Program analysis and SIF

Program analysis and SIF

For secure information flow analysis, instead of using a set of memory associated with a node of the control flow graph, during the analysis we compute the least upper bound between memories.

The memory assigned to a state is the memory with the highest security levels assigned to variables during the abstract execution of the program.

"If a concrete memory *m* occurs at node *q* at some point during an execution sequence then the collection of abstract memory at *q* must contain the corresponding abstract memory, but may contain additional abstract memories as well."

The analysis

scope(2) = 3, 4
ipd(2) = 5

x=L y=H
Implcit flow: upgrade Env. of instructions in the scope

```
1  load y
2  if 5
3  load 1
4  store x
5  halt
```

	$(M(x), M(y))$	*Stack*	*Env.*
Q_1	(L, H)	λ	L
Q_2	(L, L)	λ	L
Q_3	(L, L)	λ	L
Q_4	(L, L)	λ	L
Q_5	(L, L)	λ	L
Q_f	(L, L)	λ	L

WL = {1, 2, 3, 4, 5}
For example, when instruction 1 is executed, the state of Q2 is updated

The analysis

Each instruction i is assigned a state Q_i, representing the state in which instruction i is executed. Q_i is an abstraction of the JVM's state *before* the execution of i. Q_i is a triple (M, S, E), where $M : Registers \rightarrow \Sigma$ is a mapping from local registers to secrecy levels (the memory) , $S \in \Sigma^*$ is a mapping from the elements in the operand stack to secrecy levels (the stack) and $E \in \Sigma$ is the secrecy level of the environment.

A partial order relation on the domain of the states corresponding to the instructions is defined. This relation is induced from the ordering relation among secrecy levels.

The analysis

Partial order relation on the domain of states

- Given two states $Q_i = (M, S, E)$ and $Q'_i = (M', S', E')$, $Q_i \sqsubseteq Q'_i$ iff $M \sqsubseteq M', S \sqsubseteq S', E \sqsubseteq E'$.
- Given two memories M_1 and M_2, $M_1 \sqsubseteq M_2$ iff for each register x, it is $M_1(x) \sqsubseteq M_2(x)$.
- The domain of stacks has a bottom element, $\perp_S$, which is $\sqsubseteq$ of all stacks. Given two stacks S_1 and S_2 different from $\perp_S$, $S_1 \sqsubseteq S_2$ iff S_1 and S_2 have the same length and each item in S_1 is $\sqsubseteq$ of the item occurring in S_2 in the same position.
  Stacks with different length are unrelated.

The analysis

In the analysis:

- The domain of value is finite
- The the numer of instructions is finite

- The fixpoint of the iteration process is reached with a finite number of iterations

- The fixpoint does not depend on the order of application of the rules

The analysis

Initialization conditions:

- for each conditional instruction i the ipd is computed ($ipd(i)$).
- for each conditional instruction i the set of instruction in the scope of i is computed ($scope(i)$).
- the table of states is initialised as follows:
  Variables in all memories are assigned low level, except for variables of the memory of state Q_0, whose variables are initialised to the specified security level;
- the operand stack is assigned $\perp_S$
- all instructions is inserted in the Working List (WL)

The analysis

For each instruction in the WL the following actions are taken:

1. build the state *after* the execution of the instruction as specified by the corresponding rule;
2. using the state built at step 1, change the state of the next instructions, if any. All instructions have one successor, except for conditional branches that have two, and `return` that has none;
3. if the state of a successor instruction changes, insert the successor instruction in the WL.
4. the analysis terminates when the WL is empty.

The analysis

The least upper bound operation on states corresponding to instructions is defined point-wise on memories, stacks and environments:
$(M, S, E) \sqcup (M', S', E') = (M \sqcup M', S \sqcup S', E \sqcup E')$.
Global states are partially ordered according to the relation on the state of every instruction: $Q \sqsubseteq Q'$ iff $\forall i, Q_i \sqsubseteq Q'_i$.

The analysis

When the rule for instruction i is applied,

1. the state *after* i is calculated, abstractly executing i in state Q_i. For example, if `i: load` x is executed, and $Q_i = (M, S, E)$, the state $\bar{Q}_i$ after i has the same memory and the same environment of Q_i, and the least upper bound between the contents of x in M and the environment E is pushed onto S;

2. the state $\bar{Q}_i$ after i is *merged* with the state of all successive instructions of i. Let j be a successive instruction of i. The *merge* is performed by a least upper bound calculation between the original value of Q_j and the state $\bar{Q}_i$. For example, the state after `i: load` x is merged with Q_{i+1}.

As a consequence, if an instruction i has several predecessor instructions, Q_i is the *least upper bound* of the states after all the preceding instructions.

An example

scope(2) = 3, 4 x=L y=H
ipd(2) = 5

```
1  load y
2  if 5
3  load 1
4  store x
5  halt
```

	$(M(x), M(y))$	*Stack*	*Env.*
Q_1	(L, H)	λ	L
Q_2	(L, L)	λ	L
Q_3	(L, L)	λ	L
Q_4	(L, L)	λ	L
Q_5	(L, L)	λ	L
Q_f	(L, L)	λ	L

WL = {1, 2, 3, 4, 5}
For example, when instruction 1 is executed, the state of Q2 is updated

An example

```
1   load y
2   if 5
...
```

WL = $\{1, 2, 3, 4, 5\}$

	$(M(x), M(y))$	*Stack*	*Env.*
Q_1	(L, H)	λ	L
Q_2	(L, L)	λ	L
Q_3	(L, L)	λ	L
Q_4	(L, L)	λ	L
Q_5	(L, L)	λ	L
Q_f	(L, L)	λ	L

Instruction 1 is executed. WL = $\{2, 3, 4, 5\}$

	$(M(x), M(y))$	*Stack*	*Env.*
Q_1	(L, H)	λ	L
Q_2	(L, L)	H	L
Q_3	(L, L)	λ	L
Q_4	(L, L)	λ	L
Q_5	(L, L)	λ	L
Q_f	(L, L)	λ	L

The analysis

```
1   load y
2   if 5
...
```

WL = $\{2, 3, 4, 5\}$

	$(M(x), M(y))$	*Stack*	*Env.*
Q_1	(L, H)	λ	L
Q_2	(L, L)	H	L
Q_3	(L, L)	λ	L
Q_4	(L, L)	λ	L
Q_5	(L, L)	λ	L
Q_f	(L, L)	λ	L

Instruction 2 executed. WL = $\{3, 4, 5\}$ scope(2) = $\{3, 4\}$

	$(M(x), M(y))$	*Stack*	*Env.*
Q_1	(L, H)	λ	L
Q_2	(L, L)	λ	L
Q_3	(L, L)	λ	H
Q_4	(L, L)	λ	H
Q_5	(L, L)	λ	L
Q_f	(L, L)	λ	L

The analysis

Using this approach, the SIF analysis can be extended to cover

- 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

$\rightarrow$ THE SECURITY CONTEXT

The security context file

Propagation caused by global variables and by functions

```
type x;
type f(···);
type g(···);
```

Propagation caused by actual parameters/return of a function

```
type fun(type x1, ··· , type xn) {
        ......;
        return expr;
}

k = fun(a1, ···, an)
```

The security context file

During the analysis, the security contest contains:

- for each variable, the context file maintains the maximum security level of data stored in the variable:
  $var\_name : \sigma$

- for each function, the context file maintains how the function is called in terms of the maximum level of the calling environment, the actual parameter and the return:
  $fun\_name(param_1, \cdots, param_n) : return, calling\_evironment$

Iterative analysis

During the analysis,

- functions are analysed one at a time;
- the context file is update accordingly;
- at the end of the analysis of a function, if the context file is changed, all functions must be re-analysed starting from the new context file.
- the analysis terminates when, stating from a context file, all functions are analysed and the context file is unchanged (a fixpoint is reached).

Iterative analysis

A: security context
R: set of all functions
EXEC: the abstract execution interpreter of a function

The algorithm

A:= A^0
T:= R
while (T $\neq \emptyset$)
    select f $\in T$
    T := T - {f}
    $A' := EXEC(f, A)$
    if ($A' \neq A$)
        A := A′;
        T:=R

The analysis is repeated until **fixpoint** is reached

Iterative analysis

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

The analysis terminates, since security levels in the context file can only be upgraded, and the number of security levels is finite

Case studies

Case studies

Java cards:

Secure interactions in Java cards

Automotive:

Data secure flow in AUTOSAR models

Case studies

Java cards:

Secure interactions in Java cards

Java cards

JCSI (Java Card Secure Interaction): a tool that can be used for discovering security issues due to explicit or implicit information flows and for checking security properties of Java Card applications downloaded from untrusted sources.

Java Card 2.2.2 full instruction set

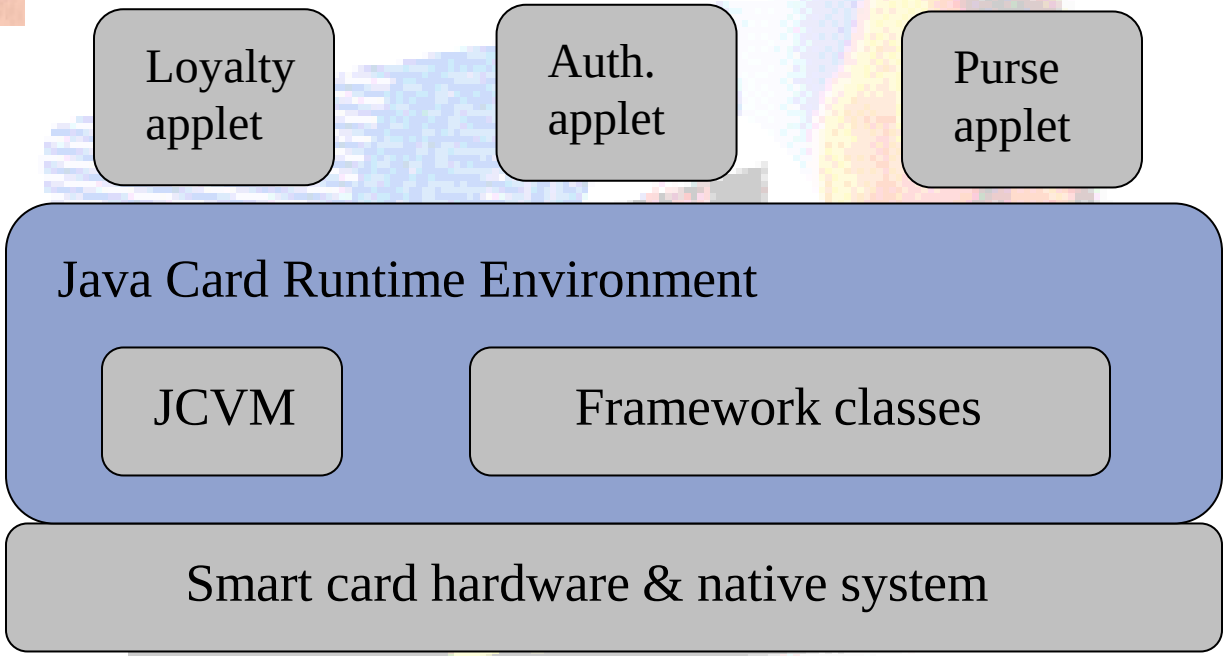
Java cards

- Smart cards: embedded systems that allow to store and process information
- Typical Aplications:
  Credit cards, Electronic cash, Loalty systems, Helthcare, 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 anouthorised accesses

Java cards

Card reader

Multiapplicative Java cards

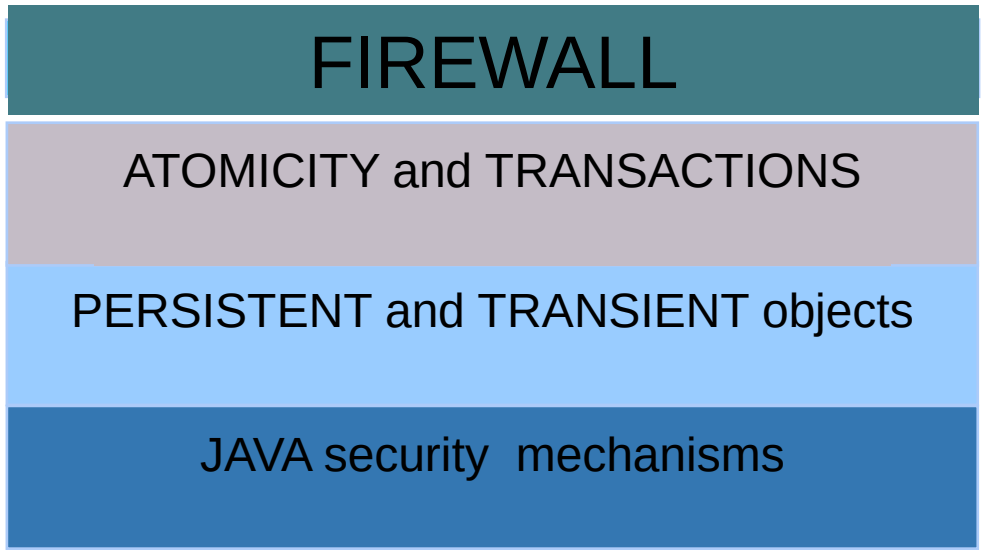


application consists of a set of applets bundled into a package

new applications can be installed after card issuance

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



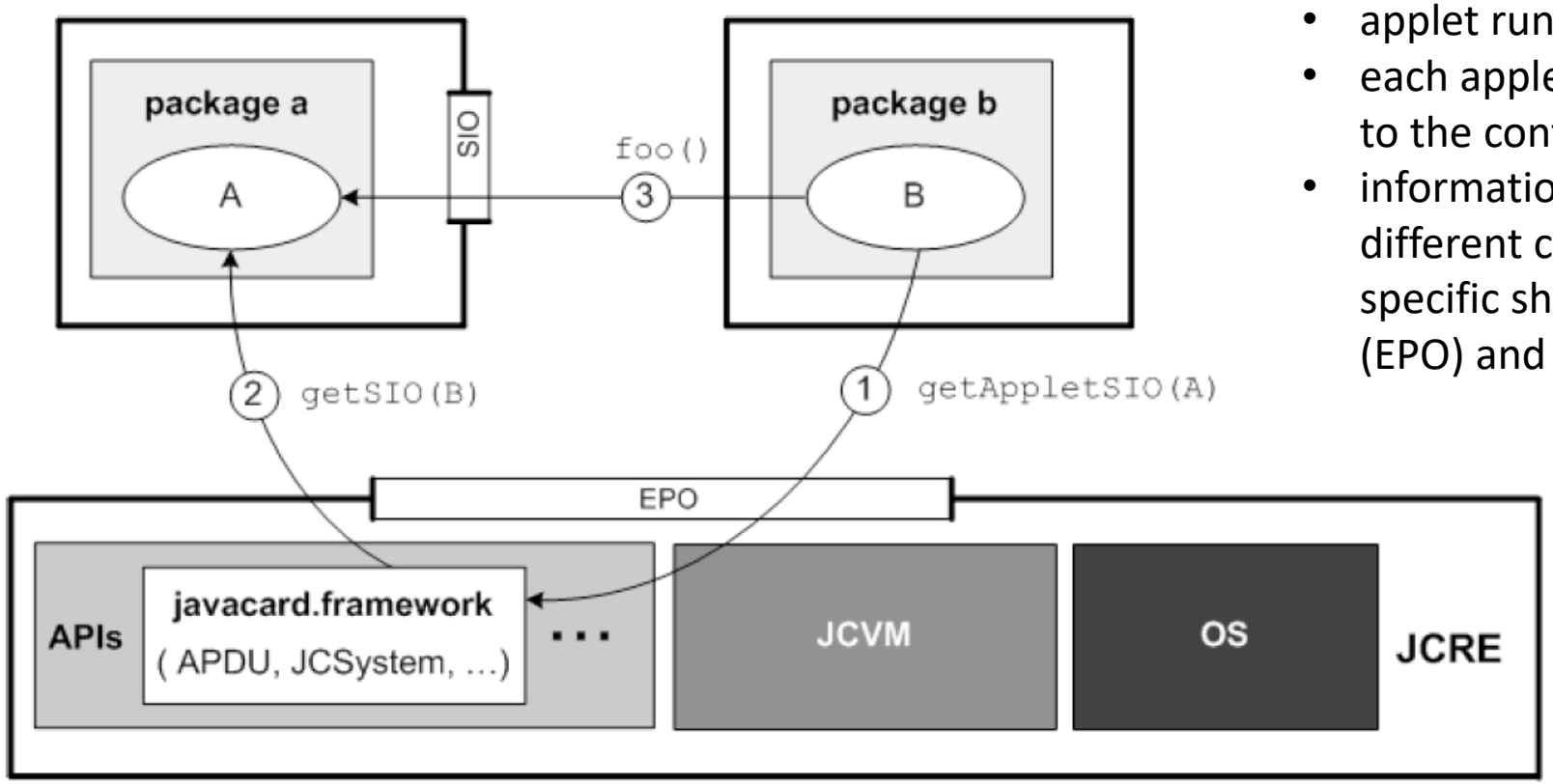
Context:

- applications are executed within protected spaces
- each application is associated with a unique context.

