VERIFICARD

Formal modeling and verification of the Java Card security architecture

Eduardo Giménez
Trusted Logic

Olivier Ly
SchlumbergerSema

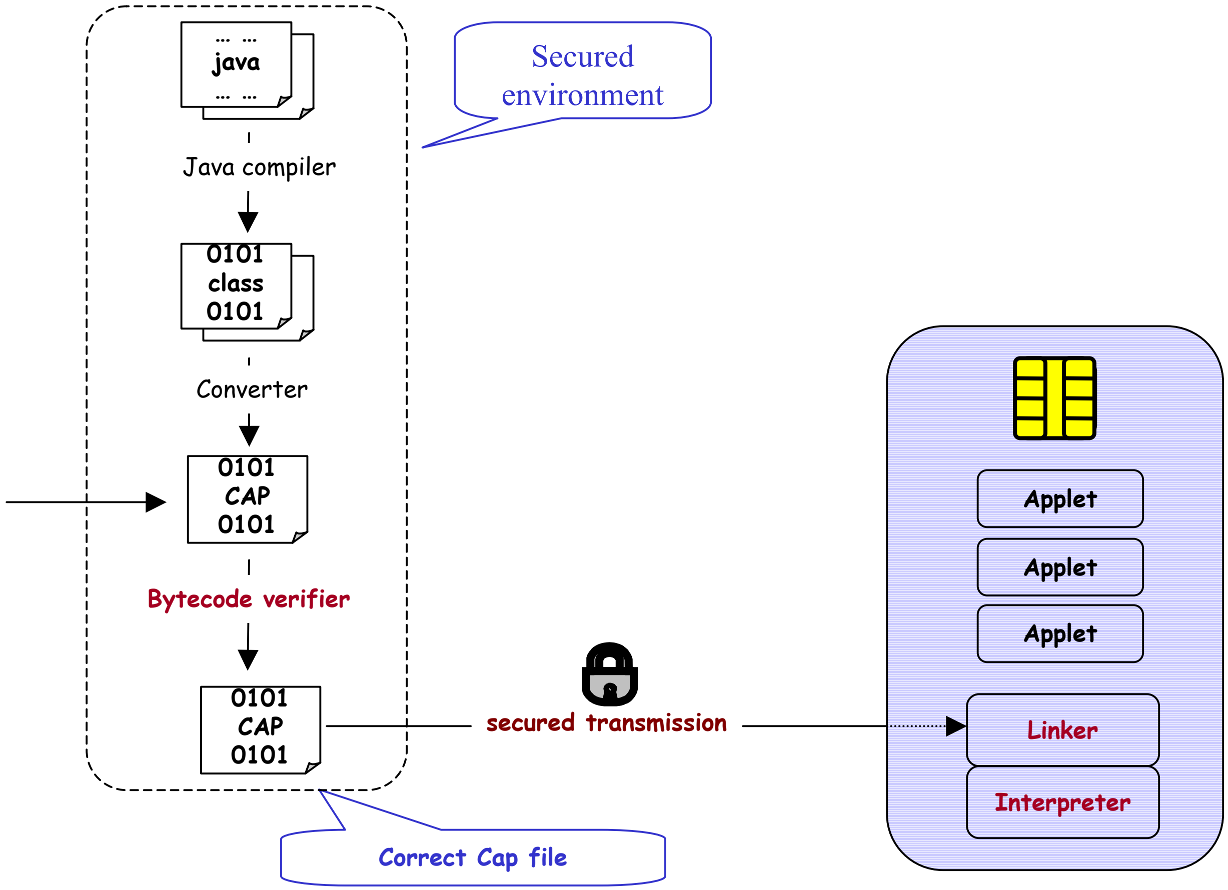
The *Formavie* project

- Partners: CP8, INRIA, Schlumberger.
- Goal: formal modeling and verification of the Java Card security architecture.
- Means: specify and prove in Coq the correctness of the critical components of a Java Card platform.
- Models developed by Trusted Logic for CP8 and Schlumberger.

Summary

1. The Java Card security chain.
2. General pattern of model.
3. The case of a Java Card platform:
   1. Formal security model.
   2. Internal consistency of the security model.
   3. Component specification.
   4. A general proof architecture for security properties.
4. Achievements and conclusions.

