Tool-Assisted Specification and Verification of the JavaCard Platform

Gilles Barthe
INRIA Sophia-Antipolis, France

Joint work with: P. Courtieu, G. Dufay, M. Huisman, L. Jakubiec, B. Serpette, S. Melo de Sousa, S. Stratulat

New generation smartcards

- Flexibility
  - High-level language for developing applets
  - Multi-application and post-issuance
- New Security Threats
  - Confidentiality
  - Integrity
  - Availability

Formal verification for smartcards

- Motivations
  - Complex software with high demands on securit[y]
  - Common Criteria require formal methods at leve[l] EAL5-EAL7

Formal verification for smartcards

- Motivations
  - Complex software with high demands on security
  - Common Criteria require formal methods at levels EAL5-EAL7
- Focus
  - Platform vs. program verification
  - Bytecode vs. source level

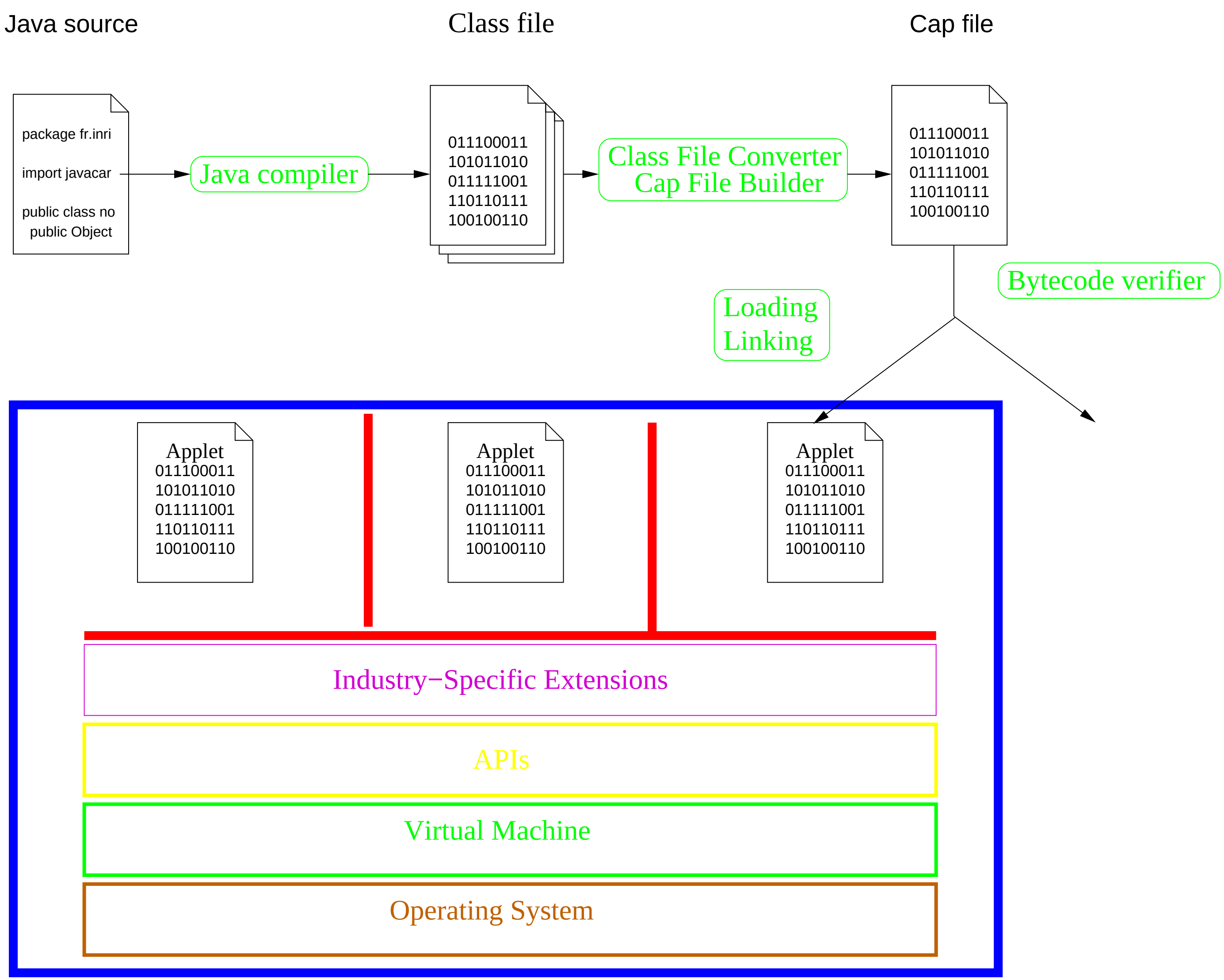
Overview

- JavaCard
- CertiCartes: verification of the JavaCard platform
- Jakarta: tool support for specification and verificatio of virtual machines

