



UNIVERSITÀ DI PISA

# Secure Information Flow in Programs

Prof. Cinzia Bernardeschi  
Dipartimento di Ingegneria dell'Informazione  
Università di Pisa  
[cinzia.bernardeschi@unipi.it](mailto:cinzia.bernardeschi@unipi.it)

IDS - Ingegneria Dei Sistemi  
Pisa, 23 Gennaio 2018

# Outline

- Data leakage
  - Security policy
  - Information flow in programs
  - Examples of illegal flow of information
- A static analysis approach for program certification
- Case studies:
  - Secure Interaction in Java cards
  - Data secure flow in AUTOSAR

# Data leakage

*GENERAL DATA PROTECTION REGULATION*(GDPR) - UE 2016/679

REGULATION OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC

Member States shall notify to the Commission the provisions of their laws which they adopt pursuant to this paragraph by 25 May 2018 and, without delay, any subsequent amendment law or amendment affecting them.

## Data leakage

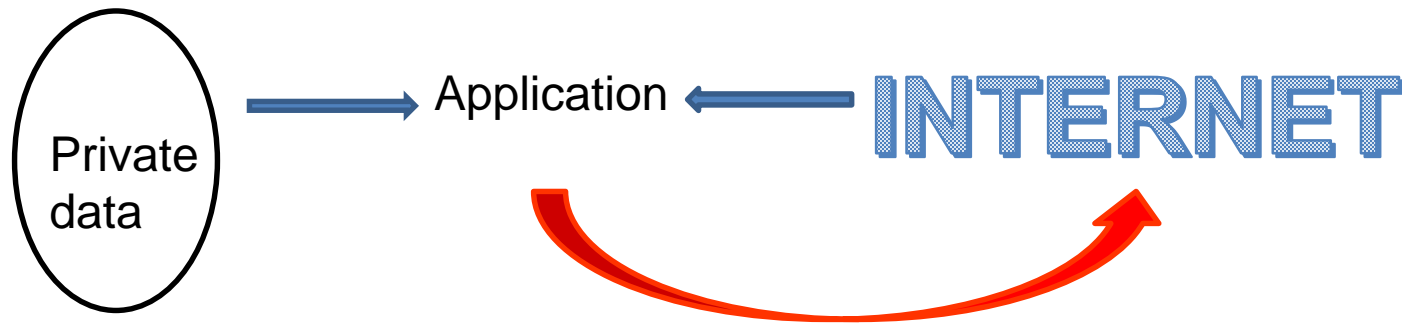
- explicit (private data made publicly available)
- interference between private and public data
- colluding apps



Information flow analysis

# Data leakage

Limit of Firewall and Access control mechanisms



Application authorized to access private data

Application authorized to access internet

Control on the information sent on the internet!!!!

Certificate that the application does not send data that may reveal any private information

**Certification of applications for secure information flow**

# Java applet

- **downloaded** by the internet and executed by web browser
- need **access** to user private data to compute some information
- possible **leakage** of user private data
- access control can be too restrictive
- security mechanisms:
  - Access control, Firewall, Sandbox

# Colluding applets

***The Independent*** (British online newspaper)

April 2017

Taken from: <http://www.independent.co.uk/life-style/gadgets-and-tech/news/android-app-steal-users-data-colluding-each-other-research-cartel-information-a7663976.html>



The team reports that the types of app fall into two major categories / Justin Sullivan/Getty Images

The biggest security risks can come from some of the least capable apps

“Android apps are mining smartphone users’ data by secretly colluding with each other, according to a new study. Pairs of apps can trade information, a capability that can lead to serious consequences in terms of security.

# Colluding applets

## Tutela contro le app mobili colluse



Le app mobili di oggi possono parlarsi facilmente. Purtroppo, questi canali comunicativi possono nascondere anche dei comportamenti insidiosi. Quando si analizzano separatamente due o più app, il comportamento di una di esse può sembrare completamente innocuo. Ma quando nello stesso dispositivo vengono installate delle app mobili che colludono, queste possono scambiarsi informazioni ed eseguire azioni ostili.



# Data leakage

## Recent works

### **Utilising K-Semantics for Collusion Detection in Android Applications**

Irina Măriuca Asăvoae, Nga Nguyen, Markus Roggenbach, Siraj Shaikh

Workshop on Formal Methods for Industrial Critical Systems International

Workshop on Automated Verification of Critical Systems, September 2016

### **Automated generation of colluding apps for experimental research**

Jorge Blasco, Thomas M. Chen

Journal of Computer Virology and Hacking Techniques, 2017

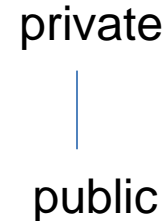
### **Detection of app collusion potential using logic programming**

Jorge Blasco, Thomas M. Chen, Igor Muttik, Markus Roggenbach

Journal of Network and Computer Applications, December 2017

# Security policy

**Multilevel Security policy:** a security policy that allows the classification of data and users based on a system of hierarchical security levels.



*Non-interference property:* the security domain private is non-interfering with domain public if no input by private can influence subsequent outputs seen by public.

Inputs and outputs are classified as either *low* sensitive (public) or *high* sensitive (private). A program has the non-interference property if and only if any sequence of low inputs will produce the same low outputs, regardless of what the high level inputs are.

The program responds in exactly the same manner on low outputs whether or not high sensitive data are changed. The low user will not be able to acquire any information about data and the activities (if any) of the high user.

# Information flow in programs

Modular programming

Information flow occurs through

- simple variables, input/output files
- array, structures, objects
- pointers, references
- objects allocated in dynamic memory
- global variables
- function calls  
(parameters by value, parameters by reference, return)

# Basics of information flow

High-level languages. Let  $x, y$  be variables

$y := x;$                       explicit flow

variable  $y$  is assigned the value of  $x$ ;  
there is an explicit flow  $x$  to  $y$

if ( $x = 0$ )                      implicit flow  
    then  $y=1$ ;  
    else  $y=0$ ;

there is an implicit flow from variable  $x$  to  $y$ , since  $y$  is assigned different values depending on the value of the condition of the control instruction (variable  $x$ )

In both cases observing the final value of  $y$  reveals information on the value of  $x$

A conditional instruction in a program causes the beginning of an implicit flow. The implicit flow begins when the conditional instruction starts (we say that we have an opened implicit flow); all the instructions in the scope of the if depend on the condition of the if.

# Basics of information flow

If a function call is executed in the scope of a conditional instruction, the function is executed under the implicit flow.

For example,

```
    if (y < 0)
        then f();
```

Function  $f()$  is invoked depending on the value of variable  $y$ .

Instructions of  $f()$  are executed under the implicit flow of the condition of the if statement.

## Termination agreement

```
    while (y > 1000) <instructions> ;
```

Covert flows in the literature

# Basics of information flow

## Bytecode instructions

<code>op</code>	pop two operands off the stack, perform the operation, and push the result onto the stack
<code>pop</code>	discard the top value from the stack
<code>push <math>k</math></code>	push the constant $k$ onto the stack
<code>load <math>x</math></code>	push the value of the variable $x$ onto the stack
<code>store <math>x</math></code>	pop off the stack and store the value into variable $x$
<code>if <math>j</math></code>	pop off the stack and jump to $j$ if non-zero
<code>goto <math>j</math></code>	jump to $j$
<code>jsr <math>j</math></code>	at address $p$ , jump to address $j$ and push return address $p + 1$ onto the operand stack
<code>ret <math>x</math></code>	jump to the address stored in $x$
<code>halt</code>	stop

# Basics of information flow

```
1 load x
2 store y
3 halt
```

explicit flow

x is loaded onto the stack, then it is stored into y, that is, y depends explicitly on x

```
1 load x
2 if 5
3 push 1
4 goto 6
5 push 0
6 store y
7 halt
```

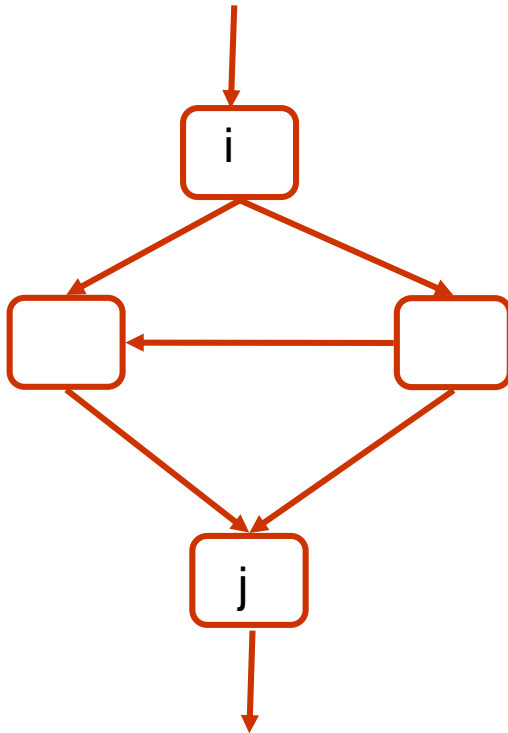
implicit flow

variable x is loaded onto the stack. Depending on the value of x, either the constant 1 or the constant 0 is pushed onto the stack, and successively stored onto y

In both cases observing the final value of y reveals information on the value of x

# Basics of information flow

## Control Flow Graph and ipd



We can use the **control flow graph** of the program to handle implicit flows.

**j=immediate postdominator of i**: the first node belonging to all paths from i

The implicit flow of an if instruction at address i terminates at the instruction with address  $ipd(i)$ .

# Basics of information flow

## Influence of the implicit flow onto the operand stack

the stack may be manipulated in different ways by the branches of a branching instruction: they can perform a different number of pop and push operations, and with a different order.

The length and the content of the operand stack may be a means by which security leakages can occur

```
1  push 1
2  load x
3  if 5
4  pop
5  halt
```

The stack is empty or not, depending on the value of x

# Basics of information flow

## Termination Agreement

```
1  load x
2  if 1
3  halt
```

# Secure Information Flow

- a program  $P$
- a lattice of security levels  $\mathcal{L}$
- every variable of  $P$  is assigned a security level in  $\mathcal{L}$
- $P$  satisfies Secure Information Flow if information at a given security level does not flow to lower levels

## Lattice

Let be given a set  $A$  and order relation  $\leq$  on  $A$ .

$(A, \leq)$  is a lattice if every pair of elements in  $A$  has both a greatest lower bound ( $glb$ ) and a least upper bound ( $lub$ ).

D. E. Denning, P. J. Denning.

Certification of programs for secure information flow.

Communications of the ACM, 20(7), 1977

# Secure Information Flow (SIF)

$\mathcal{L} = \{L, H\}$ , with  $L \leq H$

L: public, H: private

Let  $x:H$ ,  $y:L$



- Explicit information flow

$y = x;$

- Implicit information flow

$\text{if } (x) \ y=5; \text{ else } y=2$

**SIF**

the final value of each variable does not depend on the initial value of variables with higher or not related security levels.

# Termination Agreement

## TERM

it is not possible to leak high information by observing the termination of the program

```
while (x > 0) do skip;
```

# Timing Agreement

## TIME

it is not possible to leak high information by observing the number of instructions executed

```
if (x == 0) { x=1; skip; } else x=2;
```

# Secure Information flow verification

## 1) Typing approach

D. Volpano, G. Smith, C. Irvine.

A sound type system for secure flow analysis.

Journal of Computer Security, 4(3), 1996, pp. 167-187.

## 2) Semantic-based approach

Rajeev Joshi, K.Rustan M.Leino

A semantic approach to secure information flow

Science of Computer Programming, Volume 37, Issues 1–3, May 2000,

*High complexity in space and time*

## 3) Abstract interpretation of the operational semantics

Roberto Barbuti, Cinzia Bernardeschi, Nicoletta De Francesco

Abstract interpretation of operational semantics for secure information flow.

Inf. Process. Lett. 83(2): 101-108 (2002)

.....

# Abstract interpretation of the operational semantics

- the standard operational semantics of the programming language is enhanced to include information useful in the analysis.
- abstract domains are identified and abstract semantics rules are defined that execute the program on abstract domains.
- the abstract rules compute the flow of information in the program

# Secure Information flow verification

**Typing:** the security information of a variable belongs to its type, and secure Information flow is checked by means of a type system. Hierarchy between types. Types = H, L

H  
|  
L

**Semantic-based:** execute the program

**Abstract interpretation:** execute the program on abstract domains

An advantage of 3) with respect to those based on 1) is that it is semantics based and thus keeps information on the dynamic behavior of programs, allowing to check more precisely the desired properties.

y=x;  
y=0;

rejected by 1)

if 0 then y=x; else skip;

rejected by 1) and by 3)

# Abstract interpretation for analyzing Secure Information flow

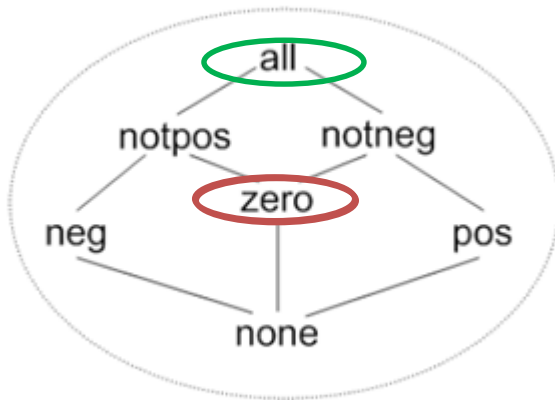
# Abstract interpretation

Abstract interpretation is a method for designing approximate semantics of programs

The abstract interpretation approach is based on lattices, continuous functions and Galois connections

Galois connections to establish a correspondence between the domain of concrete properties and the domain of abstract properties

An example of complete lattice, with  
 $D = \{all, notpos, notneg, zero, neg, pos, none\}$ :



glb: greatest lower bound

lub: least upper bound

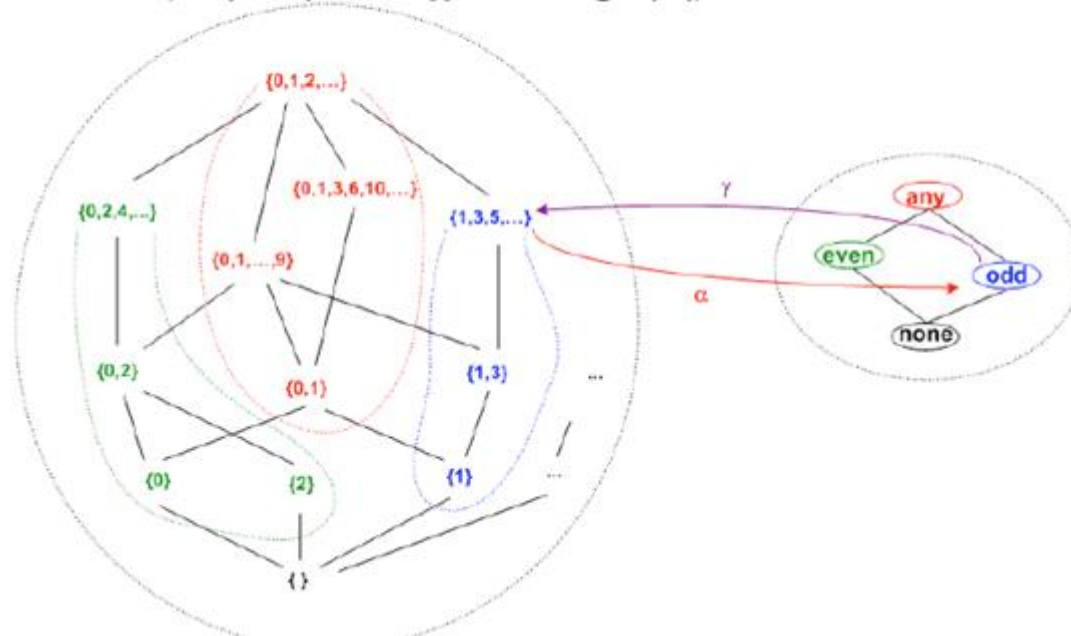
$\text{glb}(\text{notpos}, \text{notneg}) = \text{zero}$

$\text{lub}(\text{notpos}, \text{notneg}) = \text{all}$

Taken from: “Abstract interpretation and static analysis” Course held at the International Winter School on Semantics and Applications, Uruguay, 2003, by David Schmidt

# Abstract interpretation

A complete lattice  $\langle \mathcal{P}(Int), \subseteq, \{\}, Int, \cup, \cap \rangle$ , and an abstraction

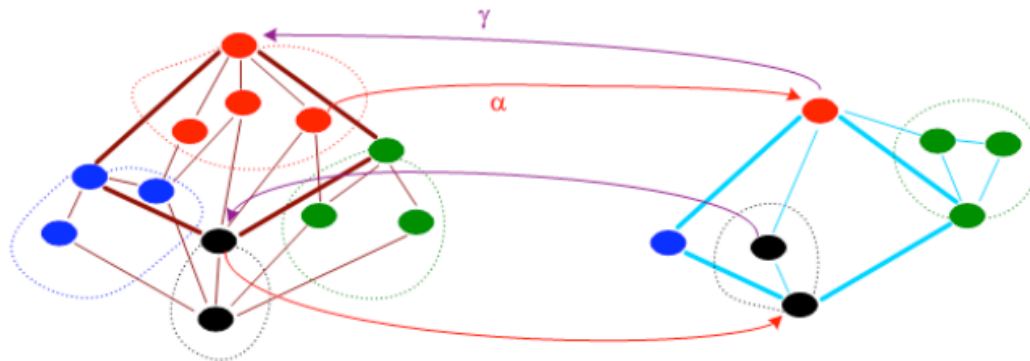


Taken from: "Abstract interpretation and static analysis" Course held at the International Winter School on Semantics and Applications, Uruguay, 2003, by David Schmidt

# Abstract interpretation

Given a complete lattice of *concrete* execution data  $\mathcal{C}$ , and a simpler complete lattice of *abstract* data  $\mathcal{A}$ , the function  $\alpha : \mathcal{C} \rightarrow \mathcal{A}$  acts like a homomorphism when we study the operations on  $\mathcal{C}$ . Let  $\gamma : \mathcal{A} \rightarrow \mathcal{C}$  be the inverse of  $\alpha$ .