Java Card applet development chain



Security properties

- Main goal is applet isolation.
  - No applet can unauthorizedly modify other applets data
  - No applet can unauthorizedly disclosure other applets data.
- Correct development and software attack prevention.
- Functional properties of the applets are not the first security concern.

Security Chain

- Critical part of the development chain :

cap → Verifier — signed cap → Protocol — cap → Linker — linked cap → Interpreter

- CAP format is taken as the reference format of the model:
  - The applet developer and the applet issuer may be different.
  - Is the format used by SUN to specify the JCVM behavior.
  - Independent from any particular vendor's implementation.
  - Compiler and converter are critical mainly for functional properties.

FORMAVIE challenges

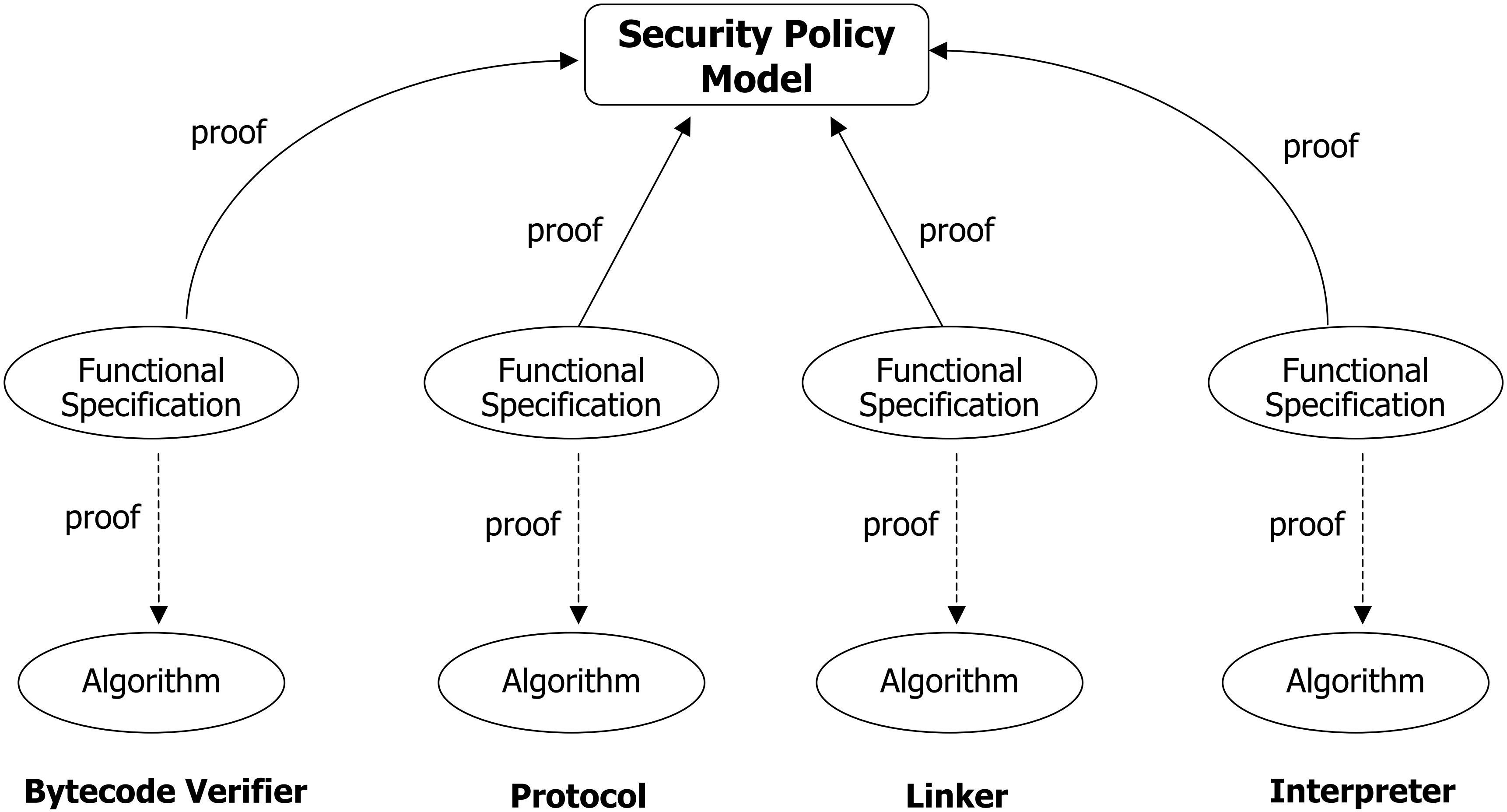
1. To understand how the different properties enhanced by each component contribute to ensure applet isolation on the card.