Firewall:

- forces the isolation between objects of applets belonging to different packages
- uses an access control mechanism to enforce security policies

Communication between packages



- applet run in a user space
- each applet can access only objects belonging to the context of the app
- information exchange between app of different contexts performed only through specific shared objects, Entry Point Objects (EPO) and Shareable Interfaces Objects (SIO)

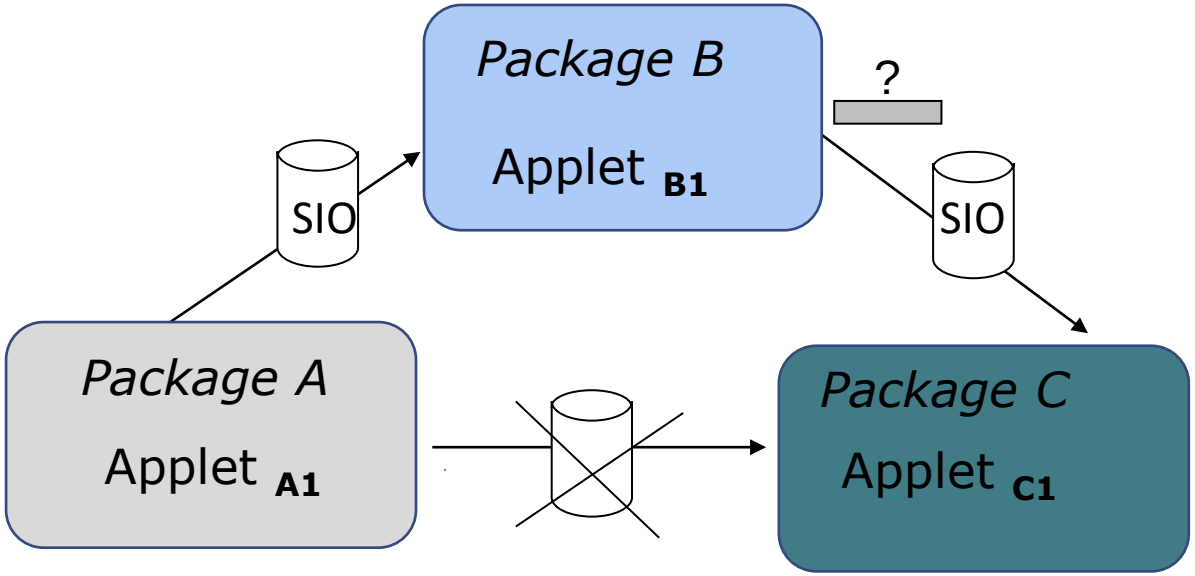
EPO: belongs to the JCRE's context. Provide methods for exchanging messages (e.g., to request access to a resource), and for customizing access control rules (e.g., to identify the identity of another application).
SIO: belongs to applications's context . Provide methods to define the functionalities of applications' shared objects

Limits of the firewall

- Based on access control checks

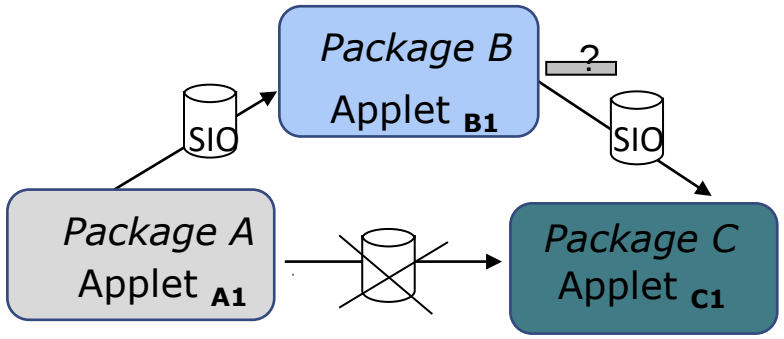
- Place restrictions on the applets that can access to methods of applets belonging to other packages

- Does not control the propagation of the information from an applet of a package towards applets of other packages



Example

A stores sensitive data in a private field balance,
A wants to share balance with B, but not with other applets.



- A implement a SIO(), which contains method foo()
  for returning the content of balance only to B.
- B executes getAppletSIO(AID, byte) implemented in a static
  EPO of the JCRE

- JCRE: dispatches a request for the shareable object to A by triggering the invocation of method getSIO(AID, byte) implemented by A, where the first parameter (AID) identifies the applet that requested the shared object (B) and byte defines a specific SIO.

- If A accepts the request, a pointer to the shareable object is returned to the JCRE, which in turn returns the pointer to B. If the request is not accepted (this may happen, for example, when the applet that requested the SIO is not authorized), A returns a null pointer to the JCRE, which in turn returns a null pointer to B.

When method foo() is invoked, the JCRE automatically performs a context switch: the current active context is changed from the context B to context A.
When the execution of the method completes, the context of the caller is set as the current active context

Example

Using access control policies to control information sharing

file BInt.java

```
import javacard.framework.Shareable;
public interface BInt extends Shareable{
   public a.AInt bar();  }
```

file B.java

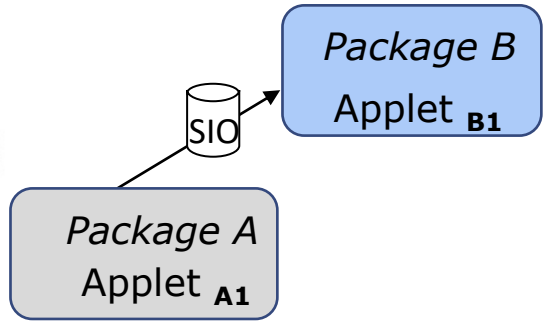
```
import javacard.framework;
import a.AInt;
public class B extends Applet implements BInt{
  private static AInt AObj;
  private short ABalance;
  private void work (){
    AObj = (AInt) (JCSystem.getAppletSIO(A, 0));
    ABalance = AObj.foo();
  }
  public Shareable getSIO(AID client, byte num){
   return this;
  }
  public AInt bar(){return AObj;}
}
```

file AInt.java

```
import javacard.framework.Shareable;
public interface AInt extends Shareable{
   public short foo();  }
```

file A.java

```
import javacard.framework;
import a.AInt;
public class A extends Applet implements AInt{
  private short balance;
  public Shareable getSIO(AID client, byte num){
    if(client.equals(B))  return this;
    return null;
  }
  public short foo(){
    AID client = getpreviouscontextAID();
    if(client.equals(B)) return balance;
    return 0;
  }
}
```



Example

Using access control policies to control information sharing

file BInt.java

```
import javacard.framework.Shareable;
public interface BInt extends Shareable{
   public a.AInt bar();  }
```

file B.java

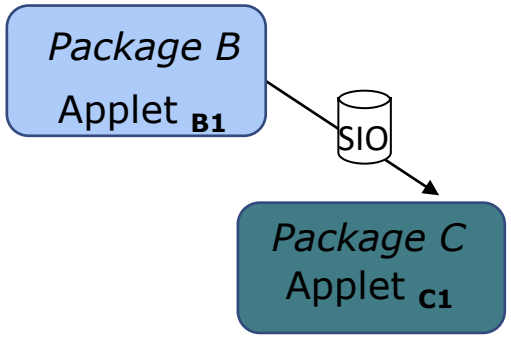
```
import javacard.framework;
import a.AInt;
public class B extends Applet implements BInt{
  private static AInt AObj;
  private short ABalance;
  private void work (){
    AObj = (AInt) (JCSystem.getAppletSIO(A, 0));
    ABalance = AObj.foo();
  }
  public Shareable getSIO(AID client, byte num){
   return this;
  }
  public AInt bar(){return AObj;}
}
```

file CInt.java

```
import javacard.framework.Shareable;
public interface CInt extends Shareable{
   public void mh();}
```

file C.java

```
import javacard.framework;
import a.AInt;
import b.BInt;
public class C extends Applet implements CInt{
  private static AInt AObject;
  private static BInt BObject;
  private short ASecret;
  public Shareable getSIO(AID client, byte num){
   return this;}
  public void work(){
   BObject = (BInt) JCSystem.getAppletSIO(B, 0);
   AObject = BObject.bar();
   ASecret = AObject.foo();
  }
}
```



Java cards security

Data leakage analysis

Security policy + static analysis

- Security lattice: the powerset of the different security levels of apps
- Static analysis for secure information flow
- Applications are analyzed one at a time
- The analysis is carried out on a per-method basis

modular analysis similar to that performed by the Java bytecode verifier, which aims to check type correctness of Java bytecode

Ambient file to store and propagate security levels of methods (i.e., methods' arguments, return, and calling environment), and the security levels of objects in the heap.

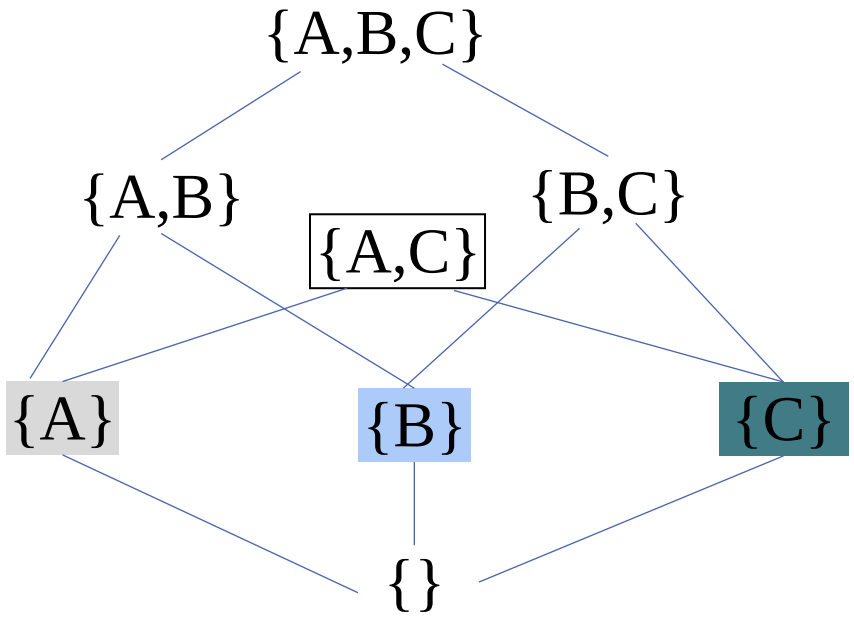
-> information flows in the bytecode are assessed by checking that the security level of the applications' shared resources do not exceed the level specified in the security policy.

Secure Information Flow

- Security levels assigned to packages:
  ID of the package
- Lattice of security levels:
  A: data which depend only on A

  A+B: data which depend only on A and B

  A+B+C: data which depend on A, B and C



Secure Information Flow

check that information exchanged between A and B has a security equal to or lower than {A,B}

Ambient file

Each entry in the ambient file specifies the security levels held by the item and, optionally, the security policy that must be checked.

The ambient file has two sections

heap section

which stores and propagates security levels of objects in the heap
Objects are abstracted into classes
Arrays are abstracted into array types
Arrays of objects and array of arrays are all abstracted into array of references type

- security level for each class field
  c.f <S>
- security level for each array type
  [t <S>

methods section

which is dedicated to methods

$$m_p^{p'} (\tau_0, \tau_1, \ldots, \tau_n) \tau_r; \tau_e \text{ <S>}$$

p is the package that implements mt
p' is the package that invokes mt
τ_0 the implicit parameter (this)
$\tau_1, \ldots \tau_n$ are the method's arguments
τ_r is the return
τ_e is the calling environment
<S> is the security policy that must be satisfied by mt

Ambient file: heap section

Heap is a private resource of the package.

Assuming we are analyzing the package p,

-> the tool initializes the security levels to the default value equal to {p}

The security level of each class field c.f is initially set to {p}

The level of class fields will be updated during the analysis according to the flow of information and the dependencies between instructions in the program

When the analysis completes

-> the security level of c . f is the max security level of the field f of all objects of class c.

Similarly, the security level of each array type is set to the security level {p}

When the analysis completes,

-> the security level of arrays of type [t is the max level saved into all array instances of type [t

Ambient file: method section

- internal methods (i.e., methods defined in p – they can be invoked only by applets belonging to the packages)
- imported methods (i.e., methods defined in other packages – they are invoked by p)
- or exported methods (i.e., methods defined in p that can be used by other packages)

The level of a method characterizes how the method is called in terms of:

-> the level of the method's actual parameters, return and calling environment

Default initialization of the ambient file

configures the analysis for the worst-case scenario: any package already installed on card may implement methods imported by the applet under analysis and may invoke exported methods of the applet under analysis.
The default policy enforces that information shared between two packages only depends on these two packages

Customized initialization of the ambient file

when the CAP files of the other applets installed on card are available, the user, assisted by the DeCAP tool, can look at those files for identifying which packages can potentially invoke the exported methods, and modify the ambient file accordingly

Ambient file: method section

$Packages(mt)$ set of packages that implement method *mt*

- imported methods
  the security policy is $\langle glb(\{\{p, \overline{p}\} \mid \overline{p} \in Packages(mt)\})\rangle$

- exported methods
  1. if mt is one of the entry point methods for JCRE and mt is not getSIO(), the security policy can be released, as the invocation of such methods propagates information to JCRE only.
     The ambient file contains an instance of mt with level p and without a security policy:

$$mt_p^p(\{p\}, \ldots, \{p\})\{p\}; \{p\}$$

  2. If the exported method is getSIO(), we need to identify all packages that may potentially request a shareable object by invoking method getAppletSIO().
     The ambient file must contain an instance of mt for each package that invokes getAppletSIO(), with the security policy: $mt_p^{p'}(\{p\}, \ldots, \{p\})\{p\}; \{p\}\ \langle\{p, p'\}\rangle$

- all other exported methods, the policy is the policy in the default initialization

Analysis of a package

p: package
D: ambient file, D^0 initial ambient file

$D := D^0$
$T := \{mt \in \mathcal{M} \mid p \texttt{ implements } mt\}$
$MT := T$
while$(MT \neq \emptyset)$
    select $mt \in MT$
    $MT := MT - \{mt\}$
    $D' := MSC(mt, D)$
    if$(D' \neq D)$
        $D := D'$
        $MT := T$

- Iterative data flow analysis
- the analysis uses an abstract interpreter, named Method Security Checker (MSC), for verifying a method

Analysis of a method

- Abstract semantics rules starting from an initial state Q^0 that represents the state of JVM on method entrance
- A method is verified stating from the current ambient file. The ambient file is set to the initial ambient file D^0 at the beginning.
- Instructions to be verified are inserted into a worklist WL, initialized with instruction 0. When WL is empty, the verification of a method completes
- Whenever an instruction i is fetched from WL, instruction i is executed starting from its current state Qi, and the after-state of instruction i is computed. The after-state is then merged with the before-state of every successor of the instruction.
- If the state of a successor j changes, or if a successor has not been visited yet, j is inserted in WL. Note that Qj stores a state that merges all possible states in which j can be executed.