**Galois connection:** for complete lattices,  $\mathcal{C}$  and  $\mathcal{A}$ , and monotonic functions,  $\alpha : \mathcal{C} \rightarrow \mathcal{A}$  and  $\gamma : \mathcal{A} \rightarrow \mathcal{C}$ , the pair  $\langle \alpha, \gamma \rangle$  forms a Galois connection, written  $\mathcal{C} \langle \alpha, \gamma \rangle \mathcal{A}$ , iff  $c \sqsubseteq_{\mathcal{C}} \gamma \circ \alpha(c)$  and  $\alpha \circ \gamma(a) \sqsubseteq_{\mathcal{A}} a$ .



Taken from: "Abstract interpretation and static analysis" Course held at the International Winter School on Semantics and Applications, Uruguay, 2003, by David Schmidt

# Abstract interpretation

For Galois connection  $\mathcal{C} \langle \alpha, \gamma \rangle \mathcal{A}$ , and functions  $f : \mathcal{C} \rightarrow \mathcal{C}$ , we can say that  $f^\# : \mathcal{A} \rightarrow \mathcal{A}$  is a *sound approximation* of  $f$  iff

$$\begin{aligned} (\alpha \circ f)(c) &\sqsubseteq_{\mathcal{A}} (f^\# \circ \alpha)(c), \text{ for all } c \in \mathcal{C} \\ \text{iff} \\ (f \circ \gamma)(a) &\sqsubseteq_{\mathcal{C}} (\gamma \circ f^\#)(a), \text{ for all } a \in \mathcal{A} \end{aligned}$$

Note that  $\alpha$  acts like a “semi-homomorphism” with respect to  $f$  and  $f^\#$ :

$$\begin{array}{ccc} c & \xrightarrow{\alpha} & \alpha(c) \\ f \downarrow & & \downarrow f^\# \\ f(c) & \xrightarrow{\alpha} & \alpha(f(c)) \sqsubseteq f^\#(\alpha(c)) \end{array}$$

Taken from: “Abstract interpretation and static analysis” Course held at the International Winter School on Semantics and Applications, Uruguay, 2003, by David Schmidt

# The method: abstract interpretation of the operational semantics

- concrete instrumented semantics recording the information flow (collecting semantics)
- abstract semantics taking only what concerns the information flow
- correctness of the abstraction

# A simple instruction set: operational semantics

$exp ::= const \mid var \mid exp \ op \ exp$

$com ::= var := exp \mid \text{if } exp \text{ then } com \text{ else } com \mid$   
 $\text{while } exp \text{ do } com \mid com ; com \mid \text{skip}$

$\longrightarrow^\epsilon \subseteq Q^\epsilon \times Q^\epsilon$  transition system

$Q^\epsilon$  set of state

state:  $\langle c, m \rangle$

**Expr<sub>const</sub>**  $\frac{}{\langle k, m \rangle \longrightarrow_{expr}^\epsilon k}$

**Expr<sub>var</sub>**  $\frac{}{\langle x, m \rangle \longrightarrow_{expr}^\epsilon m(x)}$

**Expr<sub>op</sub>**  $\frac{\langle e_1, m \rangle \longrightarrow_{expr}^\epsilon k_1 \quad \langle e_2, m \rangle \longrightarrow_{expr}^\epsilon k_2 \quad k_1 \ op \ k_2 = k_3}{\langle (e_1 \ op \ e_2), m \rangle \longrightarrow_{expr}^\epsilon k_3}$

**Ass**  $\frac{\langle e, m \rangle \longrightarrow_{expr}^\epsilon k}{\langle x := e, m \rangle \longrightarrow^\epsilon m[k/x]}$

**Skip**  $\frac{}{\langle \text{skip}, m \rangle \longrightarrow^\epsilon \langle m \rangle}$

**If<sub>true</sub>**  $\frac{\langle e, m \rangle \longrightarrow_{expr}^\epsilon true}{\langle \text{if } e \text{ then } c_1 \text{ else } c_2, m \rangle \longrightarrow^\epsilon \langle c_1, m \rangle}$

**While<sub>true</sub>**  $\frac{\langle e, m \rangle \longrightarrow_{expr}^\epsilon true}{\langle \text{while } e \text{ do } c, m \rangle \longrightarrow^\epsilon \langle c; \text{while } e \text{ do } c, m \rangle}$

**While<sub>false</sub>**  $\frac{\langle e, m \rangle \longrightarrow_{expr}^\epsilon false}{\langle \text{while } e \text{ do } c, m \rangle \longrightarrow^\epsilon \langle m \rangle}$

**Seq<sub>1</sub>**  $\frac{\langle c_1, m \rangle \longrightarrow^\epsilon \langle m' \rangle}{\langle c_1; c_2, m \rangle \longrightarrow^\epsilon \langle c_2, m' \rangle}$

**Seq<sub>2</sub>**  $\frac{\langle c_1, m \rangle \longrightarrow^\epsilon \langle c_2, m' \rangle}{\langle c_1; c_3, m \rangle \longrightarrow^\epsilon \langle c_2; c_3, m' \rangle}$

# Concrete Operational Semantics

An instrumented semantics which:

- Handles values  $(k, \sigma)$  **annotated with a security level**. During the execution of a program,  $\sigma$  indicates the least upper bound of the security levels of the information flows, both explicit and implicit, on which  $k$  depends.
- Executes instructions under a **security environment  $\sigma$** . During the execution,  $\sigma$  represents the least upper bound of the security levels of the open implicit flows.  $\sigma$  is (possibly) **upgraded** when a branching instruction begins and is (possibly) **downgraded** when all branches join.
- $C(P)$  : concrete transition system for  $P$

# Concrete operational semantics

state:  $\langle \sigma, c, M \rangle$

$$\mathbf{Expr}_{const} \quad \frac{}{\langle k^\sigma, M \rangle \longrightarrow_{expr} (k, \sigma)} \quad \mathbf{Expr}_{var} \quad \frac{M(x) = (k, \tau)}{\langle x^\sigma, M \rangle \longrightarrow_{expr} (k, \sigma \sqcup \tau)}$$

$$\mathbf{Expr}_{op} \quad \frac{\langle e_1^\sigma, M \rangle \longrightarrow_{expr} (k_1, \tau_1) \quad \langle e_2^\sigma, M \rangle \longrightarrow_{expr} (k_2, \tau_2)}{\langle (e_1 \text{ op } e_2)^\sigma, M \rangle \longrightarrow_{expr} (k_1 \text{ op } k_2, \tau_1 \sqcup \tau_2)}$$

$$\mathbf{Ass} \quad \frac{\langle e^\sigma, M \rangle \longrightarrow_{expr} v}{\langle (x := e)^\sigma, M \rangle \longrightarrow M[v/x]} \quad \mathbf{Skip} \quad \frac{}{\langle \text{skip}^\sigma, M \rangle \longrightarrow \langle M \rangle}$$

$$\mathbf{If}_{true} \quad \frac{\langle e^\sigma, M \rangle \longrightarrow_{expr} (true, \tau)}{\langle (\text{if } e \text{ then } c_1 \text{ else } c_2)^\sigma, M \rangle \longrightarrow \langle c_1^\tau, Impl(M, Mod(c_1) \cup Mod(c_2), \tau) \rangle}$$

$$\mathbf{While}_{true} \quad \frac{\langle e^\sigma, M \rangle \longrightarrow_{expr} (true, \tau)}{\langle (\text{while } e \text{ do } c)^\sigma, M \rangle \longrightarrow \langle (c; \text{while } e \text{ do } c)^\tau, Impl(M, Mod(c), \tau) \rangle}$$

$$\mathbf{While}_{false} \quad \frac{\langle e^\sigma, M \rangle \longrightarrow_{expr} (false, \tau)}{\langle (\text{while } e \text{ do } c)^\sigma, M \rangle \longrightarrow \langle Impl(M, Mod(c), \tau) \rangle}$$

$$\mathbf{Seq}_1 \quad \frac{\langle c_1^\sigma, M \rangle \longrightarrow \langle M' \rangle}{\langle c_1^\sigma; w, M \rangle \longrightarrow \langle w, M' \rangle} \quad \mathbf{Seq}_2 \quad \frac{\langle c_1^\sigma, M \rangle \longrightarrow \langle w', M' \rangle}{\langle c_1^\sigma; w, M \rangle \longrightarrow \langle w'; w, M' \rangle}$$

# Abstract Operational Semantics

the abstract semantics:

- abstracts concrete values into their security level:  $\alpha(k, \sigma) = \sigma$
- uses the same rules of the concrete semantics on the abstract domains

$A(P)$  : abstract transition system for  $P$

- finite
- multiple path
- each path of  $C(P)$  is correctly abstracted onto a path of  $A(P)$

A program  $P$  has secure information flow if in each final state of  $A(P)$ , each  $x : \sigma$  holds a value  $\tau \leq \sigma$ .

# Java bytecode: a simple instruction set

<b>op</b>	pop two operands off the stack, perform the operation, and push the result onto the stack
<b>pop</b>	discard the top value from the stack
<b>push k</b>	push the constant <code>k</code> onto the stack
<b>load x</b>	push the value of variable <code>x</code> onto the stack
<b>store x</b>	pop off the stack and store the value into <code>x</code>
<b>if j</b>	pop off the stack and jump to <code>j</code> if non-zero
<b>goto j</b>	jump to <code>j</code>
<b>jsr j</b>	at address <code>p</code> , jump to address <code>j</code> and push <code>p+1</code> onto the operand stack
<b>ret x</b>	jump to the address stored in <code>x</code>
<b>halt</b>	stop

# Standard Operational Semantics

**x: 5**  
**y: 1**

state: <program counter, memory, operand stack>

$\langle PC, [MEM(x) \ MEM(y)], STACK \rangle$

0 load *y*

1 if 4

2 push 1

3 goto 5

4 push 0

5 store *x*

6 halt

$\langle 0, [5, 1], \lambda \rangle$

↓**load**

$\langle 1, [5, 1], 1 \rangle$

↓**if<sub>true</sub>**

$\langle 4, [5, 1], \lambda \rangle$

↓**push**

$\langle 5, [5, 1], 0 \rangle$

↓**store**

$\langle 6, [0, 1], \lambda \rangle$

# Concrete Operational Semantics

x: (5,L)  
y: (1,H)

ipd: immediate post-dominator  
ipd(1) = 5

state:  $\langle \text{env}, \text{program counter}, \text{memory}, \text{operand stack}, \text{ipd stack} \rangle$

$\langle ENV, PC, [MEM(x) \ MEM(y)], STACK, IPD \rangle$

0 load y

$\langle l, 0, [(5, l), (1, h)], \lambda, \lambda \rangle$

1 if 4

$\downarrow_{\text{load}}$

$\langle l, 1, [(5, l), (1, h)], (1, h), \lambda \rangle$

2 push 1

$\downarrow_{\text{if}_{\text{true}}}$

$\langle h, 4, [(5, l), (1, h)], \lambda, (5, l) \rangle$

3 goto 5

$\downarrow_{\text{push}}$

$\langle h, 5, [(5, l), (1, h)], (0, h), (5, l) \rangle$

4 push 0

$\downarrow_{\text{ipd}}$

$\langle l, 5, [(5, l), (1, h)], (0, h), \lambda \rangle$

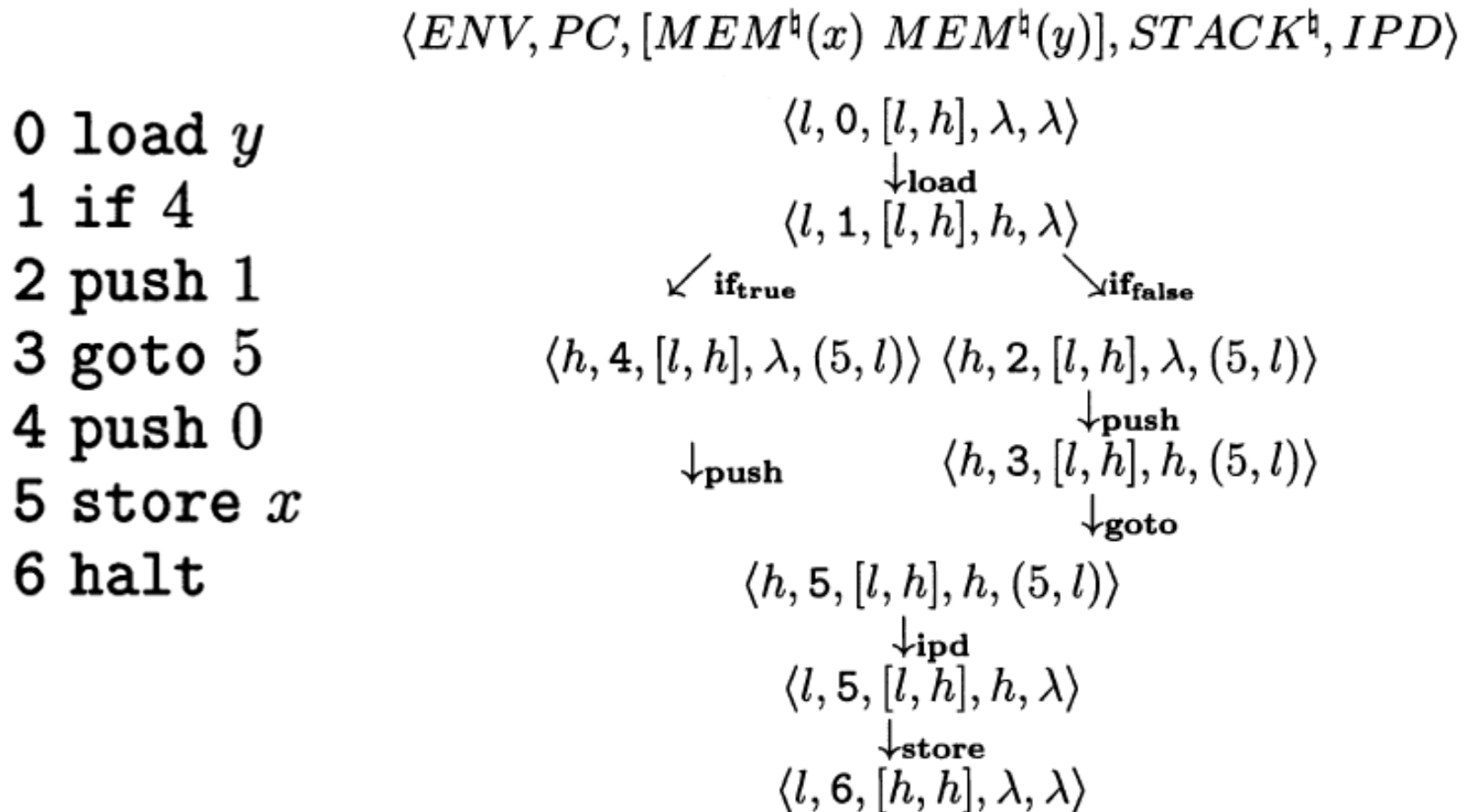
5 store x

$\downarrow_{\text{store}}$

$\langle l, 6, [(0, h), (1, h)], \lambda, \lambda \rangle$

6 halt

# Abstract Operational Semantics



# Domains of the concrete semantics

Security levels	$\mathcal{L} = \{L < H\}$	$\sigma, \tau, ..$
Constants	$V$	$k, k', ..$
Addresses	$A$	$i, j, ..$
Concrete Values	$\mathcal{V} = V \times \mathcal{L}$	$(k, \sigma)$
Concrete Addresses	$\mathcal{A} = A \times \mathcal{L}$	$(i, \sigma)$
Concrete Memories	$\mathcal{M} = \text{var} \rightarrow (\mathcal{V} \cup \mathcal{A})$	$M, M', ..$
Concrete Stacks	$\mathcal{S} = (\mathcal{V} \cup \mathcal{A})^*$	$S, S', ..$
Environments	$\mathcal{E} = \mathcal{L}$	$\sigma, \tau, ..$

# Concrete Semantics

STATES  $\mathcal{L} \times A \times \mathcal{M} \times S \times (A \cup \{0\})$

$\langle \sigma, PC, M, S, IPD \rangle$

$\sigma$

environment

PC

program counter

M

memory

S

operand stack

IPD



if  $\sigma = H$ , holds the address where  
the high implicit flow terminates

if  $\sigma = L$ , holds 0

# Transition relation rules

$P[PC] : \text{load } x, M[x] = (k, \tau), PC \neq \text{IPD}$

---

$\langle \sigma, PC, M, S, \text{IPD} \rangle \rightarrow$   
 $\langle \sigma, PC+1, M, (k, \sigma \cup \tau) \cdot S, \text{IPD} \rangle$

# Transition relation rules

$P[PC] : \text{store } \mathbf{x} , PC \neq \text{IPD}$

---

$$\langle \sigma, PC, M, (k, \tau) \cdot S, \text{IPD} \rangle \rightarrow \langle \sigma, PC+1, M[(k, \sigma \cup \tau) / \mathbf{x}], S, \text{IPD} \rangle$$

# Transition relation rules

$$PC = IPD$$

---

$$\langle \sigma, PC, M, S, PC \rangle \rightarrow \langle L, PC, M, S, 0 \rangle$$

# Transition relation rules

$P[PC] : \text{if } j, PC \neq IPD$

---

$\langle \sigma, PC, M, (0, \tau) \cdot S, IPD \rangle \rightarrow$   
 $\langle \sigma \cup \tau, PC+1, \text{up}(M), \text{up}(S), IPD' \rangle$

$IPD' = \begin{cases} \text{ipd}(PC) & \text{if } \sigma = L \text{ and } \tau = H, \text{ (an high implicit flow begins)} \\ IPD & \text{otherwise} \end{cases}$

$\text{up}(M)$  upgrades the value of the variables assigned in the scope of the implicit flow beginning at  $PC$

