JavaOne
Sun's 2002 Worldwide Java Developer Conference

On-card byte code verification

The Ultimate Step

L. Casset[1], D. Deville[2], J.-L. Lanet[1]
Research Engineer[1], PhD Student[2]
Gemplus[1], Lille University - RD2P lab[2]

Presentation Goal

Learn how Java Card becomes closer to Java.

Learning Objectives

- As a result of this presentation, you will:
  - understand the techniques used to embed full type inference into a smart card,
  - see that formal methods are of practical use in software development.

This Slide Gains Your Audience's Attention

Complete on-card byte code verification was considered impossible until now…

…we did it !

Formal development of a smart card has never been done …

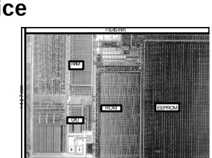
…we did it !

Agenda

- Smart Card and Applet Verification
- Type Verification in a highly constrained device a real challenge
- Java Card shows its true color
- Proof Carrying Code in practice
- Metrics

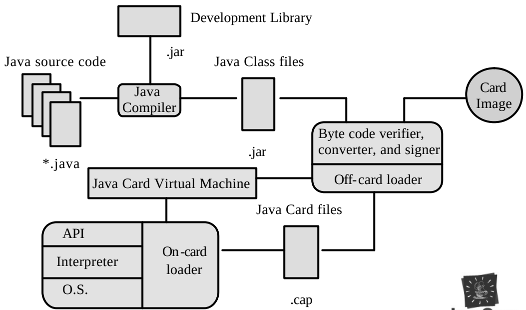
Smart Card

- **Heavily constrained device**
  - a micro module of 27mm²,
  - ISO normalization,
  - limited computing power.

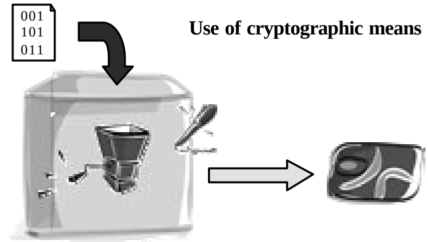


- **Mainly memory**
  - Read Only Memory (32-128 Kb),
  - Random Access Memory (128-4096 bytes),
  - EEPROM / FlashRAM (4-64Kb)
    - limited number of writes (stress),
    - low speed memory (write).

Smart Card and Java Card



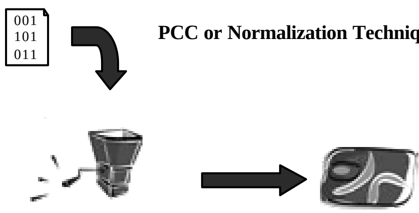
Post Issuance and Applet Verification



Use of cryptographic means

Byte Code verification in a trusted environment

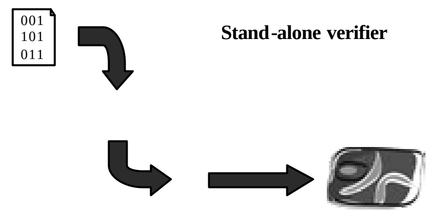
Post Issuance and Applet Verification



PCC or Normalization Technique

Byte Code processing, in a non trusted environment

Post Issuance and Applet Verification



Stand-alone verifier

No external treatment...

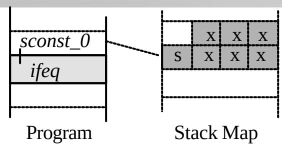
On-card Verification: a Real Challenge

- A byte code verifier contains:
  - a structural verifier,
  - a type verifier.
- Performed once during load phase.
- The verifier is a key point of the security architecture.
- We need the proof of the correct implementation of the verifier using a formal method.

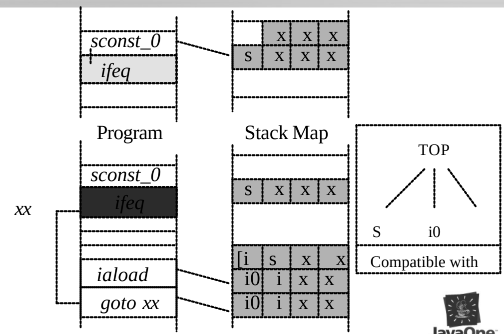
Inside On-card Verification

- Structural Verification
  - respect the CAP file format
  - perform a syntactic check of the incoming code
- Type verification
  - The verifier checks method per method that the typing rules are not violated,
  - In case of branches it must verify that types are compatible for all the paths.

Type Verification (cont.)

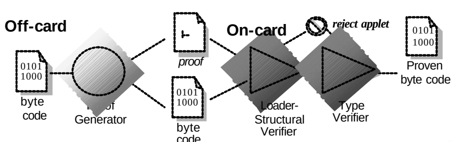


Type Verification (Cont.)



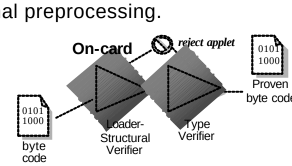
Two Solutions

- The **PCC Verifier** suits low end chip,
  - small memory usage,
  - external pre-processing: stack map like KVM.