2. To model the behavior of each component and to prove the correctness of the Java Card security architecture.

3. To build a logical framework both realistic and applicable to different vendor implementations (formal model reuse).

General architecture of the model

A modular architecture



Model architecture (1/2)

- Security policy model:
  - A collection of state machines.
    - A distinguished machine describing the computational semantics of Java Card.
    - Several « abstract » machines describing the security policies.
  - Security properties are state machine invariants.
- Functional specification
  - An abstract description of the component.
  - Specified in terms of pre- and post-conditions.
- Component contribution to the security policy model
  - The post-conditions entail some property on the execution of one of the state machines of the Security Policy model.

Model architecture (2/2)

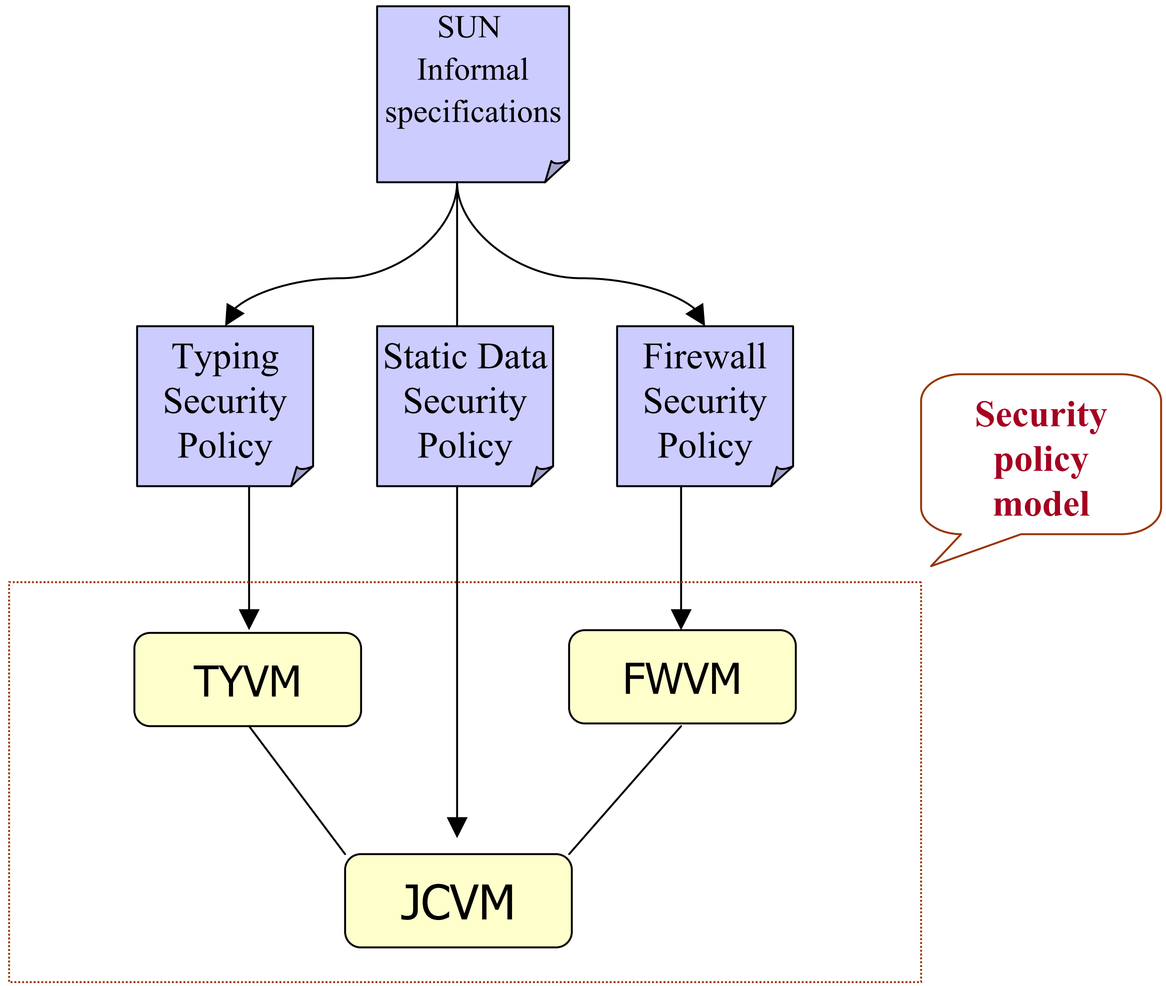
- Algorithm description
  - Should capture the complexity of the implemented solution.
  - A deterministic, potentially executable program in Coq.
  - Described as a function $f : Input \rightarrow Output + Error$.

