
\exists \ ses', s'.\text{session} = ses' \land ses'.\text{id} = id \land \\
\forall \ p : \text{Permission}, \neg ses'.\text{granted } p \land \neg ses'.\text{revoked } p
\]
State transition relation \(\leftrightarrow\):

\[
\begin{align*}
\neg \text{Pre } s \quad e & \quad \text{npre} & \quad \text{Pre } s \quad e & \quad \text{Pos } s \quad s' \quad r \quad e & \quad \text{pre} \\
\quad s & \quad e/\text{None} & \quad s & \quad s' \quad e/r & \quad s'
\end{align*}
\]

\(s \quad e/r \quad s'\): “the execution of the event \(e\) in state \(s\) results in a new state \(s'\) and produces a response \(r\)”
A session is determined by

- a suite identifier \( id \)
- an initial state \( s_0 \)
- a sequence of steps \( \langle e_i, s_i, r_i \rangle \) \( (i = 1, \ldots, n) \) s.t.

  \[
  \begin{align*}
  e_1 &= \text{start id} \\
  \text{Pre} \ s_0 \ e_1 \\
  \forall i \in \{2, \ldots, n-1\}, \ e_i &\neq \text{terminate} \\
  e_n &= \text{terminate} \\
  \forall i \in \{1, \ldots, n\}, \ s_{i-1} &\overset{e_i/r_i}{\rightarrow} s_i
  \end{align*}
  \]
A session is determined by

- a suite identifier \( id \)
- an initial state \( s_0 \)
- a sequence of steps \( \langle e_i, s_i, r_i \rangle \) (\( i = 1, \ldots, n \)) s.t.

1. \( e_1 = \text{start id} \)
2. \( \text{Pre } s_0 \ e_1 \)
3. \( \forall i \in \{2, \ldots, n-1\}, e_i \neq \text{terminate} \)
4. \( e_n = \text{terminate} \)
5. \( \forall i \in \{1, \ldots, n\}, s_{i-1} \xrightarrow{e_i/r_i} s_i \)
A session is determined by

- a suite identifier \( id \)
- an initial state \( s_0 \)
- a sequence of steps \( \langle e_i, s_i, r_i \rangle \) (\( i = 1, \ldots, n \)) s.t.

1. \( e_1 = \text{start id} \)
2. \( \text{Pre } s_0 \ e_1 \)
3. \( \forall i \in \{2, \ldots, n-1\}, e_i \neq \text{terminate} \)
4. \( e_n = \text{terminate} \)
5. \( \forall i \in \{1, \ldots, n\}, s_{i-1} \xrightarrow{e_i/r_i} s_i \)
Sessions

Inductive definition

$$S_0 \xrightarrow{\text{start id}/r_1} S_1 \xrightarrow{e_2/r_2} S_2 \xrightarrow{e_3/r_3} \ldots \xrightarrow{e_{n-1}/r_{n-1}} S_{n-1} \xrightarrow{\text{terminate}/r_n} S_n$$

$$Pre \ s_0 (\text{start id}) \quad s_0 \xrightarrow{\text{start id}/r_1} s_1 \quad \text{pses}_\text{start}$$

$$PSession \ s_0 ([ ] \bowtie \langle \text{start id}, s_1, r_1 \rangle)$$

$$PSession \ s_0 (ss \bowtie \text{last}) \quad e \neq \text{terminate} \quad \text{last.s} \xrightarrow{e/r} s' \quad \text{pses}_\text{app}$$

$$PSession \ s_0 (ss \bowtie \text{last} \bowtie \langle e, s', r \rangle)$$

$$PSession \ s_0 (ss \bowtie \text{last}) \quad \text{last.s} \xrightarrow{\text{terminate}/r} s' \quad \text{ses}_\text{term}$$

$$Session \ s_0 (ss \bowtie \text{last} \bowtie \langle \text{terminate}, s', r \rangle)$$
Sessions

Inductive definition

\[ \begin{align*}
S_0 & \xrightarrow{\text{start id} / r_1} S_1 \\
S_1 & \xrightarrow{e_2 / r_2} S_2 \\
S_2 & \xrightarrow{e_3 / r_3} \cdots \\
S_{n-1} & \xrightarrow{e_{n-1} / r_{n-1}} S_n
\end{align*} \]

Pre \(s_0\) (start id) \(\xrightarrow{\text{start id} / r_1} s_1\) \(\text{pses\_start}\)

\(\text{PSession } s_0\) ([ ] \(\bowtie\) \(\langle\text{start id}, s_1, r_1\rangle\))

\(\text{PSession } s_0\) (\(ss \bowtie\) last) \(\xrightarrow{e \neq \text{terminate}}\) last.s \(\xrightarrow{e / r} s'\) \(\text{pses\_app}\)

\(\text{PSession } s_0\) (\(ss \bowtie\) last \(\bowtie\) \(\langle e, s', r\rangle\))

\(\text{PSession } s_0\) (\(ss \bowtie\) last) last.s \(\xrightarrow{\text{terminate} / r} s'\) \(\text{ses\_term}\)

\(\text{Session } s_0\) (\(ss \bowtie\) last \(\bowtie\) \(\langle\text{terminate}, s', r\rangle\))
Sessions
Inductive definition

\[ S_0 \xrightarrow{\text{start id}/r_1} S_1 \xrightarrow{e_2/r_2} S_2 \xrightarrow{e_3/r_3} \cdots \xrightarrow{e_{n-1}/r_{n-1}} S_{n-1} \xrightarrow{\text{terminate}/r_n} S_n \]