Two Solutions

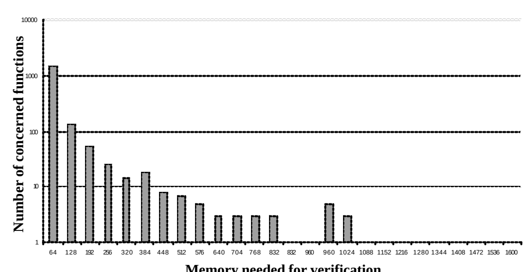
- The **PCC Verifier** suits low end chip,
  - small memory usage,
  - external pre-processing: stack map like KVM.
- The **Stand-alone Verifier** suits high end smart card,
  - no external preprocessing.



Java Card Shows its True Colour

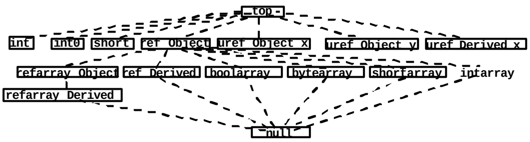
- Pros
  - no need for external assistance
  - accept all valid applets
- Cons
  - known to be unfeasible because of:
    - memory consumption
    - time complexity (mainly unification)

Memory Usage



Time Complexity

- The unification process can be complex
  - it consist of finding the Least Upper Bound (LUB) of two elements in a lattice,



  - we know the answer for primitive types.

Type Encoding: a Basic Solution

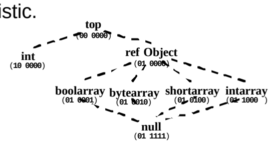
- We can pre -compute a unification table
  - easy to store in ROM or EEPROM,
  - efficient use for primitive type, simple to implement,

	top	int	ref Obj	[bool	[byte	[short	[int	null
top	top	top	top	top	top	top	top	top
int	top	int	top	top	top	top	top	top
ref Obj	top	top	ref Obj	ref Obj	ref Obj	ref Obj	ref Obj	ref Obj
[bool	top	top	ref Obj	[bool	ref Obj	ref Obj	ref Obj	[bool
[byte	top	top	ref Obj	ref Obj	[byte	ref Obj	ref Obj	[byte
[short	top	top	ref Obj	ref Obj	ref Obj	[short	ref Obj	[short
[int	top	top	ref Obj	ref Obj	ref Obj	ref Obj	[int	[int
null	top	top	ref Obj	[bool	[byte	[short	[int	null

  - we can do better.

Type Encoding: a Better Solution

- Takes into account EEPROM's stress characteristic.



- *boolarray* $\cap$ *int* = 010001 & 100000 = 000000 = *Top*

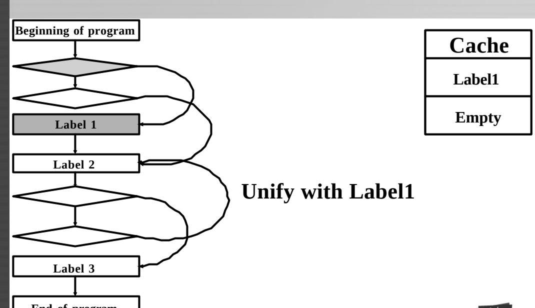
Software Cache

- We can use software cache to store stack maps
  - too big for RAM ,
  - need to be updated many times,
  - we have room in EEPROM but writes are slow and there is the stress problem.
- Cache stack maps in RAM and EEPROM.

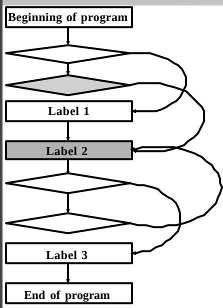
Cache Policy

- Simple Least Recently Used (LRU) policy
  - good performance and simple implementation
- We use the control graph flow of the verified method
  - no additional cost for using it
  - used during type inference initialisation phase