- Implementation soundness
  - If the input satisfies the pre-conditions, then the output produced by the function satisfies the post-conditions.

$$\forall\, x \in Input\ .\ \forall\, y \in Output\ .\ f(x)=y \rightarrow Pre(x) \rightarrow Post(x,y)$$

Java Card Security Policy Model

Java Card Security Policy Model



State Machines

$$p \in P \;; \qquad \mathrm{M}_p \equiv \langle S, \rightarrow_p, S_I, S_F \rangle$$

- P = data structures containing the program
- S = data structures describing the state of the machine
- $\rightarrow$ = transition relation (inductive predicate depending on p)
- S_I = set of possible initial states
- S_F = set of valid final states (more than having no successor)

The JCVM machine

- Formalization of Java Card execution model (JCVMS+JCRES)
- All Java Card features considered:
  - All bytecodes
  - All kind of identifiers (tokens,offsets,references,AIDs,etc)
  - All possible integer representations (big-endian,little-endian)
  - Correct access to the beginning of data (bytecode, method info,etc)
  - Native method invocation
  - Transactions and transient objects
  - Critical components of the API (input/output,Applets,PINs,etc)
- A semi-defensive and « ideal » machine.
  - All controls are performed dynamically.
  - References are separated from arithmetical values.

The JCVM as a state machine

$$cap \in P \;; \qquad \text{JCVM}_{cap} \equiv \langle\, S, \rightarrow_{cap}, S_I, S_F \,\rangle$$

- P = CAP format
- S is formed by :
  - heap
  - static field images,
  - frame stack,
  - JCRE structures (transaction log, input, output, etc)
- $\rightarrow_{cap}$ = semantics of each bytecode, as (partially) specified by SUN
- s_I = frame stack only contains the frame of the invoked method
- S_F = empty frame stack

The TYVM machine

- A formalization of « must » clauses in SUN’s specification.
- Both an abstraction and a refinement of the JCVM.
- All values of the same type are collapsed into a single point.
- Control flow is local to the current method (modular type-checking).

The TYVM as a state machine

$$p \in P \;\; ; \;\; \text{TYVM}_p \equiv \langle\, S, \rightarrow_p, s_I, S_F \,\rangle$$

- P = CAP format + well-formedness constraints.
- S is formed by :
  - The type abstraction of the operand stack of the method
  - The type abstraction of the local variables of the method
  - The current pc
- $\rightarrow$ = typing constraints associated to each bytecode
- S_I = empty stack, local variables with method type, method initial pc.
- S_F = control flows out of the method (return or uncaught exception).

CAP format constraints (examples)

- Language constraints: If a method overrides another method, then both have the same number of arguments.
- Redundant structures: searching the type of a method invocation either directly from its constant pool index or by traversing the class structure of the descriptor component leads to the same type.
- No hanged pointers: each exception handler points to the beginning of some bytecode.
- Consistent pointers: each argument of a static method invocation has an entry in the constant pool, and the entry describes a static method (and not, say, a field).
- Correspondences between components: each class in the class component has an entry in the descriptor component.

The FWVM machine

- A formalization of Java Card firewall rules.
- Obtained forgetting those conditions of the JCVM rules which do not concern firewall verifications.
- All arithmetic values collapsed into a single point.
- Structure of the operand stack and local variables forgotten.
- Control flow similar to the TYVM, but method invocations are followed (not intended for static verification).
- Intended to prove properties entailed by firewall rules (applet isolation).