Analysis of a method

Note that:

- when an instruction i is executed, every state Qj corresponding to the entry point of an exception handler protecting the instruction is also updated

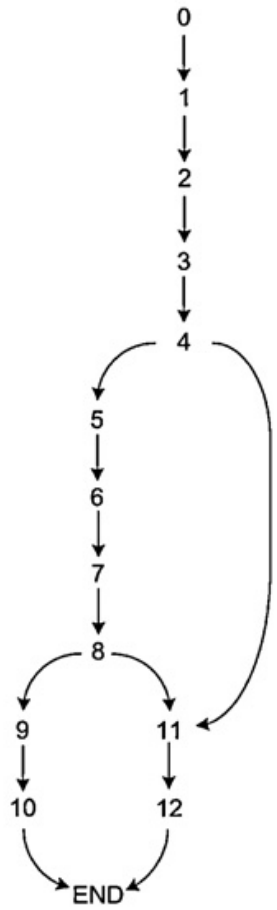
- the stack at the entry point of exception handlers contains one operand.
  in the rules, this operand is set equal to the lub between the levels in the stack positions and the level of the security environment

Analysis of a method: MSC

Bytecode of getSIO() of package B and its CFG.

```
.method public getSIO(AID, byte)Shareable
.locals 3

    0    aload r0;
    1    aload r1;
    2    getstatic b/B.C;
    3    invokevirtual JCSystem.AID.equals()
    4    ifeq 11;
    5    aload r0;
    6    getfield ABalance;
    7    push 100;
    8    if_icmple 11;
    9    aload r0;
    10   areturn;
    11   aconst_null;
    12   areturn;
```



Analysis of a method: MSC

$D^0 =$

class fields, arrays $AObj = \{b\}$ $ABalance = \{a, b\}$ ... static fields B.A = {b} B.C = {b} ...
internal methods $equals_b^b(\{\{b\}, b\})\{b\}; \{b\}$ ... $work_b^b(\{b\})\{b\}; \{b\}$
imported methods $getAppletSIO_b^b(\{b\}, \{b\})\{b\}; \{b\}\ \langle\{b\}\rangle$ $foo_a^b(\{a, b\})\{a, b\}; \{a, b\}\ \langle\{a, b\}\rangle$
exported methods $getSIO_b^c(\{b, c\}, \{b, c\}, \{b, c\})\{b, c\}; \{b, c\}\ \langle\{b, c\}\rangle$ $getSIO_b^a(\{a, b\}, \{a, b\}, \{a, b\})\{b, a\}; \{a, b\}\ \langle\{a, b\}\rangle$ $bar_b^c(\{b, c\})\{b, c\}; \{b, c\}\ \langle\{b, c\}\rangle$

$Q^0 =$

	E	M	St
0	$\{b, c\}$	$(\{b, c\}, \{b, c\}, \{b, c\})$	λ
1	{}	({}, {}, {})	λ
2	{}	({}, {}, {})	λ
3	{}	({}, {}, {})	λ
..	..	...	...
12	{}	({}, {}, {})	λ
END	{}	({}, {}, {})	λ

Analysis of a method: MSC

$D =$

class fields, array $AObj = \{b\}$ $ABalance = \{a, b\}$ ... static fields B.A = {b} B.C = {b} ...
internal methods $equals_b^b(\{b,c\},\{b,c\})\{b,c\};\{b,c\}$ ... $work_b^b(\{b\})\{b\};\{b\}$
imported methods $getAppletS\,IO_b^b(\{b\},\{b\})\{b\};\{b\}\ \langle\{b\}\rangle$ $foo_a^b(\{a,b\})\{a,b\};\{a,b\}\ \langle\{a,b\}\rangle$
exported methods $getS\,IO_b^c(\{b,c\},\{b,c\})\{a,b,c\};\{a,b,c\}\ \langle\{b,c\}\rangle$ $getS\,IO_b^a(\{a,b\},\{a,b\})\{a,b\};\{a,b\}\ \langle\{a,b\}\rangle$ $bar_b^c(\{b,c\})\{b,c\};\{b,c\}\ \langle\{b,c\}\rangle$

$Q =$

	E	M	St
0	$\{b,c\}$	$(\{b,c\},\{b,c\},\{b,c\})$	λ
1	$\{b,c\}$	$(\{b,c\},\{b,c\},\{b,c\})$	$\{b,c\}$
2	$\{b,c\}$	$(\{b,c\},\{b,c\},\{b,c\})$	$\{b,c\}\cdot\{b,c\}$
3	$\{b,c\}$	$(\{b,c\},\{b,c\},\{b,c\})$	$\{b,c\}\cdot\{b,c\}\cdot\{b,c\}$
4	$\{b,c\}$	$(\{b,c\},\{b,c\},\{b,c\})$	$\{b,c\}$
5	$\{b,c\}$	$(\{b,c\},\{b,c\},\{b,c\})$	λ
6	$\{b,c\}$	$(\{b,c\},\{b,c\},\{b,c\})$	$\{b,c\}$
7	$\{b,c\}$	$(\{b,c\},\{b,c\},\{b,c\})$	$\{a,b,c\}$
8	$\{b,c\}$	$(\{b,c\},\{b,c\},\{b,c\})$	$\{b,c\}\cdot\{a,b,c\}$
9	$\{a,b,c\}$	$(\{b,c\},\{b,c\},\{b,c\})$	λ
10	$\{a,b,c\}$	$(\{b,c\},\{b,c\},\{b,c\})$	$\{a,b,c\}$
11	$\{a,b,c\}$	$(\{b,c\},\{b,c\},\{b,c\})$	λ
12	$\{a,b,c\}$	$(\{b,c\},\{b,c\},\{b,c\})$	$\{a,b,c\}$
END	$\{b,c\}$	$(\{b,c\},\{b,c\},\{b,c\})$	λ

The analysis of another method starts from the current ambient file D

Analysis results

When the analysis of a package completes, the ambient file stores the highest security level of calling environment, actual parameters and return for each method

Illegal flows of information can be detected by looking at methods of shareable interface objects in the ambient file (exported and imported methods)

Checking exported methods

Given a security policy <S> for an exported method $mt_p^{p'}$:

the return level of the method must be $\subseteq$ S

Checking imported methods

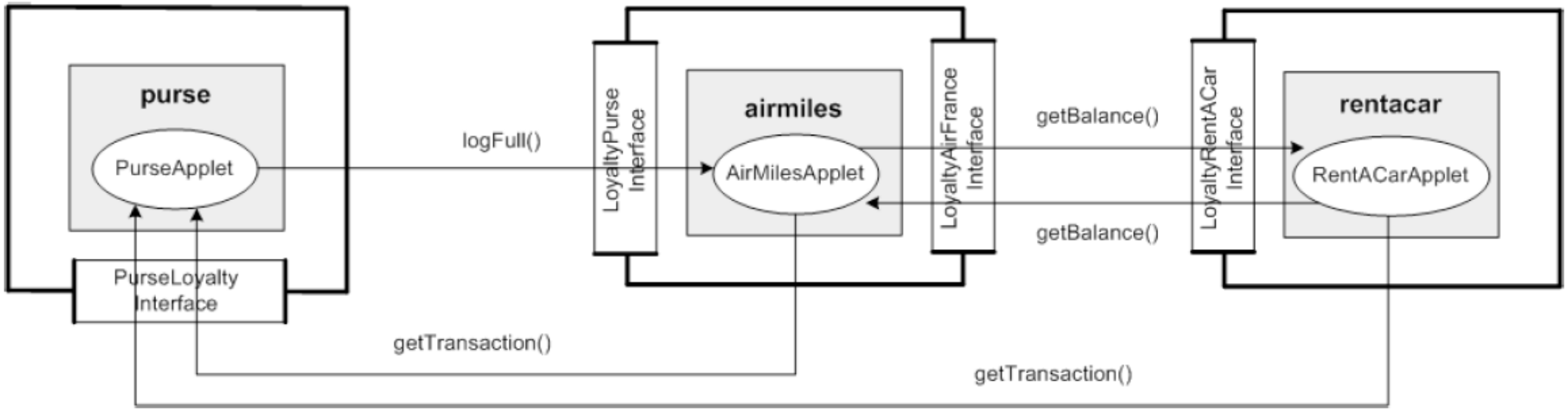
Given a security policy <S> for an imported method $mt_{p'}^{p}$:

the parameters, return, and calling environment of the method must be $\subseteq$S.

Analysis of results

- The data flow analysis is conservative, i.e., all packages that may potentially disclose confidential information are rejected, but also some secure packages might be rejected
- This is due to the fact that during the abstract execution of a method, all branches of control instructions are checked, even those that in the real execution would have never been executed
- Moreover, methods are executed under an ambient file with the maximum security levels assigned to methods and objects in the heap
- Formal proof that the analysis completes in a finite number of iterations, and independently from the order of application of the rules

A case study: Electronic Purse



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

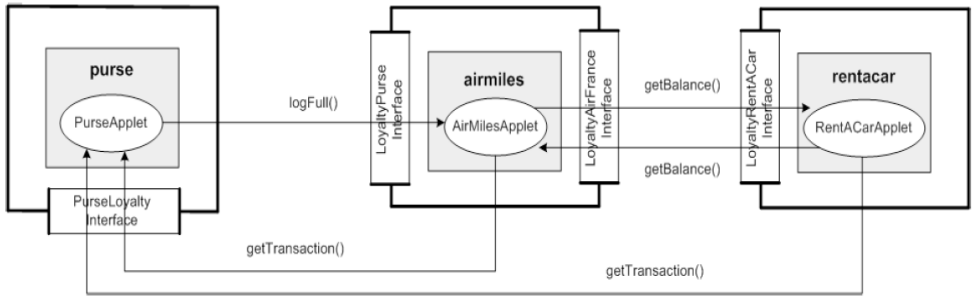
Purse: log-full service (logFull()), notifies registered applets that the transaction log is going to be over-written

Airmail: registered for the log-full service

RentACar: not registered for the log-full service

Electronic Purse

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



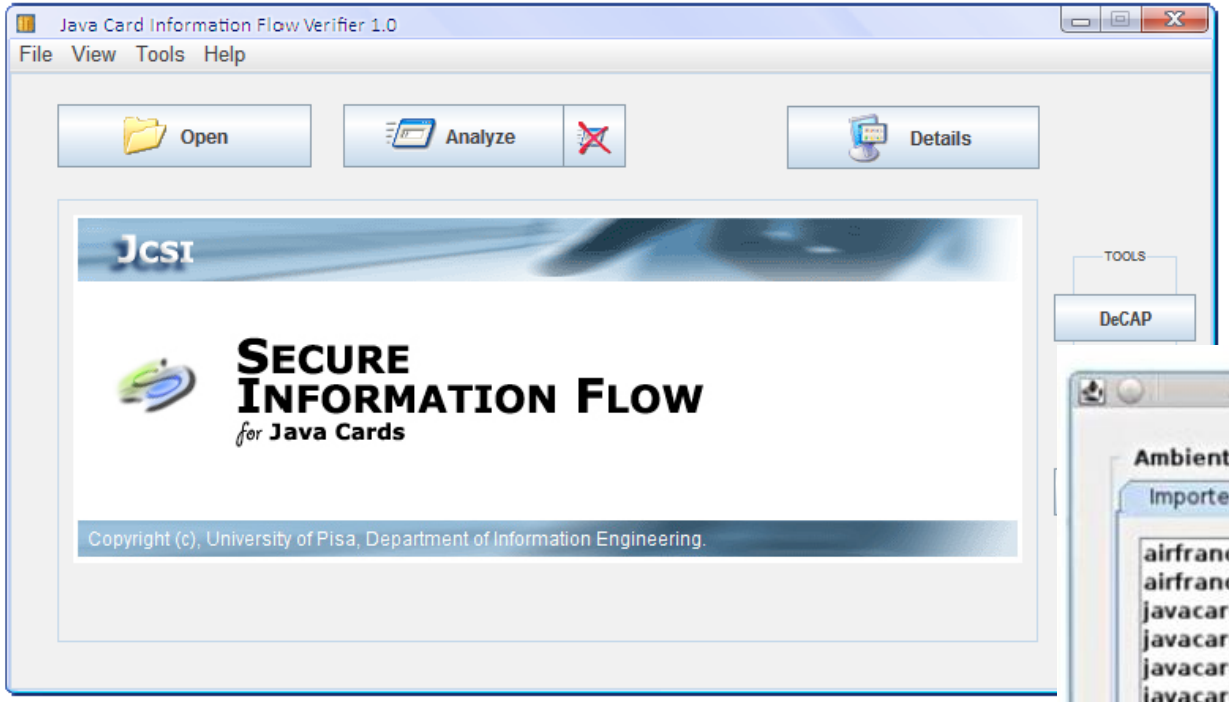
logFull() method implemented by AirFrance contains an invocation of method getTransaction() of Purse followed by an invocation of method getBalance() of RentACar.