Control Graph Flow



Control Graph Flow

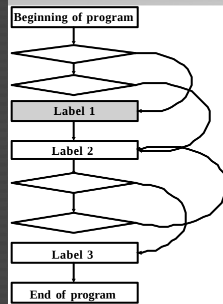


Cache
Label1
Label2

Unify with Label2

JavaOne

Control Graph Flow

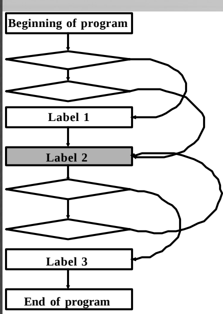


Cache
Label1
Label2

Choose next: Label1

JavaOne

Control Graph Flow

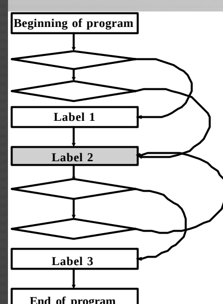


Cache
Label1
Label2

Unify with Label2

JavaOne

Control Graph Flow

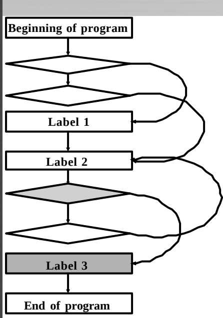


Cache
Label1
Label2

Choose next: Label2

JavaOne

Control Graph Flow

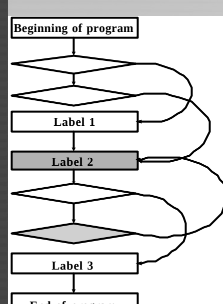


Cache
Label1 Label3
Label2

Unify with Label3

JavaOne

Control Graph Flow

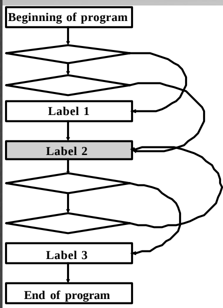


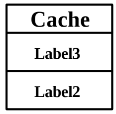
Cache
Label3
Label2

Unify with Label2

JavaOne

Control Graph Flow





Choose next: Label2
Unify with Label3
Unify with Label2
Choose next: Label3
End of verification

Cache Policy

- Results
  - only one update of data stored in EEPROM (Label1),
  - this can be avoid by control graph flow analysis
    - no need of keeping typing information for Label1.

Two Solutions

- The **Stand-alone Verifier** suits high end smart card,
  - no external preprocessing.
- The **PCC Verifier** suits low end chip,
  - small memory usage,
  - external pre-processing: stack map like KVM.

The PCC in Practice

- Pros
  - a type verification algorithm adapted to embedded device,
  - include the structural verifier part,
  - a formal model and implementation.
- Cons
  - an off-card part to compute type unification mandatory,
  - more memory (EEPROM) to store type unification results.

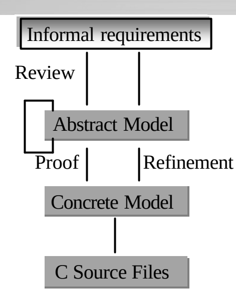
Formal Methods in Practice

- Mathematical based language,
- Provide an non ambiguous formal specification: the model,
- Propose a methodology to refine an implementation,
- Prove the correspondence between specification and implementation,

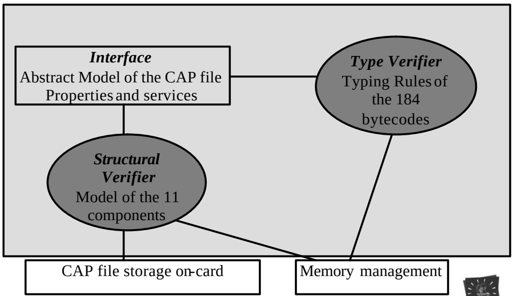
High quality code

High Quality Development

- Development with the B formal method
  - definition of the architecture,
  - formalisation of the specification in an abstract model,
  - refinement of the abstract model in a concrete model,
  - automatic code generation.



Inside the PCC verifier



The PCC Algorithm

- Needs additional typing information
  - the result of type unification,
  - stored into a custom component added to the CAP file.
- The algorithm is linear
  - check each instruction linearly,
  - for each branching instruction, checks the type compatibility of the target,
  - after each jump, take the types contained in the custom component.

Type Verifier

- Abstract model
  - the higher specification returns a boolean,
  - defines the loop on all the methods,
  - then, for each method, defines a loop on all the bytecodes,
  - specifies the typing rules of the 184 different bytecodes.
- Concrete model
  - refines the abstract model,
  - provides a proved implementation.

Structural Verifier

- Internal verifications
  - each component is modelled and checked,
  - provide access to information into the component.
- External verifications
  - models shared information between components.

Metrics

- Metrics to compare both implementation of the verifiers
  - including structural verification when available,
  - in terms of memory consumption.
- Metrics to compare both development for the type verifier
  - excluding structural verification,
  - in terms of workload and bugs.

Comparing PCC and Stand-alone Verifiers Implementation

	PCC	Stand-alone
Type ROM size (kb)	18	16
Structural ROM size (kb)	24	NYI
Total ROM size (kb)	45	24
RAM (bytes)	140	128 - 756*
Applet code overhead (%)	10-20	0

*Note that the RAM usage for the standalone verifier is dynamically tuneable

Comparing Formal and Traditional Developments - Workload

	Formal	Traditional
Development	12 weeks	10 weeks
Proof	6 weeks	NA
Test	1 week	4 weeks
Total	19 weeks	14 weeks

Comparing Formal and Traditional Developments - Bugs

	Formal	Traditional
Discovered by tests	17	54
Discovered by proof	29	NA
Total	46	54

Summary

- A real technological challenge
  - 2/3 years ago this features were considered impossible to implement,
  - formal development for smart card was considered as unrealistic,
  - Gemplus investment in new technology.
- GemClassifier: a technology breakthrough for the Java Card deployment

If You Only Remember One Thing…

GemClassifier: the first on-card proved implementation of a Java Card byte code verifier

Demo

Q&A