$\text{up}(S)$  upgrades all elements in the stack

# Concrete rules

$$\begin{array}{l}
 \text{ipd} \quad \frac{\rho = (i, \tau) \cdot \rho'}{\langle \sigma, i, M, S, \rho \rangle \longrightarrow \langle \tau, i, M, S, \rho' \rangle} \\
 \text{op} \quad \frac{c[i] = \text{op} \quad \text{not\_top}(i, \rho)}{\langle \sigma, i, M, (k_1, \tau_1) \cdot (k_2, \tau_2) \cdot S, \rho \rangle \longrightarrow \langle \sigma, i + 1, M, (k_1 \text{ op } k_2, \tau_1 \sqcup \tau_2) \cdot S, \rho \rangle} \\
 \text{pop} \quad \frac{c[i] = \text{pop} \quad \text{not\_top}(i, \rho)}{\langle \sigma, i, M, (k, \tau) \cdot S, \rho \rangle \longrightarrow \langle \sigma, i + 1, M, S, \rho \rangle} \\
 \text{push} \quad \frac{c[i] = \text{push } k \quad \text{not\_top}(i, \rho)}{\langle \sigma, i, M, S, \rho \rangle \longrightarrow \langle \sigma, i + 1, M, (k, \sigma) \cdot S, \rho \rangle} \\
 \text{load} \quad \frac{c[i] = \text{load } x \quad M(x) = (k, \tau) \quad \text{not\_top}(i, \rho)}{\langle \sigma, i, M, S, \rho \rangle \longrightarrow \langle \sigma, i + 1, M, (k, \tau \sqcup \sigma) \cdot S, \rho \rangle} \\
 \text{store} \quad \frac{c[i] = \text{store } x \quad \text{not\_top}(i, \rho)}{\langle \sigma, i, M, (k, \tau) \cdot S, \rho \rangle \longrightarrow \langle \sigma, i + 1, M[(k, \tau)/x], S, \rho \rangle} \\
 \text{goto} \quad \frac{c[i] = \text{goto } j \quad \text{not\_top}(i, \rho)}{\langle \sigma, i, M, S, \rho \rangle \longrightarrow \langle \sigma, j, M, S, \rho \rangle} \\
 \text{if}_{\text{false}} \quad \frac{c[i] = \text{if } j \quad \text{not\_top}(i, \rho)}{\langle \sigma, i, M, (0, \tau) \cdot S, \rho \rangle \longrightarrow \langle \tau, i + 1, \text{up}_M(M, \text{mod}^P(F^P(i)), \tau), \text{up}_S(S, \tau), (\text{ipd}(i), \sigma) \cdot \rho \rangle} \\
 \text{if}_{\text{true}} \quad \frac{c[i] = \text{if } j \quad \text{not\_top}(i, \rho)}{\langle \sigma, i, M, (k \neq 0, \tau) \cdot S, \rho \rangle \longrightarrow \langle \tau, j, \text{up}_M(M, \text{mod}^P(F^P(i)), \tau), \text{up}_S(S, \tau), (\text{ipd}(i), \sigma) \cdot \rho \rangle}
 \end{array}$$

# Abstract semantics

Abstract constants

$$\mathcal{V}^{\#} = \{ \cdot \}$$

Abstract security levels

$$\mathcal{L}^{\#} = \mathcal{L}$$

Abstract Values

$$\mathcal{V}^{\#} : \mathcal{V}^{\#} \times \mathcal{L}^{\#} \equiv \mathcal{L}$$

Abstract Addresses

$$\mathcal{A}^{\#} = \mathcal{A}$$

Abstract Memories

$$\mathcal{M}^{\#} : \text{var} \rightarrow (\mathcal{L} \cup \mathcal{A})$$

Abstract Stacks

$$\mathcal{S}^{\#} : (\mathcal{L} \cup \mathcal{A})^*$$

Abstract Environments

$$\mathcal{E}^{\#} = \mathcal{E} = \mathcal{L}$$

Abstract States:

$$\mathcal{L} \times \mathcal{A} \times \mathcal{M}^{\#} \times \mathcal{S}^{\#} \times (\mathcal{A} \cup \{ 0 \})$$

# Abstract operational semantics

the abstract semantics:

- abstracts concrete values into their security level:

$$\alpha(k, \sigma) = \sigma$$

- uses the same rules of the concrete semantics on the abstract domains

Both rules for if are always applied -  $\text{if}_{\text{true}}$   
 $\text{if}_{\text{false}}$

$A(P)$  : abstract transition system for  $P$

- finite
- multiple path
- each path of  $C(P)$  is correctly abstracted onto a path of  $A(P)$

# Results

## Theorem 1

A program  $P$  satisfies SIF if for each state of  $A(P)$  such that  $P[PC] = \text{halt}$ , then for each  $x : L$  it is:

$M[x] = L$

or

$M[x] = (i, L)$  for some  $i$ .

# Results

## Theorem 2

A program  $P$  satisfies TERM if each state of  $A(P)$

$\langle \sigma, PC, M, S, IPD \rangle$  such that  $\sigma = H$

does not belong to a cycle.

# Results

## Theorem 3

A program  $P$  satisfies TIME if:

- all paths in  $A(P)$  starting from a state satisfying  $TOP(S)=H$  and  $P[PC] = \text{if}$  and ending with a state satisfying  $PC=\text{ipd}(i)$  **have the same length.**

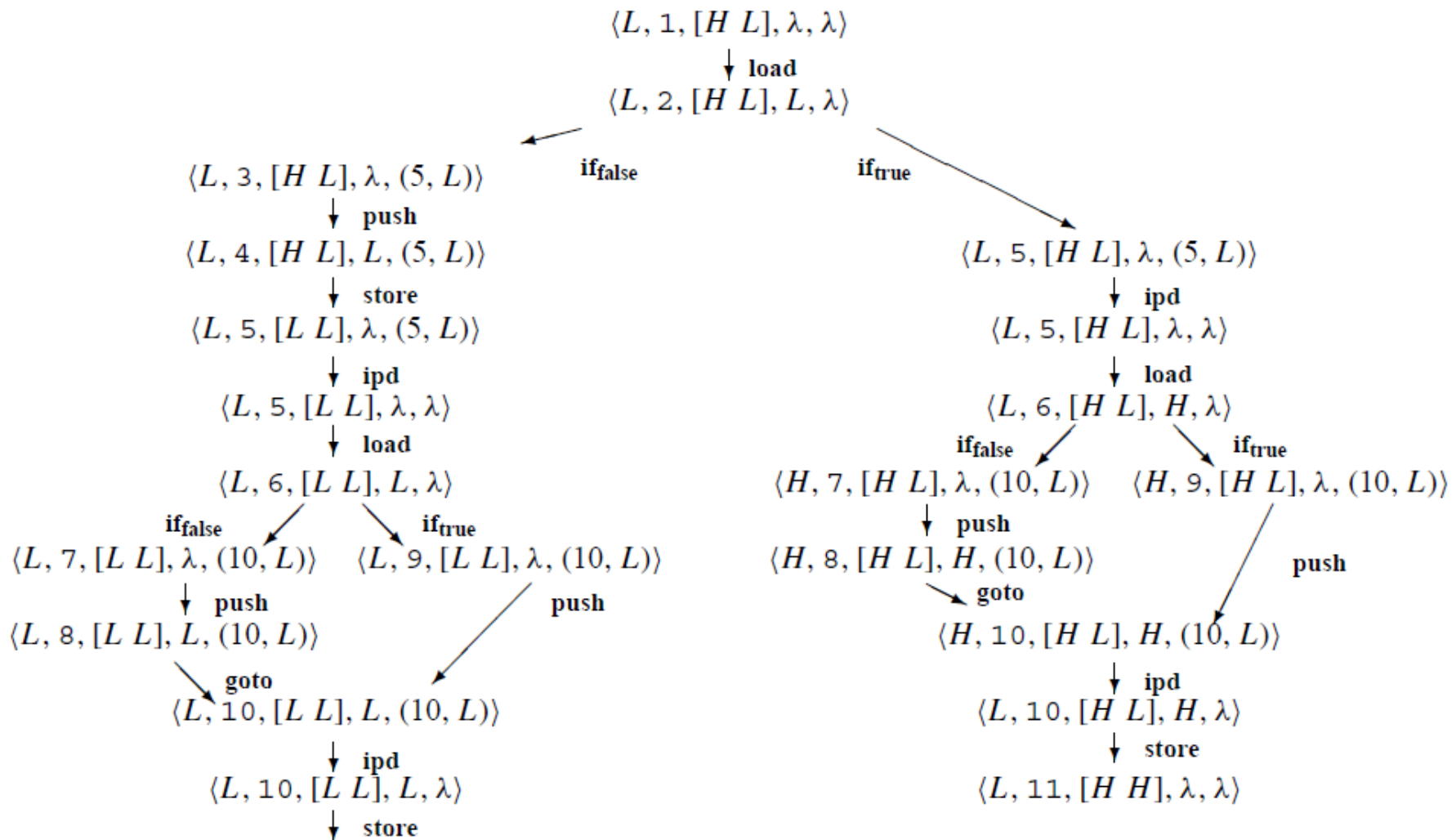
# Another example: concrete semantics

$x:(0,H)$   $y:(1,L)$

$\text{ipd}(2) = 5, \text{ipd}(6)=10$

1	load y	↓ load	$\langle L, 1, [(0, H)(1, L)], \lambda, \lambda \rangle$
2	if 5		$\langle L, 2, [(0, H)(1, L)], (1, L), \lambda \rangle$
3	push 3	↓ if <sub>true</sub>	
4	store x		$\langle L, 5, [(0, H)(1, L)], \lambda, (5, L) \rangle$
5	load x	↓ ipd	
6	if 9		$\langle L, 5, [(0, H)(1, L)], \lambda, \lambda \rangle$
7	push 1	↓ load	
8	goto 10		$\langle L, 6, [(0, H)(1, L)], (0, H), \lambda \rangle$
9	push 0	↓ if <sub>false</sub>	
10	store y		$\langle H, 7, [(0, H)(1, L)], \lambda, (10, L) \rangle$
11	halt	↓ push	
			$\langle H, 8, [(0, H)(1, L)], (1, H), (10, L) \rangle$
		↓ goto	
			$\langle H, 10, [(0, H)(1, L)], (1, H), (10, L) \rangle$
		↓ ipd	
			$\langle L, 10, [(0, H)(1, L)], (1, H), \lambda \rangle$
		↓ store	
			$\langle L, 11, [(0, H)(1, H)], \lambda, \lambda \rangle$

# An example: abstract semantics



Abstract semantics for  $M_0^h(x) = H$  and  $M_0^h(y) = L$ .

# Works

R. Barbuti, C. Bernardeschi, N. De Francesco.  
Abstract Interpretation of Operational Semantics for Secure Information Flow.  
Inf. Process. Lett. 83(2): 101-108 (2002)

R. Barbuti, C. Bernardeschi, N. De Francesco.  
Checking Security of Java Bytecode by Abstract Interpretation.  
Special Track on Security at the ACM Symposium  
on Applied Computing (SAC2002), March 10-14, Spain 2002.

Since the model is based on operational semantics, it is fully automatic.  
Moreover it can be integrated with model checking

Cinzia Bernardeschi, Nicoletta De Francesco:  
Combining Abstract Interpretation and Model Checking for Analysing  
Security Properties of Java Bytecode. VMCAI 2002: 1-15

# Method invocation and shared objects: the security context

Data propagation caused by any method invocation and the access to common data structures in the heap is studied by executing each method inside a security context

For each variable: the highest level of data stored

*var*:  $\sigma$

For each method: the highest level of input/output parameters, return and the security environment.

*mt*( $\sigma_1, \dots, \sigma_n$ ):  $\sigma, \sigma'$

SECURITY CONTEXT

classfields, arrays

A.f1 = L

B.f2 = H

methods of class A

methods of class B

mt1(L, H) L; L

mt2(L) L; L

mt3(L) L; L

A is a class with data member f1 and method member mt1; B a class with data member f2 and method member mt2 and mt3.

# Secure information flow analysis

The secure information flow analysis corresponds to an iterative verification of all methods within a common security context: it stops when a fixpoint is reached.

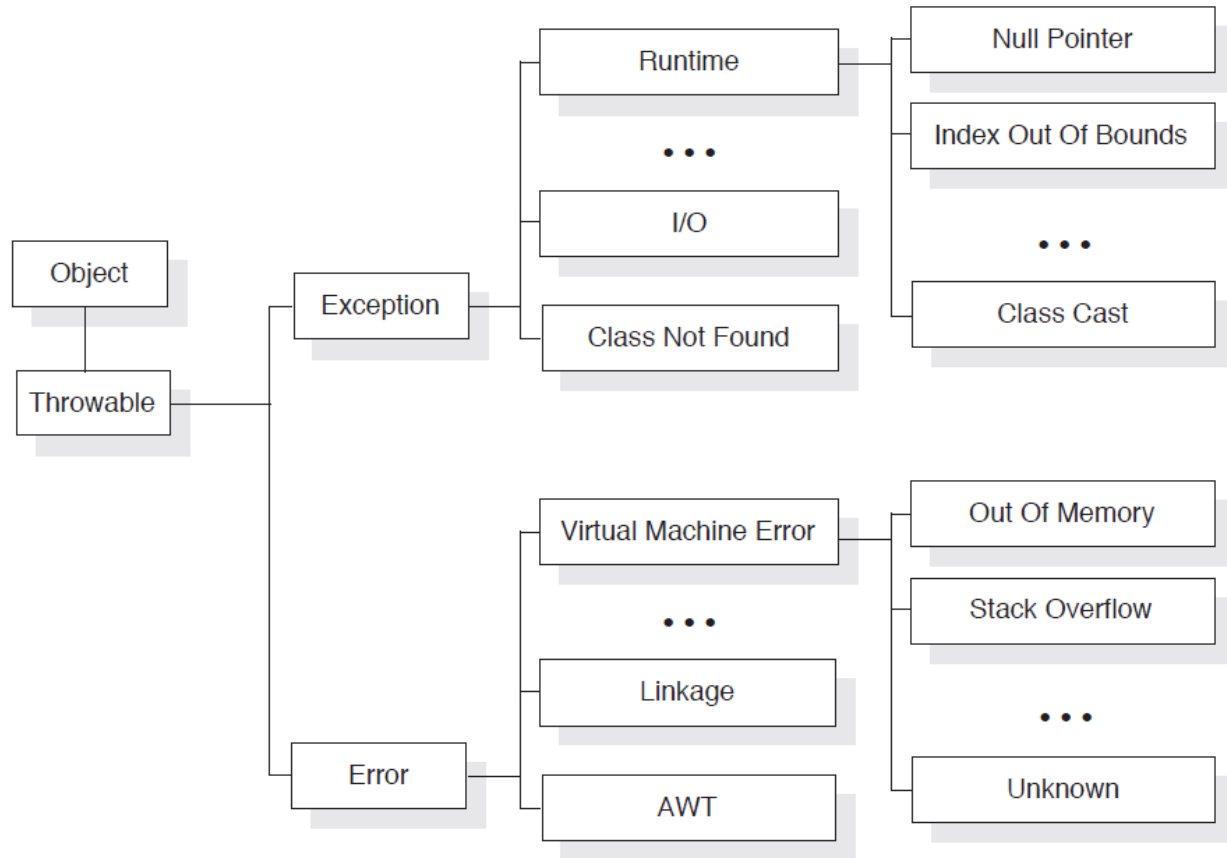
Let EXEC be the abstract interpreter of a method

Let  $C^0$  be the initial context

Let M be the set of methods

```
 $C := C^0, T := M$   
while ( $T \neq 0$ )  
    select  $mt$  in  $T$   
     $T := T - mt$   
     $C' := EXEC(mt, C)$   
    if ( $C' \neq C$ )  
         $C := C'$   
         $T := M$ 
```

# Information flow through exceptions



Java Exception Hierarchy (incomplete)

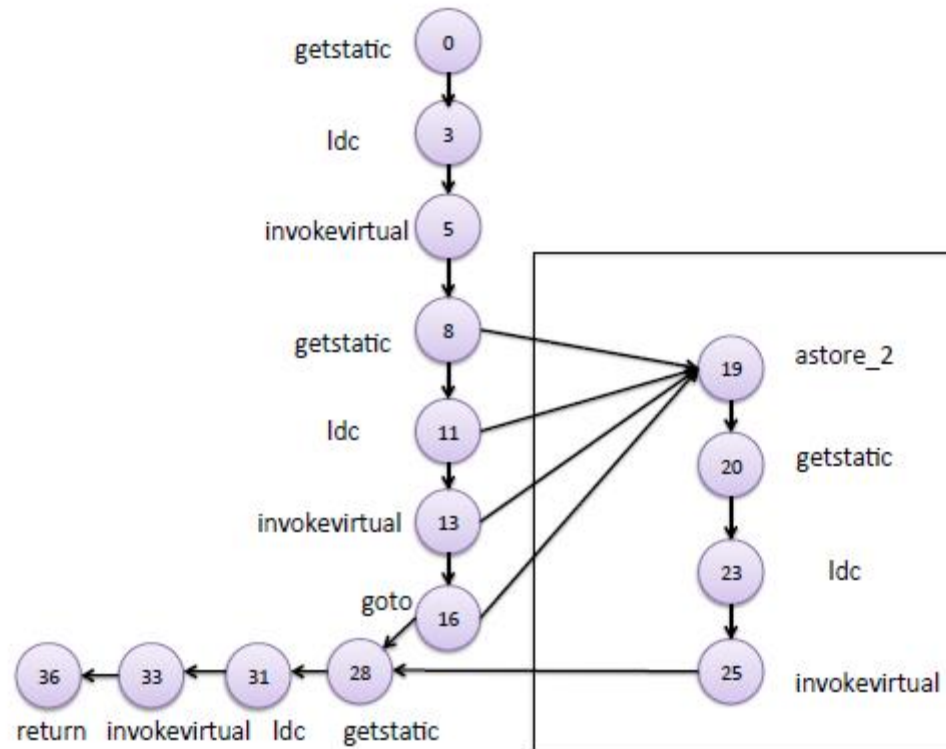
# Java code

```
1
2 import java.io.IOException;
3 public class Hello {
4     public static void main(String[] args) {
5     }
6
7     public void stampa(String s){
8         System.out.println("Outside");
9
10        try { System.out.println("Inside try");}
11        catch (ArithmeticException e) {System.out.println("Print catch");}
12
13        System.out.println("Print ...");
14    }
15 }
```

# The bytecode

```
1 Compiled from "Hello.java"
2 public class Hello {
3     public Hello();
4     Code:
5         0: aload_0
6         1: invokespecial #1 //Method java/lang/Object."<init>":()V
7         4: return
8
9     public static void main(java.lang.String []);
10    Code:
11        0: return
12
13    public void stampa(java.lang.String);
14    Code:
15        0: getstatic #2 //Field java/lang/System.out:Ljava/io/PrintStream;
16        3: ldc      #3 //String Outside
17        5: invokevirtual #4 //Method java/io/PrintStream.println:(Ljava/lang/String;)V
18        8: getstatic #2 //Field java/lang/System.out:Ljava/io/PrintStream;
19       11: ldc      #5 //String Inside try
20       13: invokevirtual #4 //Method java/io/PrintStream.println:(Ljava/lang/String;)V
21       16: goto     28
22       19: astore_2
23       20: getstatic #2 //Field java/lang/System.out:Ljava/io/PrintStream;
24       23: ldc      #7 //String Print catch
25       25: invokevirtual #4 //Method java/io/PrintStream.println:(Ljava/lang/String;)V
26       28: getstatic #2 //Field java/lang/System.out:Ljava/io/PrintStream;
27       31: ldc      #8 //String Print ...
28       33: invokevirtual #4 //Method java/io/PrintStream.println:(Ljava/lang/String;)V
29       36: return
30
31    Exception table:
32        from    to  target type
33         8      16    19    Class java/lang/ArithmeticException
34 }
```

# Extended control flow graph



Extended control flow graph of the bytecode

# PinCloner applet

Pin\_file and Clone\_file are the input and the output files

Pin\_file is a private file containing a secret PIN (a sequence of 0/1 characters, for simplicity)

Let us suppose that the PINcloner application can read from the private file. After every character has read, it will be written in a the public file by the handler of the exception.