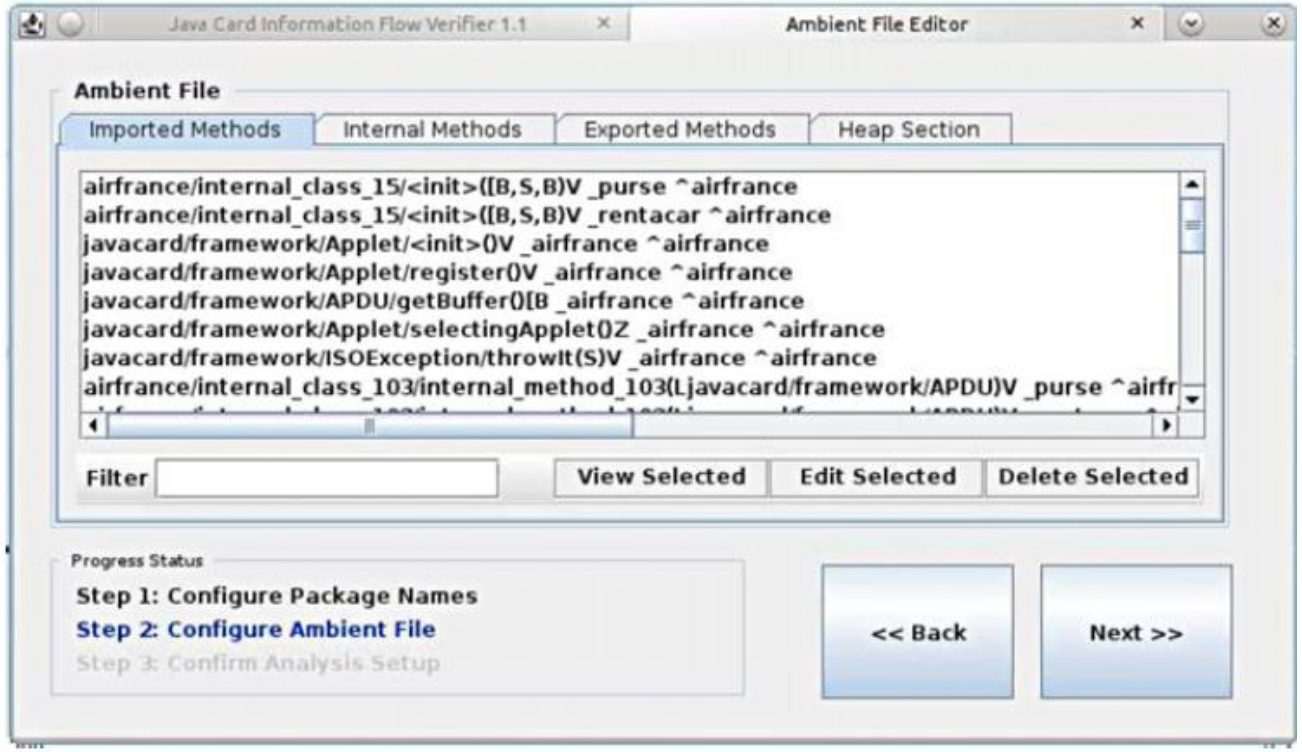
Applet RentaACar, whenever observes an invocation of getBalance(),
can infer that Purse is going to over-write the transaction log.

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

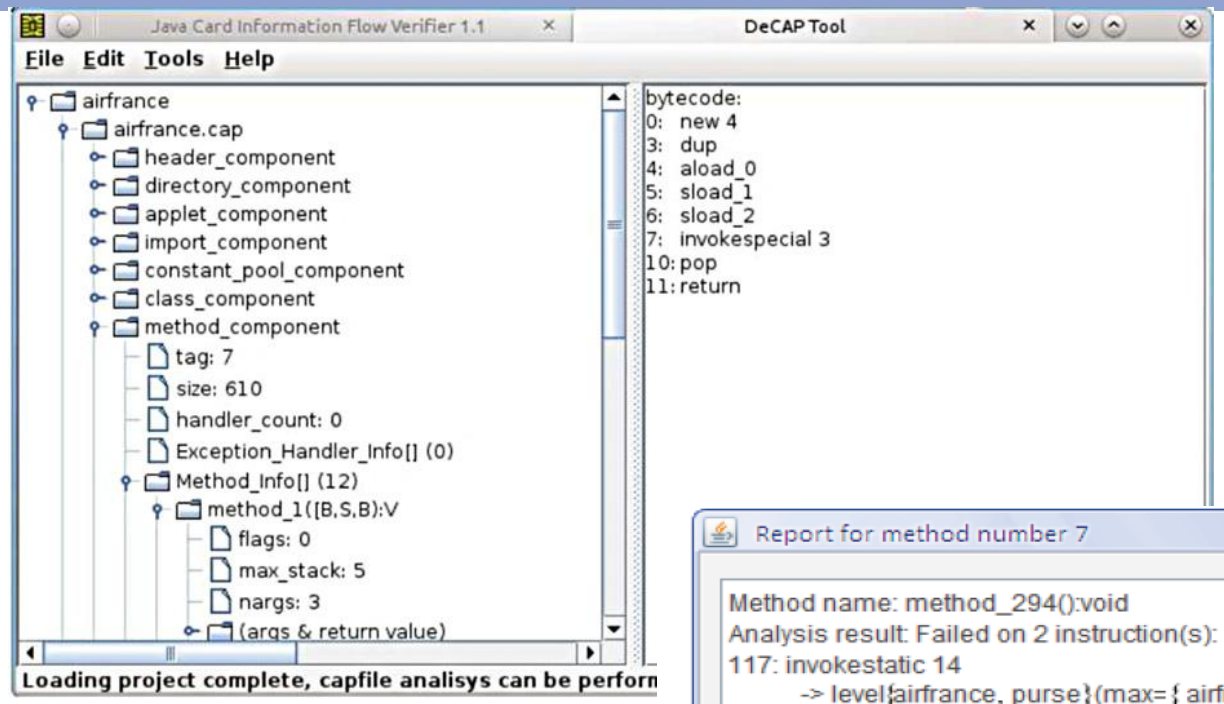
Purse is not able to detect such information flow.

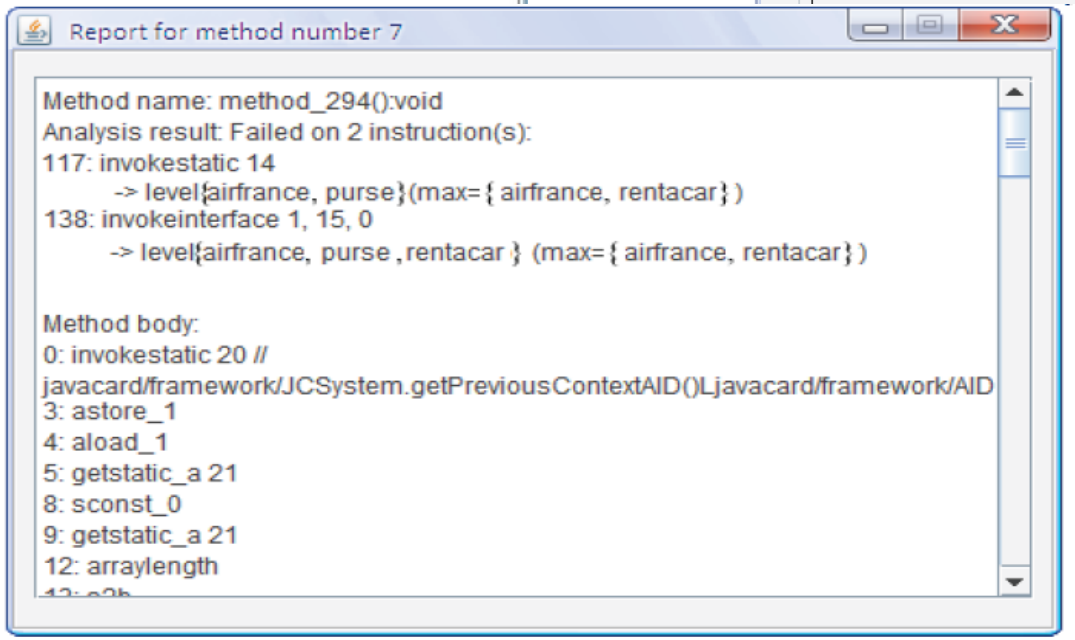
JCSI tool

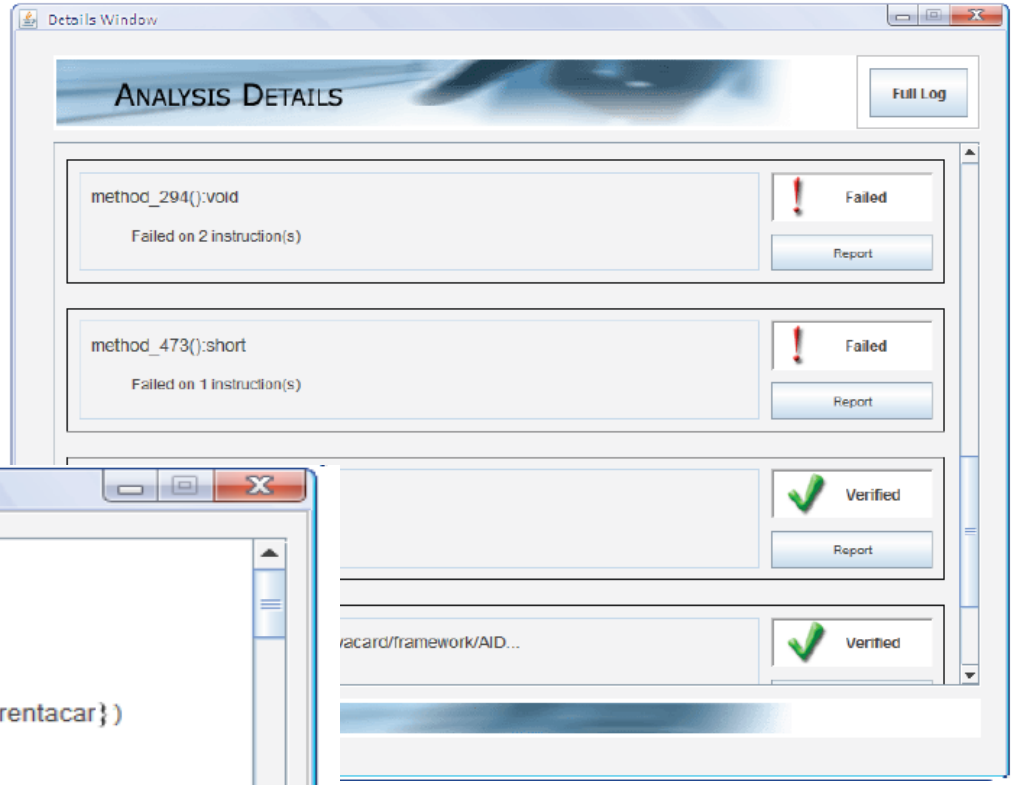

Java Card Information Flow Verifier 1.0
File View Tools Help
Open
Analyze
Details
JCSI
SECURE INFORMATION FLOW
for Java Cards
Copyright (c), University of Pisa, Department of Information Engineering.
TOOLS
DeCAP


Java Card Information Flow Verifier 1.1
Ambient File Editor
Ambient File
Imported Methods
Internal Methods
Exported Methods
Heap Section
airfrance/internal_class_15/<init>([B,S,B)V _purse ^airfrance
airfrance/internal_class_15/<init>([B,S,B)V _rentacar ^airfrance
javacard/framework/Applet/<init>()V _airfrance ^airfrance
javacard/framework/Applet/register()V _airfrance ^airfrance
javacard/framework/APDU/getBuffer()[B _airfrance ^airfrance
javacard/framework/Applet/selectingApplet()Z _airfrance ^airfrance
javacard/framework/ISOException/throwIt(S)V _airfrance ^airfrance
airfrance/internal_class_103/internal_method_103(Ljavacard/framework/APDU)V _purse ^airfr
Filter
View Selected
Edit Selected
Delete Selected
Progress Status
Step 1: Configure Package Names
Step 2: Configure Ambient File
Step 3: Confirm Analysis Setup
<< Back
Next >>

JCSI tool


Java Card Information Flow Verifier 1.1
DeCAP Tool
File Edit Tools Help
airfrance
airfrance.cap
header_component
directory_component
applet_component
import_component
constant_pool_component
class_component
method_component
tag: 7
size: 610
handler_count: 0
Exception_Handler_Info[] (0)
Method_Info[] (12)
method_1([B,S,B):V
flags: 0
max_stack: 5
nargs: 3
(args & return value)
bytecode:
0: new 4
3: dup
4: aload_0
5: sload_1
6: sload_2
7: invokespecial 3
10: pop
11: return
Loading project complete, capfile analisys can be perform


Report for method number 7
Method name: method_294():void
Analysis result: Failed on 2 instruction(s):
117: invokestatic 14
-> level{airfrance, purse}(max={ airfrance, rentacar})
138: invokeinterface 1, 15, 0
-> level{airfrance, purse ,rentacar } (max={ airfrance, rentacar})
Method body:
0: invokestatic 20 //
javacard/framework/JCSystem.getPreviousContextAID()Ljavacard/framework/AID
3: astore_1
4: aload_1
5: getstatic_a 21
8: sconst_0
9: getstatic_a 21
12: arraylength


Details Window
ANALYSIS DETAILS
Full Log
method_294():void
Failed on 2 instruction(s)
Failed
Report
method_473():short
Failed on 1 instruction(s)
Failed
Report
Verified
Report
Verified

Discussion

- JCSI tool is able to certify applets against secure information flow of sensitive data saved present on the card

- JCSI tool is able to certify an applet before its installation on the card

- Verification based on abstract Interpretation
  - JCSI 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

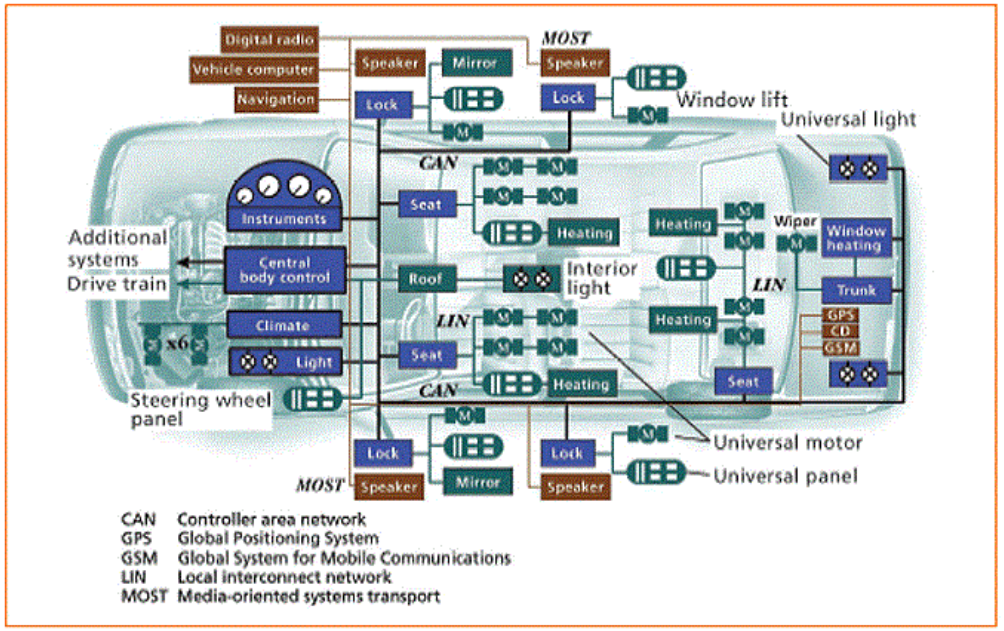
Case studies

Automotive:

Data secure flow in AUTOSAR models

Secure information flow in AUTOSAR models

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



Over 80 different embedded processors, interconnected with each other.

Key ECUs (Electronic Control Unit):

- Engine Control Modul (ECM)
- Electronic Brake Control Module (EBCM)
- Transmission Control Module (TCM)
- Vehicle Vision System (VVS)
- Navigation Control Module (NCM)
- …

A case study: secure information flow in AUTOSAR models

Automotive systems: Mixed-criticality safety critical systems

High criticality
Braking system, Throttle system, …

Low criticality
Infotainment system, ..