The FWVM as a state machine

$$cap \in P \ ; \qquad \text{FWVM}_{\text{cap}} \equiv \langle S, \rightarrow_{\text{cap}}, s_I, S_F \rangle$$

- P = CAP format
- S is formed by :
  - Frame stack (active context, pc, known references)
  - Static field images abstraction (field values collapsed)
  - Heap abstraction (field values collapsed)
- $\rightarrow$ = firewall verifications associated to each bytecode
- S_I = single frame with initial pc, context and known references.
- S_F = empty frame stack

Example: arraylength bytecode

$$\frac{rf \neq \text{null} \qquad \text{hp}(rf) = \langle \text{o}, [a1,\ldots,an] \rangle \qquad \text{FirewallConditions}(\text{c}, o)}{\langle \text{sfi};\ \text{hp};\ \langle \text{c};\text{pc};\text{lv};[rf,\ldots]\rangle :: \ldots \rangle \xrightarrow{\text{arraylength}} \langle \text{sfi};\ \text{hp};\ \langle \text{c};\text{pc+1};\text{lv};[\text{n},\ldots]\rangle :: \ldots \rangle}$$

JCVM

$$\frac{}{\langle \text{pc};\ \text{lv};\ [\text{Array}(T),\ldots] \rangle \xrightarrow{\text{arraylength}} \langle \text{pc+1};\ \text{lv};\ [\texttt{short},\ldots] \rangle}$$

TYVM

$$\frac{rf \in \text{refs} \qquad rf \neq \text{null} \qquad \text{hp}(rf) = \langle \text{o}, \_ \rangle \qquad \text{FirewallConditions}(\text{c}, o)}{\langle \text{sfi};\ \text{hp};\ \langle \text{c};\text{pc};\text{refs}\rangle :: \ldots \rangle \xrightarrow{\text{arraylength}} \langle \text{sfi};\ \text{hp};\ \langle \text{c};\text{pc+1};\text{refs}\rangle :: \ldots \rangle}$$

FWVM

Internal consistency of the security model

Type abstraction soundness

The typing rules express sufficient conditions for the program code to completely determine the execution in the computational model (JCVM).

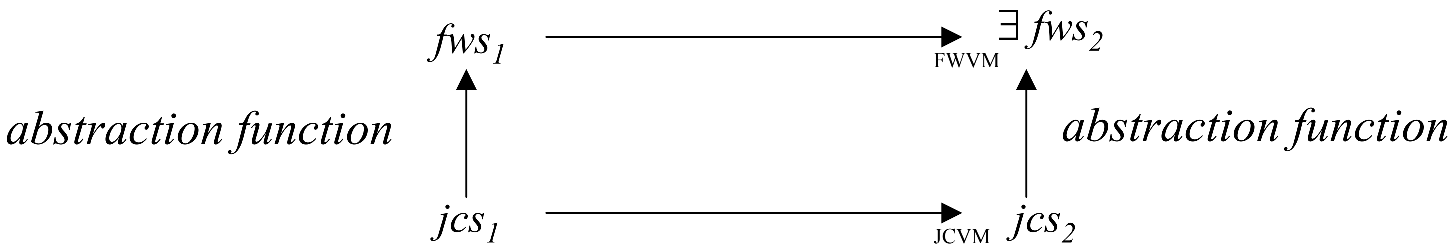
Safe(M_p) = any trace of M generated by the program p leads to a valid final state of M.

$$Safe\ (tyvm_p) \Rightarrow Safe\ (jcvm_p)$$

Example: arraylength $\langle$sfi; hp; $\langle$c;pc;lv, [ ]$\rangle$::…$\rangle$

Firewall abstraction soundness

Any trace of the computation model corresponds to some trace of the machine stating the firewall rules through an abstraction function.



Component modeling

Embedded interpreter: functional specification

- A new state machine is introduced (EMVM).
- It focuses on the modifications of the card memory.
- Differences with respect to the JCVM:
  - Works on linked CAP format.
  - Less defensive than the JCVM.
  - Less typed model (every piece of data is a block of bytes).
  - Considers potentially side effects resulting from:
    - Overflow of data structures (operand stack, objects, etc)
    - Access to non-initialized memory blocks;
    - Bounded resources
- Observational point of view (abstract state, memory services)

