

# Reasoning about JavaCard

Gilles Barthe

`Gilles.Barthe@inria.fr`

`http://www-sop.inria.fr/lemme/personnel/Gilles.Barthe`

INRIA Sophia-Antipolis, France

Joint work with:

Pierre Courtieu, Guillaume Dufay, Marieke Huisman,  
Line Jakubiec, Daniel Perovich, Bernard Serpette,  
Simão Sousa, Sorin Stratulat, Kwong-Cheong Wong

# Overview

- JavaCard
- Formal methods for smartcards
- CertiCartes
- Jakarta
- Conclusions

## New generation smartcards

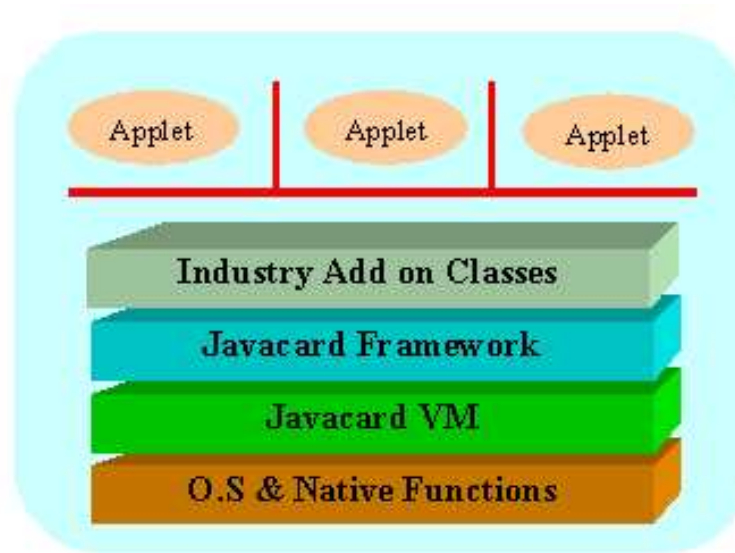
- Flexibility
  - High-level language for developing applets
  - Multi-application
  - Post-issuance
- Security
  - Applets must be communicating
  - Some of their information must remain private
  - Resource control

# JavaCard

- A superset of a subset of Java:
  - A **subset**: No multi-threading, double, security manager, dynamic class loading, garbage collection, etc.
  - A **superset**: Firewall, entry points, shareable interfaces, transactions, etc.
- JavaCard developers benefit from Java technology: JavaCard programs are built using the **JavaCard APIs** and compiled with any Java compiler

# The JavaCard Platform

- Compiled .class files are **converted** into .cap files
- Converted .cap files are **bytecode verified**, loaded on-card and executed with the **JavaCard Runtime Environment**



# Formal verification for JavaCard

- Why
  - High demands on security
  - Common Criteria require formal methods
- What and how
  - Platform verification (this talk)
  - Program verification
  - Source vs bytecode level?

# CertiCartes: specification

Formal **executable** specifications in Coq of

- the **Defensive** JavaCard Virtual Machine
- the JavaCard **ByteCode Verifier**
  - Abstract JavaCard Virtual Machine
  - Data Flow Analysis
- the **Offensive** JavaCard Virtual Machine
- some **APIs**

# CertiCartes: verification

Formal **correctness** proof in Coq

- data flow analyzer **terminates**
- abstraction **commutes** with execution
- offensive JCVM and defensive JCVM **coincide** on programs that pass bytecode verification

## Assessment

CertiCartes is an **in-depth feasibility study**. Accurate and complete formalization of the JavaCard platform is possible, but its cost is prohibitive.

Some problems:

- **Low level** of automation makes proofs tedious
- **Difficult to make variations** (or refinements) on the specification
- Definitions are **(a bit) cluttered**

This is bad because we want to use the methodology for other analyses.

# Jakarta

A dedicated specification language can have a positive impact on formal specification and formal verification.

Jakarta is designed to support:

- **concise** and **executable** specifications
- **refinement** and **abstractions** of specifications
- **automation** of correctness proofs

## Current focus

- Input: **Defensive** Virtual Machine
- Output:
  - **Offensive** and **Abstract** Virtual Machines
  - Diagrams **commute**
  - Offensive and Defensive machines **coincide** on well-typed programs

Automating the correctness proof of the BCV is yet out of reach

# The Jakarta Specification Language

- JSL types: first-order polymorphic types
- JSL expressions: **first-order algebraic terms** built from variables, constructor symbols (data type declarations), defined symbols (function definitions):

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

- Functions defined by **conditional rewrite rules**:

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

where  $r_i$  are **patterns with fresh variables**

## Execution model and compilation

- Execution model provided by **term-rewriting**
- Compilation into intermediate **language with case-expressions**  
(decompilation is also supported)
- Allows a limited form of **non-determinism**

## Abstractions with the Jakarta Transformation Kit

- For each datatype  $\sigma$  define  $\hat{\sigma}$  and  $[\cdot]_{\sigma} : \sigma \rightarrow \hat{\sigma}$
- For each defined function  $f : \sigma \rightarrow \tau$ , define  $\hat{f} : \hat{\sigma} \rightarrow \hat{\tau}$  by transforming

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

into

$$[l_1] \twoheadrightarrow [r_1], \dots, [l_n] \twoheadrightarrow [r_n] \Rightarrow [g] \rightarrow [d]$$

- Not a legal rule: **substitution** and **cleaning** required

# Verification with the Jakarta Automation Kit

- Tailored towards [Coq](#)
- Automatic generation of [principles](#) that reduce goals such as  $\forall \vec{x}. \phi(\vec{x}, f \vec{x})$  to subgoals of the form

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

- Reduce correctness proofs to [equational reasoning](#)

## Jakarta status

- **Translations** between Coq and the Jakarta Specification language (specification of the JCVM in JSL)
- Generated **offensive** virtual machine, **abstract** machine underway
- Used **JAK tactics** to good effect
- **Equational reasoning** needs to be further automated (there are good systems around)

## Conclusions

- Formal specification and verification of the JavaCard platform is **feasible but labor-intensive**
- Our line of research:
  - **develop tools** to ease formal verification
  - Apply these tools to develop **enhanced Byte Code Verifiers**
- For our work on program verification please visit  
<http://www-sop.inria.fr/lemme/verificard>