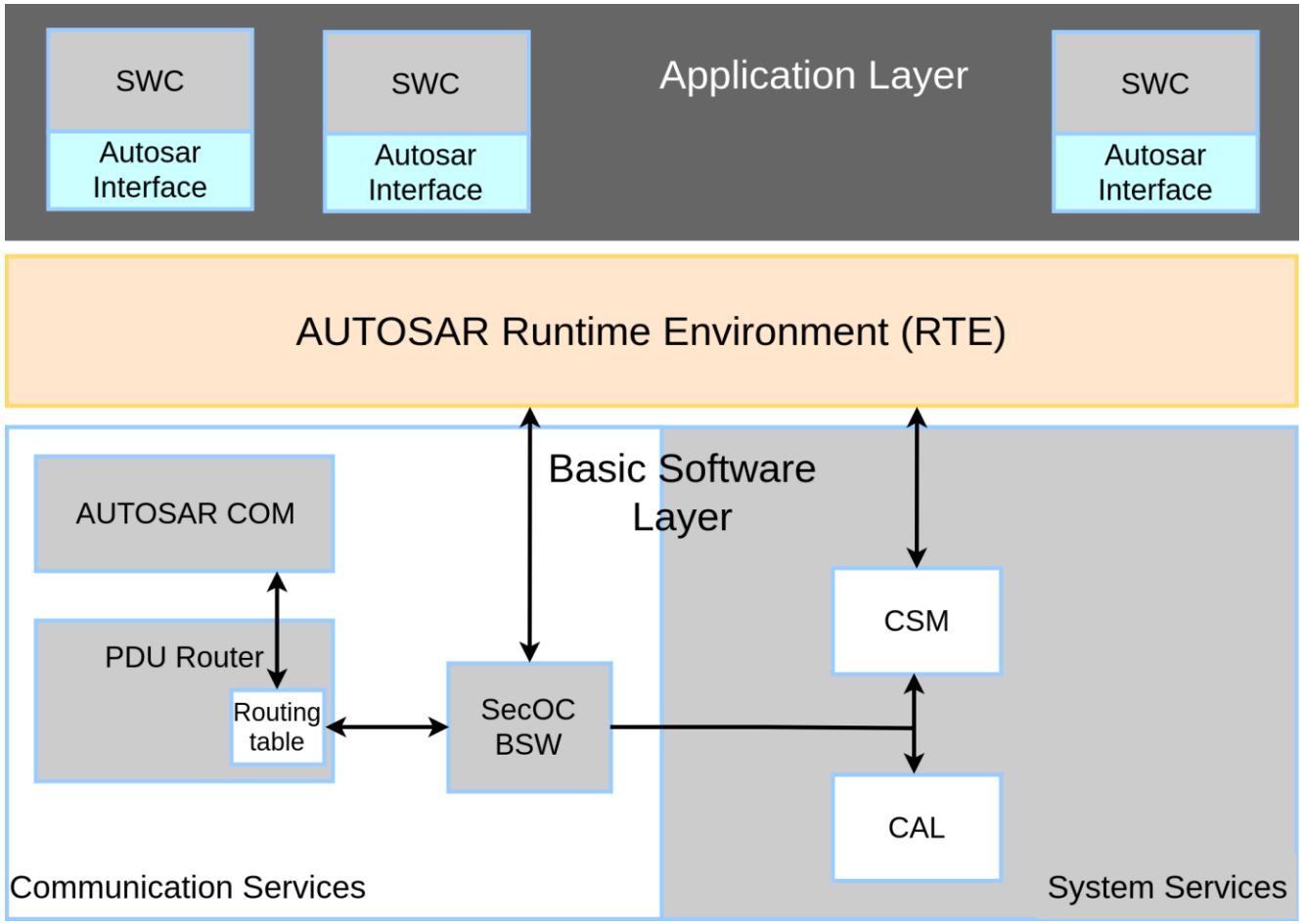
Recent research has shown that it is possible for external intruders to compromise the proper operation of safety functions getting access to the infotainment system.

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

Secure flow in AUTOSAR models

AUTomotive **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

Secure flow in AUTOSAR models

AUTOSAR

- offers a set of security-related services, which provides security functions such as:

  Encryption, Integrity and Authentication of messages exchanged over the car networks

- makes security mechanisms available to the developers in three dierent modules:

  - the Secure On-board Communication (SecOC) module, which routes IPDUs (Interaction layer Protocol Data Units) with security requirements;

  - Abstraction Library (CAL), which implements a library of cryptographic functions;

  - the Crypto Service Manager (CSM), which provides software components with cryptographic functions implemented in software or hardware.

Secure flow in AUTOSAR models

AUTOSAR does not provide any means to specify security requirements at the level of application components, but rather requires the application developers to directly use the standard security services.

EXTENDS AUTOSAR models with a set of security annotations at application level that are assigned to system components and communication links between components

Annotations specify

- Integrity and confidentiality requirements of a link
- The level of trust we have to place in a SWC to provide the expected function

Design code generation features to automatically synthesize the right services to achieve secure communications exploiting security annotations.

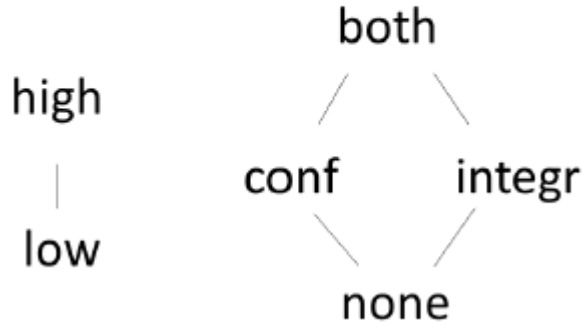
AUTOSAR security levels

- **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**
  - **Trust levels are assigned to ports of SWC**

- **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

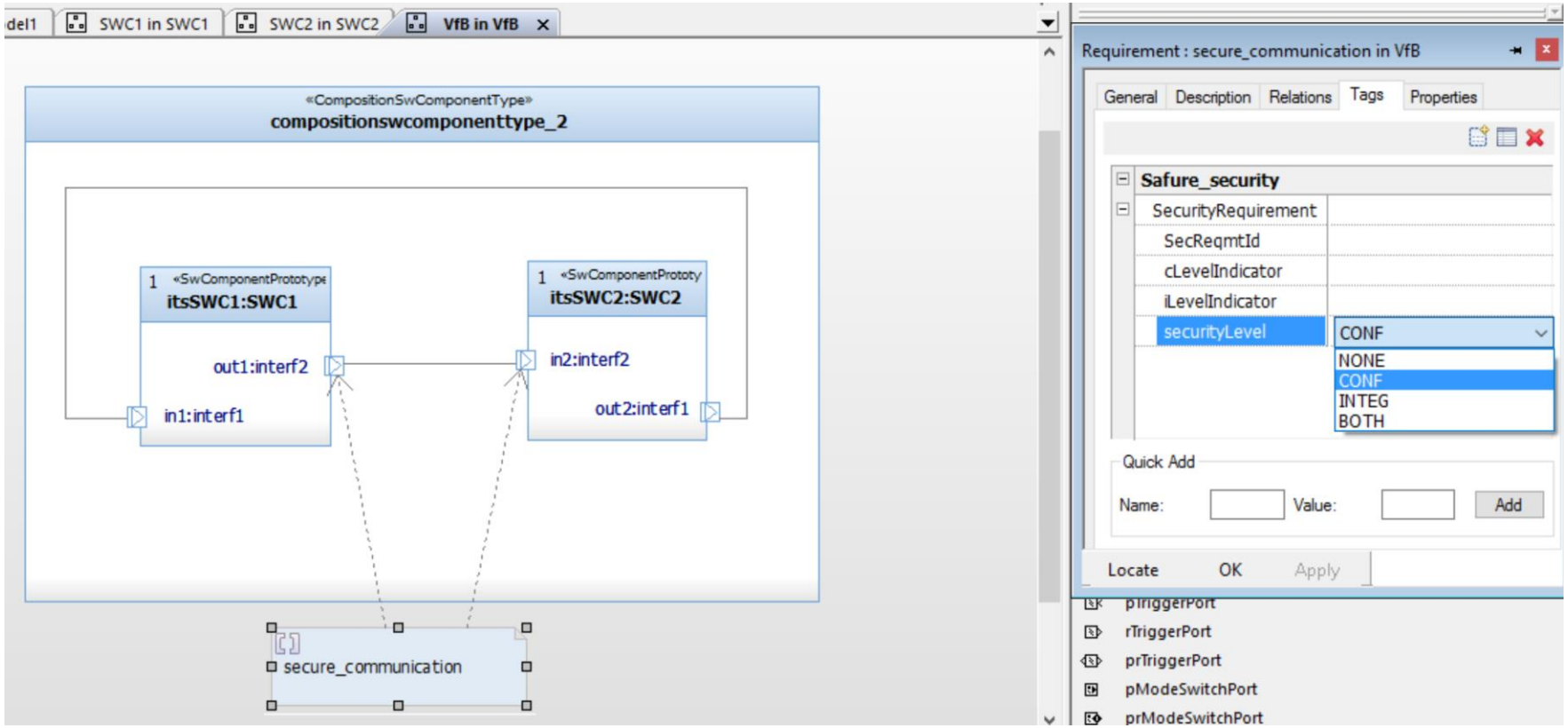
  The proposed security extensions are:
  confidentiality and integrity of the exchanged information
  - The security requirement can assume one of the following values: **none**, **conf**, **integr**, **both**
  - **Security requirement levels are assigned to communication links of SWC**

Lattice of security levels



AUTOSAR extensions in Rhapsody

Security requirements are assigned to swc and communication links



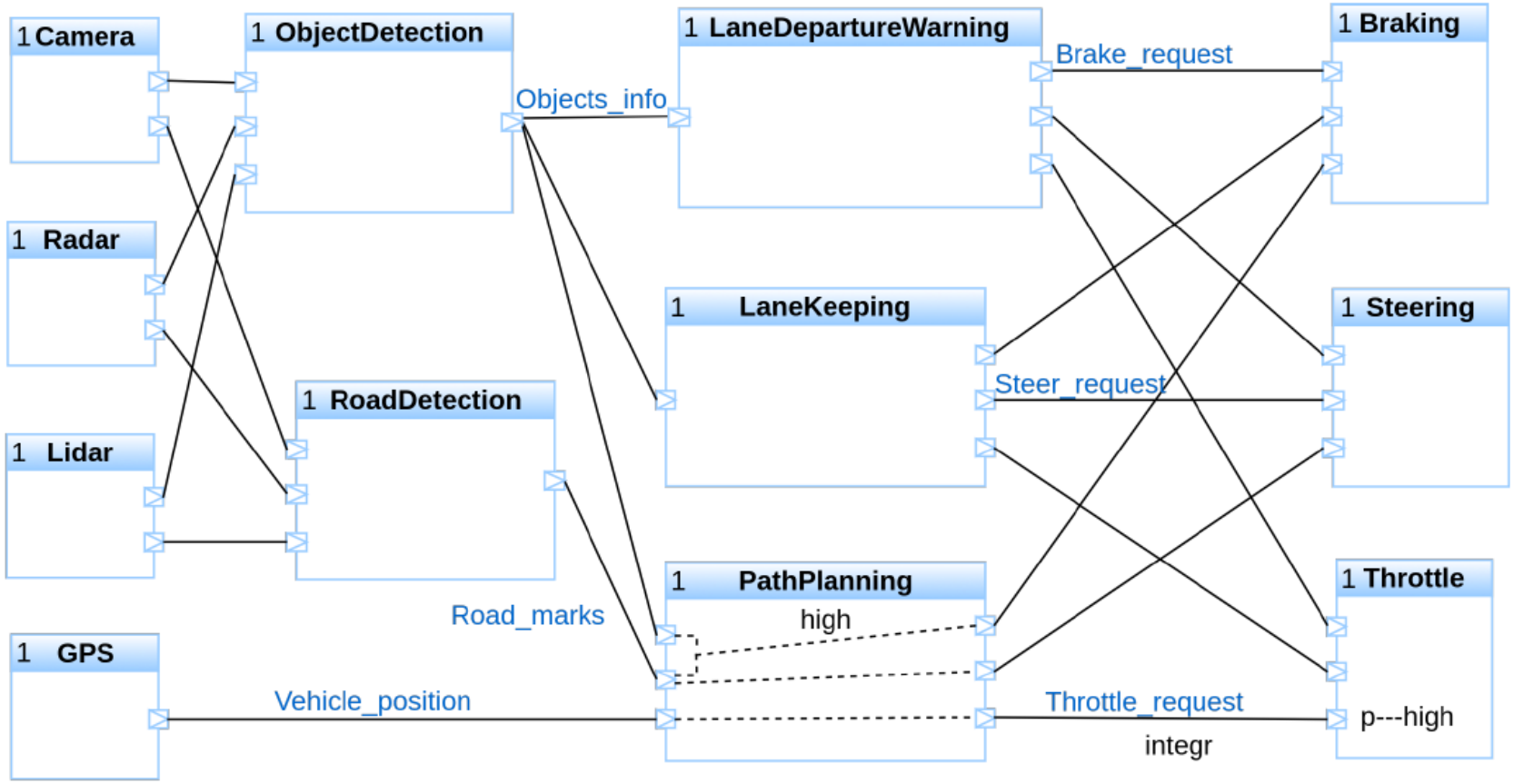
Mixed-criticality

Autonomous driving

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).

- Throttle component is assigned the high trust level;

- Throttle request link is assigned the integrity security requirement.



AUTOSAR models extended with security annotations.
By default the lowest level

Mixed-criticality

Data received by Throttle on the link Throttle_request must satisfy integrity security requirement

however, 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

Secure flow in AUTOSAR models

AUTOSAR provides information of which functional entity (runnable) reads from or writes onto ports of components

-> but using this information for deriving the causality between data and for assigning correctly the annotations demands a huge utilisation of security operations thus introducing performance penalties

assigns integrity to all links **directly or indirectly connected** to Throttle_request

critical issue in the automotive domain where software components run on a resource-limited computer network and have real-time constraints

Security policy

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

More efficient solution: information flow theory of programs can be exploited

The approach

Port dependencies

Given an AUTOSAR model, data written at port pj does not depend on a data read from port pi (pj does not depend on pi for short) if, changing the data at pi, the data written onto pj are always the same.

Deps(p):
ports on which data sent at port p depends

Abstract interpretation: execution of the runnables using port levels instead of real data

Deps(p) = {p, p1, …, pk}

over-approximation of the set of ports on which data sent at port p depends

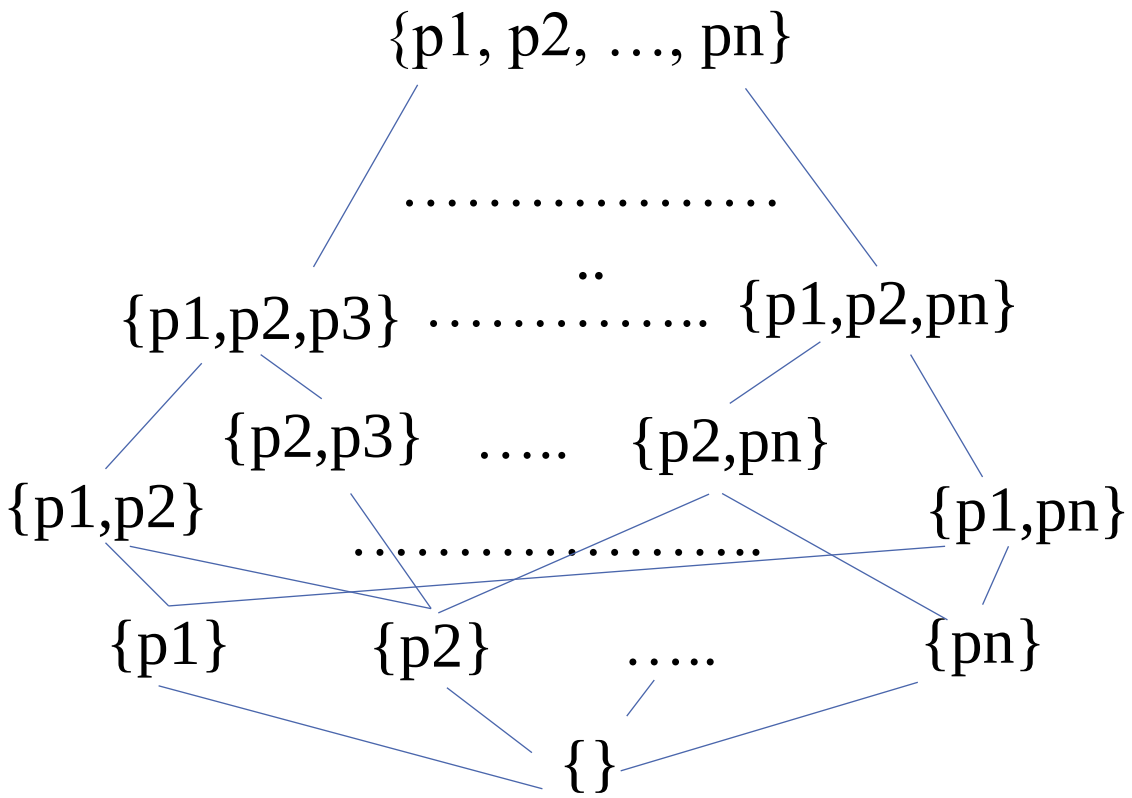
Information flow analysis: Deps(p)

Levels

- the set of identifiers of ports of SWCs

Let p1, p2 and p3 be the ports of the SWCs.