PINCloner application clones the characters of the Pin\_file by throwing different kind of exceptions depending on the value read (NullPointerException and ArithmeticException).

The exceptions are thrown directly with a throw statement

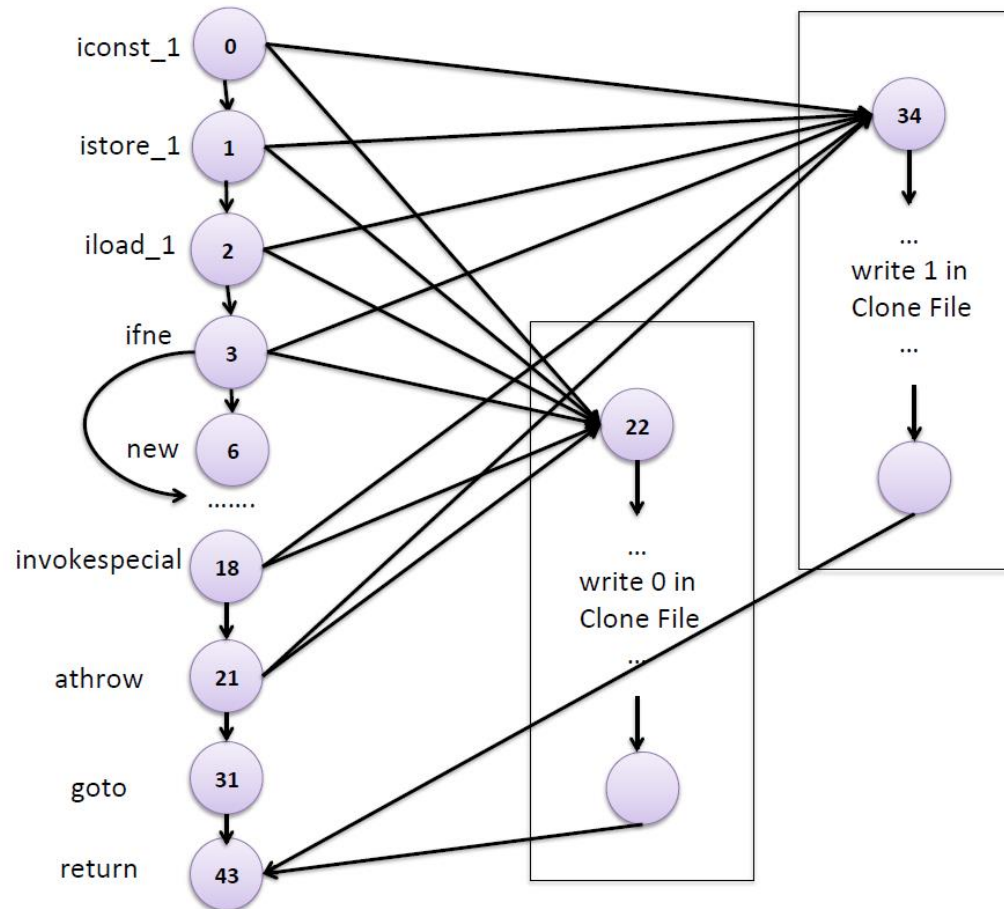
# PinCloner Java code

```
1 import java.io.IOException;
2
3 public class PinCloner {
4     public static void main(String[] args) {
5     }
6
7     public void PinCloner() {
8         try {
9             int p;
10             for (...) { // reads a char from PIN_FILE and store it in p;
11                 .....
12                 if (p == 0) { throw new ArithmeticException(); }
13                 else { throw new NullPointerException(); }
14             }
15         }
16         catch (ArithmeticException e) { // write 0 in Clone_File
17         }
18         catch (NullPointerException e) { // write 1 in Clone_File
19         }
20         .....
21     }
22 }
```

# PinCloner Java bytecode

```
1
2 Compiled from "PinCloner.java"
3 public class PinCloner {
4     public PinCloner();
5     Code:
6         0: aload_0
7         1: invokespecial #1 // Method java/lang/Object."<init>":()V
8         4: return
9
10    public static void main(java.lang.String []);
11    Code:
12        0: return
13
14    public void PinCloner();
15    Code:
16        0: iconst_1
17        1: istore_1
18        2: iload_1
19        3: ifne          14
20        6: new           #2 // class java/lang/ArithmeticException
21        9: dup
22       10: invokespecial #3 // Method java/lang/ArithmeticException."<init>":()V
23       13: athrow
24       14: new           #4 // class java/lang/NullPointerException
25       17: dup
26       18: invokespecial #5 // Method java/lang/NullPointerException."<init>":()V
27       21: athrow
28       22: .....        // write 1 in Clone_File
29       31: goto          43
30       34: ...           // write 1 in Clone_File
31       43: return
32
33    Exception table:
34        from    to    target type
35         0      22    22    Class java/lang/ArithmeticException
36         0      22    34    Class java/lang/NullPointerException
37 }
```

# Extended control flow graph



control region of instruction i: set of instructions under the implicit flow of instruction i  
(set of instruction on any path from i to ipd(i), ipd(i) excluded)

# PinCloner applet

The control region of 3 includes the instructions of the exception handlers, and consequently these instructions are executed in a security environment given by the condition of the ifne.

Since the condition depends on the 0/1 value of PIN character read from a high security file, the implicit flow is high.

The handler of the exception, write such value into the low security Clone\_ file.

The application violates the secure information flow because high security data are written on a public file and our methodology successfully detects this data leakage.

Data Leakage in Java applets with Exception Mechanism  
Cinzia Bernardeschi, Paolo Masci, and Antonella Santone  
ITASEC 2018, Febbraio 2018.

# Case studies

# Case studies

Java cards:

Secure interactions in Java cards

Automotive:

Data secure flow in AUTOSAR models

Marco Avvenuti, Cinzia Bernardeschi, Nicoletta De Francesco, Paolo Masci:  
JCSI: A tool for checking secure information flow in Java Card applications. *Journal of Systems and Software* 85(11): 2479-2493 (2012)

Cinzia Bernardeschi, Marco Di Natale, Gianluca Dini, Maurizio Palmieri:  
Verifying Data Secure Flow in AUTOSAR Models by Static Analysis. *ICISSP 2017*: 704-713

# Java cards

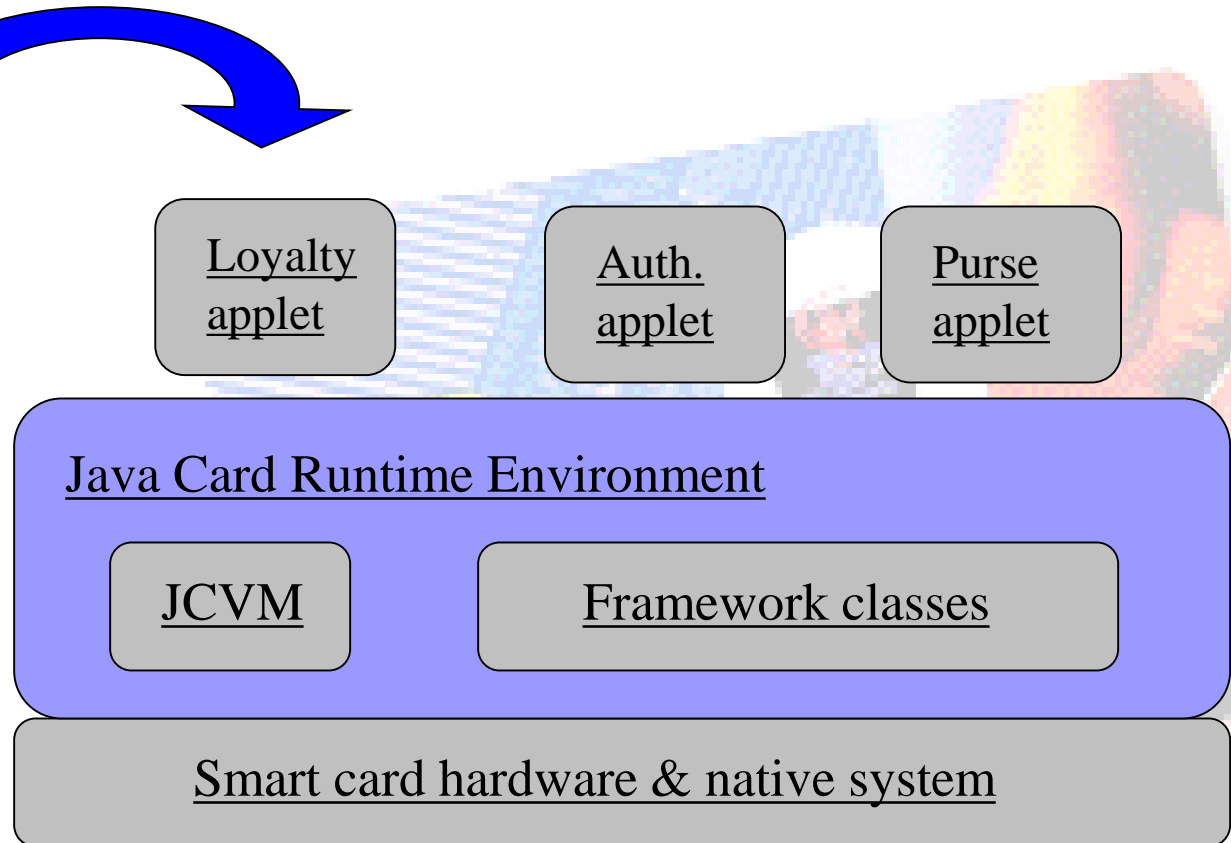
- Smart cards: embedded systems that allow to store and process information
- Typical Applications: Credit cards, Electronic cash, Loyalty systems, Healthcare, Government identification ....
- Java cards:
  - Java Virtual machine / applications (applets) are portable
  - Multiapplicative Java cards: applets can be downloaded and installed on card after the card issuance
  - Applet's sensitive data must be protected against unauthorised accesses

# Java cards



Card reader

## Multiapplicative Java cards



# Java cards security

- Security in Java cards is a combination of the security mechanisms in Java and additional security procedures imposed by the card platform

## **FIREWALL**

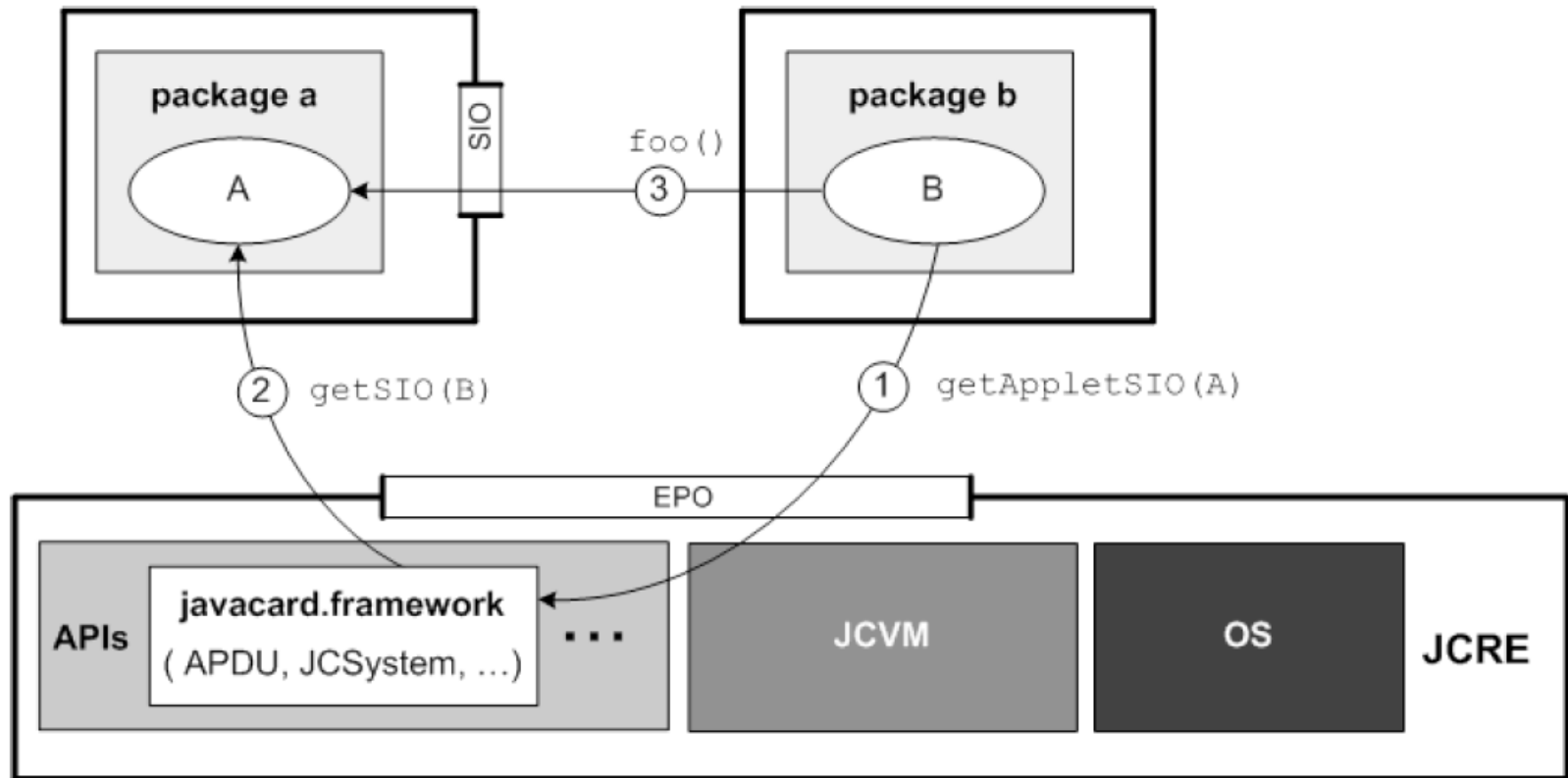
ATOMICITY and TRANSACTIONS

PERSISTENT and TRANSIENT objects

JAVA security mechanisms

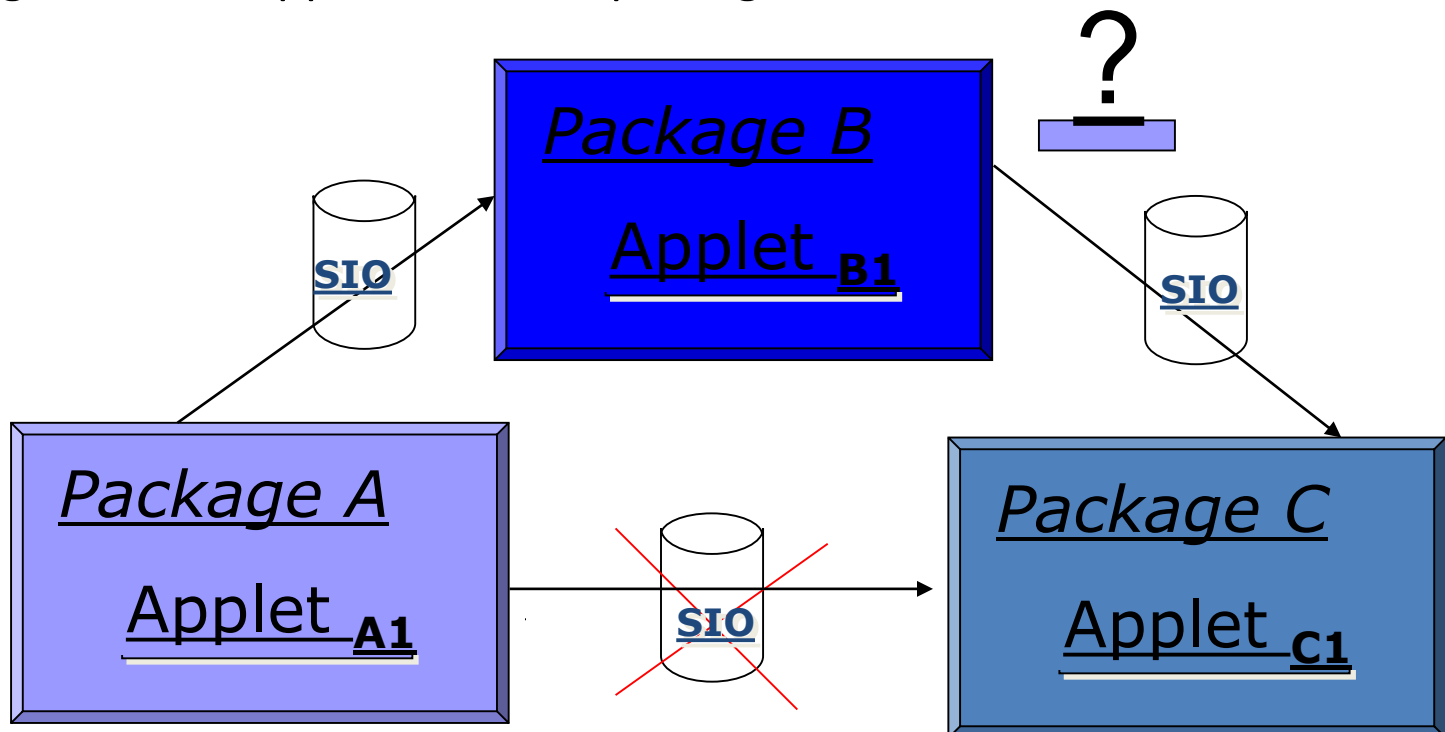
- The Firewall forces the isolation between objects of applets belonging to different packages

