Checking Secure Interactions of Smart Card Applets: extended version*

[Published in *Journal of Computer Security* **10**(4):369–398, 2002.]

P. Bieber[1], J. Cazin[1], P. Girard[2], J.-L. Lanet[2], V. Wiels[1], and G. Zanon[1]

[1] ONERA-CERT/DTIM
BP 4025, 2 avenue E. Belin,
F-31055 Toulouse Cedex 4, France
{bieber,cazin,wiels,zanon}@cert.fr
[2] GEMPLUS
avenue du pic de Bertagne, 13881 Gémenos cedex, France
{Pierre.GIRARD,Jean-Louis.LANET}@gemplus.com

Abstract. This paper presents an approach enabling a smart card issuer to verify that a new applet securely interacts with already downloaded applets. A security policy has been defined that associates levels to applet attributes and methods and defines authorized flows between levels. We propose a technique based on model checking to verify that actual information flows between applets are authorized. We illustrate our approach on applets involved in an electronic purse running on Java enabled smart cards.

Introduction

A new type of smart cards is getting more and more attractive: multiapplication smart cards. The main characteristics of such cards are that applications can be loaded after the card issuance and that several applications run on the same card. A few operating systems have been proposed to manage multiapplication smart cards, namely Java Card [22], Multos [15] and more recently Windows for Smart Cards [14]. In this paper, we will focus on Java Card. Following this standard, applications for multiapplication smart cards are implemented as interacting Java applets.

Multiapplication smart cards involve several participants: the card provider, the card issuer that proposes the card to the users, application providers and card holders (users). The card issuer is usually considered responsible for the security of the card. The card issuer does not trust application providers: applets could be malicious or simply faulty.

As in a classical mobile code setting, a malicious downloaded applet could try to observe, alter, use information or resources it is not authorized to. Of course,

* The Pacap project was partially funded by MENRT décision d'aide 98.B.0251

a set of JavaCard security functions were defined that severely restrict what an applet can do. But these functions do not cover a class of threats we call illicit applet interactions that cause unauthorized information flows between applets.

Our goal is to provide techniques and tools enabling the card issuer to verify that new applets interact securely with already loaded applets.

The first section introduces security concerns related to multiapplication smart cards. The second section of the paper describes the electronic purse functionalities and defines the threats associated with this application. The third section presents the security policy and information flow property we selected to verify that applets interact securely. The fourth section shows how we verify secure interaction properties on the applets byte code. The fifth section relates our approach to other existing work.

1 Security Concerns

1.1 Java Card security mechanisms

Security is always a big concern for smart cards but it is all the more important with multiapplication smart cards and post issuance code downloading.

Opposed to monoapplicative smart cards where Operating System and application were mixed, multiapplication smart cards have drawn a clear border between the operating system, the virtual machine and the applicative code. In this context, it is necessary to distinguish the security of the card (hardware, operating system and virtual machine) from the security of the application. The card issuer is responsible for the security of the card and the application provider is responsible for the applet security, which relies necessarily on the security of the card.

The physical security is obtained by the smart card media and its tamper resistance. The security properties that the OS guarantees are the quality of the cryptographic mechanisms (which should be leakage resistant, i.e. resistant against side channel attacks such Differential Power Analysis [11]), the correctness of memory and I/O management.

A Java Card virtual machine relies on the type safety of the Java language to guarantee the innocuousness of an applet with respect to the OS, the virtual machine [12], and the other applets. However this is guaranteed by a byte-code verifier which is not on board, so extra mechanisms have been added. A secure loader (like OP [24]) checks before loading an applet that it has been signed by an authorized entity (namely the card issuer). Even if an unverified applet is successfully loaded on the card, the card firewall [23], which is part of the virtual machine, will still deny to an aggressive applet the possibility to manipulate data outside its memory space.

To allow the development of multiapplication smart cards, the Java Card has introduced a new way for applets to interact directly. An applet can invoke another applet method through a shared interface. An applet can decide to share or not some data with a requesting applet based on its identifier.

1.2 Applets providers and end users security needs

Applet providers have numerous security requirements for their applications. Classical one are secret key confidentiality, protection from aggressive applets, integrity of highly sensitive data fields such as electronic purse balances, etc. These requirements are widely covered by the existing security mechanisms at various levels from silicon to secure loaders and are not covered in this paper. However, new security requirements appear with the growing complexity of applets and their ability to interact directly with other applets.

Particularly, application providers do not want information to flow freely inside the card. They want to be sure that the commercial information they provide such as marketing information and other valuable data (especially short term ones such as stock quotes, weather forecast and so on) will not be retransmitted by applets without their consent. For example, marketing information will be collected by a loyalty applet when an end user buys some goods at a retailer. This information will be shared with partner applets but certainly not with competitor applets. However it will certainly be the case that some partners applets will interact with competitor applets. As in the real world trust between providers should not be transitive and so should be the authorized information flows.

It is also expected that pay services inside the card, as information, will not be able to be re-sold by rogue applets. In a very constrained environment, services, even as little as an array sorting method can be valuable as they can help other applets to save precious non volatile memory space and thus money.

Finally, the end user security requirement will be related mainly with privacy. As soon as medical data or credit card record are handled by the card and transmitted between applets great care should be taken with the information flow ([6] details such a privacy threat).

1.3 Applet Certification

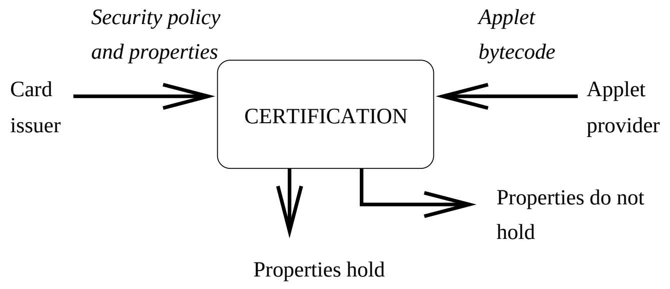

Fig. 1. Applet Certification

Applet providers and end users cannot control that their information flow requirements are enforced on the card because they do not manage it. Our goal is to provide techniques and tools enabling the card issuer to verify that new applets respect existing security properties defined as authorized information flows. The approach is described on figure 1.

If the applet provider wants to load a new applet on a card, she provides the bytecode for this applet. The card issuer has a security policy for the card and security properties that must be satisfied. This security policy should take into account the requirements of applet providers and end users. We provide techniques and tools to decide whether the properties are satisfied by the new applet (these techniques are applied on the applet bytecode). If the properties hold, the applet can be loaded on the card; if they do not hold, it is rejected.

2 Electronic Purse

2.1 Electronic Purse Functionalities

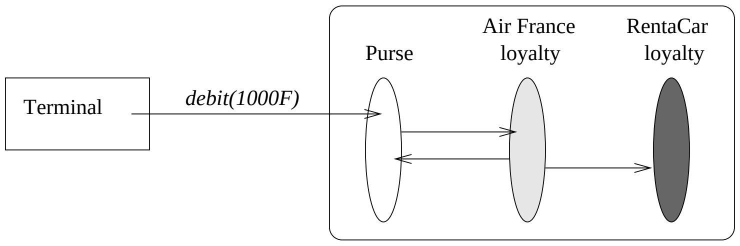

Fig. 2. An electronic purse

A typical example of a multiapplication smart card is an electronic purse with one purse applet and two loyalty applets: a frequent flyer (Air France) application and a car rental (RentaCar) loyalty program. The purse applet manages debit and credit operations and keeps a log of all the transactions. When the card owner wants to subscribe to a loyalty program, the corresponding loyalty applet is loaded on the card. This applet must be able to interact with the purse to get to know the transactions made by the purse in order to update loyalty points according to these transactions. For instance, the Air France applet will add miles to the account of the card owner whenever an Air France ticket is bought with the purse. The card owner can use these miles to buy a discounted Air France ticket. Agreements may also exist between loyalty applets to allow exchanges of points. For instance, loyalty points granted by RentaCar could be summed with Air France miles to buy a discounted ticket.

The electronic purse has been chosen as case study for our project. It has been implemented in Java by Gemplus.

2.2 Electronic Purse Threats

We suppose that all the relevant Java and JavaCard security functions are used in the electronic purse. But these functions do not cover all the threats. We are especially interested in threats particular to multiapplication smart cards like illicit applet interactions. An example of illicit interaction in the case of the electronic purse is described on figure 3.

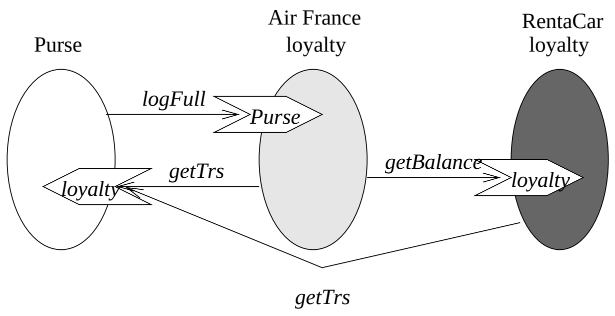

Fig. 3. Applet Interactions

The purse applet has a shared interface for loyalty applets to get their transactions and the loyalty applet has a shared interface for partner loyalty applets to get loyalty points.