JavaCard

- A superset of a subset of Java:
  - A subset: no large datatypes, security manager, dynamic class loading, (garbage collection). . .
  - A superset: firewall, entry points, shareable interfaces, transactions, etc.
- JavaCard programs use the JavaCard APIs

The JavaCard Platform



CertiCartes

Formal specification/verification of:

- JCVMs: small-step semantics

```
exec: state -> returned_state
```

  - written in Coq but use a neutral style
  - executable with the JCVM Tools
- BCV: executable in Caml
- part of the JCRE

Program model

```
Record jcprogram : Set := { interfaces : (list Interfac
                             classes    : (list Class);
                             methods    : (list Method)
Record Method : Set := {
 is_static      : bool;
 signature      : ((list type)*type);
 local          : nat;                    (* Number of local
 handler_list   : (list handler_type); (* Exception hand
 bytecode       : (list Instruction);  (* instructions to
 method_id      : method_idx;             (* Index of the me
  owner          : class_idx              (* Index of the ov
```

Memory model

- Stack as a list of frames

```
Record frame : Set := {
   locvars          : (list valu);        (* Local Vari
   opstack          : (list valu);        (* Operand st
   p_count          : bytecode_idx;       (* Program co
   method_loc       : method_idx;         (* Location o
   context_ref      : package;            (* Context In
```

- State

```
Definition state := static_heap*heap*stack.
```

Instruction

```
Definition NEW := [idx:cap_class_idx][state:jcvm_state]
Cases (stack_f state) of
(cons h lf) =>
  (* Extract the owner class from thew cap_file *)
  Cases (Nth_elt (classes cap) idx) of
  (* then a new instance is created and pushed into the heap *)
  (Some cl) => let new_obj = ... in
      (Normal
         (Build_jcvm_state
            (sheap_f state)
              (app (heap_f state) new_obj)
 (* the reference of the created object is pushed  into the opstack *)
 (cons
   (update_opstack (cons (vRef (vRef_instance idx (S (length (heap_f state)
           lf))) |
    None => (AbortCode class_membership_error state)
  end |
_ => (AbortCode state_error state)
```

Virtual Machines Specification

- Defensive JCVM is closest to specification:
  - It manipulates typed values
  - Types are checked at run-time

Virtual Machines Specification

- Defensive JCVM is closest to specification:
  - It manipulates typed values
  - Types are checked at run-time
- Offensive JCVM is closest to implementation:
  - It manipulates untyped values
  - Type correctness enforced by BCV

Virtual Machines Specification

- Defensive JCVM is closest to specification:
  - It manipulates typed values
  - Types are checked at run-time
- Offensive JCVM is closest to implementation:
  - It manipulates untyped values
  - Type correctness enforced by BCV
- Abstract JCVM used in bytecode verification:
  - Manipulates types as values
  - Operates on a method-per-method basis

Cross-Validation

Defensive and offensive VMs coincide on programs that are well-typed for the abstract VM

Cross-Validation

Defensive and offensive VMs coincide on programs that are well-typed for the abstract VM

- offensive and defensive VMs coincide on programs well-typed for the defensive VM

Cross-Validation

Defensive and offensive VMs coincide on programs that are well-typed for the abstract VM

- offensive and defensive VMs coincide on programs well-typed for the defensive VM
- programs that are well-typed for the abstract VM are well-typed with the defensive VM

Cross-Validation

Defensive and offensive VMs coincide on programs that are well-typed for the abstract VM

- offensive and defensive VMs coincide on programs well-typed for the defensive VM
- programs that are well-typed for the abstract VM are well-typed with the defensive VM