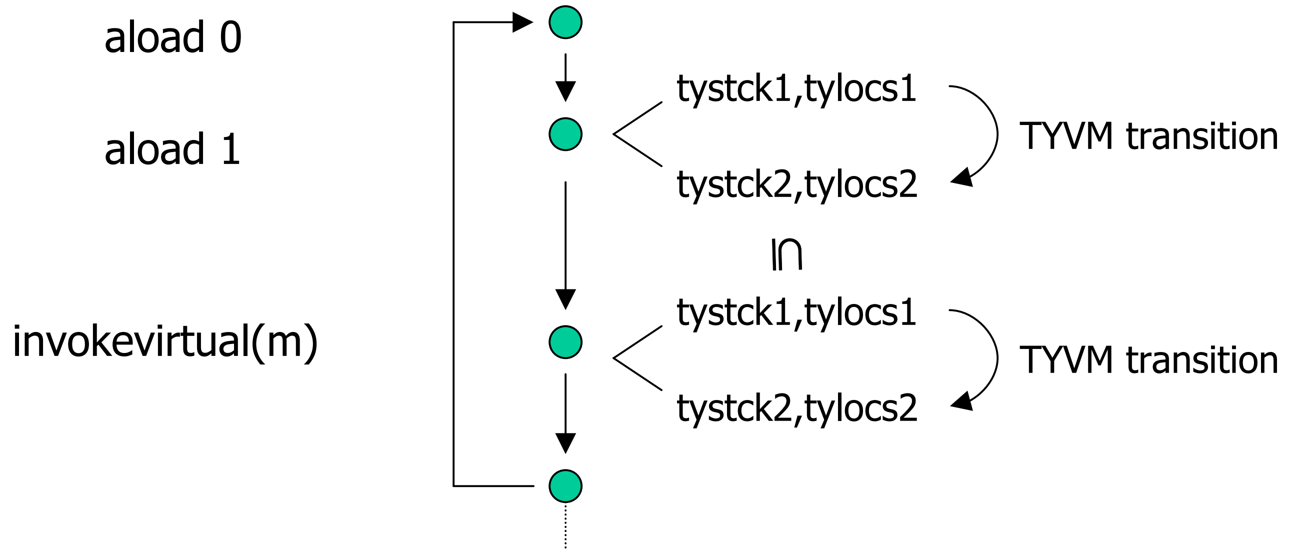
Embedded Linker: functional specification

- A new program format is introduced (linked format).
- Specification consists in two relations between a cap file and a card memory state.
  - Resolution post-condition: the linked format of the CAP file can be observed from the card memory.
  - Preparation post-condition: the static field image described in the CAP file and the initial static arrays can be observed from the card memory.

Off-card bytecode verifier: functional specification

- Type assignment: a mapping associating a (pair of) type stacks and local variable type mappings to each point of a method.

- Specification consists in a collection of conditions on a type assignment for the program.