A "logFull" service is proposed by the purse to the loyalty applets: when the transaction log is full, the purse calls the $logFull$ method of the loyalty applets that subscribed to the service to warn them that the log is full and they should get the transactions before some of them are erased and replaced by new ones. We suppose the Air France applet subscribed to the logFull service, but the RentaCar applet did not. When the log is full, the purse calls the $logFull$ method of the Air France applet. In this method, the Air France applet gets transactions from the purse but also wants to update its extended balance that contains its points plus all the points it can get from its loyalty partners. To update this extended balance, it calls the *getBalance* method of the RentaCar loyalty applet. In that case, the car rental applet can guess that the log is full when the Air France applet calls its *getBalance* method and thus get the transactions from the purse. There is a leak of information from the Air France applet to the RentaCar one and we want to be able to detect such illicit information flows. This illicit behaviour would not be countered by the applet firewall as all the invoked methods belong to shared interfaces.

3 Multiapplication Security Policy

To implement the applet certification approach, we first have to choose a security policy adapted to multiapplication cards and associated security properties.

3.1 Security policy

We propose to use a multilevel security policy [6] that was designed for multiapplication smart cards. Each applet provider is assigned a security level and we consider special levels for shared data. On the example of the electronic purse, we have a level for each applet: AF for Air France, P for purse and RC for RentaCar and levels for shared data: $AF + RC$ for data shared by Air France and RentaCar, $AF + P$ for data shared by Air France and purse, etc. The relation between levels $\preceq$ is used to authorize or forbid information flows between applets. In the policy we consider, $AF + P \preceq AF$ and $AF + P \preceq P$, this means that information whose level is $AF + P$ is authorized to flow towards information whose level is P or AF. So shared information from Air France and Purse may be received by Air France and Purse applets. To model that applets may only communicate through shared interfaces, direct flows between levels AF, P and RC are forbidden. So we have: $AF \npreceq P$, $P \npreceq AF$, $AF \npreceq RC$, $RC \npreceq AF$, $P \npreceq RC$ and $RC \npreceq P$.

The levels together with the $\preceq$ relation have a lattice structure, so there are a bottom level *public* and a top level *private*.

Figure 4 shows some of the authorized information flows.

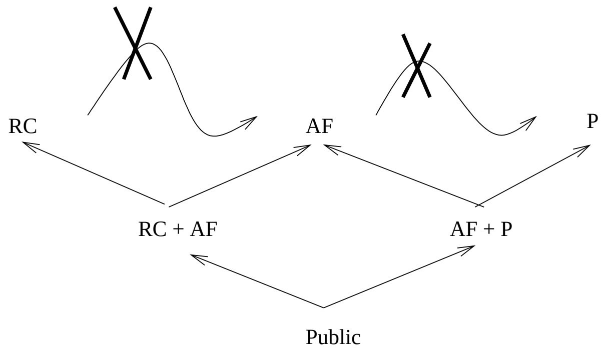

Fig. 4. Authorized information flows

3.2 Security properties

Now we have to define the security properties to be enforced. We have chosen the secure dependency model [1] that applies to systems where applications might, maliciously or involuntarily, communicate confidential information to other applications. Like other information flow models such as non-interference [7], this model ensures that dependencies between system objects cannot be exploited to establish an indirect communication channel. We apply this model to the electronic purse: illicit interactions will be detected by controlling the dependencies between objects of the system.

A program is described by a set of evolutions that associate a value with each object at each date. We note $Ev \subseteq Objects \times Dates \rightarrow Values$ the set of evolutions of a program. The set $Objects \times Dates$ is made of three disjoint subsets: input objects that are not computed by the program, output objects that are computed by the program and are directly observable and internal objects that are not observable. We assume that function lvl associates a security level with input and output objects.

The secure dependency property $SecDep$ requires that the value of output objects with security level l only depends on the value of input objects whose security level is dominated by l:
$\forall o_t \in Output, \forall e \in Ev, \forall e' \in Ev, e \sim_{aut(o_t)} e' \Rightarrow e(o_t) = e'(o_t)$
where $aut(o_t) = \{o'_{t'} \in Input \mid t' < t, lvl(o'_{t'}) \preceq lvl(o_t)\}$ and $e \sim_{aut(o_t)} e'$ iff $\forall o'_{t'} \in aut(o_t), e(o'_{t'}) = e'(o'_{t'})$.

This property cannot be directly proved with a model checker such as SMV [13] because it is neither a safety or liveness property nor a refinement property. So we look for sufficient conditions of $SecDep$ that are better handled by SMV. By analysing the various instructions in a program, it is easy to compute for each object the set of objects it syntactically depends on. The set $dep(i, o_t)$ contains objects with date $t-1$ used by instruction at program location i to compute the value of o_t. The program counter is an internal object such that pc_{t-1} determines the current instruction used to compute the value of o_t. Whenever an object is modified (i.e. o_{t-1} is different from o_t) then we consider that pc_{t-1} belongs to $dep(i, o_t)$.

Hypothesis 1. The value of o_t computed by the program is determined by the values of objects in $dep(e(pc_{t-1}), o_t)$:
$\forall o_t \in Output, \forall e \in Ev, e' \in Ev, e \sim_{dep(e(pc_{t-1}),o_t)} e' \Rightarrow e(o_t) = e'(o_t)$

The latter formula looks like $SecDep$ so to prove $SecDep$ it could be sufficient to prove that the security level of any member of $dep(e(pc_{t-1}), o_t)$ is dominated by o_t security level, in that case we would have proved that $dep(e(pc_{t-1}), o_t) \subseteq aut(o_t)$. But $dep(e(pc_{t-1}), o_t)$ may contain internal objects, as lvl is not defined for these objects we might be unable to check this sufficient condition. To overcome this problem we define function $lvldep$ that associates, for each evolution, a computed level with each object. If o_t is an input object then $lvldep(e, o_t) = lvl(o_t)$ otherwise $lvldep(e, o_t) = max\{lvldep(e, o'_{t-1}) \mid o'_{t-1} \in dep(e(pc_{t-1}), o_t)\}$ where max denotes the least upper bound in the lattice of levels.

Theorem 1. *A program satisfies SecDep if the computed level of an output object is always dominated by its security level:*
$\forall o \in Output, \forall e \in Ev, lvldep(e, o_t) \preceq lvl(o_t)$

Proof: We first prove by induction on t that the program satisfies property $DepL$: $\forall o_t \in Output, \forall e \in Ev, e' \in Ev, e \sim_{depL(e,o_t)} e' \Rightarrow e(o_t) = e'(o_t)$ where $depL(e, o_t) = \{o'_{t'} \in Input \mid lvl(o'_{t'}) \preceq lvldep(e, o_t)\}$.
As there is no output initially,$DepL$ is trivially true for $t = 0$. We suppose that $DepL$ is true for output objects until time $t-1$. Let e and e' be two

evolutions such that $e \sim_{depL(e,o_t)} e'$ where o_t is an output object, we want to prove that $e(o_t) = e'(o_t)$. By definition of *lvldep* if $o'_{t-1} \in dep(e(pc_{t-1}), o_t)$ then $lvldep(e, o'_{t-1}) \preceq lvl(o_t)$ so $\forall o''_{t''} \in Input, lvl(o''_{t''}) \preceq lvldep(e, o'_{t-1}) \Rightarrow lvl(o''_{t''}) \preceq lvl(o_t)$, hence $depL(e, o'_{t-1}) \subseteq depL(e, o_t)$. So, for all o'_{t-1} in $dep(e(pc_{t-1}), o_t)$, $e \sim_{depL(e,o'_{t-1})} e'$ and by induction hypothesis $e(o'_{t-1})= e'(o'_{t-1})$. We have $e \sim_{dep(e(pc_{t-1}),o_t)} e'$ and by hypothesis 1, we can conclude that $e(o_t) = e'(o_t)$.

We now show that, under the theorem assumption, if a program satisfies *DepL* it also satisfies *SecDep*. Let e and e' be two evolutions such that $e \sim_{aut(o_t)} e'$ where o_t is an output object. We show that $depL(e, o_t)$ is included in $aut(o_t)$ so $e \sim_{depL(e,o_t)} e'$ and by *DepL* we can conclude that $e(o_t) = e'(o_t)$.

Let $o'_{t'}$ be a member of $depL(e, o_t)$, by definition of $depL(e, o_t)$, $lvl(o'_{t'}) \preceq lvldep(e, o_t)$. According to the theorem assumption, $lvldep(e, o_t) \preceq lvl(o_t)$ because o_t is an output object. Hence we have $lvl(o'_{t'}) \preceq lvl(o_t)$ so $o'_{t'}$ belongs to $aut(o_t)$.

As we want to use model checkers to verify the security properties, it is important to restrict the size of value domains in order to avoid state explosions during verifications. To check security, it is sufficient to prove that the previous property holds in any state of an abstracted program where object values are replaced with object computed levels. If Ev is the set of evolution of the concrete program, then we note Ev^a the set of evolutions of the corresponding abstract program.

Hypothesis 2. We suppose that the set of abstract evolution Ev^a is such that the image of Ev by *abs* is included in Ev^a, where $abs(e)(o_t) = lvldep(e, o_t)$ if $o \neq pc$ and $abs(e)(pc_t) = e(pc_t)$.