# Communication between packages



# Limits of the firewall

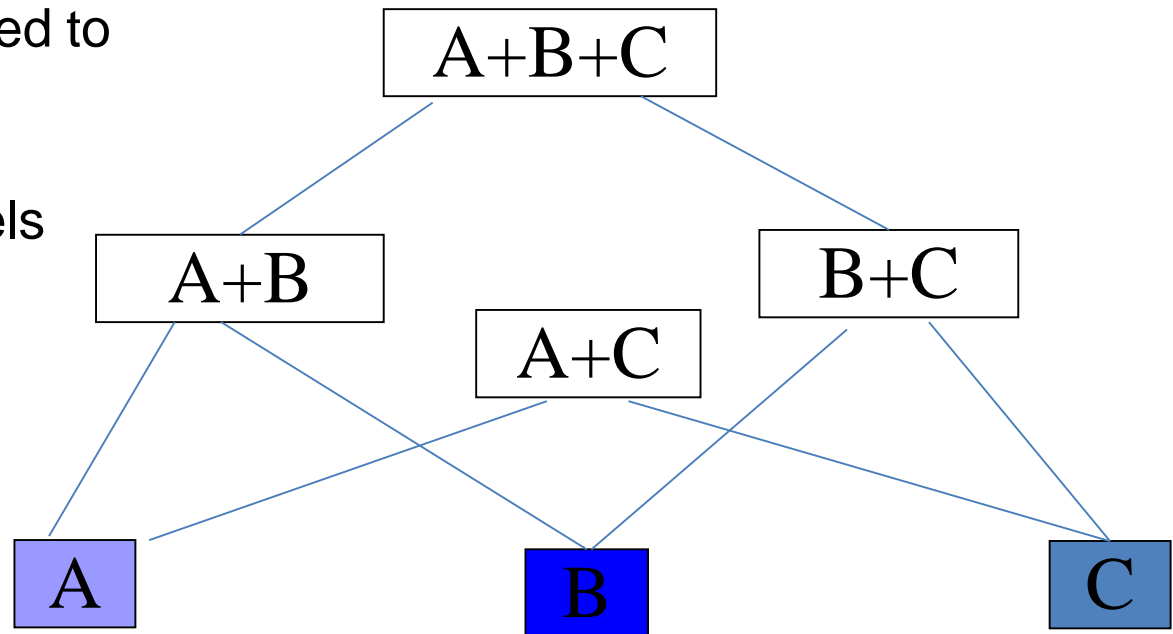
- Based on access control checks
- Place restrictions on the applets that can access to methods of applets belonging to other packages
- Does not control the propagation of the information from an applet of a package towards applets of other packages



# Secure Information Flow

- Security levels assigned to packages

- Lattice of security levels



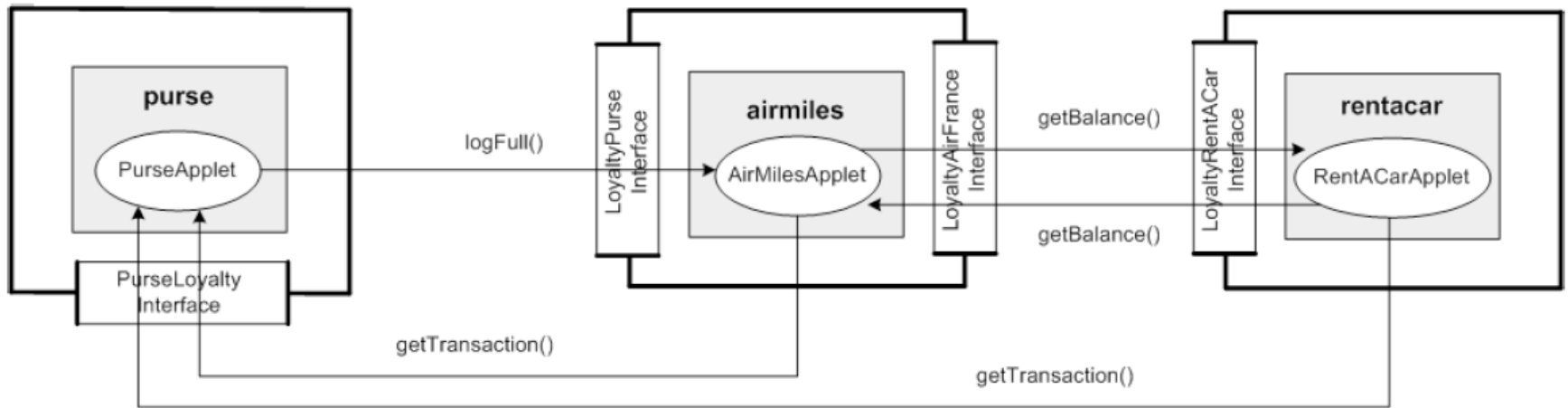
- Abstract Interpretation framework: abstract execution of the applets using security levels instead of real data
- Secure Information Flow: Check that information exchanged between A and B has a security equal to or level lower than A+B

# Java Card Information Flow Verifier

JCIFV performs the analysis according to the following main steps

1. Unique security levels are automatically assigned to packages and shareable interface objects. An initial security level is assigned to the other methods and object fields
2. CAP file (native code of an applet) is decoded and saved as a bytecode
3. Abstract interpretation of the bytecode is performed
4. The analysis stops when the state of the abstract interpreter does not longer change and all methods have been analyzed
5. Secure information flow is checked

# Electronic Purse



illicit information flow from Purse to RentACar caused by a method invocation (no parameters) from AirMiles and RentAcar

Purse: log-full service (**logFull()**), which notifies registered applets that the transaction log is going to be over-written.

Airmail: registered for the log-full service

RentACar: not registered for the log-full service

# Electronic Purse

Assume that AirFrance requests RentACar the amount of miles (`getBalance()`) every time Purse notifies AirFrance that the transaction log is full.

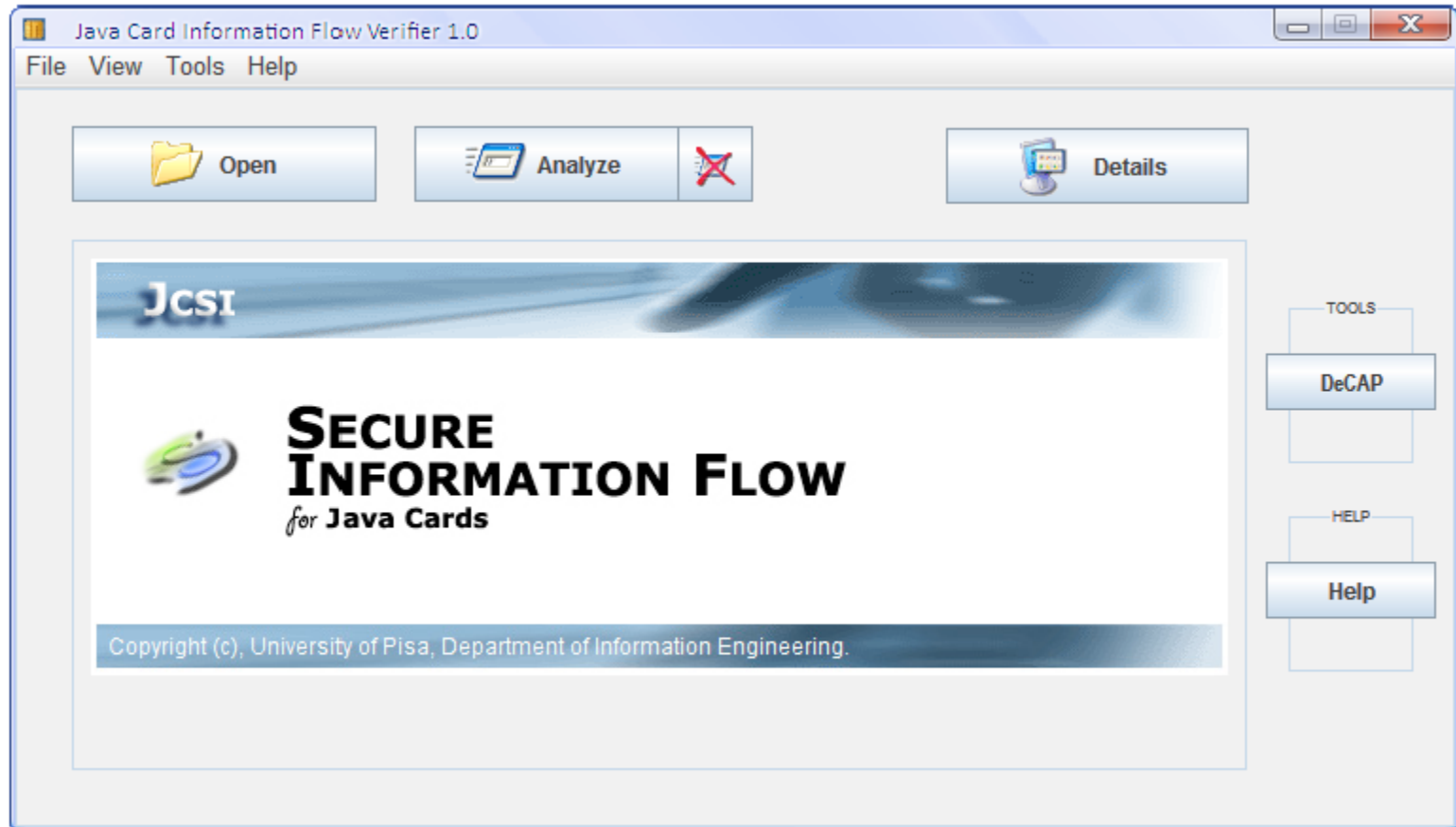
`logFull()` method implemented by AirFrance contains an invocation of method `getTransaction()` of Purse followed by an invocation of method `getBalance()` of RentACar.

Applet RentaACar, whenever observes an invocation of `getBalance()`, can infer that Purse is going to over-write the transaction log.

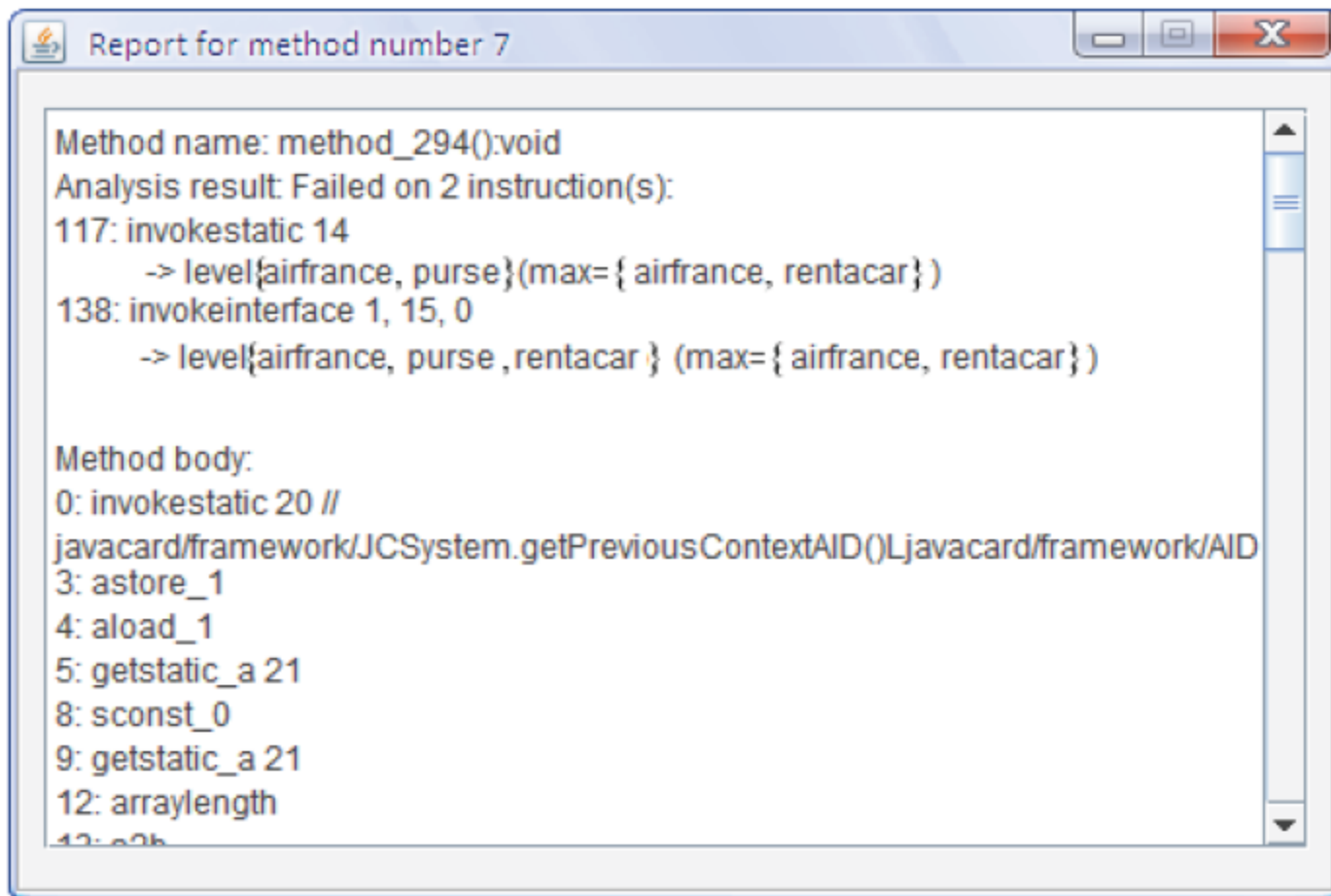
Thus, even without subscribing to the log-full service, RentACar is able to benefit from such a service.

Purse is not able to detect such information flow.

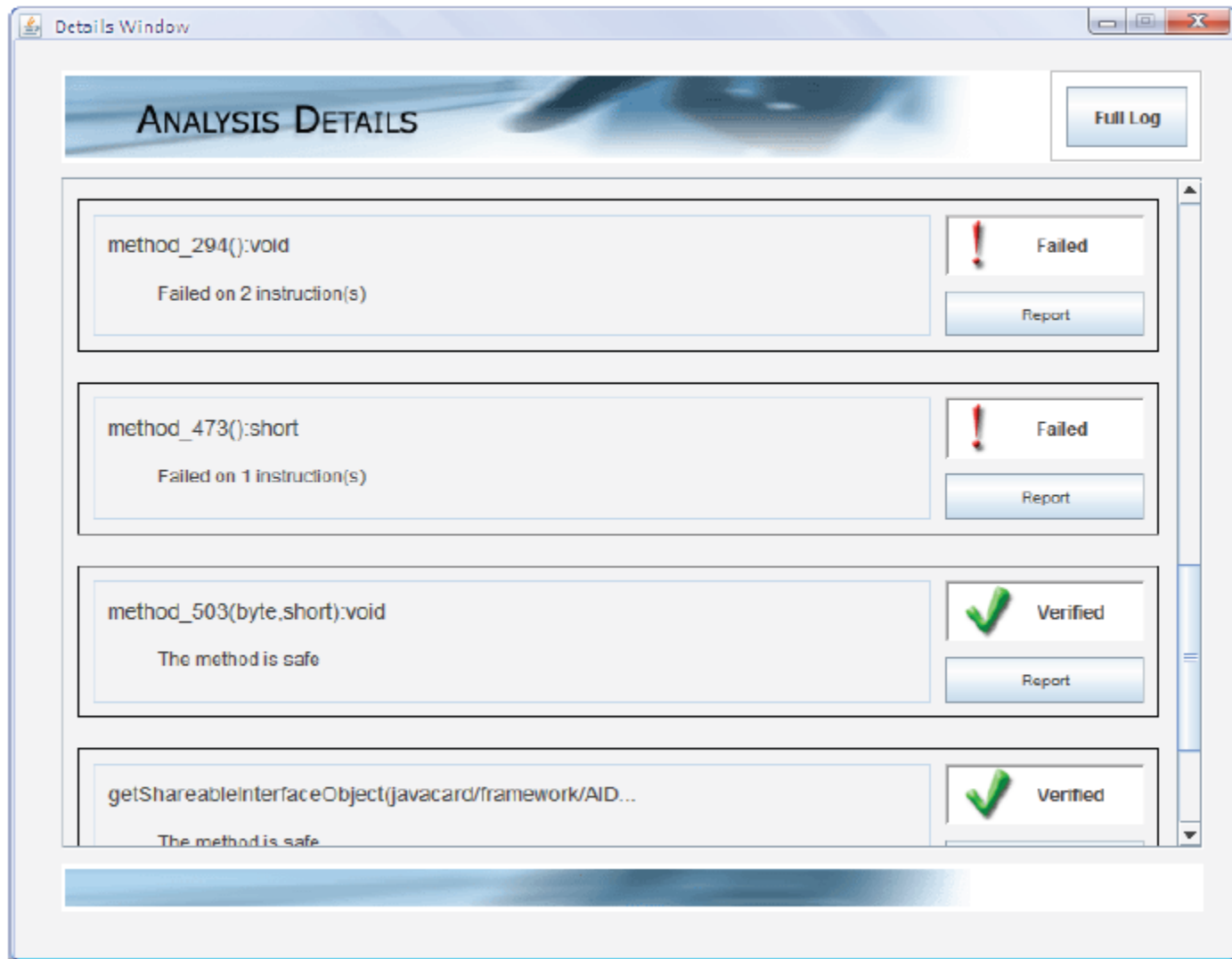
# The tool



# Report



# Analysis



# Discussion

- JCIFV tool is able to certify applets against secure information flow of sensitive data saved present on the card
- Verification (Abstract Interpretation)
  - JVIFV certifies only secure applets
  - some correct applets could also be rejected
    - due to the characteristics of the applet code the percentage of erroneously rejected applets is very low

# Automotive

Modern automotive electronics systems are real-time embedded system running over networked Electronic Control Units (ECUs) interconnected by wired networks such as the Controller Area Network (CAN) or Ethernet.

