Checking Secure Interactions of Smart Card Applets

extended version

P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, G. Zanon

CS 711 ● 5 November 2003

René Rydhof Hansen

Overview

- Java Card and Applet Security
- Example: Electronic Purse
- Security Policy and Property
- Modelling and Verifying Applets

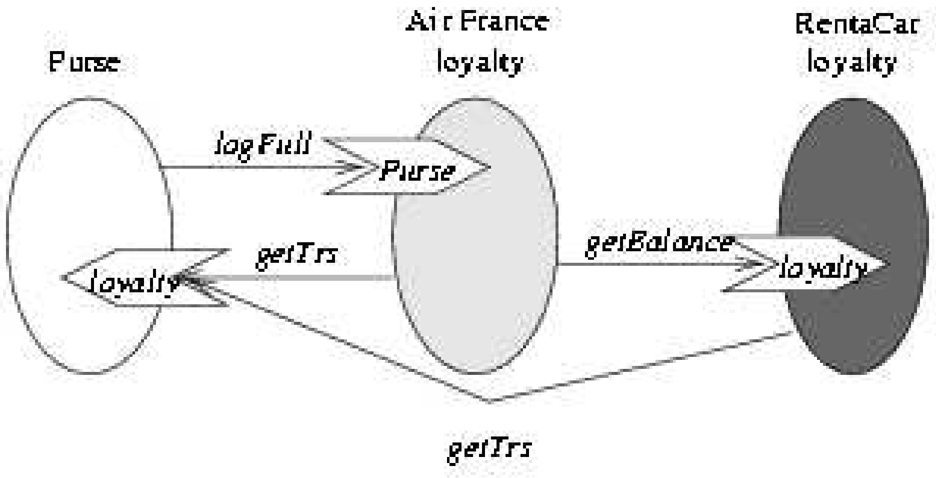
Java Card

- Java Card: variant of Java for smart cards
  - No threads
  - No reflection
  - No Security Manager
  - No long, float, double, character, string, . . .
  - No garbage collection
- Java smart cards
  - On-card Java Card Virtual Machine
  - Multiapplet platform
  - Dynamic download of applets
- Java Card Bytecode

Applet Security

- Security features
    - Type safety
    - Byte-code verification
    - Applet firewall
- Security problems
    - Inter-applet communication
        - Across firewall boundaries
        - Information leaks
    - Dynamic download of applets
    - Timing and power-consumption attacks
    - . . .
- Verify and certify applets