Theorem 2. *If* $\forall o_t \in Output, \forall e^a \in Ev^a, e^a(o_t) \preceq lvl(o_t)$ *then the concrete program guarantees SecDep.*

Proof: Let o_t be an output object and e be a concrete evolution in Ev, by Hypothesis 2 $abs(e)$ is an abstract evolution hence $abs(e)(o_t) \preceq lvl(o_t)$. By definition of *abs*, $abs(e)(o_t) = lvldep(e, o_t)$ so $lvldep(e, o_t) \preceq lvl(o_t)$ and by applying theorem 1, *SecDep* is satisfied.

4 Applet Certification

An application is composed of a finite number of interacting applets. Each applet contains several methods. For efficiency reasons, we want to limit the number of applets and methods analysed when a new applet is downloaded or when the security policy is modified.

We first present how we decompose the global analysis of interacting applets on a card into local verifications of a subset of the methods of one applet. Then we explain how the local verifications are implemented.

4.1 Global Analysis Technique

There are two related issues in order to be able to apply the approach in practice: we first have to determine what is the program we want to analyse (one method of one applet, all the methods in one applet, all the methods in all the applets on the card); then we have to identify inputs and outputs and to assign them a level.

We suppose the complete call graph of the application is given. Two kinds of methods will especially interest us because they are the basis of applet interactions: interface methods that can be called from other applets and methods that invoke external methods of other applets. We have decided to analyse subsets of the call graph that include such interaction methods. Furthermore an analysed subset only contains methods that belong to the same applet.

Let us consider for instance the Air France applet. Method $logFull$ is an interface method that calls two internal methods: $askForTransactions$ and *update*. $askForTransactions$ invokes method *getTransaction* of Purse and credit the attribute *balance* with the value of the transactions; *update* invokes method *getBalance* of RentaCar and updates the value of the *extendedbalance* attribute. The program we are going to analyse is the set of 3 methods $logFull$, $askForTransactions$ and *update*.

Level Association. For a given program, we consider as inputs results from external invocations and read attributes. We take as outputs parameters of external invocations and modified attributes. We thus associate security levels with applet attributes and with method invocations between applets. By default, we associate level AF (resp. P and RC) with all the attributes of Air France (resp. Purse and RentaCar) applet. As Air France can invoke the *getBalance* or *debit* methods of RentaCar, we assign the shared security level $RC + AF$ to these interactions. Similarly, as the Purse applet can invoke the $logFull$ method of Air France, we associate level $AF + P$ with this interaction. And we associate level $P + AF$ (resp. $P + RC$) with the invocation of *getTransaction* method of the Purse applet by the Air France applet (resp. by RentaCar applet).

Security Properties. A security property is associated with each output variable. This property follows from the sufficient conditions defined in the previous section :

- $Sresult$: when a result is produced, the computed level of the result is dominated by the level of the interaction.
- *Smethod*: when an external method is invoked, the computed level of parameters is dominated by the security level of this interaction.
- $Sfield$: when an attribute is modified, the computed level of the attribute is dominated by the security level of this attribute.

In order to verify these properties, we have to compute all the evolutions of the program. First, for each instruction i and each object o of the program, we

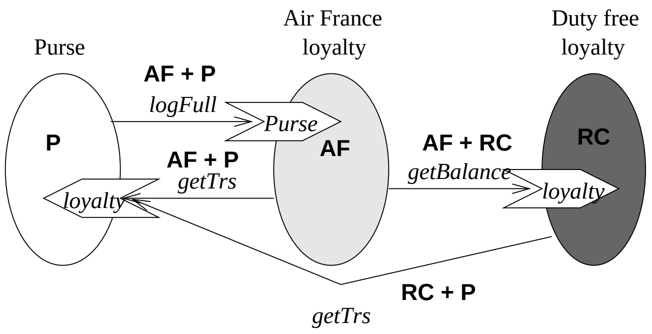

Fig. 5. Level Association

compute the set $dep(i, o_t)$. For Java byte code, we take into account, in addition to output and input objects defined previously, the operand stack, the program counter and other objects that will be given in the next section. For example, an instruction such as `iadd` computes the sum of the two integers at the top of the stack, pops these integers and pushes the result of the addition. If the current instruction at program location i is `iadd`, then the set $dep(i, o_t)$, where o_t is the top of the stack at time t, includes objects representing the top of the stack and its predecessor at time $t-1$ and object pc_{t-1}. The definition of dep for each instruction of the subset of the JavaCard byte code we have taken into account is defined in the first appendix.

Verification of Hypotheses 1 and 2. We have access to the computed level of objects by means of the sets of syntactical dependencies $dep(i, o_t)$. But we have to ensure that the hypotheses used to define the sufficient conditions are satisfied. The first hypothesis is trivially true because byte code instructions are deterministic. However, we must take care to include indirect dependencies in $dep(i, o_t)$. For example, a conditional instruction such as `if` can give two different values to the program counter depending on the the value on top of the stack. To satisfy hypothesis 1, it is thus necessary to include the top of the stack at time $t-1$ into the set $dep(if, pc_t)$.

The second hypothesis is also satisfied because each Java byte code instruction corresponds to one abstract byte code instruction that manipulates levels instead of values. We can thus associate with each sequence of instruction of a concrete evolution a sequence of instructions that corresponds to an abstract evolution. This abstract evolution associates with each object its computed level except for the program counter that keeps the same value as in the concrete evolution, which satisfies the condition of hypothesis 2.

Assume-Guarantee verification. Even if it is sufficient to verify the security properties on abstract evolutions where the value domain for variables is small, model checkers such as SMV cannot always handle call graphs with a large number of methods. Consequently we have decided to decompose the analysis.

We propose an assume-guarantee discipline that allows to verify a set of methods locally on each applet even if the methods call methods in other applets through shared interfaces (see figure 6). For instance, method *update* of applet Air France calls method $getBalance$ of RentaCar, we will analyse both methods separately. We check that the $Smethod$ property holds for the $update$ method of the Air France applet: the level of parameters of the $getBalance$ method invocation is dominated by the level of this interaction (i.e. $RC + AF$). And we assume that $RC + AF$ is the level of the result of this method invocation. When we analyse the $getBalance$ method in the RentaCar applet, we will check that the $Sresult$ property holds: the level of the result of this method is dominated by the level of the interaction and we will assume that $RC + AF$ is the level of the parameters of the $getBalance$ method.

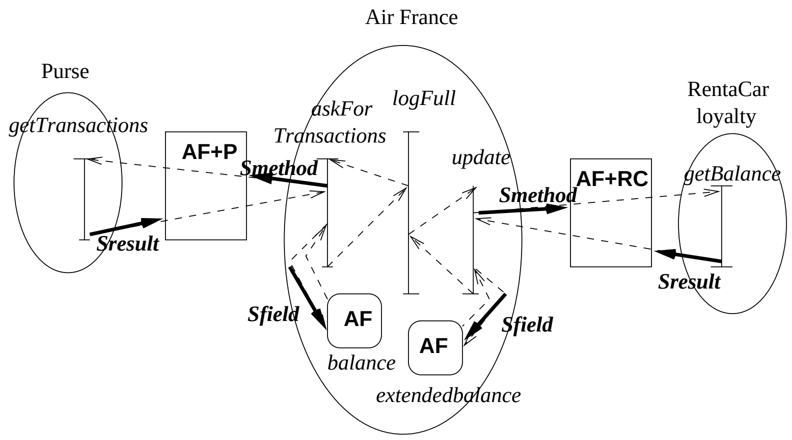

Fig. 6. Assume-Guarantee Verification

We adopt the same discipline for the attributes inside an applet. When an attribute is read, we assume that its level is the security level that was associated with it. When the attribute is modified, we check that the new level is dominated by the security level of this attribute. This assume-guarantee discipline inside an applet allows to verify only a subset of methods of an applet at a time (not the whole set of methods of this applet).

Thanks to this decomposition principle, it is possible to focus the analysis on the new applet that the card issuer wants to download on the cards. If the policy is unchanged there is no need to analyze again the already existing applets because levels associated with input and output objects will not change. If the security policy changes the security level associated with some input or output objects then only methods of already existing applets that use these objects should be checked again.

4.2 Local Analysis Technique

Our method to verify the security property on the application byte code is based on three elements:

- abstraction: following [3] we abstract all values of variables by computed levels;
- sufficient condition: we verify an invariant that is a sufficient condition of the security property;
- model checking: we verify this invariant by model checking (SMV).

We consider a set of methods at a time. Our approach uses the modularity capabilities of SMV: it first consists in building an SMV module that abstracts a method byte code for each method, then to build a `main` module that contains instances of each of these method abstraction modules and describes the interconnections between these modules. We begin by briefly presenting SMV, then we explain how we build a module for each method, we describe the `main` module and finally the properties. We illustrate the approach on the logFull example presented above that involves the analysis of methods *logFull*, *update* and *askForTransactions*. The full SMV model is presented in appendix 2.

SMV. The model checker we use is SMV, from Cadence Labs [3]. We only give here a brief presentation of the specification language. The interested reader is referred to [13] for more information.

Basic types in SMV are booleans, enumerated and integer intervals. Arrays can also be used (several dimension arrays if necessary). An SMV specification is a set of equations describing evolution of variable values. Two main ways exist to specify the value of a variable: either we give its value at each time instant `variable := expression`, or we give the initial value and the next value `init(variable) := expr1; next(variable) := expr2`. A particularity of SMV is that non deterministic assignments are possible, for example `b := {0,1}` specifies that b takes either value 0 or value 1.
Remark : ";" does not represent sequence in SMV, it is only a separator.