Automotive systems: Mixed-criticality safety critical systems

Braking system, Throttle system, ...  
Infotainment system

Recent research has shown that it is possible for external intruders to compromise the proper operation of safety functions getting access to the infotainment system.

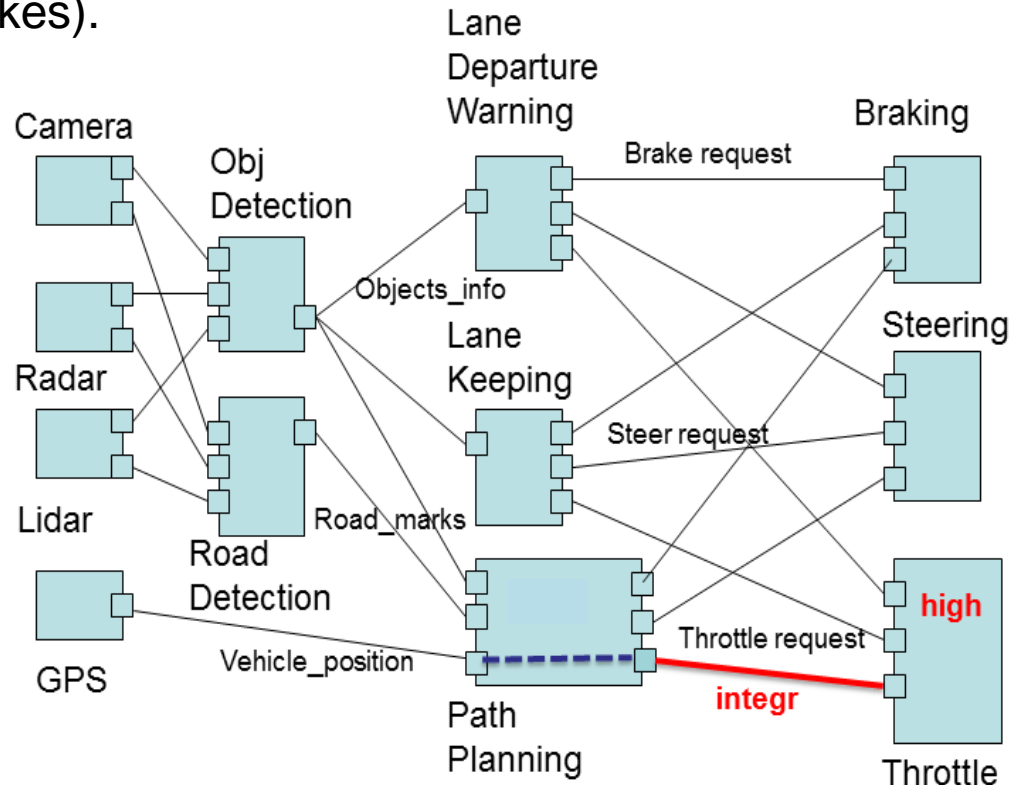
**Low security level data must not compromise the computation of high criticality functions**

# Automotive

Path Planning, Lane Keeping and Lane Departure Warning are active safety functions that receive such data and send commands to actuators (steering, throttle and brakes).

Autonomous driving

Mixed-criticality safety critical systems



AUTOSAR models are extended with security annotations. In the example,

- Throttle component is assigned the high trust level;
- Throttle request link is assigned the integrity security requirement.

# Automotive

Data received by Throttle on the link Throttle\_request must satisfy high trust level and integrity security requirement

The point is that: the way in which security annotations are specified must consider the causal dependencies between data that traverse the model.

If Throttle requires integrity on its input data sent by Path Planning, then integrity must be guaranteed also along the path from the data originator (GPS) to Path Planning (the Vehicle\_position link), otherwise, the security constraint cannot be satisfied and the set of annotations is not correct.

Similarly, Path Planning and GPS must have high trust level.

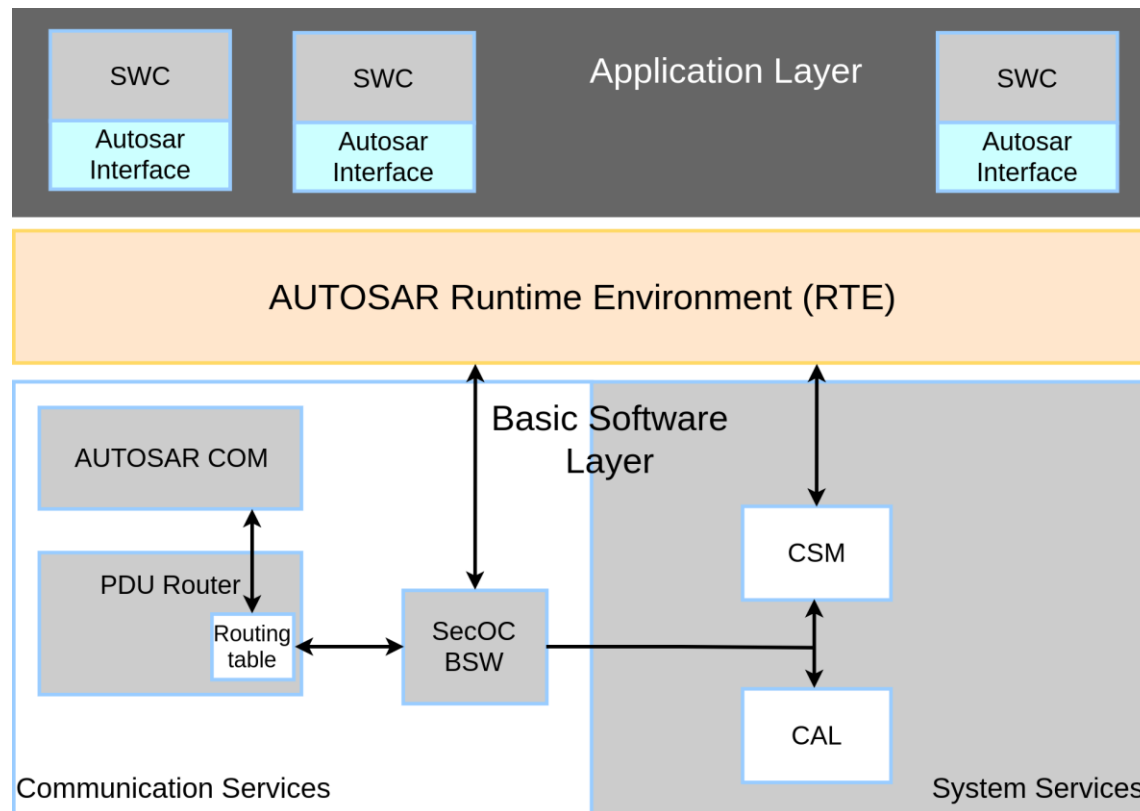
The simplest solution assigns integrity/high to all links/components directly or indirectly connected to Throttle/Throttle\_request.



In order to obtain a more efficient solution, **information flow** theory can be exploited to compute the dependency between data

# AUTOSAR

**AUT**omotive **O**pen **S**ystems **AR**chitecture: open industry standard for automotive software architectures, spanning all levels, from device drivers, to operating system, communication abstraction layers and the specification of application-level components

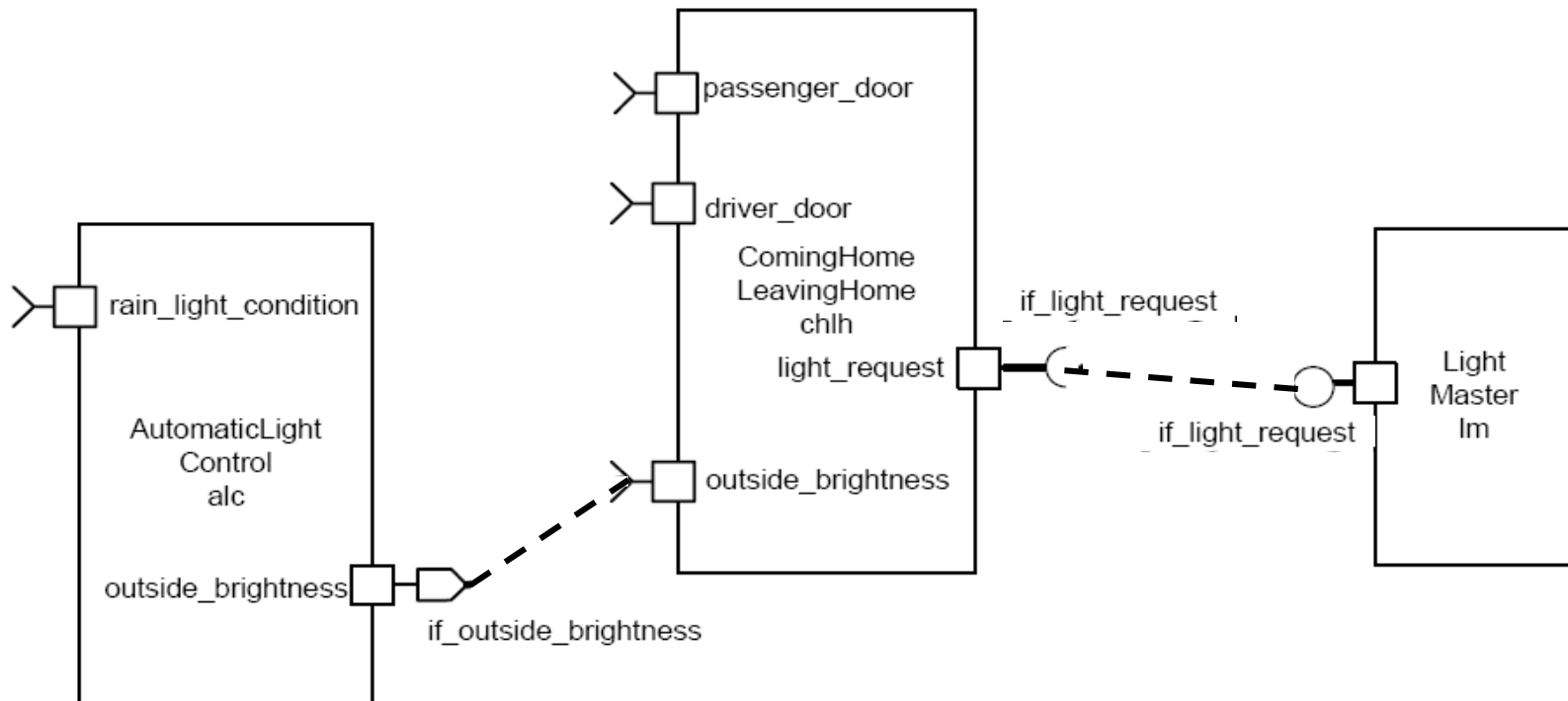


# AUTOSAR architecture

A fundamental concept of AUTOSAR is the separation between:

- **application** and
- **infrastructure**.

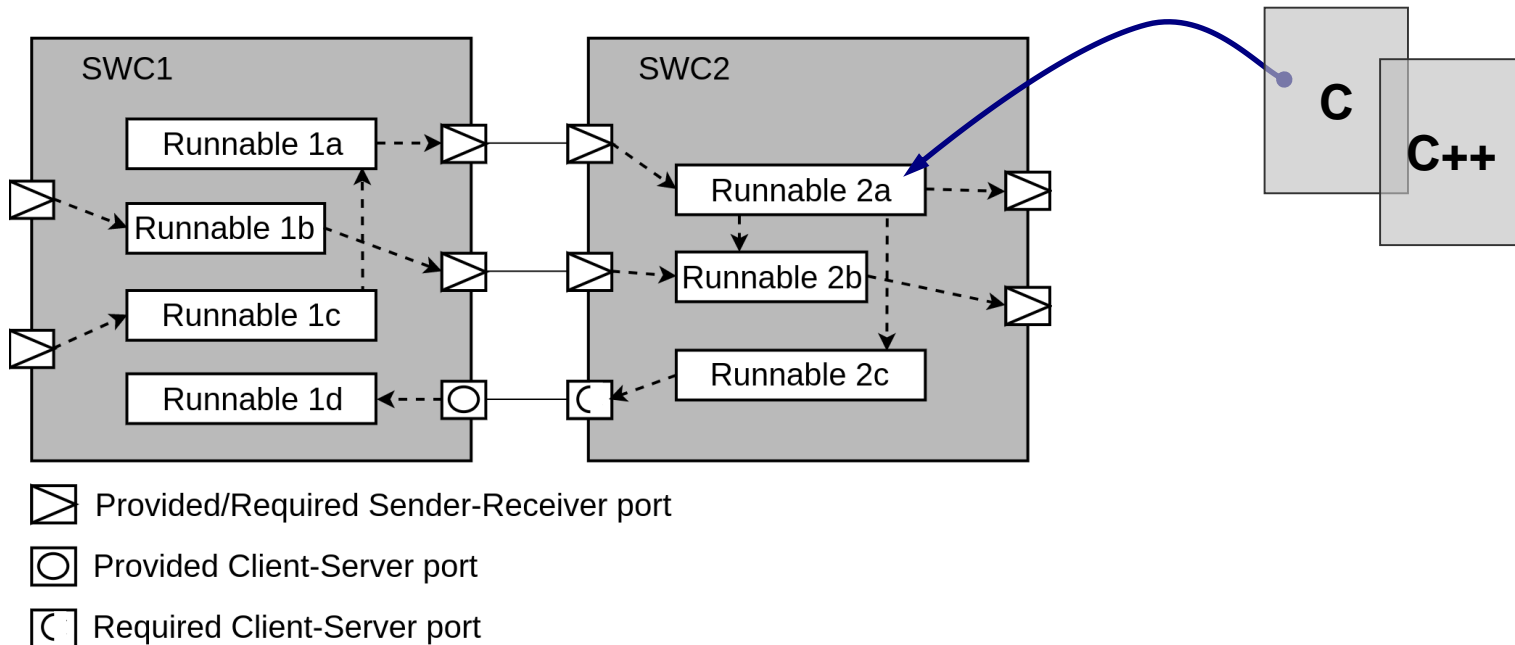
An application in AUTOSAR consists of Software Components interconnected by connectors



# Runnables

**Runnables** define the behavior of components

- Runnables are entry points to code-fragments and are (indirectly) a subject for scheduling by the operating system.