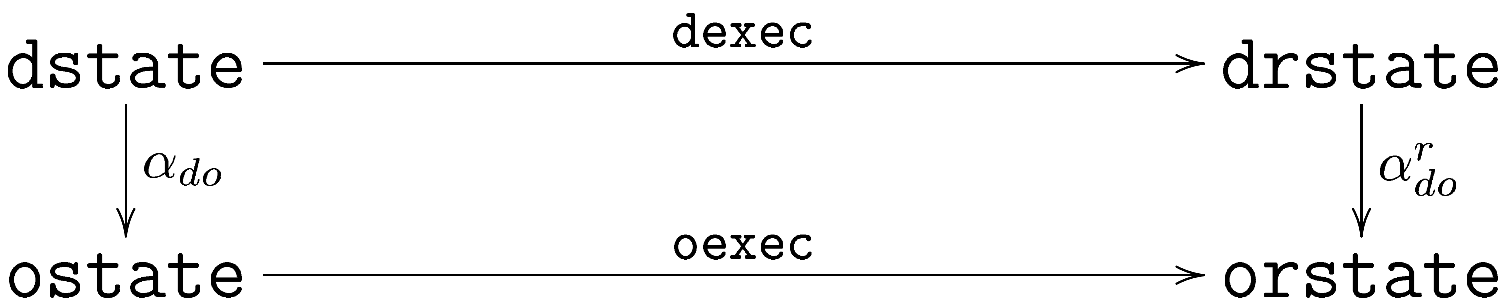
Best viewed as some form of correctness of abstract interpretations

Offensive vs. Defensive

- Abstraction function: $\alpha_{do} : (t, z) \mapsto z$

Offensive vs. Defensive

- Abstraction function: $\alpha_{do} : (t, z) \mapsto z$
- Diagram commutes



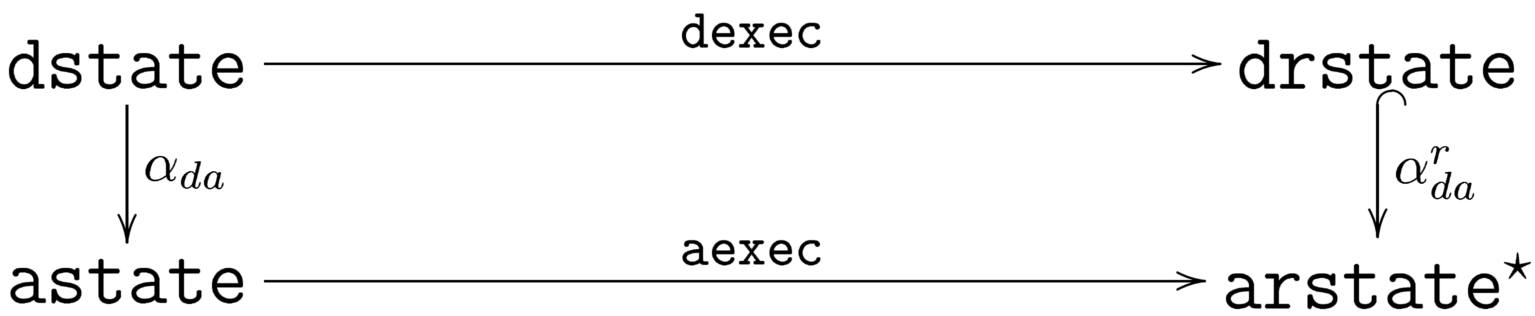
if defensive VM does not raise typing errors

Abstract vs. Defensive

- Abstraction function: $\alpha_{da} : (t, z) \mapsto t$

Abstract vs. Defensive

- Abstraction function: $\alpha_{da} : (t, z) \mapsto t$
- Diagram commutes



if “execution keeps in the same frame”

Bytecode verifier

- Reject programs which go wrong (on the abstract VM) using dataflow analysis (Kildall's algorithm)

Bytecode verifier

- Reject programs which go wrong (on the abstract VM) using dataflow analysis (Kildall's algorithm)
- Defensive and offensive machines coincide on programs that pass bytecode verification

Bytecode verifier

- Reject programs which go wrong (on the abstract VM) using dataflow analysis (Kildall's algorithm)
- Defensive and offensive machines coincide on programs that pass bytecode verification
- Proof builds upon commuting diagrams, correctness of DFA, methodwise verification, and monotonicity of abstract VM