SMV specifications are organised in modules. A module gathers declarations and assignments and, like a procedure, has got input and output parameters. A module can thus be instanciated several times, with different parameters.

The main control structures are:

- if: `if (c) x:=expr1 else x:= expr2`,
  that can also be written `x := c ? expr1 : expr2 ;`

[3] http://www-cad.eecs.berkeley.edu/~kenmcmil/

– case:

```
case{
<cond1> : <stmt1>
..
<condn> : <stmtn>
[default : <dfstmt>]
}
```

– switch:

```
switch (<expr>){
<case1> : <stmt1>
...
<casen> : <stmtn>
[default : <dfstmt>]
}
```

– for: `for (i=0; i<3; i=i+1){ t[i] := i;}`
– `default` is less classical but is useful for a more compact description of equations:

```
default
  <stmt1>
in
  <stmt2>
```

This expression means that equations of `<stmt1>` are used for all cases where variables are not defined by `<stmt2>`.

Properties can be defined that have to be satisfied by every execution of the system, using `assert`. Properties are linear temporal logic formulas, temporal operators are `G` (always), `F` (eventually), `X` (next) et `U` (until). Assertions can be labeled, for example `Prop1: assert G(x=0)`. One assertion can also be used as hypothesis to prove another property by using the statement `using ... prove` : `using Prop1 prove Prop2`.

Method abstraction module. We illustrate our technique on a simplified version of Air France *update*() method. This method directly invokes method *getBalance* of the RentaCar applet and updates the *extendedbalance* field.

```
Method void update()
   0 aload_0
   1 invokespecial 179 <Method int getBalance()>
   4 istore_1
   5 aload_0
   6 dup
   7 getfield 220 <Field int extendedbalance>
```

```
  10 iload_1
  11 iadd
  12 putfield 220 <Field int extendedbalance>
  15 return
```

Abstraction. The *update*() byte code abstraction is modelled by an SMV module. This module has got parameters (which are instantiated in the main module):

– *active* is an input of the module, it is a boolean that is true when the method is active (as we consider several methods, we have to say which one is effectively executing);
– *context* is a boolean representing the context level of the caller;
– *param* is an array containing the levels of the parameters of the method (only one here, the object to which the method is applied, *this*);
– $field$ is an array containing the levels of the attributes used by the method (only one here *extendedbalance*);
– *method* is an array containing the levels of the results of the external methods invoked by the *update* method (only one here *getBalance*).
– *public* is a boolean representing the bottom in the lattice of levels.

The module also involves the following variables:

– *pc*: program counter;
– *lpc*: the level of the program counter, the context level of the method;
– *mem*[*i*]: an array modelling the memory locations;
– *stck*[*i*]: an array modelling the operand stack;
– *stckP*: stack pointer;
– $ByteCode$: the name of the current instruction.

The values of the variables are abstracted into levels. Levels are defined in a module called Levels in such a way that a level is represented by a boolean; least upper bound is modelled by disjunction, sharing by conjunction and implication encodes the fact that a level is dominated by another.

Hence the types of abstracted variables are boolean or array of booleans. We do not abstract the value of the program counter that gives the sequencing of instructions, we keep unchanged the value of the stack pointer that gives the index of the first empty slot.

```
module update(active, context, param, field, method, public){
  pc : -2..9;
  lpc: boolean;
  stck : array 0..1 of boolean;
  stckP : -1..2;
  mem : array 0..1 of boolean;
  ByteCode : {invoke_179, load_0, return, nop, store_1, dup,
             load_1, getfield_0,op, putfield_0};
```

The byte code execution starts at program location -1. Initially, the stack is empty.

```
init(pc):= -1; init(stckP):= 1;
for (i=0; i< 3; i=i+1) {init(stck[i]) := public; }
```

The control loop defines the value of the program counter and of the current instruction. It is an almost direct translation of the Java byte code. When *pc* is equal to -2 then the execution is finished and the current instruction is nop that does nothing. As in [8], each instruction we consider models various instructions of the Java byte code. For instance, as we do not care about the type of memory and stack locations, instruction `load_i` represents Java instructions (`aload_i`, `iload_i`, `lload_i`,...). Similarly, the `op` instruction models all the binary operations as (`iadd`, `ladd`, `iand`, `ior`, ...). Instruction `start` initializes the memory location with the method parameters and the level *lpc* with the context level of the caller *context*.

Although methods in the Electronic Purse case study tend to be rather short, the value domain of the *pc* variable is generally the larger domain in the SMV model we generate. As a large value domain can be a source of state explosion for model checkers, it is interesting to reduce it. We consider that the next value of *pc* is $pc + 1$ (except of course in the case of conditional or jump instructions). In this example, the domain of *pc* is -2..9 that is smaller that the original domain 0..15.

```
 if (active) {
  (next(pc), ByteCode) :=
   switch(pc) {
    -2: (-2, nop);
    -1: (0, start);
    0 : (pc+1, load_0 );
    1 : (pc+1, invoke_179 );
    2 : (pc+1, store_1 );
    3 : (pc+1, load_0 );
    4 : (pc+1, dup );
    5 : (pc+1, getfield_0 );
    6 : (pc+1, load_1 );
    7 : (pc+1, op );
    8 : (pc+1, putfield_0 );
    9 : (-2, return);
  };}
else {next(pc) := pc; ByteCode := nop;}
```

The following section of the SMV model describes the effect of the instructions on the variables. The instructions compute levels for each variable. The *load* instruction pushes the level of a memory location on the stack, the *store* instruction pops the top of the stack and stores the least upper bound of this level and *lpc* in a memory location. The least upper bound of levels l_1 and l_2 is modelled by the disjunction of two levels $l_1 \vee l_2$. The *dup* instruction duplicates on the stack the top of the stack. The *op* instruction computes the least upper bound of the levels of the two first locations of the stack. The *invoke* instruction pops from the stack the parameter and pushes onto the stack the result of this method invocation. Instruction $getfield$ pushes on the top of the stack the level of attribute *extendedbalance*. And, finally, instruction $putfield$ pops from the stack the level of attribute *extendedbalance*.

```
switch(ByteCode) {
 nop :;
 load_0 : {next(stck[stckP]):= mem[0];next(stckP):=stckP-1;}
 load_1 : {next(stck[stckP]):= mem[1];next(stckP):=stckP-1;}
 store_1 : {next(mem[1]):=(stck[stckP+1]|lpc);
           next(stckP):=stckP+1;}
 dup : {next(stck[stckP]):= stck[stckP+1];next(stckP):=stckP-1;}
 op : {next(stck[stckP+2]):=(stck[stckP+1]|stck[stckP+2]);
       next(stckP):=stckP+1;}
 invoke_179 : {next(stck[stckP]):=method[0];
               next(stckP):= stckP+1;}
 getfield_0 : {next(stck[stckP+1]):=field[0];}
 putfield_0 : {next(stckP):=stckP+2;}
 return : ;
 start : {next(stckP):= 1; next(mem[0]):= param[0];
          for (i=0; i< 3; i=i+1) {next(stck[i]) := public; }
          next(lpc) := context;}
}
}
```

"Conditional" instructions. No conditional instruction such as `ifne`, `table-switch` ... occurs in the example presented above. The behaviour of these instructions is to jump at various locations of the program depending on the value of the top of the stack. As this value is replaced by a level, the abstract program cannot decide precisely which is the new value of the program counter. So an SMV non-deterministic assignment is used to state that there are several possible values for *pc* (generally $pc + 1$ or the target location of the conditional instruction).

It is well known that conditional instructions introduce a dependency on the condition. This dependency is taken into account by means of the *lpc* variable. When a conditional instruction is encountered, *lpc* is modified and takes as new value the least upper bound of its current level and of the condition level. As each modified variable depends on *lpc*, we keep trace of the implicit dependency between variables modified in the scope of a conditional instruction and the condition.

lpc should also get back to its old level when going out of the conditional. However, there is no explicit `endif` in the byte code and it can be difficult to compute the end of a conditional. We will not handle this problem here but a solution is proposed in [2].

Main module. We have an SMV module for the *update* method. We can build in a similar way a module for the *askForTransaction* method and for the *logFull* method. We also have a module Levels. The SMV `main` module is composed of two parts: the first part manages the connections between methods (definition of the active method, parameter passing); the second one assigns levels to attributes and interactions.

The `main` module includes an instance of each of the method abstraction modules and one instance of the level module. To describe the call graph, we use the method call stack *calls* and its associated stack pointer *prev*. Variable *active* defines the currently active method.

```
module main(){
  L: Levels;
  active : {logFull, askForTransaction, update, stop};
  calls : array 0..2 of {logFull,askForTransaction,update} ;
  prev: -1..2 ;
  init(prev) := 2 ;
  for(i=0; i < 3; i=i+1) init(calls[i]):= logFull;
  m_lgf : logFull(active_lgf,context_lgf,param_lgf,method_lgf,
                  L.public);
  m_aft : askForTransaction(active_aft,context_aft,param_aft,
                            field_aft,method_aft,L.public);
  m_ud : update(active_ud,context_ud,param_ud,field_ud,method_ud,
                L.public);
```

The `main` module contains a set of equations that define the value of *active* according to the call graph. Initially, the active method is *logFull*.
When *askForTransactions* is invoked (`m_lgf.ByteCode=invoke_108`), method *askForTransactions* becomes active.
When it terminates (`m_aft.pc=-1`), *logFull* becomes active again.
When *update* is invoked (`m_lgf.ByteCode=invoke_55`) method *update* becomes active. When this method terminates (`m_ud.pc=-1`), *logFull* is active until the end.