# AUTOSAR runnable interaction

## Runnable interaction

### Global variables

**Ports** define interaction points between (runnables belonging to) different SWCs.

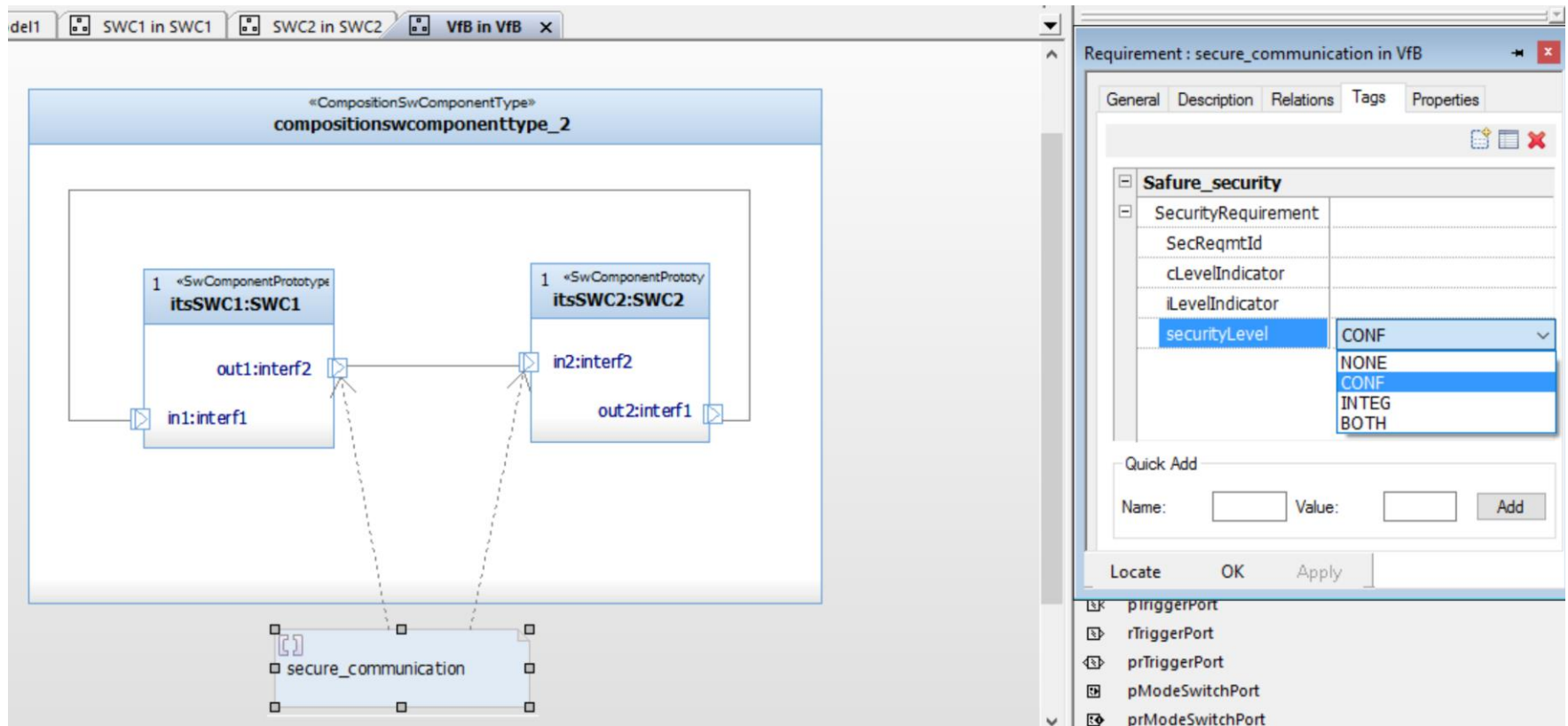
For interactions among runnables belonging to the same component  
**Inter Runnable Variables** (IRVs)

The RTE provides protection mechanisms for IRVs (as opposed to global variables)

# AUTOSAR security policy

- **Trust level** of a software component  
software components with high trust level are executed on secure and reliable hardware
  - we assume two trust levels: **high, low**
- **Security requirement** of a communication link  
the level of security that data sent on links must satisfy to protect in-vehicle communications from cyber threats such as eavesdropping, integrity and spoofing.  
The proposed security extensions are:  
confidentiality and integrity of the exchanged information
  - The security requirement can assume one of the following values:  
**none, conf, integr, both.**

# AUTOSAR extensions in Rhapsody

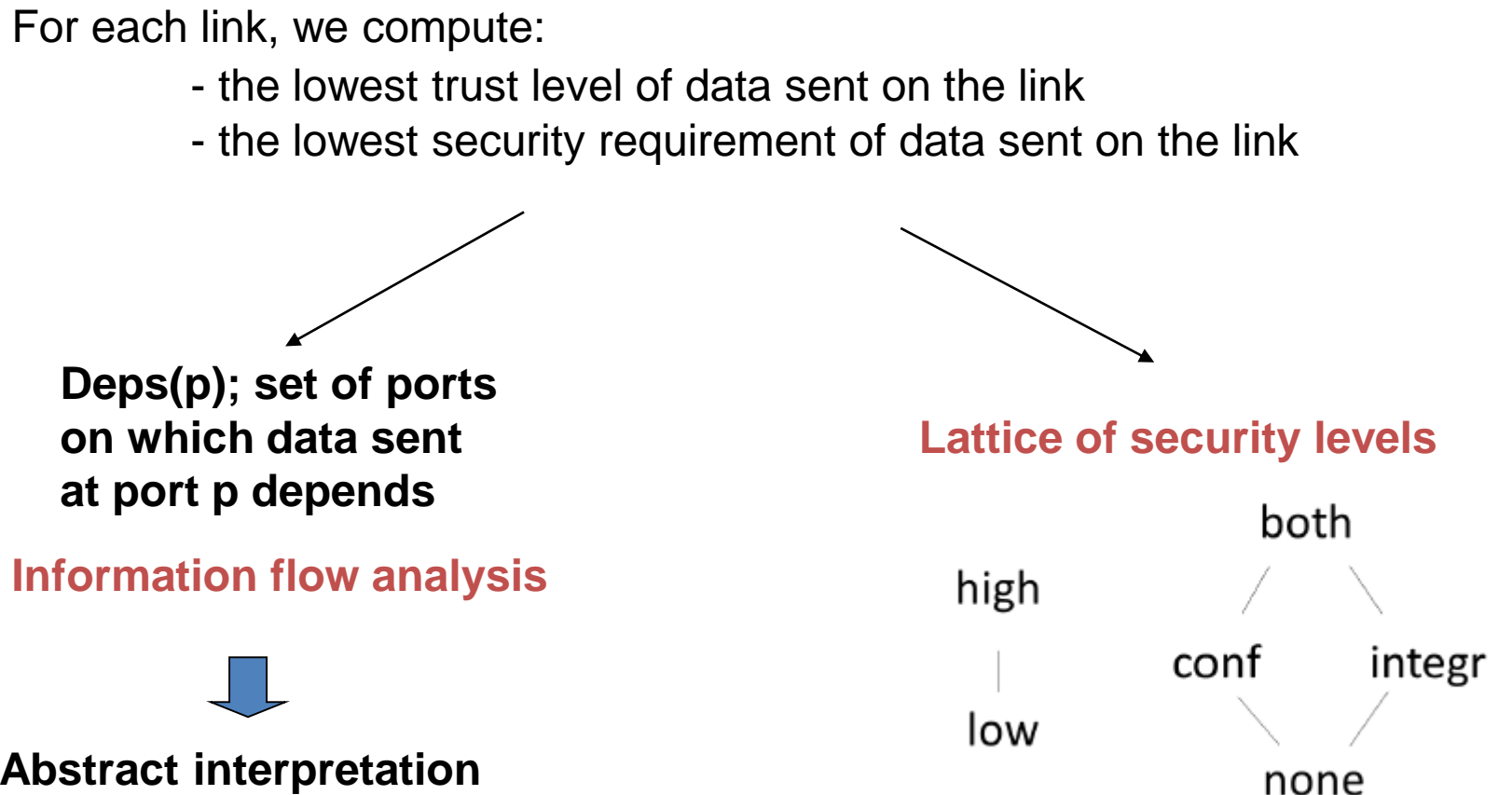


# AUTOSAR secure flow analysis

An AUTOSAR model satisfies data secure flow if data sent on a link at run-time, always have a security requirement and a trust level not lower than those specified by the security annotations.

For each link, we compute:

- the lowest trust level of data sent on the link
- the lowest security requirement of data sent on the link



**Deps(p); set of ports  
on which data sent  
at port p depends**

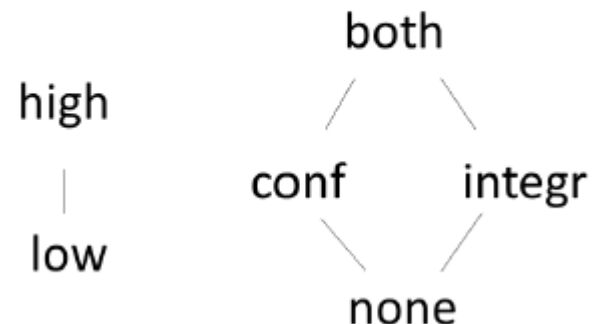
The diagram illustrates the process of AUTOSAR secure flow analysis. It starts with a link, from which two arrows point to 'Deps(p)' and the 'Lattice of security levels'. 'Deps(p)' leads to 'Information flow analysis', which then leads to 'Abstract interpretation' via a blue arrow. The 'Lattice of security levels' is a diagram showing a hierarchy of security levels: 'both' at the top, branching to 'high' and 'integr', 'high' branching to 'low' and 'conf', and 'conf' and 'integr' both leading to 'none' at the bottom.

**Information flow analysis**



**Abstract interpretation**

**Lattice of security levels**



# AUTOSAR secure flow analysis

Given a link  $l = (p_i, p_j) \in L$ ,

1.  $\langle \delta_l, \mu_l \rangle = \langle high, both \rangle$
2.  $\forall p \in Deps(p_i)$   
 $\delta_l = glb(\delta_l, trustlevel(cmp(p)))$
3.  $\forall l' = (q, q'), | q, q' \in Deps(p)$   
 $\mu_l = glb(\mu_l, securityrequirement(l'))$

**Definition 3** (Data secure flow property). *Given an AUTOSAR model with security annotations, the model satisfies the data secure flow property if, for each link  $l = (p_i, p_j) \in L$ , with  $\langle \delta_l, \mu_l \rangle$  the data security of data sent at  $l$ :*

$$\begin{aligned} & trustlevel(cmp(p_j)) \sqsubseteq \delta_l \\ & \wedge securityrequirement(l) \sqsubseteq \mu_l \end{aligned}$$

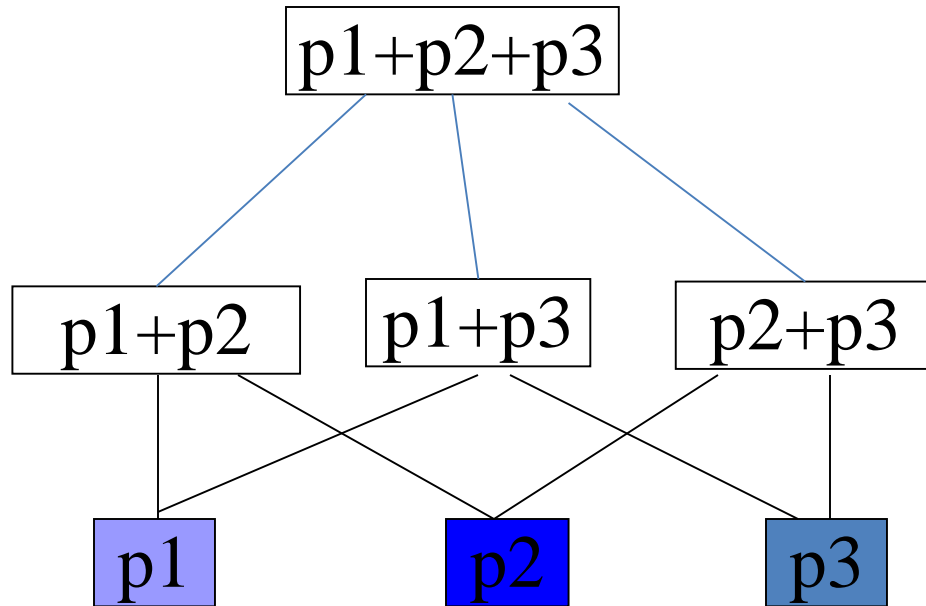
glb: greatest lower bound between levels

lub: least upper bound between levels

# Data flow analysis: $\text{Deps}(p)$

We have a level for each port

Lattice of levels



Abstract Interpretation framework: abstract execution of the runnables using port levels instead of real data

$\text{Deps}(p)$ : ports on which data sent at port  $p$  depends

# The abstract interpreter: EXEC

Each runnable is executed starting from the abstract memory and the context file, and applying the abstract rules.

All branches of conditional/iterative instructions are always executed, due to the loss of real data in the abstract semantics

# Abstract semantics

A POINTER is assumed to be simple variable, that maintains the dependencies of the pointer, plus the dependencies of the pointed data in the abstract execution.

An ARRAY is assumed to be a simple variable, that maintains the whole dependencies of each element in the array.

A STRUCTURED VARIABLE is mapped to a set of simple variables, one for each member (we use the notation, as usual). If we have a variable data that is a structure with two fields a and b, we map such variable into two simple variables, data:a and data:b, respectively.

RTE function for reading from or writing onto ports are mapped to read and write of the port variable.

For simplicity, the name of the port variable is equal to the name of the port.

RTE functions that invoke remote services trigger the runnable that implements the service. The function implementing the service is invoked.

# The context file

A context file is defined to record information stored in the shared memory of a SWC and information about runnable calls.

The context file maintains:

- for each variable  $v \in Var$ , the entry  $v : \sigma$ , where  $\sigma$  is the dependency level of data assigned to  $v$ ;
- for each runnable  $r \in R$ , the entry  $r(\sigma_1, \dots, \sigma_k)\sigma; \sigma'$ , where  $\sigma_1, \dots, \sigma_k$  are the levels for the actual parameter,  $\sigma$  is the level for the result and  $\sigma'$  is the level of the calling environment.

# The context file

In particular, during the analysis, for each variable, the context file maintains the maximum dependency level of data recorded in the variable.

For each runnable, the context file maintains how the runnable is called in terms of the maximum level of the calling environment, the actual parameters and return. We assume parameters and return of runnables, for generality.

# Analysis of an AUTOSAR model

Iterative analysis until the fixpoint is reached

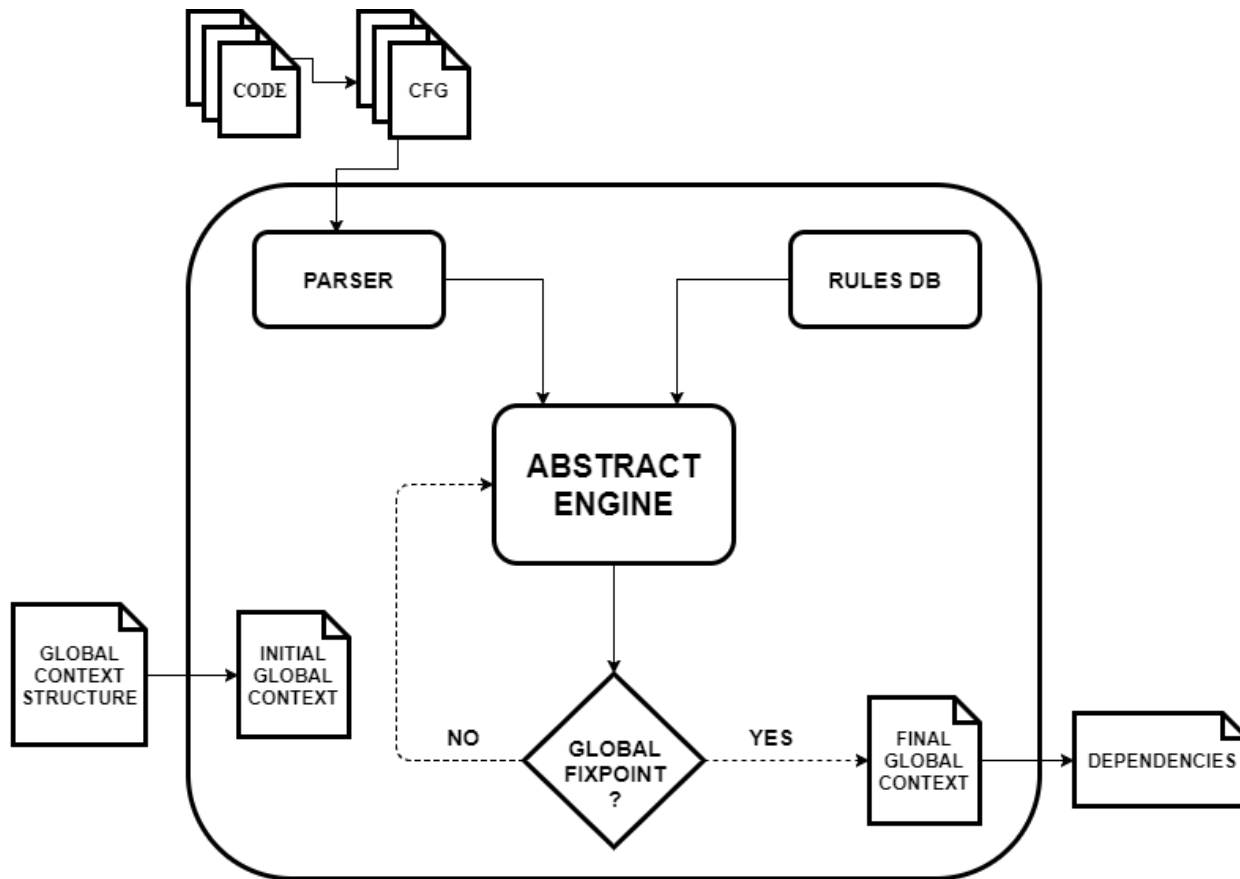
R: set of all runnables

A: security context

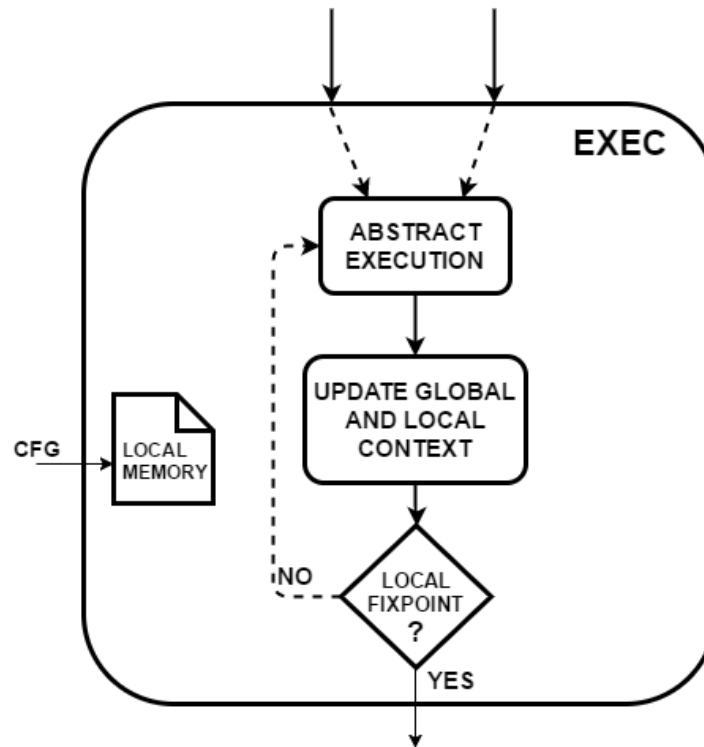
```
 $A := A^{\emptyset}$   
 $T := R$   
while( $T \neq \emptyset$ )  
  select  $r \in T$   
   $T := T - \{r\}$   
   $A' := EXEC(r, A)$   
  if( $A' \neq A$ )  
     $A := A'$   
     $T := R$ 
```

The analysis of Dets scales up, since runnables of AUTOSAR software components are analysed separately.