Assessment

- Positive evaluation from Gemplus but CertiCartes is an in-depth feasibility study

Assessment

- Positive evaluation from Gemplus but CertiCartes is an in-depth feasibility study
- A complete formalization of the JavaCard platform is labour intensive (E. Giménez)

Assessment

- Positive evaluation from Gemplus but CertiCartes is an in-depth feasibility study
- A complete formalization of the JavaCard platform is labour intensive (E. Giménez)
- The methodology works well and could be used for other analyses

Assessment

- Positive evaluation from Gemplus but CertiCartes is an in-depth feasibility study
- A complete formalization of the JavaCard platform is labour intensive (E. Giménez)
- The methodology works well and could be used for other analyses
- High-level of automation is possible

Assessment

- Positive evaluation from Gemplus but CertiCartes is an in-depth feasibility study
- A complete formalization of the JavaCard platform is labour intensive (E. Giménez)
- The methodology works well and could be used for other analyses
- High-level of automation is possible
- Specifications use a restricted language and proofs use well-understood techniques

Jakarta

- A dedicated environment for formal specification an
verification of typed low-level languages

Jakarta

- A dedicated environment for formal specification an verification of typed low-level languages
- Designed to support:
  - executable specifications
  - abstractions (and refinement) of specifications
  - automation of correctness proofs

Current focus

- Input: defensive virtual machine

Current focus

- Input: defensive virtual machine
- Output:

Current focus

- Input: defensive virtual machine
- Output:
  - offensive and abstract virtual machines

Current focus

- Input: defensive virtual machine
- Output:
  - offensive and abstract virtual machines
  - offensive and defensive machines coincide on well-typed programs

Current focus

- Input: defensive virtual machine
- Output:
  - offensive and abstract virtual machines
  - offensive and defensive machines coincide on well-typed programs
  - programs that are ill-typed for the defensive VM are ill-typed with the abstract VM

Current focus

- Input: defensive virtual machine
- Output:
  - offensive and abstract virtual machines
  - offensive and defensive machines coincide on well-typed programs
  - programs that are ill-typed for the defensive VM are ill-typed with the abstract VM
  - the abstract virtual machine is monotone

Jakarta Specification Language

- JSL types are first-order polymorphic types
- JSL expressions are first-order algebraic terms

$$\mathcal{E} := \mathcal{V} \mid \mathcal{E} == \mathcal{E} \mid c\,\vec{\mathcal{E}} \mid f\,\vec{\mathcal{E}}$$

- Functions defined by conditional rewrite rules

$$l_1 \twoheadrightarrow r_1, \ \ldots\ , l_n \twoheadrightarrow r_n \Rightarrow g \rightarrow d$$

where r_i are patterns with fresh variables

Compiling JSL Specifications

- Specifications are executed by rewriting engines
- Deterministic specifications are compiled into case-expressions then CAML, Coq, Isabelle, PVS
- Non-deterministic specifications $f : \sigma \to \tau$ are translated into $f^\star : \sigma \to \tau^\star$
- Partial specifications $f : \sigma \to \tau$ are translated into $f_\perp : \sigma \to \tau_\perp$

Abstractions

- For each datatype σ define $\hat{\sigma}$ and $\lceil . \rceil_{\sigma} : \sigma \to \hat{\sigma}$

Abstractions

- For each datatype σ define $\hat{\sigma}$ and $\lceil . \rceil_\sigma : \sigma \to \hat{\sigma}$
- For each defined function $f : \sigma \to \tau$, define $\hat{f} : \hat{\sigma} \to$ by transforming $l_1 \twoheadrightarrow r_1, \ \ldots \ , l_n \twoheadrightarrow r_n \Rightarrow g \to d$ into $\lceil l_1 \rceil \twoheadrightarrow \lceil r_1 \rceil, \ \ldots \ , \lceil l_n \rceil \twoheadrightarrow \lceil r_n \rceil \Rightarrow \lceil g \rceil \to \lceil d \rceil$