When *logFull* invokes *askForTransactions* or *update*, it involves parameter passing between methods. In the example, there is only one parameter in each case(*this*), so it is sufficient to copy the top of logFull stack into the parameter array of the invoked method. We also transfer the context level of the caller to the invoked method by copying *lpc* in the *context* parameter.

```
if(m_lgf.ByteCode = invoke_108) {
          next(param_aft[0]) := m_lgf.stck[m_lgf.stckP+1];
          next(context_aft) := m_lgf.lpc; }
```

```
if(m_lgf.ByteCode = invoke_55) {
         next(param_ud[0]) := m_lgf.stck[m_lgf.stckP+1];
         next(context_ud) := m_lgf.lpc; }
```

Remark: the methods in the example do not have result, but in the general case, we would also have to transfer the result (i.e. the top of the stack) from the invoked method to the caller.

It now remains to assign levels to attributes and interactions. In the example, we have two attributes with level AF: *balance* (225) which is a parameter of $askForTransactions$, and *extendedbalance* (220) which is a parameter of *update*; and two interactions: *getBalance* (179) between Air France and RentaCar (so its level is $AF + RC$), parameter of $askForTransactions$, and *getTransaction* (163) between Air France and Purse (so its level is $AF + P$), parameter of *update*.
In our boolean encoding of levels, $l1 + l2$ is expressed by $L.l_1 \& L.l_2$.

```
field_aft[1] := L.AF;    method_aft[3] := L.AF & L.P;
field_ud[0] := L.AF;     method_ud[0] := L.AF & L.RC;
```

Invariant. We explained above how to compute a level for each variable. We also explained what security level we assigned to attributes and interactions. The invariant we verify is the sufficient condition we previously described: the computed level of each output is always dominated by its security level.

For the *update* method we should check two properties: one to verifiy that the interaction between *update* and *getBalance* is correct and the other one to check that *update* correctly uses attribute *extendedbalance*.
Property *Smethod_179* means that, whenever the current instruction is the invocation of method *getBalance*, then the level of the transmitted parameters (the least upper bound of *lpc* and the top of the stack) is dominated by the level of the interaction (the value of `method[0]` is $AF+RC$). In our boolean encoding of levels, l_1 is dominated by l_2 if `L.l1` implies `L.l2`. Property $Sfield\_220$ means that, whenever the current instruction is the modification of field *extendedbalance*, then the level of the new value (the least upper bound of *lpc* and the top of the stack) is dominated by the level of the attribute (the value of `field[0]` is AF).

```
Sfield_220 :
  assert G (ByteCode=putfield_0 -> ((stck[stckP+1]|lpc) ->
                                                         field[0]));
Smethod_179 :
  assert G(ByteCode = invoke_179 -> ((stck[stckP+1]|lpc) ->
                                                        method[0]));
```

For the initial method (here *logFull*), we also have to verify the $Sresult$ property which means that whenever the method is finished the level of the return value (the least upper bound of the top of the stack and the level of *pc*) is dominated by the level of the interaction $AF + P$. As $logFull$ does not return any value, the property to be checked is just that the level lpc is domintaed by the level of the interaction.

```
Sresult     :
  assert G (m_lgf.ByteCode=return ->(m_lgfl.lpc -> L.AF & L.P));
```

4.3 Analysis.

Once we have the abstract model and the invariant, we model check the invariant properties on the model using SMV. If the property does not hold the model checker produces a counter-model that represents an execution of the byte code leading to a state where the property is violated.

A security problem is detected when checking property $Smethod\_179$ of method *update*. Indeed, the $logFull$ interaction between purse and Air France has $AF + P$ level. The $getBalance$ channel has $AF + RC$ level and we detect that the invocation of the $getBalance$ method depends on the invocation of the $logFull$ method. There is thus an illicit dependency from a variable of level $AF + P$ to an object of level $AF + RC$ (cf figure 7).

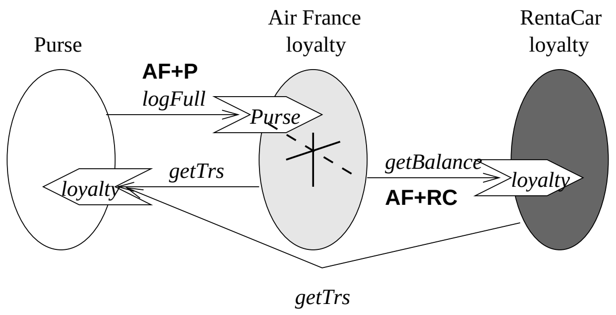

Fig. 7. Illicit interaction detection

A tool has been implemented which automates the production of SMV models. The automatization is relatively straightforward provided that preliminary treatments are made to prepare the model construction, such as construction of the call graph, method name resolution, subroutine elimination. The tool also handles exceptions. The tool and associated treatments are described in [2].

The electronic purse case study was completely analysed using this tool (132 SMV models, 421 properties). A fourth of the properties are verified within 5 seconds and 90 % are verified in less than 10 minutes (see [2] for a more detailed description of the experiments). The electronic purse application we considered is a big application with respect to smart card memory size limitations, it is more complex than current applets developed by Gemplus for customers. Hence we think we have demonstrated the ability of our technique to verify real world applications.

5 Related Work and Conclusion

In this paper, we have presented an approach for the certification of applets that are to be loaded on a Javacard. The security checks we propose are complementary to the security functions already present on the card. The applet firewall controls the interaction between two applets, while our analysis has a more global view and is able to detect illicit transitive information flow between several applets. Our approach is also complementary with the kind of verification described in [19]. The authors model application isolation requirements in a multi-application smart-card context. They are interested in showing that operating system functions enforce secure interaction between applets but their approach does not deal with transitive informations flows. Hence the model described in [19] could provide the basis of a formal model of applet isolation enforced by the JavaCard applet firewall.

Our approach is based on a security policy that defines how information is authorized to flow between applets. The policy associates levels with applet attributes and method invocations. We used the secure dependency property to control information flows between applets. We showed that to check this property it was sufficient to prove a set of security invariants on a finite model of the applets. We proposed to abstract the Java byte-code by replacing actual values with security levels. We used the standard model checker SMV to verify the security invariants. A tool has been implemented that automates the production of SMV models. However, a complete automatization is hardly possible: an interaction with the user is needed for the definition of security policy and level assignments.

A lot of work has been going on about the analysis of security properties of Java byte code. The major part of this work is concerned with properties verified by SUN byte code verifier like correct typing, no stack overflow, etc. These properties are not directly connected with the information flow control property we have based our work on. Among this work, two kinds of approaches can be distinguished depending on the technique used for the verification. Most of the approaches are based on static analysis techniques, particularly type systems [4, 21]. One approach has used model checking (with SMV) to specify the correct typing of Java byte code [18]. We also based our approach on model checking tools because they tend to be more generic and expressive than type checking algorithms. This allowed us to obtain results faster because we did not have to implement a particular type checking algorithm. This should also enable us to perform experiments with other security policies and properties without developing new verification algorithms.

Recently, several researchers investigated the static analysis of information flow properties quite similar to our secure dependency property but, to our knowledge, none of them applied their work on Java byte-code. Girard et alter

[9] defined a type system to check that a program written in a subset of the C language does not transfer High level information in Low level variables. In [5], the typing relation was related to the secure dependency property. Volpano and Smith [20] proposed a type system with a similar objective for a language that includes threads. They relate their typing relation to the non-interference property. The secure dependency property was compared with non-interference in [1].

Myers and Liskov [17] propose a technique to analyze information flows of imperative programs (the case of Java is taken into account in [16]) where variables are associated with security levels. In this context, a security level contains for each possible creator of the value a set of authorized observers. A comparison relation $\preceq$ is defined that authorizes information to flow from l_1 to l_2 whenever for all creators listed in l_1, the set of authorized observers in l_2 is contained in the set of authorized observers in l_2. As in our approach, $\preceq$ has a lattice structure so the level associated with the result of a binary operation is the least upper bound of the levels associated with that operation parameters. One of the interesting features of this policy is the declassify operation that allows creators to modify levels. The authors propose an almost linear constraint solving algorithm to verify that the level associated with each value is dominated by the level associated with each variable it may affect. This algorithm is more efficient than the model checking based verification but it is restricted to one kind of security property.

We think that our approach could deal with other kind of security properties with very limited modifications. In [10] the authors propose an automatic method for verifying that an implementation using local security checks satisfies a global security property. Their approach is focused on control flow properties such as Java Virtual Machine stack inspection. A Java program is modelled by its call stack traces, the only instructions of interest are: method invocation, method return and security checks. Methods are tagged with properties such as their origin, electronic signature status, etc. The authors propose to state security properties as Linear Time Temporal logic formulae. For instance an interesting JavaCard security property could be: once a method has been invoked by the terminal no internal applet method should be invoked until the user is authenticated. As our byte code models contain the call graph stacks (variable `calls`), and as temporal operator *Until* is defined in SMV we should be able to state and verify the previous property in our framework.

Finally, it might be interesting to investigate whether the results obtained in the case of JavaCard could be useful in the broader setting of Java hostile applets.