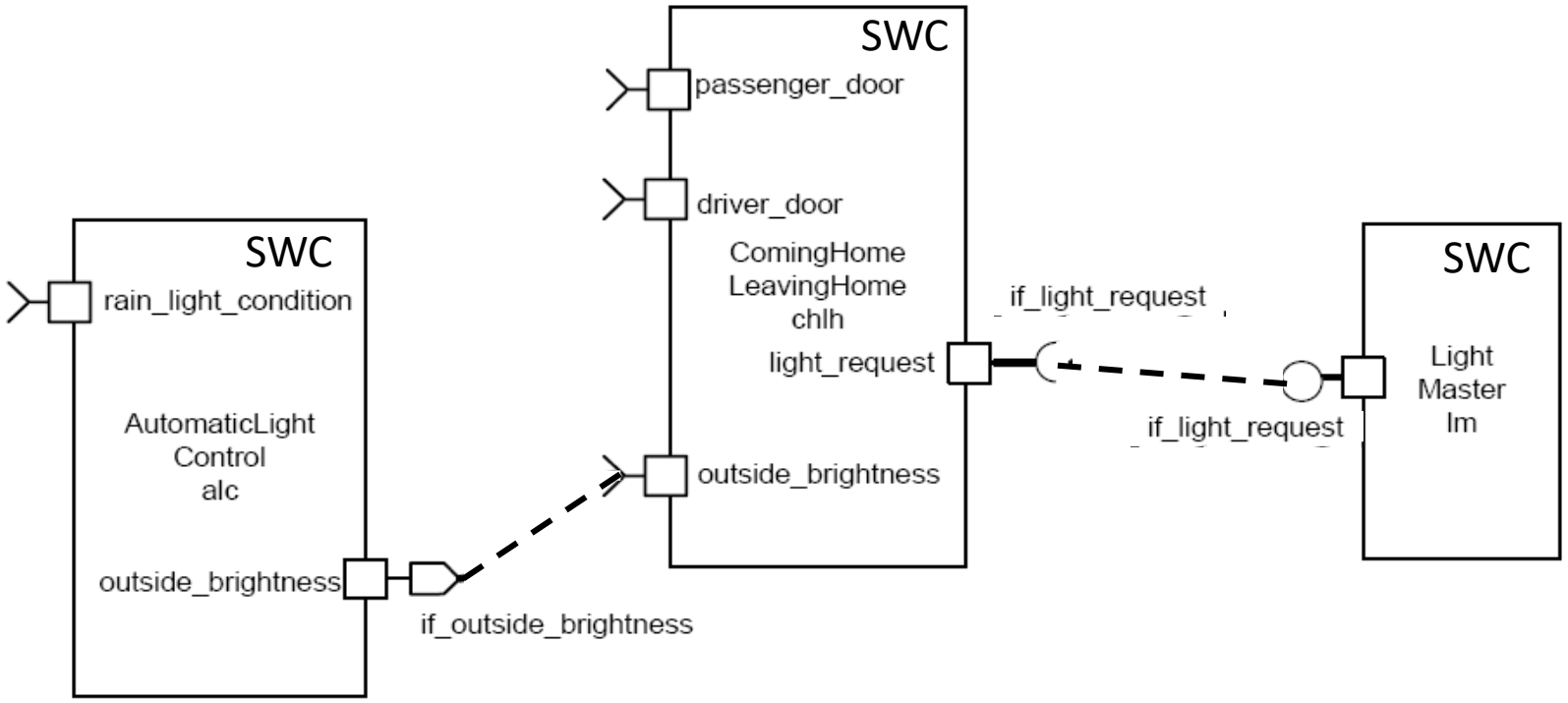
The lattice of levels is the following



Static analysis. Abstract execution of the code of the SWCs using port levels instead of real data. Abstract rules for analysing informaton flow in programs

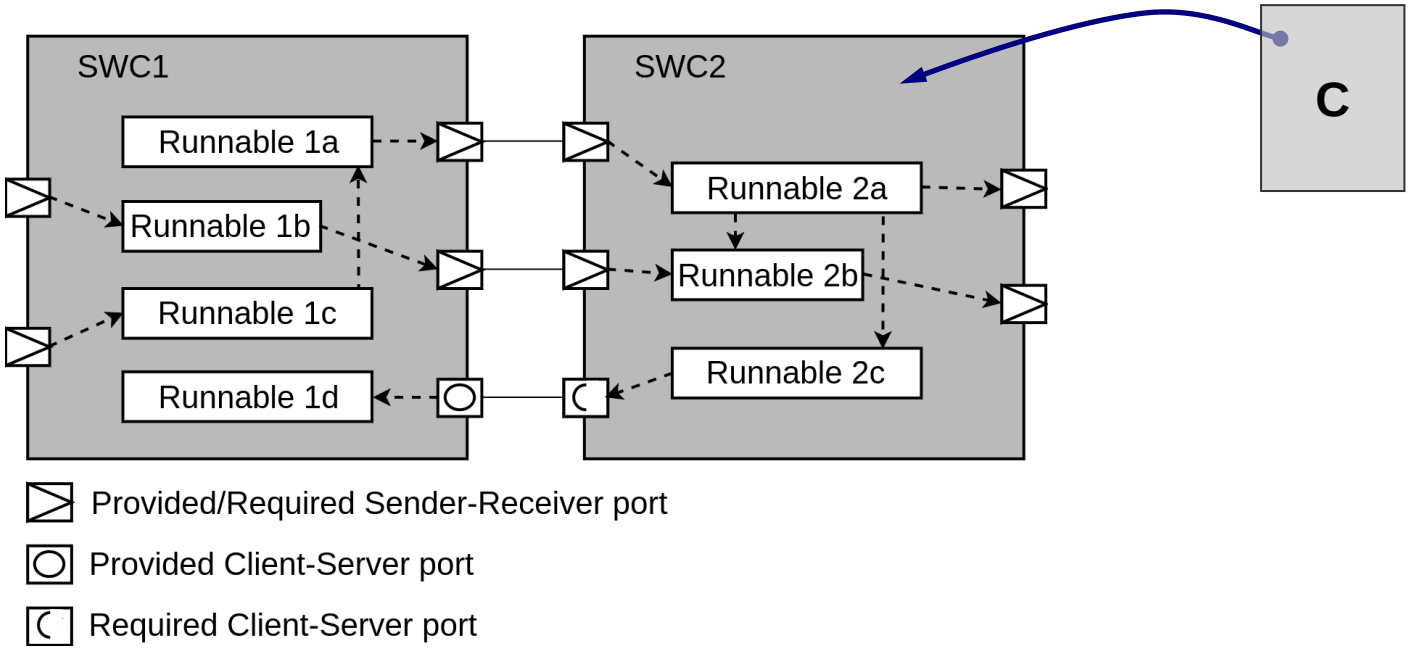
AUTOSAR architecture: SWC

An application in AUTOSAR consists of Software Components interconnected by connectors



SWC: Runnables

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



Static analysis : 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)

The abstract interpreter: EXEC

Analysis of a SWC

- Each runnable is executed starting from the abstract memory, a calling environment and the context file, by 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 PORT is a variable.

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

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

The analysis

- The analysis of an AUTOSAR model is based on an iterative process that performs the abstract execution of all runnables in R, using the global context file. If during the analysis a level in the global context file changes, all runnables must be re-executed.

- The analysis uses the abstract interpreter EXEC to analyses a single runnable. EXEC performs an abstract execution of the runnable starting from a global context file A and producing a new global context le A0.

- The analysis terminates when, starting from a global context file, all runnables are executed and the global context is not changed.

Context file

During the analysis,

for **each variable**, the context file maintains the maximum dependency level of data recorded in the variable

for each variable v, the entry v; σ

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

for each runnable r, the entry r(σ1, σ2, …, σn) σ; σ'

An initial global context

%Begin Global Context

% global variables of SWCs

gv1 = { }

gv2 = { }

. . . . .

% IRV o f SWCs

irv1 = { }

irv2 = { }

. . . . .

% po r t s o f SWCs

p1 = {p1}

p2 = {p2}

. . . .

% runnabl e s o f SWCs

run (a , { }) { } ; { }

run1 ((b , { }) , (c&, {})) { } ; { }

run2 ( ) { } ; { }

. . . . .

%End Global Context

{ } the lowest level

Iterative analysis

Iterative analysis until fixpoint is reached

A: global context file
R: set of all runnables

$$
\begin{aligned}
&A := A^{\emptyset} \\
&T := R \\
&\texttt{while}(T \neq \emptyset) \\
&\quad \texttt{select}\ r \in T \\
&\quad T := T - \{r\} \\
&\quad A' := EXEC(r, A) \\
&\quad \texttt{if}(A' \neq A) \\
&\qquad A := A' \\
&\qquad T := R
\end{aligned}
$$

The analysis scales up, since runnables of AUTOSAR SWCs are analysed separatly

EXEC abstract interpreter

A runnable is executed by an abstract interpreter EXEC which takes as input the CFG of the runnable. In the CFG the instructions are grouped in Basic Blocks (bb).

Each type of instruction is assigned an abstract execution rule. The abstract execution of an instruction, updates the local context (M;Env), and the global context A. EXEC examines one bb at a time and abstractly executes each instruction of the block, and propagates the updates ((M;Env) and A after the execution of the instructions) to successors blocks.

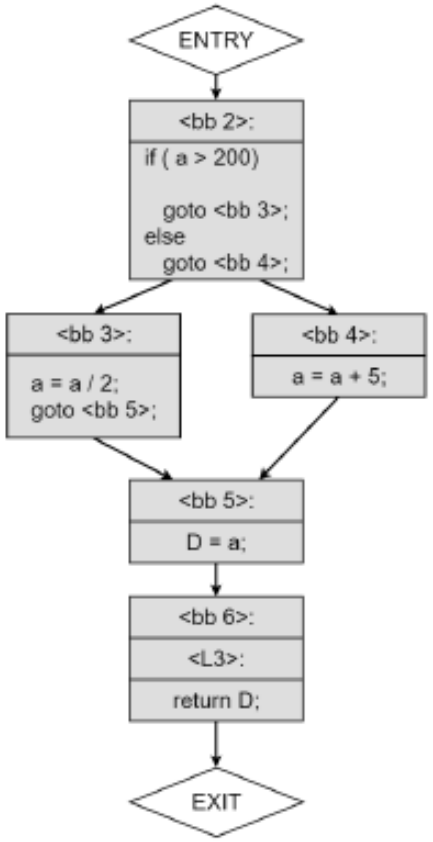
Instructions in the scope of conditional block, are executed under the implicit flow of the condition of the control instruction in the conditional block.

AUTOSAR runnable

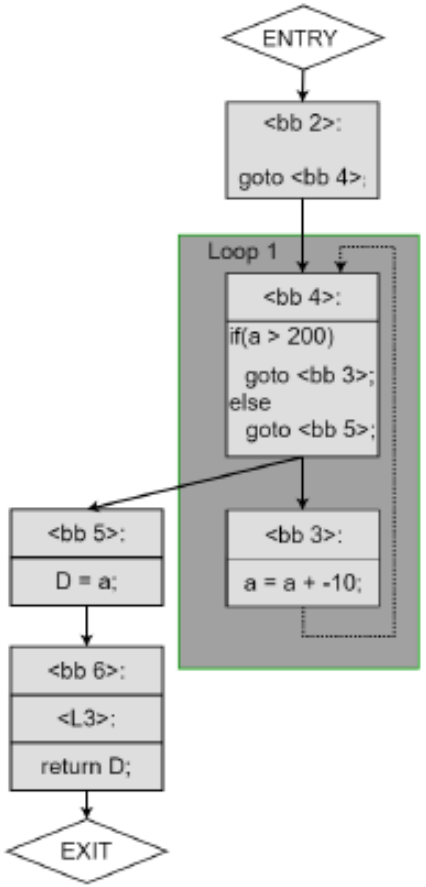
A runnable is modelled by its CFG

```
int runnable1 (int a) {
   if (a>200)   a = a/2;
     else   a= a+5;
   return a;
}
int runnable2 (int a) {
   while (a>200)
        a = a -10;
   return a;
}
```