# Tool



# EXEC



# Generating models that satisfy data secure flow property using Deps

Let us consider a link  $L = (q, p)$ .

- Let  $A$  be the set of components that are the owner of ports in  $\text{Deps}(p)$ .  
Data sent on  $L$  depends only on the components in  $A$
- Let  $B$  be the set of links such that the source and the destination port of the link belong to  $\text{Deps}(p)$
- Data sent on  $L$  depends only on the links in  $B$

We use this information to assign the correct levels to components and links in the AUTOSAR model.

In the following,  $\text{level}(X)$  is the annotation assigned to  $X$  in the model.

# Generating models that satisfy data secure flow property using Deps

Given an AUTOSAR security annotated model, if the model does not satisfy data secure flow, we assign the correct levels as follows:

For each link  $L$ ,

let  $\langle T, S \rangle$  be the pair  $\langle \text{trust level}, \text{security requirement} \rangle$  that data sent on  $L$  must satisfy according to the annotations.

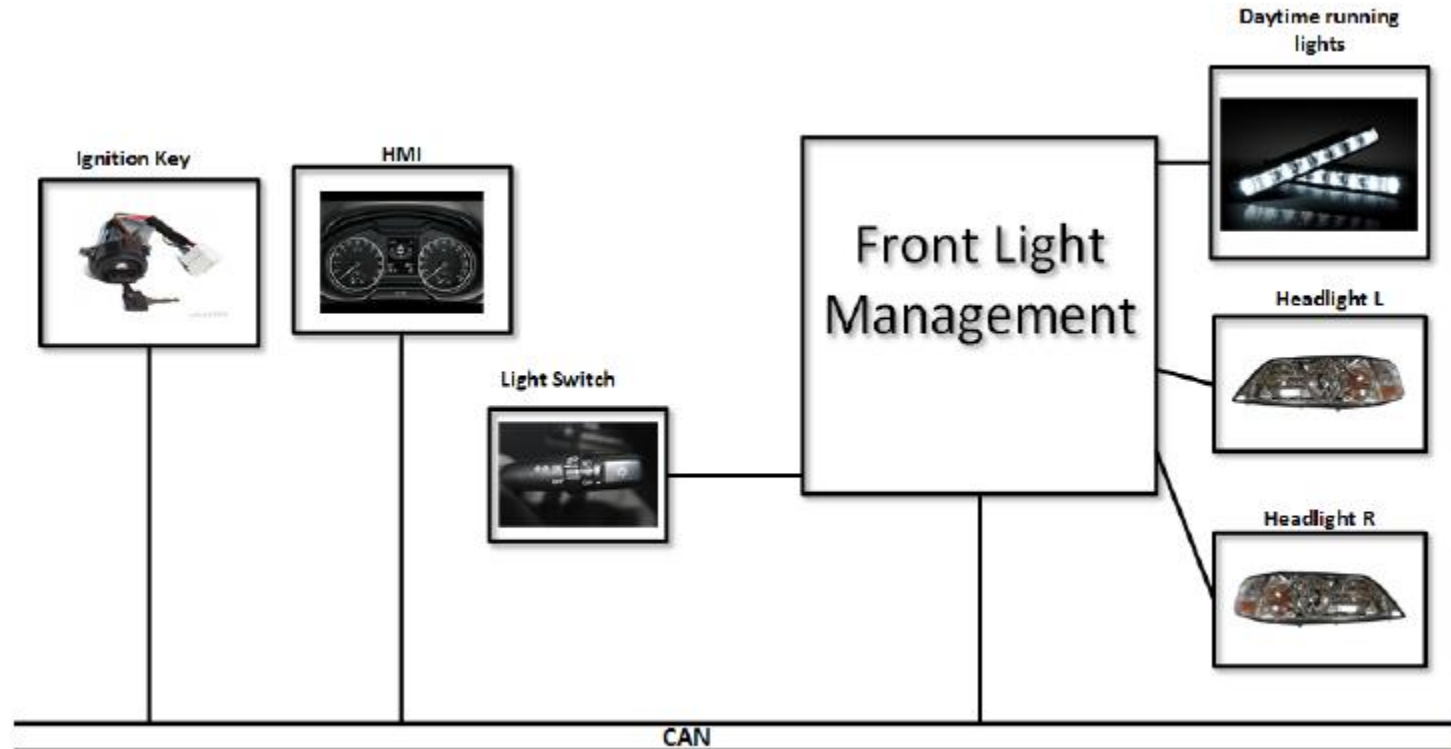
Let  $A$  and  $B$  be the set of components and the set of links on which data sent on  $L$  depend, computed using Deps.

For each component  $A1$  in  $A$ , the level of  $A1$  is set equal to  $\text{lub}(\text{level}(A1), T)$ , the component must have no lower trust level than  $T$ .

For each link  $B1$  in  $B$ , the level of the link is set equal to  $\text{lub}(\text{level}(B1), S)$ , the link must have no lower security requirement than  $S$ .

# An example: Front Light Manager

Front Light Manager use case described in the standard documents of AUTOSAR



Safety Use Case Example, release 4.2.2. [http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and security/auxiliary/AUTOSAR\\_EXP\\_SafetyUseCase.pdf](http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and security/auxiliary/AUTOSAR_EXP_SafetyUseCase.pdf)

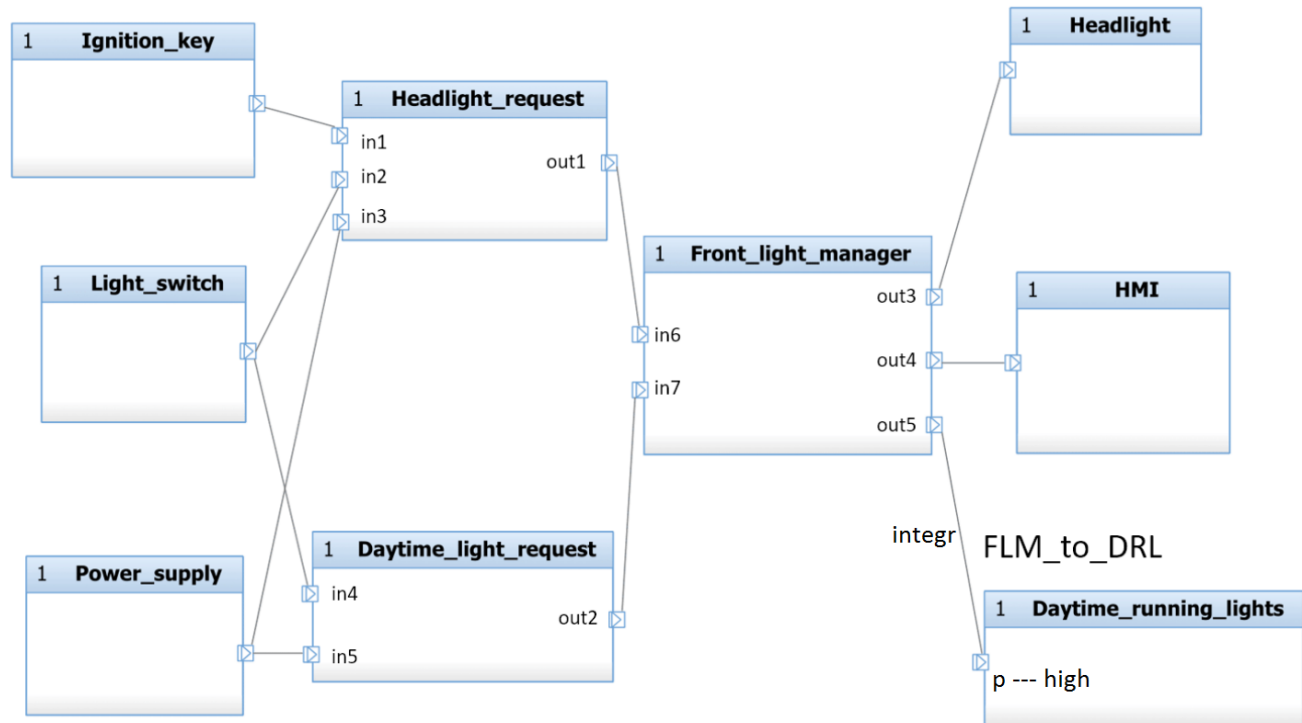
# Front Light Manager model

Security annotations:    Daytime\_running\_lights : High    FLM\_TO\_DRL : integr

Data secure flow  
is not satisfied



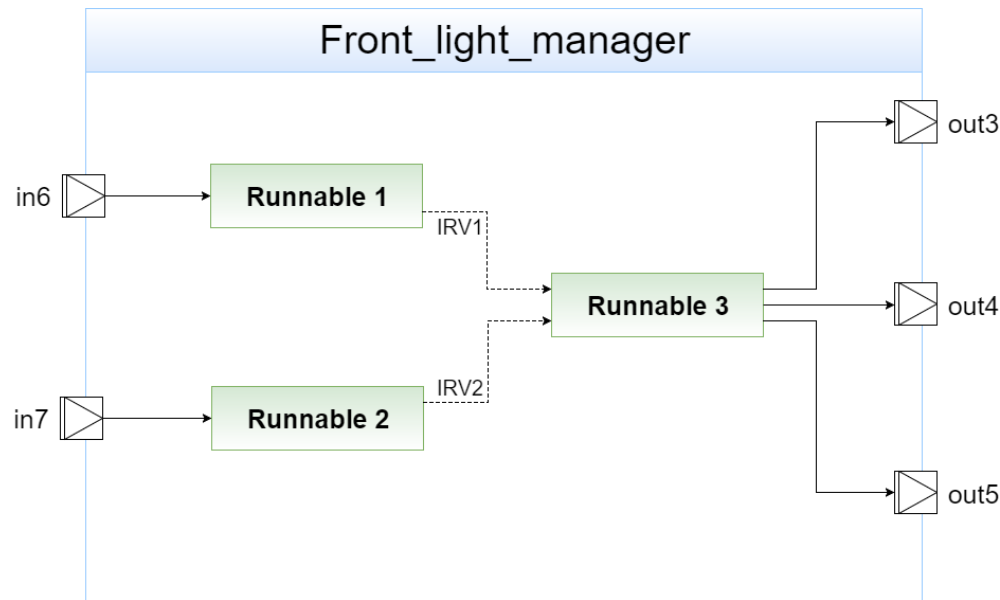
data sent on the  
link FLM\_TO\_DRL  
are not protected  
along the path  
from the sources to  
the destination



Simplest solution: assignment of high trust level to Front\_light\_manager, Headlight\_request, Daytime\_light\_request, Light\_switch, Ignition\_key, Power\_supply. Similarly for links.

We use Deps to correctly annotate the model.

# An example of component: Front\_light\_manager



```
void FLM_Runnable3(void) {  
    Rte_IWrite_Runnable3_PPort_out3(Rte_IrvIRead_Runnable3_IRV1());  
    Rte_IWrite_Runnable3_PPort_out5((Rte_IrvIRead_Runnable3_IRV2()));  
    if ( (Rte_IrvIRead_Runnable3_IRV1() == REQ_HEADLIGHT_ON) ||  
        (Rte_IrvIRead_Runnable3_IRV2() == REQ_DAYTIME_ON) ){  
        Rte_IWrite_Runnable3_PPort_out4(LIGHTS_ON);  
    }  
    else{  
        Rte_IWrite_Runnable3_PPort_out4(LIGHTS_OFF);  
    }  
}
```

# Information for generating the context

```
% global variables
int HR_voltage_threshold1;
int HR_voltage_threshold2;
int DLR_voltage_threshold1;
...
% inter runnable variables
int16_t FLM_IRV1;
int16_t FLM_IRV2;
int16_t DLR_IRV1;
...
% ports
int in1;
int in2;
...
int out1;
int out2;
...
% functions
void flm_Runnable1() 0;
void flm_Runnable2() 0;
.....
% links
out2 -> in7;
out1 -> in6;
```



# Analysis of dependencies

## Final dependencies

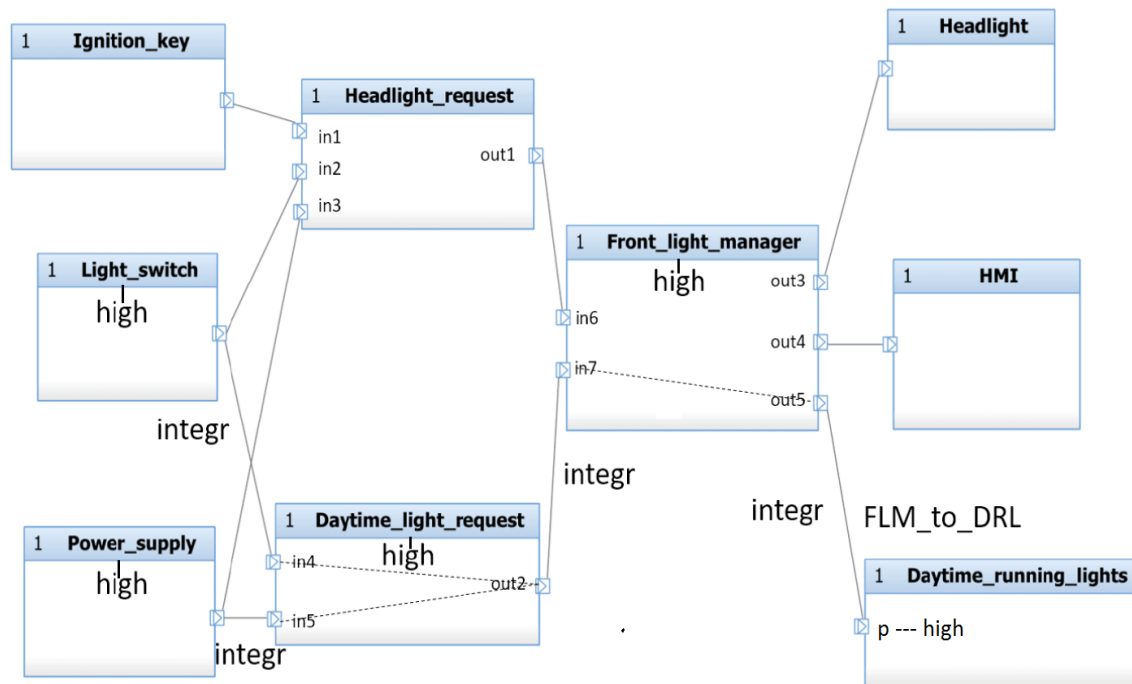
Global Environment:

	in1	in2	in3	in4	in5	in6	in7	out1	out2	out3	out4	out5
HR_voltage_threshold1	0	0	0	0	0	0	0	0	0	0	0	0
HR_voltage_threshold2	0	0	0	0	0	0	0	0	0	0	0	0
DLR_voltage_threshold1	0	0	0	0	0	0	0	0	0	0	0	0
DLR_voltage_threshold2	0	0	0	0	0	0	0	0	0	0	0	0
FLM_IRV1	1	1	1	0	0	1	0	1	0	0	0	0
FLM_IRV2	0	0	0	1	1	0	1	0	1	0	0	0
DLR_IRV1	0	0	0	1	0	0	0	0	0	0	0	0
HR_IRV1	1	0	0	0	0	0	0	0	0	0	0	0
HR_IRV2	0	1	0	0	0	0	0	0	0	0	0	0
in1	1	0	0	0	0	0	0	0	0	0	0	0
in2	0	1	0	0	0	0	0	0	0	0	0	0
in3	0	0	1	0	0	0	0	0	0	0	0	0
in4	0	0	0	1	0	0	0	0	0	0	0	0
in5	0	0	0	0	1	0	0	0	0	0	0	0
in6	1	1	1	0	0	1	0	1	0	0	0	0
in7	0	0	0	1	1	0	1	0	1	0	0	0
out1	1	1	1	0	0	0	0	1	0	0	0	0
out2	0	0	0	1	1	0	0	0	1	0	0	0
out3	1	1	1	0	0	1	0	1	0	1	0	0
out4	1	1	1	1	1	1	1	1	1	0	1	0
out5	0	0	0	1	1	0	1	0	1	0	0	1

For example, port in6 depends on ports in1,in2,in3, in6 and out1

# Front Light Manager final model

the output port of Front light manager connected to the Daytime\_running\_lights (out5 in our implementation) does not depend on the input port connected to the Headlight request component (in6 in our implementation)



Model that satisfies data secure flow property

# Conclusions

Abstract interpretation allows automated verification of secure information flow in programs

Intermediate level between typing approaches and semantics-based approaches

Analysis can be improved to reduce the number of false positive

Other works

- Secure information flow in .NET applications (master thesis)
- Secure information flow in concurrent programs (paper)

Future work

- Privacy of data in Android smart phones
- Malicious Colluding apps
- Privacy of data in medical app

Thank you for your attention!

Prof. Cinzia Bernardeschi  
Dipartimento di Ingegneria dell'Informazione  
Università di Pisa  
[cinzia.bernardeschi@unipi.it](mailto:cinzia.bernardeschi@unipi.it)

## Combining Abstract Interpretation and Model

# Security by Abstract Interpretation + Model Checking

## Abstract interpretation

- abstract domain of values
- execute the program operating over abstract domain
- abstract semantics = structure representing all executions in a finite way

## Model Checking

Model check the abstract semantics against temporal logic formulae expressing the expected behavior

# Implementation in SMV

- SMV model checker
- Kripke structure
- CTL logic
- Specifications = assertions on the state variables  
and on the paths of the system.

# Implementation in SMV

$\Phi_{SIF} =$

and  $\mathbf{x} \text{ in } L \text{ AG } ((PC = i) \text{ and } P[PC] = \text{halt})$   
 $\Rightarrow ((MEM[x] = L) \text{ or } (MEM[x] = (j, L))) ;$