References

1. P. Bieber and F. Cuppens. A Logical View of Secure Dependencies. *Journal of Computer Security*, 1(1):99–129, 1992.
2. P. Bieber, J.Cazin, A. El Marouani, P. Girard, J.-L. Lanet, V. Wiels, and G. Zanon. The pacap prototype: a tool for detecting java card illegal flow. In *Proceedings of Java Card workshop*, 2000.

3. D. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.
4. Stephen N. Freund and John C. Mitchell. A type system for object initialization in the java byte code language. In *Proceedings of OOPSLA 98*, 1998.
5. P. Girard. *Formalisation et mise en oeuvre d'une analyse statique de code en vue de la verification d'applications securisees.* PhD thesis, ENSAE, 1996.
6. Pierre Girard. Which security policy for multiapplication smart cards? In *USENIX workshop on smartcard technology*, 1999.
7. J. Goguen and J. Meseguer. Unwinding and Inference Control. In *IEEE Symposium on Security and Privacy*, Oakland, 1984.
8. Pieter H. Hartel, Michael J. Butler, and Moshe Levy. The operational semantics of a java secure processor. Technical Report DSSE-TR-98-1, Declarative systems and Software Engineering group, University of Southampton,Highfield, Southampton SO17 1BJ, UK, 1998.
9. C. O'Halloran J. Cazin, P. Girard and C. T. Sennett. Formal Validation of Software for Secure Systems. In *Anglo-french workshop on formal methods, modelling and simulation for system engineering*, 1995.
10. T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security policies. In *Proceedings of the 20th IEEE Security and Privacy Symposium*, 1999.
11. Paul Kocher, Joshua Jaff, and Benjamin Jun. Differential power analysis: Leaking secrets. In *Advances in Cryptology – CRYPTO'99 Proceedings.* Springer-Verlag, 1999.
12. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1997.
13. K.L. McMillan. *The SMV language.* Cadence Berkeley Labs, 1999.
14. Microsoft. *Windows for Smart Cards.* http://www.microsoft.com/smartcard.
15. Mondex. *Multos Specification.* http://www.multos.com.
16. A. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of 26th ACM Symposium on Principles of Programming Languages (POPL99)*, 1999.
17. A.C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM symposium on operating systems principles*, 1997.
18. Joachim Posegga and Harald Vogt. Offline verification for java byte code using a model checker. In *Proceedings of ESORICS*, number 1485 in LNCS. Springer, 1998.
19. G. Shellhorn, W. Reif, A. Schairer, P. Karger, V. Austel, and D. Toll. Verification of a formal security model for multiapplicative smart cards. In *Proceedings of ESORICS*, number 1895 in LNCS. Springer, 2000.
20. G. Smith and D.M. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of POPL*, 1998.
21. Raymie Stata and Martin Abadi. A type system for java bytecode subroutines. In *Proc. 25th Symposium on Principles of Programming Languages*, 1998.
22. Sun Microsystems. *JavaCard Specification.* http://java.sun.com/products/javacard.
23. Sun Microsystems. *Java Card 2.1 Realtime Environment (JCRE) Specification*, February 1999.
24. Visa. *Open Platform, Card Specification*, April 1999. Version 2.0.

Appendix 1: Definition of the Pacap byte code

In this appendix, we define the subset of the Java byte code that we have taken into account. In the following table, each instruction of this subset is associated with a Pacap abstract instruction. A Pacap abstract instruction is a representative for all Java byte code instructions with similar syntactical dependencies. For instance, the `op` Pacap instruction represents all binary operations such as `iadd`, `isub`, `ior`,... as for all these instructions, the top of the stack at $t+1$ depends of the first and second value in the stack at t and all other objects are unchanged. In the table we note `*add` for all the Java byte code instructions matching with this expression such as `iadd`, `fadd`. In the `load_a` and `store_a` Pacap instructions, a denotes a memory location address. In the `invoke_i`, `getstatic_i`, `getfield_i`, `putfield_i` and `putstatic_i` Pacap instructions, i is an attribute or method identifier.

Java	**Pacap**
`nop, *neg, *shl, *ushr, i2*` `chekcast , wide, goto, goto_w`	`nop`
`*const*, *ipush, ldc*`	`const`
`*load*`	`load_a`
`*store*`	`store_a`
`pop`	`pop`
`pop2`	`pop2`
`dup`	`dup`
`dup_x1`	`dup_x1`
`dup_x2`	`dup_x2`
`dup2`	`dup2`
`dup2_x1`	`dup2_x1`
`dup2_x2`	`dup2_x2`
`swap`	`swap`
`*add, *sub, *mul, *div, *rem, *and, *or`	`op`
`if*, ifnonnull`	`if1`
`if_*, ifnull`	`if2`
`*return`	`returnres`
`return`	`return`
`getstatic`	`getstatic_i`
`putstatic`	`putstatic_i`
`getfield`	`getfield_i`
`putfield`	`putfield_i`
`invokevirtual, invokespecial` `invokestatic, invokeinterface`	`invoke_i` `r_invoke_i`

Fig. 8. Java byte code translation table

Several Java byte code instructions are not taken into account because they are not supported by JavaCard virtual machines. For instance, as there is no thread in JavaCard, instructions `monitorenter` and `monitorexit` are not represented in the Pacap byte code. We also assume that due to some preliminary transformations on the Java source code or byte code some instructions will not be used. For instance, if we suppose that sub-routines calls are replaced by the sub-routine codes we need not take into account instructions `jsr`, `ret` and `jsr_w`. In [2], we explain transformations that can be performed in order to treat exceptions (instruction `athrow`), arrays (instructions `iaload`, `iastore`) or object creation (instruction `new`).

Table in figure 9 defines the dependency function $dep(i, o_t)$ for any instruction i of the Pacap byte code and for the objects of interest. The table defines dep for instructions modifying memory locations (noted mem_a), program counter (noted pc) and the stack (noted s). The stack pointer is noted sp, s_{sp+1} is the top of the stack. The symbol $=$ means that the object is unchanged by instruction i, in that case $dep(i, o_t) = \{o_{t-1}\}$. The symbol $\emptyset$ means that $dep(i, o_t)$ is empty.

inst	sp	s_{sp-1}	s_{sp}	s_{sp+1}	s_{sp+2}	s_{sp+3}	s_{sp+4}	mem_a	pc
`nop`	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$
`pop`	$sp+1$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$
`pop2`	$sp+2$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$
`dup`	$sp-1$	$=$	s_{sp+1}	$=$	$=$	$=$	$=$	$=$	$=$
`dup_x1`	$sp-1$	$=$	s_{sp+1}	s_{sp+2}	s_{sp+1}	$=$	$=$	$=$	$=$
`dup_x2`	$sp-1$	$=$	s_{sp+1}	s_{sp+2}	s_{sp+3}	s_{sp+1}	$=$	$=$	$=$
`dup2`	$sp-2$	s_{sp+1}	s_{sp+2}	$=$	$=$	$=$	$=$	$=$	$=$
`dup2_x1`	$sp-2$	s_{sp+1}	s_{sp+2}	s_{sp+3}	s_{sp+1}	s_{sp+2}	$=$	$=$	$=$
`dup2_x2`	$sp-2$	s_{sp+1}	s_{sp+2}	s_{sp+3}	s_{sp+4}	s_{sp+1}	s_{sp+2}	$=$	$=$
`swap`	$=$	$=$	$=$	s_{sp+2}	s_{sp+1}	$=$	$=$	$=$	$=$
`op`	$sp+1$	$=$	$=$	s_{sp+2}, s_{sp+1}	$=$	$=$	$=$	$=$	$=$
`const`	$sp-1$	$=$	$\emptyset$	$=$	$=$	$=$	$=$	$=$	$=$
`const_pop`	$sp+1$	$=$	$=$	$=$	$\emptyset$	$=$	$=$	$=$	$=$
`load_a`	$sp-1$	$=$	mem_a	$=$	$=$	$=$	$=$	$=$	$=$
`invoke_i`	$sp+n$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$
`r_invoke_i`	$sp-1$	$=$	met_i	$=$	$=$	$=$	$=$	$=$	$=$
`getfield_i`	$=$	$=$	$=$	$field_i$	$=$	$=$	$=$	$=$	$=$
`getstatic_i`	$sp-1$	$=$	$field_i$	$=$	$=$	$=$	$=$	$=$	$=$
`store_a`	$sp+1$	$=$	$=$	$=$	$=$	$=$	$=$	pc, s_{sp+1}	$=$
`if1`	$sp+1$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	pc, s_{sp+1}
`if2`	$sp+2$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	pc, s_{sp+1}, s_{sp+2}
`returnres`	$sp+1$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$
`return`	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$
`putstatic_i`	$sp+1$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$
`putfield_i`	$sp+2$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$