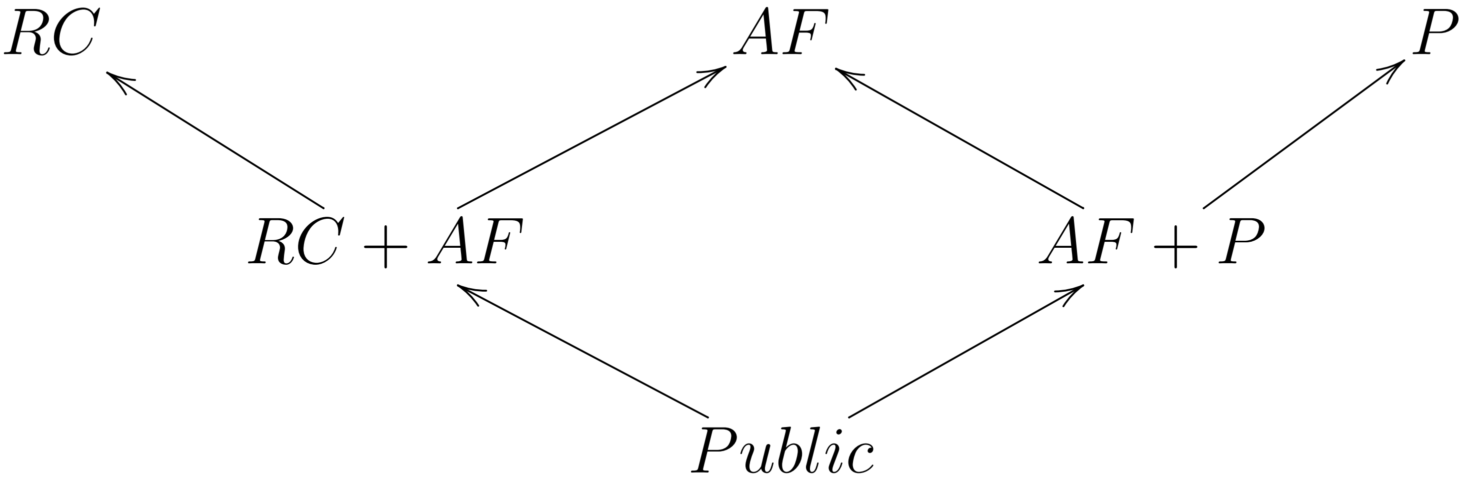
Example: Electronic Purse



- An electronic purse with two *loyalty applets*: AirFrance and RentaCar
- *logFull* invocation results in leak from AirFrance to RentaCar
- Not caught by the applet firewall

Security Policy for Electronic Purse

- Assume lattice of security levels: $(Levels, \preceq)$
- Seperate levels for each applet: P, AF, RC
- Separate levels for sharing data: $AF + P$ and $AF + RC$



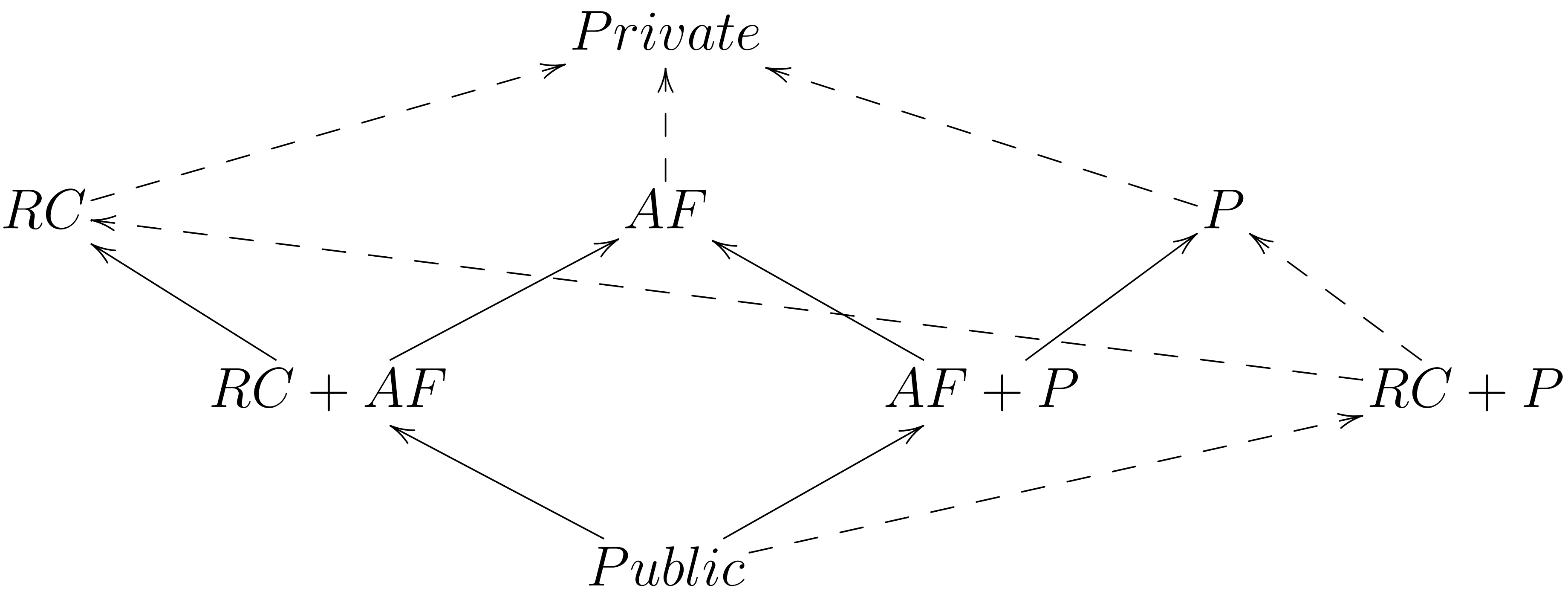
RC AF P

$RC + AF$ $AF + P$

$Public$

Security Policy for Electronic Purse

- Assume lattice of security levels: $(Levels, \preceq)$
- Seperate levels for each applet: P, AF, RC
- Separate levels for sharing data: $AF + P$ and $AF + RC$