Visual Studio C++
gcc *-fdump-tree-cfg*

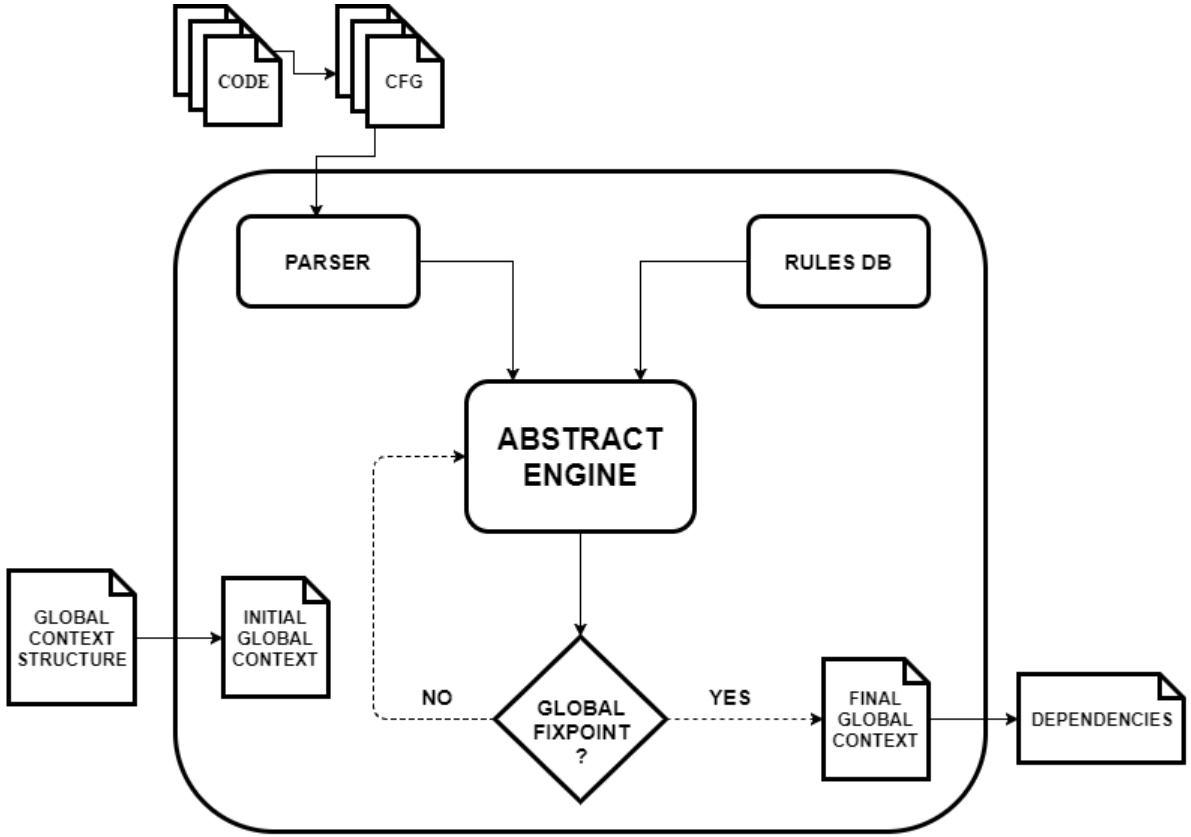


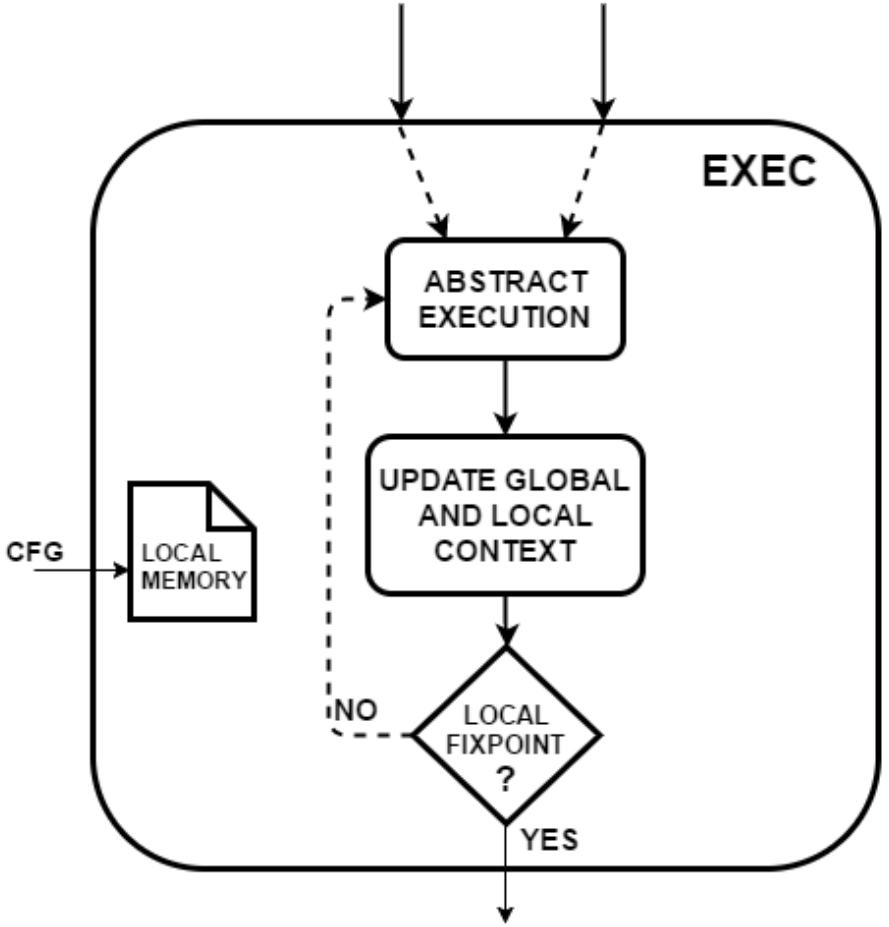
runnable1



runnable2

Tool architecture





Runnable 2

Block	**Code**	**Succ.**	**Scope**
Q_{entry}		2	
Q_2	goto bb4;	4	
Q_3	a = a-10;	4	
Q_4	if (a > 200) goto bb3 else goto bb5	3, 5	3
Q_5	d = a	6	
Q_6	return d	exit	
Q_{exit}			

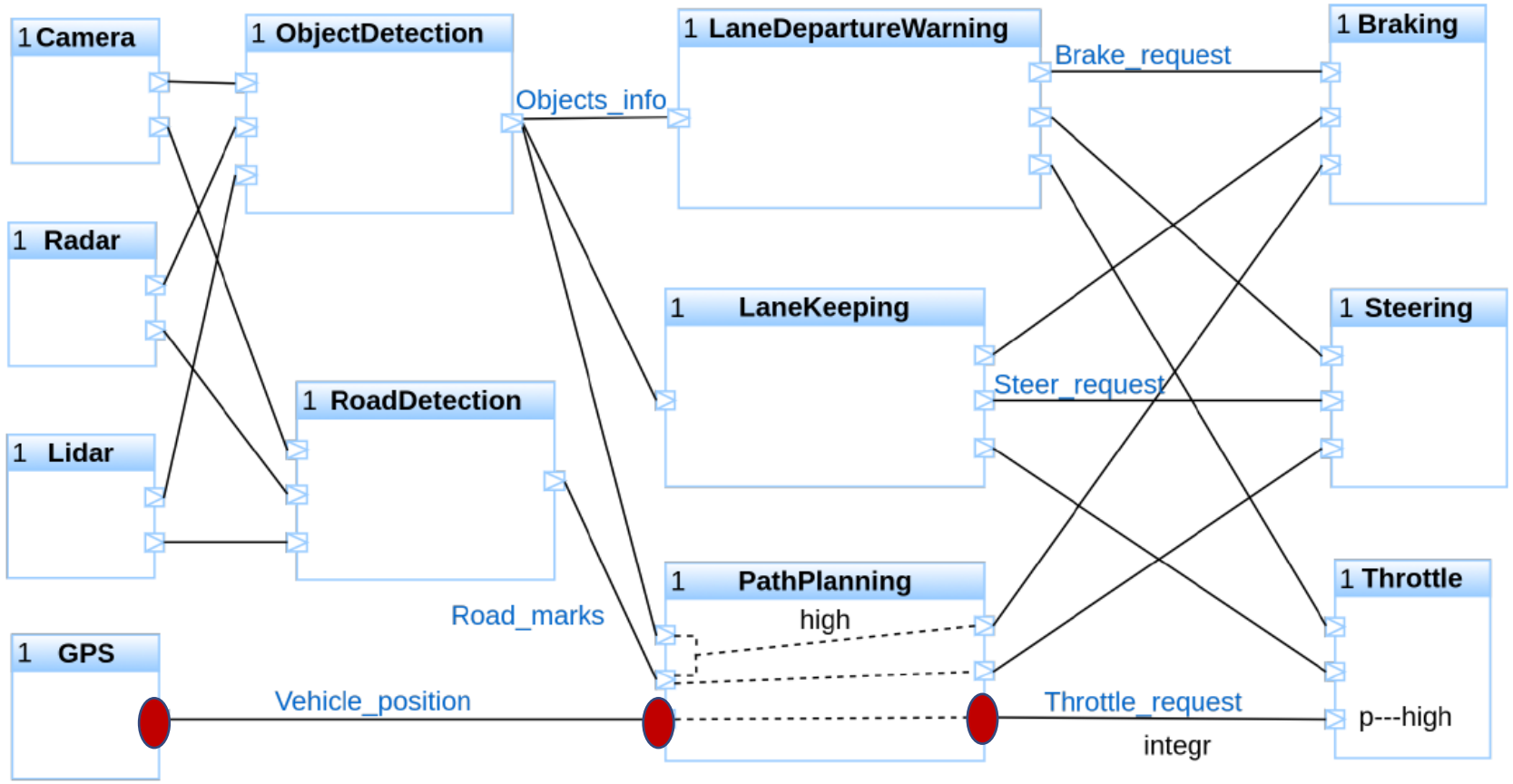
Initial local context

Block	**Memory**		**Env**
Q_{entry}	$(a, p2)$	$(d, \emptyset)$	$p1$
Q_2	$(a, \emptyset)$	$(d, \emptyset)$	$p1$
Q_3	$(a, \emptyset)$	$(d, \emptyset)$	$p1$
Q_4	$(a, \emptyset)$	$(d, \emptyset)$	$p1$
Q_5	$(a, \emptyset)$	$(d, \emptyset)$	$p1$
Q_6	$(a, \emptyset)$	$(d, \emptyset)$	$p1$
Q_{exit}	$(a, \emptyset)$	$(d, \emptyset)$	$p1$

Local context after first iteration

Block	**Memory**		**Env**
Q_{entry}	$(a, p2)$	$(d, \emptyset)$	$p1$
Q_2	$(a, p2)$	$(d, \emptyset)$	$p1$
Q_3	$(a, \{p1, p2\})$	$(d, \emptyset)$	{p1, p2}
Q_4	$(a, \{p1, p2\})$	$(d, \emptyset)$	$p1$
Q_5	$(a, \{p1, p2\})$	$(d, \emptyset)$	$p1$
Q_6	$(a, \{p1, p2\})$	$(d, \{p1, p2\})$	$p1$
Q_{exit}	$(a, \{p1, p2\})$	$(d, \{p1, p2\})$	$p1$

Information flow analysis: Deps(p)



From Deps(p) to security levels

Algorithm for data security of link l

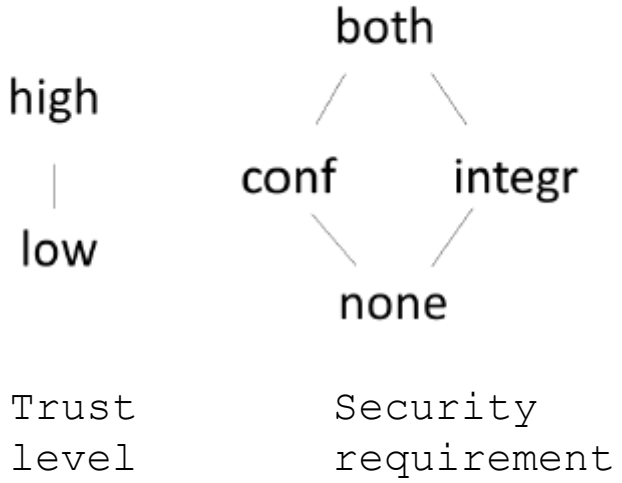
l = <trust level, security requirement>

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

1. $\langle \delta_l, \mu_l \rangle = \langle high, both \rangle$
2. $\forall p \in Deps(p_i)$:
   $\delta_l = glb(\delta_l, \texttt{trustlevel}(cmp(p)))$
3. $\forall l' = (q, q') \mid q, q' \in Deps(p)$:
   $\mu_l = glb(\mu_l, \texttt{securityreq}(l'))$

glb: greatest lower bound
cmp(p) : component of port p

Lattice of security levels



AUTOSAR data secure flow

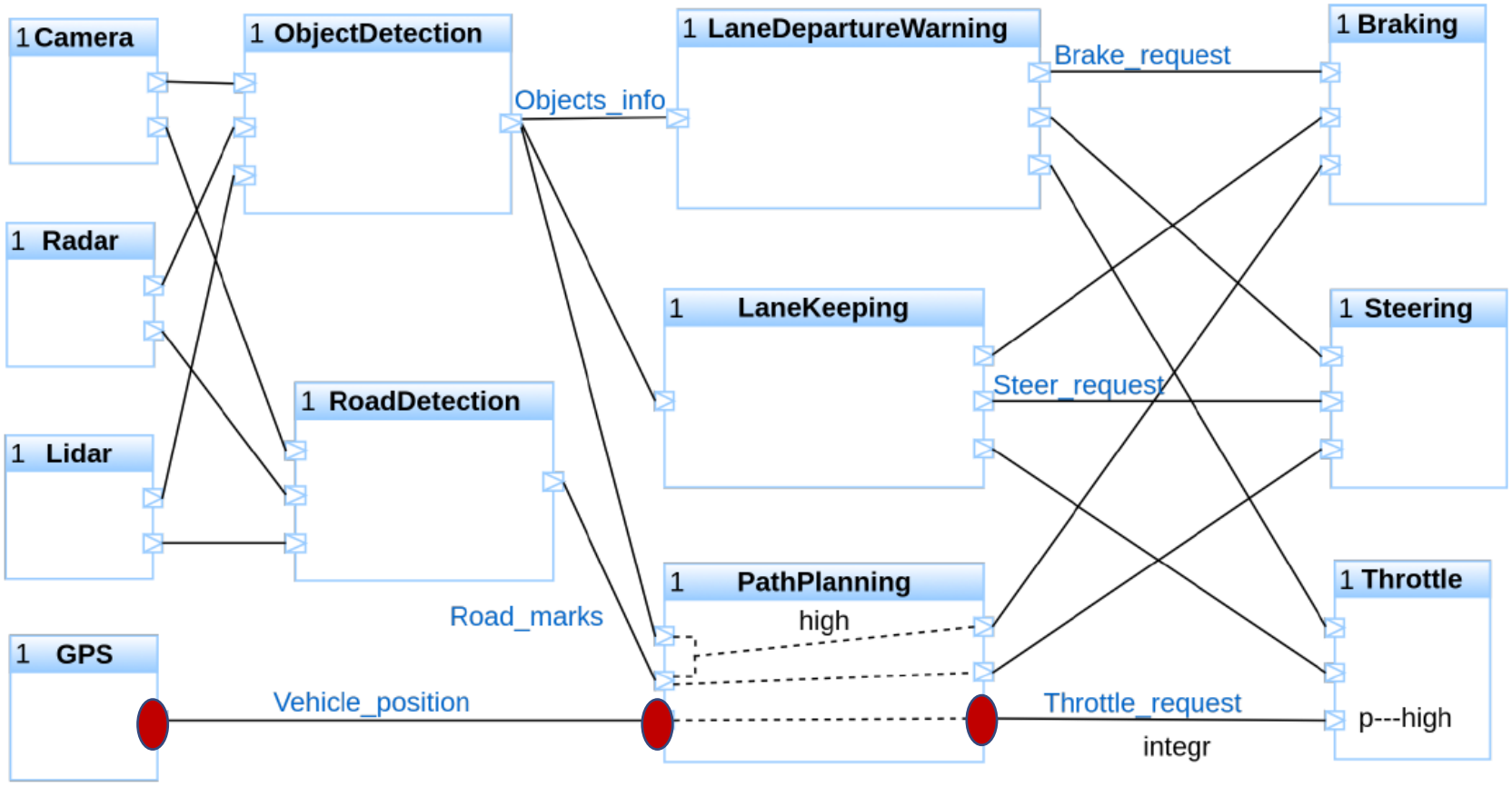
Definition (Data secure flow property) *Given an AUTOSAR model with security annotations, the model satisfies the data secure flow property if for each link* $l = (p_i, p_j) \in L$*:*

$$\delta_l \not\sqsubseteq \texttt{trustlevel}(c, p_j) \wedge \mu_l \not\sqsubseteq \texttt{securityreq}(l)$$

where $c = cmp(p_j)$ *and,* δ_l *and* μ_l *are the lowest trust level and the lowest security requirement of data sent onto link* l *.*