Fig. 9. Definition of *dep* for instructions modifying the stack

Appendix 2: `LogFull` SMV model

```
module Levels() {
/*                                                              */
/* Modelling of levels :
   private=1, public=0, other levels are booleans,
   l1 < l2 modeled by l1 -> l2, l1+l2 by l1 & l2,
   max(l1,l2) by l1 | l2 */
/*                                                              */
  public,  private, AF, RC, P : boolean;
  public := 0; private := 1;
  init(AF) := {0,1};   next(AF) := AF;
  init(RC) := {0,1};   next(RC) := RC;
  init(P) := {0,1};   next(P) := P;
}

 module logFull (active, context, param, method, public) {
 /*                                      */
 /*  Declarations                        */
 /*                                      */
  INPUT context : boolean; /* bottom level*/
  INPUT active : boolean; /* method activation */
  INPUT param : array 0..0 of boolean; /* method parameters */
  INPUT method : array 0..1 of boolean;/* result of called methods */
  /* 0 : 108 askForTransaction  */
  /* 1 : 55 update */
  pc : -2..4; /* program counter  */
  lpc : boolean;/* program counter level */
  stck : array 0..0 of boolean; /* Stack */
  stckP : -1..0; /* Stack Pointer*/
  mem : array 0..0 of boolean; /* memory locations */
  ByteCode : { invoke_108,invoke_55,load_0,nop,return,start };
 /*                                      */
 /*  Init                                */
 /*                                      */
  init(pc) := -1;  init(lpc) := context;
  init(stckP) := 0; init(mem[0]) := param[0];
  for (i=0; i< 1; i=i+1) {init(stck[i]) := public; }
 /*                                      */
 /*  Control                             */
 /*                                      */
  if (active) {
   (next(pc), ByteCode) :=
    switch(pc) {
     -2: (-2, nop);
     -1: (0, start);
     0 : (pc+1, load_0 );
     1 : (pc+1, invoke_108 );
     2 : (pc+1, load_0 );
     3 : (pc+1, invoke_55);
     4 : (-2, return);
   };
  } else { next(pc):=pc;  ByteCode := nop; } ;
 /*                                      */
 /*  Instructions                        */
 /*                                      */
   default {
     next(lpc):=lpc; next(stckP):=stckP;
     next(mem[0]) := mem[0];
     for (i=0; i< 1; i=i+1) {next(stck[i]) := stck[i]; }
   } in {
    switch(ByteCode) {
     invoke_108 : {next(stckP):= stckP + 1 ;}
     invoke_55 : {next(stckP):= stckP + 1 ;}
     load_0 : {next(stck[stckP]):= mem[0];next(stckP):=stckP-1;}
     nop : ;
     return :  ;
     start : {next(lpc) := context; next(stckP) := 0; next(mem[0]) := param[0];
```

```
            for (i=0; i< 1; i=i+1) {next(stck[i]) := public;}}
   }
  }
}

module askForTransaction (active, context, param, field, method, public) {
/*                                    */
/*  Declarations                      */
/*                                    */
 INPUT public : boolean; /* bottom level*/
 INPUT context : boolean; /* bottom level*/
 INPUT active : boolean; /* method activation */
 INPUT param : array 0..0 of boolean; /* method parameters */
 INPUT field : array 0..4 of boolean;/* used attributes */
 /* 4 : #227 aid[] */
 /* 3 : #120  visitCounter */
 /* 2 : #146  accumulator */
 /* 1 : #225  balance */
 /* 0 : #119  P_AID[] */
 INPUT method : array 0..3 of boolean;/* result of called methods */
 /* 1 : 130 throwIt  */
 /* 0 : 192 lookupAID  */
 /* 2 : 235 getAppletShareableInterfaceObject  */
 /* 3 : 163 getTransaction */
 pc : -2..50; /* program counter  */
 lpc : boolean;/* program counter level */
 stck : array 0..3 of boolean; /* Stack */
 stckP : -1..3; /* Stack Pointer*/
 mem : array 0..4 of boolean;
 ByteCode : { const,getstatic_0,getfield_1,getfield_2,getfield_3,getfield_4,
    putfield_1,putfield_2,putfield_3,if1,invoke_130,invoke_192,invoke_235,
    invoke_163,load_0,load_1,load_2,load_3,load_4,nop,pop,r_invoke_192,
    r_invoke_235,r_invoke_163,return,store_1,store_2,store_3,store_4,op,dup,
    start};
/*                                    */
/*  Init                              */
/*                                    */
 init(pc) := -1;     init(lpc) := context;
 init(stckP) := 2;  for (i=0; i< 3; i=i+1) {init(stck[i]) := public; }
 init(mem[4]) := public; init(mem[3]) := public; init(mem[2]) := public;
 init(mem[1]) := public; init(mem[0]) := param[0];
/*                                    */
/*  Control                           */
/*                                    */
 if (active) {
  (next(pc), ByteCode) :=
   switch(pc) {
    -2: (-2, nop);
    -1: (0, start);
    0 : (pc+1, getstatic_0 );
    1 : (pc+1, const );
    2 : (pc+1, const );
    3 : (pc+1, invoke_192 );
    4 : (pc+1, r_invoke_192 );
    5 : (pc+1, store_1 );
    6 : (pc+1, load_1 );
    7 : ({pc+1, 10 } ,if1);
    8 : (pc+1, const );
    9 : (pc+1, invoke_130 );
    10 : (pc+1, load_1 );
    11 : (pc+1, const );
    12 : (pc+1, invoke_235 );
    13 : (pc+1, r_invoke_235 );
    14 : (pc+1, store_2 );
    15 : (pc+1, load_2 );
    16 : ({pc+1, 50 } ,if1);
    17 : (pc+1, load_2 );
    18 : (pc+1, nop );
    19 : (pc+1, store_3 );
    20 : (pc+1, load_3);
    21 : (pc+1, load_0);
    22 : (pc+1, getfield_4);
    23 : (pc+1, invoke_163);
    24 : (pc+1, r_invoke_163);
    25 : (pc+1, store_4);
    26 : (pc+1, load_0 );
    27 : (pc+1, dup );
    28 : (pc+1, getfield_1 );
    29 : (pc+1, load_4 );
    30 : (pc+1, op );
    31 : (pc+1, nop );
    32 : (pc+1, putfield_1 );
    33 : (pc+1, load_0 );
    34 : (pc+1, dup );
    35 : (pc+1, getfield_2 );
    36 : (pc+1, load_4 );
    37 : (pc+1, op );
    38 : (pc+1, nop );
    39 : (pc+1, putfield_2 );
    40 : (pc+1, load_0 );
    41 : (pc+1, dup );
    42 : (pc+1, getfield_3 );
    43 : (pc+1, const );
    44 : (pc+1, op );
    45 : (pc+1, nop );
    46 : (pc+1, putfield_3 );
    47 : (50, nop);
    48 : (pc+1, pop );
    49 : (50, nop);
    50 : (-2, return);
  };
}
 else {next(pc):=pc;  ByteCode := nop; } ;
/*                                      */
/*  Instructions                        */
/*                                      */
  default {
    next(lpc):=lpc; next(stckP):=stckP; next(mem[3]) := mem[3];
    next(mem[2]) := mem[2]; next(mem[1]) := mem[1]; next(mem[0]) := mem[0];
    for (i=0; i< 3; i=i+1) { next(stck[i]) := stck[i]; }
  }
  in {
   switch(ByteCode) {
    const : {next(stck[stckP]):=public;next(stckP):=stckP-1;}
    getstatic_0 : {next(stckP):=stckP-1;next(stck[stckP]):=field[0];}
    getfield_1 : next(stck[stckP+1]):=field[1];
    getfield_2 : next(stck[stckP+1]):=field[2];
    getfield_3 : next(stck[stckP+1]):=field[3];
    putfield_1 : { next(stckP):=stckP+2;}
    putfield_2 : { next(stckP):=stckP+2;}
    putfield_3 : { next(stckP):=stckP+2;}
    if1 : {next(lpc):=(stck[stckP+1] | lpc); next(stckP):=stckP+1;}
    invoke_130 : {next(stckP):= stckP + 1 ;}
    invoke_192 : {next(stckP):= stckP + 3 ;}
    invoke_235 : {next(stckP):= stckP + 2 ;}
    invoke_163 : {next(stckP):= stckP + 2 ;}
    load_0 : {next(stck[stckP]):= mem[0];next(stckP):=stckP-1;}
    load_1 : {next(stck[stckP]):= mem[1];next(stckP):=stckP-1;}
    load_2 : {next(stck[stckP]):= mem[2];next(stckP):=stckP-1;}
    load_3 : {next(stck[stckP]):= mem[3];next(stckP):=stckP-1;}
    load_4 : {next(stck[stckP]):= mem[4];next(stckP):=stckP-1;}
    nop : ;
    pop : {next(stckP):=stckP+1;}
    r_invoke_192 : {next(stck[stckP]):=method[0];next(stckP):=stckP-1;}
    r_invoke_235 : {next(stck[stckP]):=method[2];next(stckP):=stckP-1;}
    r_invoke_163 : {next(stck[stckP]):=method[3];next(stckP):=stckP-1;}
    return : ;
    store_1 : {next(mem[1]):= (stck[stckP + 1] | lpc) ;next(stckP):=stckP+1;}
    store_2 : {next(mem[2]):= (stck[stckP + 1] | lpc) ;next(stckP):=stckP+1;}
    store_3 : {next(mem[3]):= (stck[stckP + 1] | lpc) ;next(stckP):=stckP+1;}
    store_4 : {next(mem[4]):= (stck[stckP + 1] | lpc) ;next(stckP):=stckP+1;}
    op : {next(stck[stckP+2]):=(stck[stckP+1]|stck[stckP+2]);
          next(stckP):=stckP+1;}
    dup : {next(stck[stckP]) := stck[stckP+1]; next(stckP) := stckP-1;}
    start : { next(lpc) := context; next(stckP) := 2;
              for (i=0; i< 3; i=i+1) {next(stck[i]) := public; }
              next(mem[4]) := public; next(mem[3]) := public;
              next(mem[2]) := public; next(mem[1]) := public;
              next(mem[0]) := param[0]; }
 } }
 /*      Security Properties      */

  Sfield_120: assert G (ByteCode=putfield_3 ->
                      ((stck[stckP+1]|lpc) ->  field[3]));  /* visitCounter */