$Private$

RC

AF

P

$RC + AF$

$AF + P$

$RC + P$

$Public$

The “semantics” of Programs

Programs are represented as *objects* that *evolve* over time:

$$Ev \subseteq Objects \times Dates \longrightarrow Values$$

where

$$\begin{array}{rcll} Objects & = & Input & \text{not computed \& observable} \\ & \uplus & Output & \text{computed \& observable} \\ & \uplus & Internal & \text{computed \& not observable} \end{array}$$

Security level assigned to input and output objects

$$lvl : Input \uplus Output \longrightarrow Levels$$

Secure Dependency (SecDep)

Output objects should only depend on input objects of a lower level:

$$\forall o_t \in \mathit{Output}.\forall e \in \mathit{Ev}.\forall e' \in \mathit{Ev}. \quad e \sim_{aut(o_t)} e' \Rightarrow e(o_t) = e'(o_t)$$

where

$$aut(o_t) = \left\{ o'_t \in \mathit{Input} \,\middle|\, t' < t, lvl(o'_t) \preceq lvl(o_t) \right\}$$

and

$$e \sim_{aut(o_t)} e' \iff \forall o'_{t'} \in aut(o_t).\, e(o'_{t'}) = e'(o'_{t'})$$

Sufficient Conditions for SecDep

- Problem: SecDep is not well-suited for model-checking with SMV
- Solution: Find checkable sufficient conditions for SecDep
  - Exploit dependencies given by program structure:
    - $dep(i, o_t)$: contains objects at $t - 1$ used by instruction at i to compute calue of o_t (explicit flows)
    - Whenever $o_{t-1} \neq o_t$ then $pc_{t-1} \in dep(i, o_t)$ (implicit flows)
  - Reformulate SecDep in terms of $dep(i, o_t)$

Hypothesis 1 (SecDep Reformulated)

- **Hyp 1:** The value of o_t computed by the program is determined by the values of objects in $dep(e(pc_{t-1}), o_t)$:

$$\forall o_t \in Output. \forall e \in Ev. e' \in Ev. \\ e \sim_{dep(e(pc_{t-1}),o_t)} e' \Rightarrow e(o_t) = e'(o_t)$$

- Need to prove only that:

$$\forall o'_{t'} \in dep(e(pc_{t-1}), o_t) : lvl(o'_{t'}) \preceq lvl(o_t)$$

- But what about internal objects?

Internal Objects

- Problem: Internal objects are not assigned a security level
- Solution: Trace internal objects back to input
  - For input objects: $lvldep(e, o_t) = lvl(o_t)$
  - Otherwise:

$$lvldep(e, o_t) = \bigsqcup \left\{ lvldep(e, o'_{t-1}) \,\middle|\, o'_{t-1} \in dep(e(pc_{t-1}), o_t) \right\}$$

Theorem 1

- **Thm 1:** A program satisfies SecDep if the computed level of an output object is always dominated by its security level:

$$\forall o \in Objects. \forall e \in Ev. lvldep(e, o_t) \preceq lvl(o_t) \;\Rightarrow\; \text{“SecDep”}$$

- Proof by induction on t and using Hypothesis 1.
- Still not quite there yet. . .

Hypothesis 2 (Abstract Interpretation?)

- To avoid state explosion, work on abstract evolutions.
- **Hyp 2:** We suppose that the set of abstract evolution Ev^a is such that the image of Ev under abs is included in Ev^a, where $abs(e)(o_t) = lvldep(e, o_t)$ if $o \neq pc$ and $abs(e)(pc_t) = e(pc_t)$.
- In other words: leave the program counter alone and abstract all other objects to their (computed) security level.

Theorem 2

- **Thm 2:** If $\forall o_t \in Output . \forall e^a \in Ev^a .\ e^a(o_t) \preceq lvl(o_t)$ then the concrete program guarantees SecDep.
- Proof by Theorem 1 and Hypothesis 2.
- Finally: checkable and sufficient condition for SecDep.