**Pre s_0 (start id)**

\[ \text{Pre } s_0 (\text{start id}) \]

\[ s_0 \xrightarrow{\text{start id}/r_1} s_1 \]

**PSession s_0 ([ ] ∩ ⟨start id, s_1, r_1⟩)**

**PSession s_0 (ss ∩ last)**

\[ e \neq \text{terminate} \]

\[ \text{last.s} \xrightarrow{e/r} s' \]

**PSession s_0 (ss ∩ last ∩ ⟨e, s', r⟩)**

**PSession s_0 (ss ∩ last)**

\[ \text{last.s} \xrightarrow{\text{terminate}/r} s' \]

**Session s_0 (ss ∩ last ∩ ⟨terminate, s', r⟩)**

Zanella Béguelin, Betarte, Luna

A Formal Specification of the MIDP 2.0 Security Model
Outline

1. Motivation
2. Specification
3. Verification
4. Refinement

A Formal Specification of the MIDP 2.0 Security Model
The formal specification defines a theory
Properties of the security model are theorems
We state and prove some of these theorems with the help of Coq
State validity is an invariant

\[ \forall (s \, s' : \text{State}) \, (e : \text{Event}) \, (r : \text{Response}) \]

\[ \text{Valid } s \rightarrow s \xrightarrow{e/r} s' \rightarrow \text{Valid } s' \]

A state is valid if (among other things)

- Suite identifiers are unique;
- The current suite is an installed suite;
- Granted permissions are consistent with corresponding protection domains and application descriptors;
- Permissions required as critical by a suite are not forbidden by its protection domain.
Some Proved Theorems

Motivation Specification Verification Refinement

Revocation of permissions is correctly enforced

Whenever a permission is revoked in session mode, subsequent authorization requests are refused

Generalization of invariants

- Sufficient and necessary conditions for invariants
- Theorem: one-step invariants remain true once established
1. Motivation
2. Specification
3. Verification
4. Refinement
Why Should We Care?

Remarks
- We have a higher-order specification
- Transition relation defined implicitly
- Coq program extraction mechanism cannot be used

What is the pay off of refinement?
- An executable prototype
- An oracle for testing
- Test case extraction (*black box testing*)
For each type $T$, a concrete type $\overline{T}$ is defined

$x \sqsubseteq \overline{x}$ is read “$x$ is refined by $\overline{x}$”

**Example (Predicates as lists)**

Let $P : A \rightarrow Prop$ and $l : list \overline{A}$, then $l \sqsubseteq P$ iff

\[
(\forall a, P a \rightarrow \exists \overline{a}, \overline{a} \in l \land a \sqsubseteq \overline{a}) \wedge \\
(\forall \overline{a}, \overline{a} \in l \rightarrow \exists a, P a \land a \sqsubseteq \overline{a})
\]

Whenever $A = \overline{A}$ this simplifies to

$\forall a, P a \leftrightarrow a \in l$
Concrete State

\[
\text{State} := \{ \text{suite} : \text{list} \text{Suite}, \\
\text{session} : \text{option} \text{SessionInfo}, \\
\text{granted, revoked} : \text{SuiteID} \rightarrow \text{list} \text{Permission} \} 
\]
For every state $\overline{s}'$ and response $r$ computed by $\text{interp}$ there must exist a corresponding abstract state $s'$ refined by $\overline{s}'$ reachable from $s$ by $\xhookrightarrow{}$ with the same response.

\[
\forall (s : \text{State}) \left( \overline{s} : \overline{\text{State}} \right) \left( e : \text{Event} \right) \left( \overline{e} : \overline{\text{Event}} \right) \left( r : \text{Response} \right),
\]
\[
s \sqsubseteq \overline{s} \rightarrow e \sqsubseteq \overline{e} \rightarrow
\]
\[
\text{let } \left( \overline{s}', r \right) := \text{interp} \overline{s} \overline{e} \text{ in } \exists s' : \text{State}, s' \sqsubseteq \overline{s}' \land s \xrightarrow{e/r} s'
\]
For every state $\overline{s}'$ and response $r$ computed by $\text{interp}$ there must exist a corresponding abstract state $s'$ refined by $\overline{s}'$ reachable from $s$ by $\leftrightarrow$ with the same response

$$\forall (s : \text{State}) \left( \overline{s} : \overline{\text{State}} \right) (e : \text{Event}) \left( \overline{e} : \overline{\text{Event}} \right) (r : \text{Response}),$$

$$s \sqsubseteq \overline{s} \rightarrow e \sqsubseteq \overline{e} \rightarrow$$

let $(\overline{s}', r) := \text{interp} \overline{s} \overline{e}$ in $\exists s' : \text{State}, s' \sqsubseteq \overline{s}' \wedge s \xrightarrow{e/r} s'$

Zanella Béguelin, Betarte, Luna  
A Formal Specification of the MIDP 2.0 Security Model
Main Contributions

- The first formalization of the MIDP 2.0 security model
- Formal machine-checked verification of the model
- Investigated some aspects unclear in the informal specification
- A refinement methodology

The complete development in Coq may be obtained from

We have not completed a full refinement
Relax hypothesis assumed about the model
  More than one active suite
  Dynamic security policies in Protection Domains
Consider extensions to the existing model
  Hierarchical permissions
  Multiplicities (Besson et al. – ESORICS’06)
Thank you!

Additional Information


Santiago.Zanella@inria.fr