  Sfield_146: assert G (ByteCode=putfield_2 ->
                      ((stck[stckP+1]|lpc) ->  field[2]));  /* accumulator */
  Sfield_225: assert G (ByteCode=putfield_1 ->
                      ((stck[stckP+1]|lpc) ->  field[1]));  /* balance */
 Smethod_163: assert G(ByteCode = invoke_163 ->
                        ((stck[stckP+1]|stck[stckP+2]|lpc) -> method[3]));
 }


 module update(active, context, param, field, method, public){
 /*                                    */
 /*  Declarations                      */
 /*                                    */
  INPUT public : boolean; /* bottom level*/
  INPUT context : boolean; /* bottom level*/
  INPUT active : boolean; /* method activation */
  INPUT param : array 0..0 of boolean; /* method parameters */
  INPUT field : array 0..0 of boolean;/* used attributes */
  /* 0 : 220 extendedBalance */
  INPUT method : array 0..0 of boolean;/* result of called methods */
  /* 0 : 179 getBalance */
  pc : -2..9; /* program counter */
  lpc: boolean;/* program counter level */
  stck : array 0..2 of boolean; /* Stack */
  stckP : -1..2; /* Stack Pointer*/
  mem : array 0..1 of boolean;
 /* Programme instructions    */
  ByteCode : {invoke_179, load_0, return, nop, store_1, dup,
             load_1, getfield_0,op, putfield_0,start};
 /*                                    */
 /*  Init                              */
 /*                                    */
 init(pc):= -1; init(stckP):= 1; init(mem[0]):= param[0];
 for (i=0; i< 3; i=i+1) {init(stck[i]) := public; }
 init(lpc) := context;
 /*                                    */
 /*  Control                           */
 /*                                    */
if (active) {
  (next(pc), ByteCode) :=
   switch(pc) {
    -2: (-2, nop);
    -1: (0, start);
    0 : (pc+1, load_0 );
    1 : (pc+1, invoke_179 );
    2 : (pc+1, store_1 );
    3 : (pc+1, load_0 );
    4 : (pc+1, dup );
    5 : (pc+1, getfield_0 );
    6 : (pc+1, load_1 );
    7 : (pc+1, op );
    8 : (pc+1, putfield_0 );
    9 : (-2, return);
  };}
else {next(pc) := pc; ByteCode := nop;}

 /*                                    */
 /*  Instructions                      */
 /*                                    */
   default {
     next(lpc):=lpc; next(stckP):=stckP;
     next(mem[1]) := mem[1]; next(mem[0]) := mem[0];
     for (i=0; i< 3; i=i+1) { next(stck[i]) := stck[i]; }
   }
   in {
   switch(ByteCode) {
    nop :;
    load_0 : {next(stck[stckP]):= mem[0];next(stckP):=stckP-1;}
    load_1 : {next(stck[stckP]):= mem[1];next(stckP):=stckP-1;}
    store_1 : {next(mem[1]):=(stck[stckP+1]|lpc) ;next(stckP):=stckP+1;}
    dup : {next(stck[stckP]):= stck[stckP+1]; next(stckP):=stckP-1;}
    op : {next(stck[stckP+2]):=(stck[stckP+1]|stck[stckP+2]);
          next(stckP):=stckP+1;}
    invoke_179 : {next(stck[stckP]):=method[0];next(stckP):= stckP+1;}
    getfield_0 : {next(stck[stckP+1]):=field[0];}
    putfield_0 : {next(stckP):=stckP+2;}
    return : ;
    start : {next(stckP):= 1; next(mem[0]):= param[0];
             for (i=0; i< 3; i=i+1) {next(stck[i]) := public; }
             next(lpc) := context;}
   }
   }

/*    Security properties     */
Sfield_220 : assert G (ByteCode=putfield_0 ->
                      ((stck[stckP+1]|lpc) ->  field[0]));
 Smethod_179 : assert G(ByteCode = invoke_179 ->
                       ((stck[stckP+1]|lpc) -> method[0]));



}


/*                                     */
/*     Main Module                     */
/*                                     */
module main(){
  L: Levels; /* Levels module instance */
/*                                      */
/* call stack declaration and init      */
/*                                      */
  active : {logFull, askForTransaction, update, stop};
  calls : array 0..2 of {logFull,askForTransaction,update} ;  /* call stack */
  prev: -1..2 ; /* call stack pointer */
  init(prev) := 2 ;
  for(i=0; i < 3; i=i+1) init(calls[i]):= logFull;
/*                                      */
/* logFull module instanciation         */
/*                                      */
  active_lgf : boolean; /* method activation */
  context_lgf : boolean; /* invokation level */
  param_lgf : array 0..0 of boolean; /* method parameters */
  method_lgf : array 0..1 of boolean;/* result of called methods */
  m_lgf : logFull(active_lgf, context_lgf, param_lgf, method_lgf, L.public);
  active_lgf := (active=logFull);
  init(method_lgf[0]) := L.public ;/* 0 : 108 askForTransaction  */
  init(method_lgf[1]) := L.public ;  /* 1 : 55 update */
  init(param_lgf[0]) := L.public ;
/*                                     */
/* askForTransaction module instanciation */
/*                                     */
  active_aft : boolean; /* method activation */
  context_aft : boolean; /* invokation level */
  param_aft : array 0..0 of boolean; /* method parameters */
  field_aft : array 0..4 of boolean;/* used attributes */
  method_aft : array 0..3 of boolean;/* result of called methods */
  m_aft : askForTransaction(active_aft, context_aft, param_aft, field_aft,
                            method_aft, L.public);
  active_aft := (active=askForTransaction);
  field_aft[0] := L.public;   /* 0 : #119  P_AID[] */
  field_aft[1] := L.AF;     /* 1 : #225  balance */
  field_aft[2] := L.AF;  /* 2 : #146  accumulator */
  field_aft[3] := L.AF;  /* 3 : #120  visitCounter */
  field_aft[4] := L.public;   /* 4 : #227 aid[] */
  method_aft[1] := L.public;    /* 1 : 130 throwIt  */
  method_aft[0] := L.public;   /* 0 : 192 lookupAID  */
  method_aft[2] := L.public;    /* 2 : 235 getAppletShareableInterfaceObject  */
  method_aft[3] := L.AF & L.P;  /* 3 : 163 getTransaction */
  init(param_aft[0]) := L.public ;
/*                                     */
/* update module instanciation         */
/*                                     */
  active_ud : boolean; /* method activation */
  context_ud : boolean; /* invokation level */
  param_ud : array 0..0 of boolean; /* method parameters */
  field_ud : array 0..0 of boolean;/* used attributes */
  method_ud: array 0..0 of boolean; /* result of called methods */
  m_ud : update(active_ud, context_ud, param_ud, field_ud, method_ud, L.public);
  active_ud := (active=update);
  field_ud[0] := L.AF;    /* 0 : #220  extendedBalance */
  method_ud [0] := L.AF & L.RC;   /* 0 : #179 getBalance() */
  init(param_ud[0]) := L.public ;
/*                                     */
/*  context initialization             */
/*                                     */
  context_lgf := L.AF & L.P;
  init(context_aft) := L.public; init(context_ud) := L.public;
/*                                     */
/*  activity Transfer                  */
/*                                     */
  init(active) := logFull;
  default {
    next(active):=active; next(prev):= prev;
    for (i=0; i < 3  ;i=i+1) next(calls[i]):=calls[i];
  } in {
    switch (active) {
      logFull :  default {
        switch  (m_lgf.ByteCode) {
          invoke_108 : { next(active):= askForTransaction;
            next(calls[prev]):= logFull; next(prev):= prev-1;}
          invoke_55 : {next(active) := update;
            next(calls[prev]):= logFull; next(prev) := prev-1;}
         }
      } in { if(m_lgf.pc=-2) { next(active):= stop; }}
      askForTransaction :  if(m_aft.pc=-2) {
         next(prev):=prev+1; next(active):= calls[prev]; }
      update : if(m_ud.pc=-2) {
         next(prev):=prev+1; next(active):= calls[prev]; }
    }
  }
/*                                     */
/*            parameter transfer       */
/*                                     */
  switch (active) {
    logFull : {
     if(m_lgf.ByteCode = invoke_108) {
       next(context_aft) := m_lgf.lpc;
       next(param_aft[0]) := m_lgf.stck[m_lgf.stckP+1];
     } else {
       next(context_aft) := context_aft;
       next(param_aft[0]) := param_aft[0];
     }
     if(m_lgf.ByteCode = invoke_55) {
       next(context_ud) := m_lgf.lpc;
       next(param_ud[0]) := m_lgf.stck[m_lgf.stckP+1];
     } else {
       next(context_ud) := context_ud;
       next(param_ud[0]) := param_ud[0];
     }
    }
    askForTransaction : {
     if (m_aft.pc = -2) {next(method_lgf[0]) := m_aft.lpc;}
    }
    update : {
     if (m_ud.pc = -2) {next(method_lgf[1]) := m_ud.lpc;}
    }
  }
/*                                        */
/*           Security properties          */
/*                                        */
 Sresult : assert G (m_lgf.pc = -2 -> (m_lgf.lpc -> L.AF & L.P));
}
```