Algorithm design

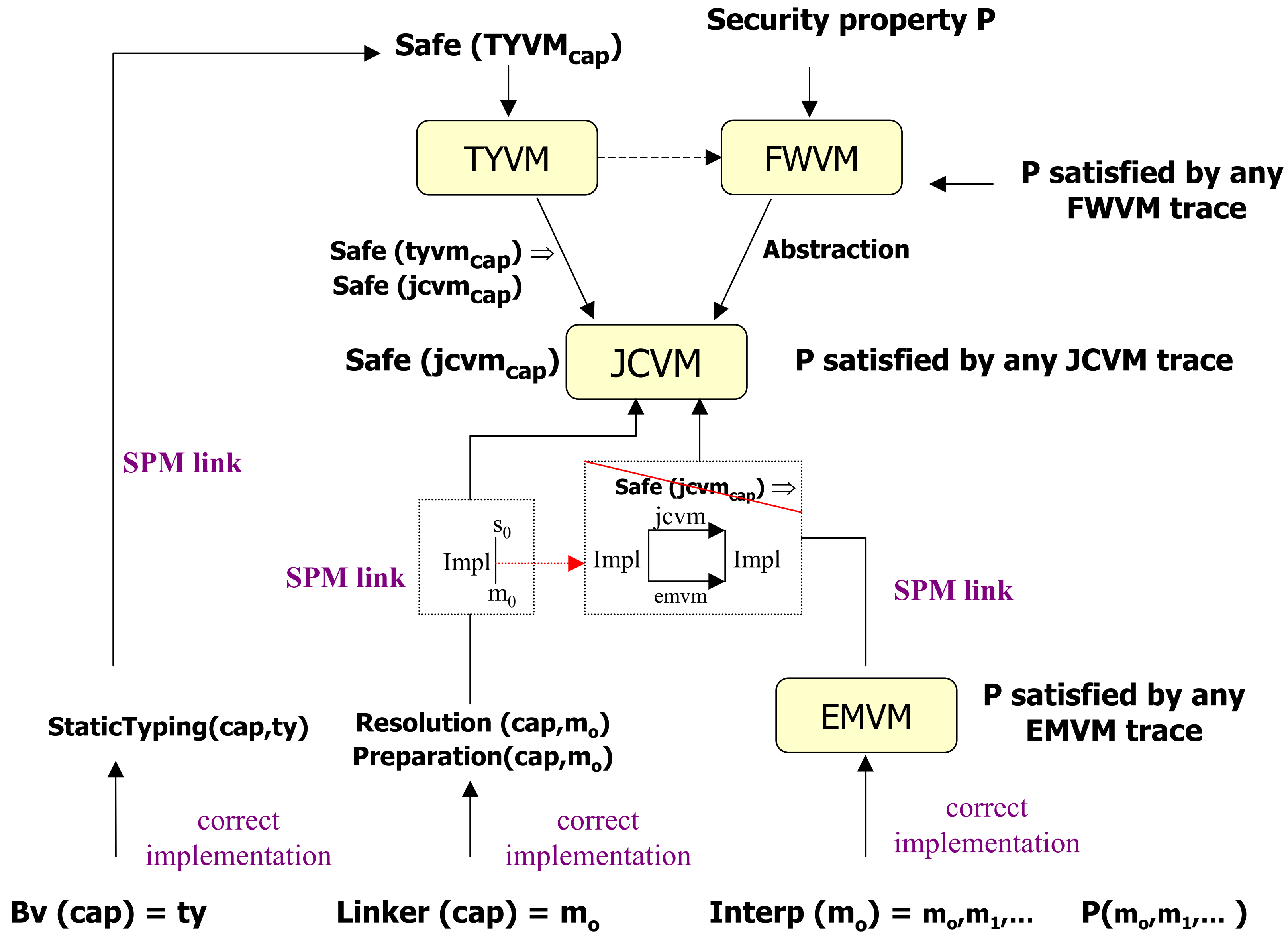
- Written in a functional programming language (Coq).

- Bytecode verifier: $bv : cap \rightarrow type$
  - A variant of Kildall's algorithm
  - Can deal with sub-routine polymorphism
- Embedded linker : $linker: CardMemory \rightarrow cap \rightarrow CardMemory$
- Embedded interpreter: $interp : CardMemory \rightarrow CardMemory$

- Could be extracted into Ocaml executable functions.
- Provides a way of testing the specifications.

A general architecture for proving security properties

Proof architecture



Some statistics

	LIB	SPEC	PROOF	Total
Lines	13423	35593	71904	**120920**
Modules	20	142	116	**278**

- Definitions : 2600
- Inductive definitions : 788
- Theorems : 2422
- Axioms : 236
- Model parameters : ≈230
- Six year/men of work (including documentation)
- Several people (from 2 to 5) working in parallel.

Conclusions

Modeling contributions (1/2)

- A general proof architecture for security properties.
  - Factorizes part of the proof effort.
  - Adaptable for a particular vendor's implementation.

- A complement to SUN's specifications (some examples)
  - CAP file information access.
  - Native method: invocation and resources.
  - Transaction effects on bytecode semantics.
  - The whole state of the API.
  - A useful bytecode abstraction.

Modeling contributions (1/2)

- Enhanced organization of the specification
- Logical dependencies between concepts are put forward.
  - Spread descriptions collected and completed.
  - What does the "JCRE" actually cover?
- Some specification imprecisions and omissions detected.
  - Example: what active context shall the JCRE use to call the `Applet.install` method?
- Slight refinements of SUN specification proposed.

Feedback about the Coq proof assistant

- Using Coq for industrial applications is feasible (not true 5 years ago).
- A challenge for the future: proving in the large.
  - Proof maintenance?
    - No experience in specification evolution.
    - An example: automatic generation of hypotheses names should be avoided as much as possible.
  - How development time can be reduced?
    - Tools for managing huge models become necessary.
    - Hypertext navigation, fold/unfold tools, find tools, etc.
    - Context sensitive information.
    - Should not be "external" tools!

Future work

- Verification of security properties.
- Customization for particular implementations an application domains (GSM, etc).
- Specification evolution (Java Card 2.2)
- Migration to Coq V7.2
- Integration into a certification tool (TL-FIT).