Abstractions

- For each datatype σ define $\hat{\sigma}$ and $\lceil . \rceil_\sigma : \sigma \to \hat{\sigma}$
- For each defined function $f : \sigma \to \tau$, define $\hat{f} : \hat{\sigma} \to$ by transforming $l_1 \twoheadrightarrow r_1, \ \ldots \ , l_n \twoheadrightarrow r_n \Rightarrow g \to d$ into $\lceil l_1 \rceil \twoheadrightarrow \lceil r_1 \rceil, \ \ldots \ , \lceil l_n \rceil \twoheadrightarrow \lceil r_n \rceil \Rightarrow \lceil g \rceil \to \lceil d \rceil$
- Not a legal rule: substitution and cleaning steps declared in abstraction scripts

Abstractions

- For each datatype σ define $\hat{\sigma}$ and $\lceil . \rceil_\sigma : \sigma \to \hat{\sigma}$
- For each defined function $f : \sigma \to \tau$, define $\hat{f} : \hat{\sigma} \to$ by transforming $l_1 \twoheadrightarrow r_1, \ \ldots \ , l_n \twoheadrightarrow r_n \Rightarrow g \to d$ into $\lceil l_1 \rceil \twoheadrightarrow \lceil r_1 \rceil, \ \ldots \ , \lceil l_n \rceil \twoheadrightarrow \lceil r_n \rceil \Rightarrow \lceil g \rceil \to \lceil d \rceil$
- Not a legal rule: substitution and cleaning steps declared in abstraction scripts
- Generated offensive and abstract JCVMs

Proof automation using Coq

- Mostly case analysis + equational reasoning

Proof automation using Coq

- Mostly case analysis + equational reasoning
- Built tactics that reduce $\forall \vec{x}.\ \phi(\vec{x}, f\ \vec{x})$ to

$$\forall \vec{x} : \vec{\sigma}.\forall \vec{y} : \vec{\sigma'}.l_1 = r_1 \ \wedge \ \ldots \ \wedge \ l_n = r_n \Rightarrow \phi\ (\vec{x},\ f\ \vec{x})$$

and perform some equational reasoning

Proof automation using Coq

- Mostly case analysis + equational reasoning
- Built tactics that reduce $\forall \vec{x}.\ \phi(\vec{x}, f\ \vec{x})$ to
  $\forall \vec{x} : \vec{\sigma}.\forall \vec{y} : \vec{\sigma'}.l_1 = r_1\ \wedge\ \ldots\ \wedge\ l_n = r_n \Rightarrow \phi\,(\vec{x},\ f\ \vec{x})$
  and perform some equational reasoning
- Further automation of equational reasoning is highly desirable

Proof automation using Coq

- Mostly case analysis + equational reasoning
- Built tactics that reduce $\forall \vec{x}.\ \phi(\vec{x}, f\ \vec{x})$ to
  $\forall \vec{x} : \vec{\sigma}.\forall \vec{y} : \vec{\sigma}'.l_1 = r_1\ \wedge\ \ldots\ \wedge\ l_n = r_n \Rightarrow \phi\,(\vec{x},\ f\ \vec{x})$
  and perform some equational reasoning
- Further automation of equational reasoning is highly desirable
- Exploiting abstraction scripts seems promising

Proof automation using Spike

- Spike is a first-order prover for “inductive theorems”

Proof automation using Spike

- Spike is a first-order prover for “inductive theorems”
- Cross-validation of the VMs for 2/3 of bytecodes

Proof automation using Spike

- Spike is a first-order prover for "inductive theorems"
- Cross-validation of the VMs for 2/3 of bytecodes
- Now applying Spike to prove the monotonicity of abstract VM

Conclusions

- Formal specification and verification of the JavaCard platform is feasible but labor-intensive

Conclusions

- Formal specification and verification of the JavaCard platform is feasible but labor-intensive
- Tool support for formal specification and verification of (type safety for) low-level typed languages

Conclusions

- Formal specification and verification of the JavaCard platform is feasible but labor-intensive
- Tool support for formal specification and verification of (type safety for) low-level typed languages
- Some interesting topics:
  - extracting code or tests from specifications
  - tools for certifying certifying compilers

Conclusions

- Formal specification and verification of the JavaCar[d] platform is feasible but labor-intensive
- Tool support for formal specification and verification of (type safety for) low-level typed languages
- Some interesting topics:
  - extracting code or tests from specifications
  - tools for certifying certifying compilers
- For further information www.inria.fr/lemme/verificar[d]