Modelling Applets

- Assume: given complete call graph
- Analyse only methods that interact with other applets
  - Example: *logFull*, *askfortransactions*, *update*
- Identify input and output
  - Input: Read attributes and results of external invocations
  - Output: Modified attributes and parameters of external invocations
- Assign security levels to input and output
  - Example: *logFull* is assigned level $AF + P$

Modelling Applets

- Use “assume/guarantee” discipline for local verification of method invocation
  - Assume: return values dominated by security level
  - Guarantee: method parameters dominated by security level
- Allows for modular (re-)verification (call graph?)

Modelling Methods

- Methods are abstracted into parameterised SMV modules:
  - *active*: current method is invoked
  - *context*: context of caller
  - *param*: method parameters
  - *field*: attributes' security levels
  - *method*: security levels of invoked (external) methods
- Main module
  - Instantiate other modules,
  - Assign security levels
  - Simulate call graph

Modelling the update Method

```
module update(active, context, param, field, method){
  L: levels;
  pc: -1..9;
  lpc: boolean;
  mem: array 0..1 of boolean;
  stck: array 0..1 of boolean;
  sP: -1..1;
  ByteCode : {invoke_108, load_0, return, nop, store_1, dup,
              load_1, getfield_220,op, putfield_220};

  init(pc):= 0; init(sP):= 1; init(mem[0]):= param[0];
  for(i=0; i< 2; i=i+1) {init(stck[i]) := L.public; }
  init(lpc) := context;
```

Modelling the `update` Method

```
if (active) {
  (next(pc), ByteCode) :=
   switch(pc) {
    -1: (-1, nop);
     0: (pc+1, load_0 );
     1: (pc+1, invoke_108 );
     2: (pc+1, store_1 );
     3: (pc+1, load_0 );
     4: (pc+1, dup );
     5: (pc+1, getfield_220 );
     6: (pc+1, load_1 );
     7: (pc+1, op );
     8: (pc+1, putfield_220 );
     9: (-1, return);
   };}
 else {next(pc) := pc; next(ByteCode) := nop;}
```

Modelling the `update` Method

```
switch(ByteCode) {
  nop :;
  load_0 : {next(stck[sP]) := mem[0];next(sP):=sP-1;}
  load_1 : {next(stck[sP]) := mem[1];next(sP):=sP-1;}
  store_1 : {next(mem[1]):=(stck[sP+1]|lpc) ;next(sP):=sP+1;}
  dup : {next(stck[sP]):= stck[sP+1]; next(sP):=sP-1;}
  op : {next(stck[sP+2]):=(stck[sP+1]|stck[sP+2]);
        next(sP):=sP+1};
  invoke_108 : {next(stck[sP]):=method[0];next(sP):= sP+1;}
  getfield_220 : {next(stck[sP+1]):=field[0];}
  putfield_220 : {next(sP):=sP+2;}
```

Verifying Properties for `update`

- Formulate properties as Linear Temporal Logic formulae
  - Check interaction with *getbalance*:

```
Smethod_108 :
  assert G (m_update.ByteCode=invoke_108 ->
   ((m_update.stck[sP+1]|m_update.lpc) -> L.AF & L.RC));
```

$$\mathtt{m\_update.stck}[\mathtt{sP}+1] \sqcup \mathtt{m\_update.lpc} \sqsubseteq AF + RC$$

  - Check use of attribute *extendedbalance*:

```
Sfield_220 :
  assert G (m_aft.ByteCode=putfield_220 ->
             ((m_aft.stck[sP+1]|m_aft.lpc) -> L.AF));
```

$$\mathtt{m\_aft.stck}[\mathtt{sP}+1] \sqcup \mathtt{m\_aft.lpc} \sqsubseteq AF$$

Verification Results

- The information leak is found and a counterexample is produced
- To check the full purse example: 20 analyses, 100 methods, and 60 properties
- No more than 3 minutes/property

Questions

- Relevant mechanism for purse example?
- Relevant security property? How do you know?
- Model validation? How?
- Reasonable hypotheses?
- Scope of conditionals?
- Which methods to analyse?
- Formal enough? Level of assurance?
- Precision? Label creep?
- What properties to be checked?

Quote of the Day

We also based our approach on model-checking tools because they tend to be more generic and expressive than type-checking algorithms. This allowed us to obtain results faster because we did not have to implement a particular type-checking algorithm. This shold also enable us to perform experiments with other security policies and properties.