AUTOSAR data secure flow

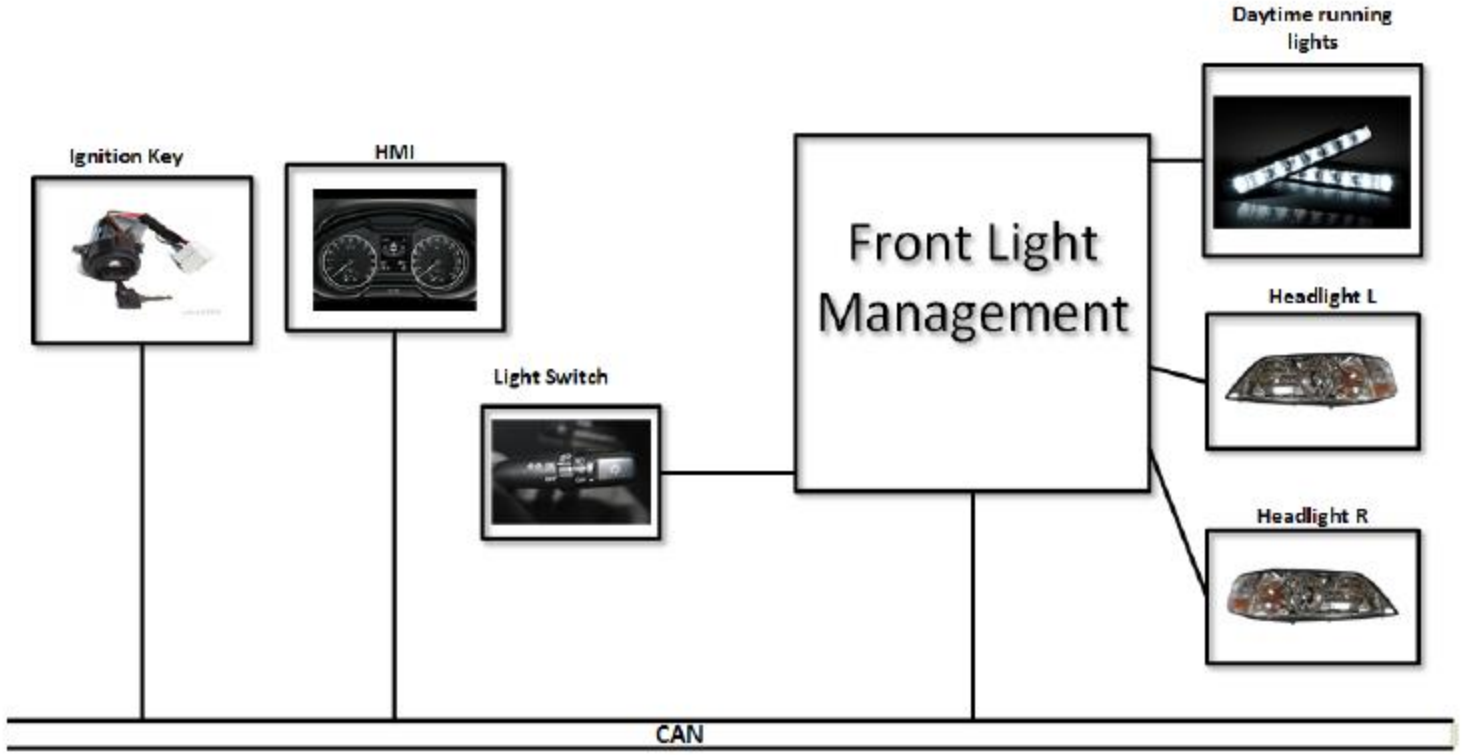
Security annotations: **Throttle_request = <high, integr>**



Data secure flow is not satisfied

Throttle_request = <low, none>

An example: Front Light Manager



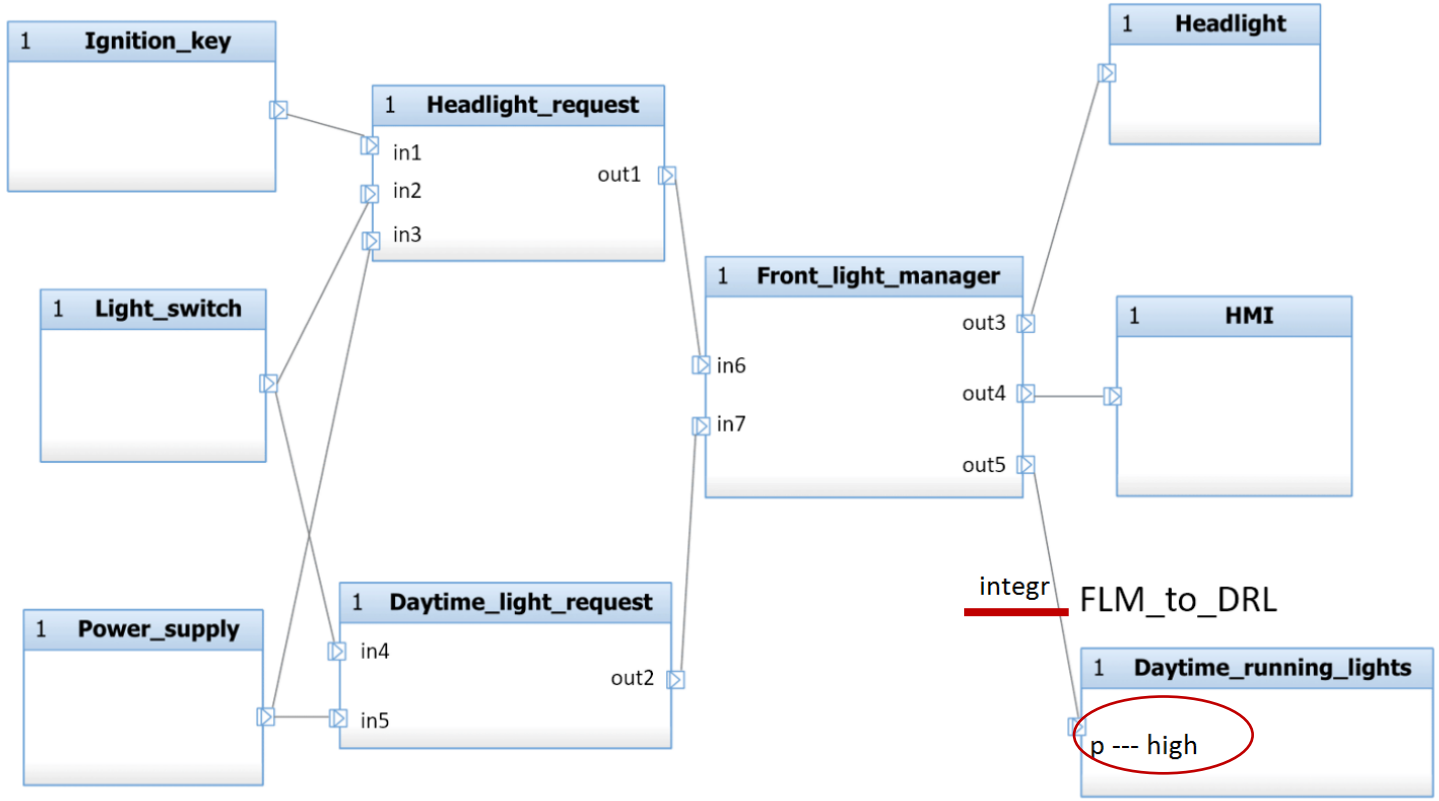
Safety Use Case Example, release 4.2.2. http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and security/auxiliary/AUTOSAR_EXP_SafetyUseCase.pdf

Front Light Manager

Security annotations: FLM_to_DRL = <high, integr>

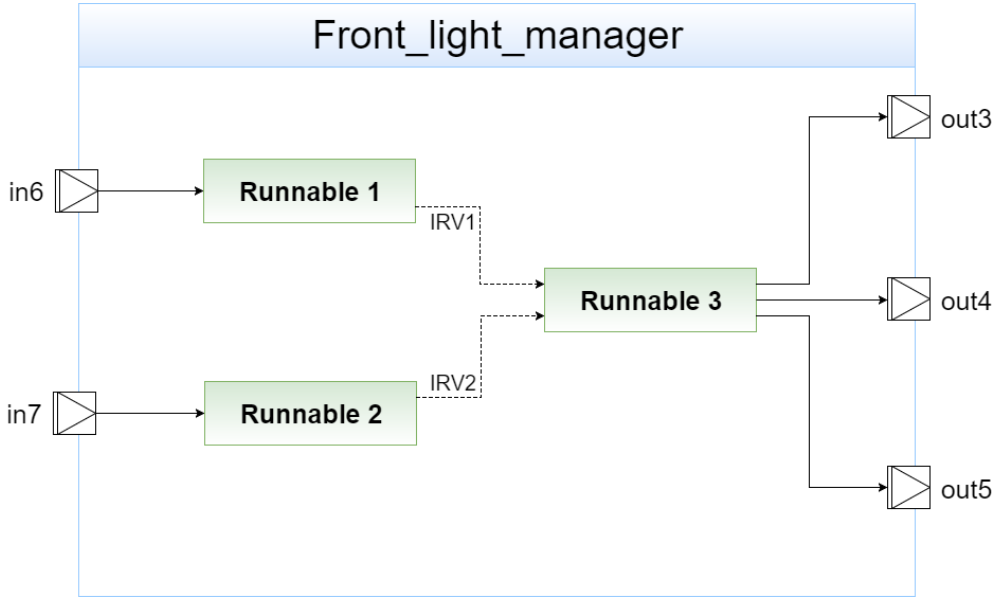
Data secure flow is not satisfied

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

Global Context le at the beginning of the analysis

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	0	0	0	0	0	0	0	0	0	0	0	0
FLM_IRV2	0	0	0	0	0	0	0	0	0	0	0	0
DLR_IRV1	0	0	0	0	0	0	0	0	0	0	0	0
HR_IRV1	0	0	0	0	0	0	0	0	0	0	0	0
HR_IRV2	0	0	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	0	0	0	0	0	1	0	0	0	0	0	0
in7	0	0	0	0	0	0	1	0	0	0	0	0
out1	0	0	0	0	0	0	0	1	0	0	0	0
out2	0	0	0	0	0	0	0	0	1	0	0	0
out3	0	0	0	0	0	0	0	0	0	1	0	0
out4	0	0	0	0	0	0	0	0	0	0	1	0
out5	0	0	0	0	0	0	0	0	0	0	0	1

Analysis of dependencies

Global Context le at the end of the analysis

Global Environment:

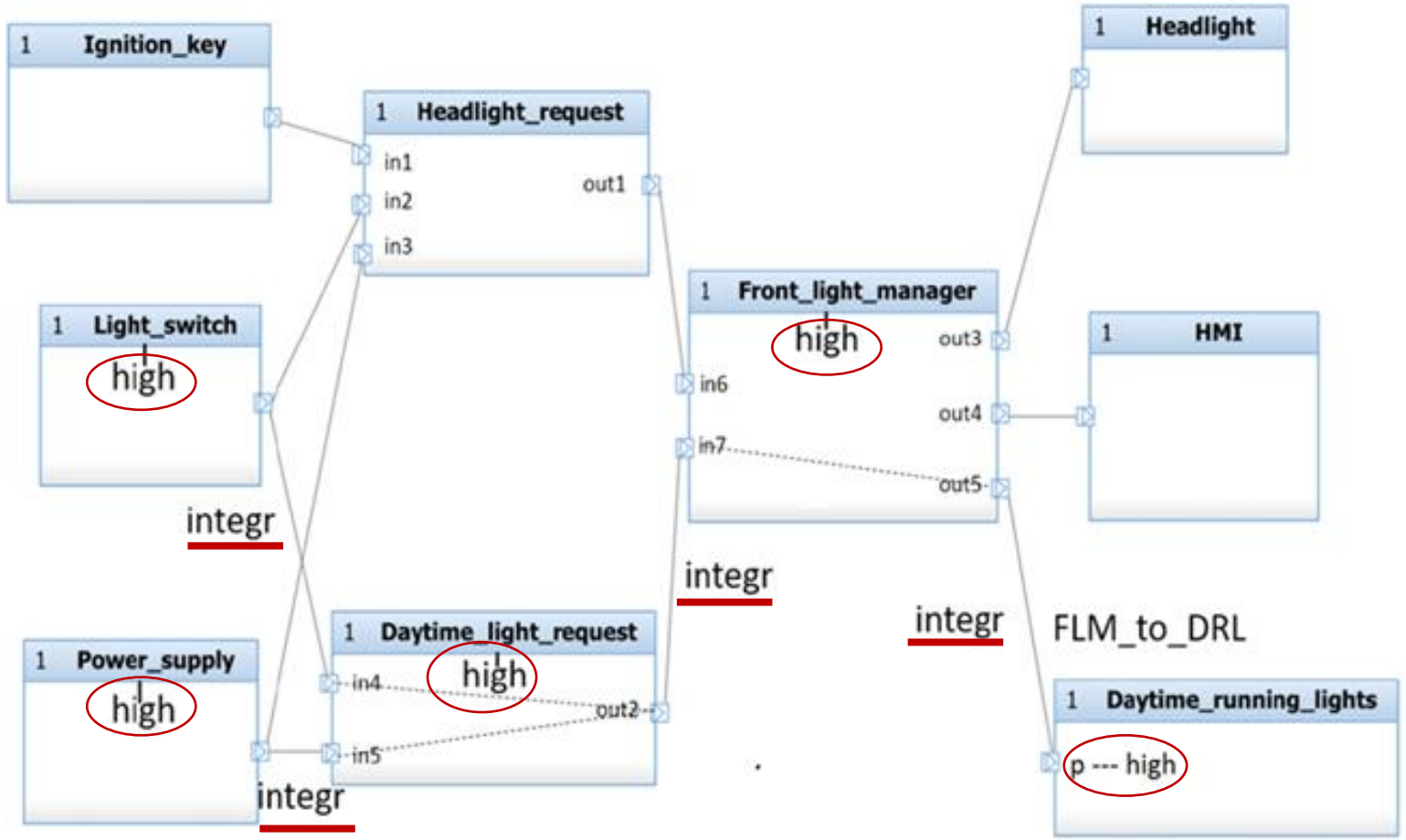
	in1	in2	in3	in4	in5	in6	in7	out1	out2	out3	out4	out5
HR_voltage_threshold1	0	0	0	0	0	0	0	0	0	0	0	0
HR_voltage_threshold2	0	0	0	0	0	0	0	0	0	0	0	0
DLR_voltage_threshold1	0	0	0	0	0	0	0	0	0	0	0	0
DLR_voltage_threshold2	0	0	0	0	0	0	0	0	0	0	0	0
FLM_IRV1	1	1	1	0	0	1	0	1	0	0	0	0
FLM_IRV2	0	0	0	1	1	0	1	0	1	0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

Generating a correctly annotated model

Using the results of the analysis a correctly annotated model can be generated

For example: the output port of Front light manager connected to theDaytime_running_lights (out5 in our implementation) does not depend on the input port connected to the Headlight request component (in6 in our implementation)



Data secure flow is satisfied

AUTOSAR data secure flow

- At the end of the analysis the global context file records the dependencies for ports of all the SWCs.
- The approach is conservative, in the sense that all possible dependencies for any real execution of the runnables are detected.
- False dependencies are possible, since, for example, in the abstract analysis all branches of control instructions are executed, even those that in real execution would have never been executed