Certification d'un porte-monnaie électronique*

[Published in *Formalisation des Activités Concurrentes (FAC2000)*, Toulouse, France, May 18, 2000.]

Pierre Bieber[1], Jacques Cazin[1], Pierre Girard[2], Jean-Louis Lanet[2], Virginie Wiels[1], and Guy Zanon[1]

[1] ONERA/CERT/DTIM
2 avenue E. Belin, 31055 Toulouse cedex, France
[2] GEMPLUS
Avenue du pic de Bertagne, 13881 Gémenos cedex, France

Abstract. L'article décrit l'état d'avancement d'un projet commun à Gemplus et à l'ONERA. Gemplus a développé un porte-monnaie électronique s'exécutant sur des cartes à puces supportant Java. Le but du projet est de vérifier des propriétés de sécurité qui doivent être satisfaites par les applets participant à l'application. Une politique de sécurité a été définie qui associe des niveaux aux attributs et aux méthodes et qui définit les flux d'information autorisés entre les niveaux. Nous proposons une technique basée sur le model checking pour vérifier que les flux d'information entre applets sont autorisés.

Introduction

La section 1 présente le contexte des cartes multi-applicatives et le cas d'étude choisi pour le projet. Dans la section 2, nous expliquons les mécanismes de sécurité existants sur les cartes et montrons qu'ils ne couvrent pas toutes les menaces. Notre but est de compléter ces mecanismes en détectant les interactions illicites entre applets. Pour cela, nous construisons un modèle SMV à partir du byte code des applets et vérifions que ce modèle satisfait des propriétés de sécurité. La section 3 donne la politique de sécurité choisie et les propriétés de sécurité associées. La section 4 explique comment construire le modèle SMV et vérifier les propriétés. Conclusion et perspectives sont données en section 5.

1 Contexte et cas d'étude

1.1 Cartes à puces multi-applicatives

Une nouvelle génération de cartes à puces arrive sur le marché : les cartes à puces multi-applicatives. Les principales caractéristiques de ces cartes sont que

* Le projet Pacap est partiellement financé par le MENRT décision d'aide 98.B.0251

des applications peuvent être chargées après la délivrance de la carte et que plusieurs applications différentes peuvent s'exécuter sur une même carte. Des systèmes d'exploitation ont été proposés pour les cartes multi-applicatives : Java Card [1], Multos [2] et plus récemment Windows for Smart Cards [3]. Dans ce papier, nous nous intéresserons à Java Card. Suivant cette norme, les applications pour les cartes multi-applicatives sont programmées sous forme d'applets Java.

1.2 Cas d'étude

Un exemple classique de carte multi-applicative est un porte-monnaie électronique constitué d'une applet purse et de deux applets loyalty: une application de fidélité Air France et un programme de fidélité d'une agence de location de voiture (RentaCar). L'applet purse gère les opérations de débit et crédit du porte-monnaie ainsi qu'un historique des transactions. Les achats pouvant être effectués dans plusieurs devises (par exemple francs et euros) cette applet gère également une table de taux de conversion. A cette applet de base s'ajoutent des applets (nommées Loyalty) qui correspondent à des programmes de fidélité qui octroient des points (par exemple, des miles pour un programme de fidélité d'une compagnie aérienne) en fonction des achats effectués avec ce porte-monnaie. Lorsque le détenteur de la carte à puce souhaite participer à un programme de fidélité, il suffit de charger sur sa carte l'applet Loyalty correspondante. L'applet Loyalty doit pouvoir interagir avec l'applet Purse pour connaître les transactions effectuées afin de mettre à jour le nombre de points de fidélite. Il est aussi prévu que des accords existent entre programmes de fidélité. Dans ce cas, différentes applets Loyalty pourront interagir pour s'échanger des points de fidélité. Par exemple, les points gagnés avec RentaCar pourraient être ajoutés aux miles Air France pour acheter un billet.

Le porte-monnaie électronique a été choisi comme cas d'étude pour notre projet et a été implémenté en Java par Gemplus.

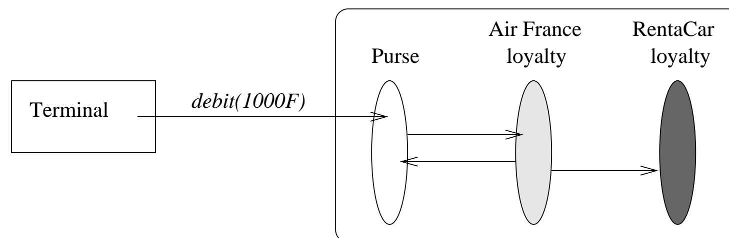

Fig. 1. Porte-monnaie électronique

[1] http://java.sun.com/products/javacard
[2] http://www.multos.com
[3] http://www.microsoft.com/smartcard

2 Sécurité

La sécurité est toujours un sujet très important pour les cartes à puces, mais elle est d'autant plus importante pour les cartes multi-applicatives. Ce type de cartes met en jeu plusieurs participants : le fabricant de cartes, l'émetteur qui propose la carte aux utilisateurs, les fournisseurs d'applications et les utilisateurs. L'émetteur est généralement considéré comme responsable de la sécurité de la carte. Il ne fait pas confiance aux fournisseurs d'applications : les applets peuvent être malveillantes ou simplement incorrectes.

Les fonctions de sécurité Java comme le vérificateur de byte code ou le security manager [7] ont été conçues pour empêcher des applets malveillantes d'endommager les ressources locales. Ces fonctions ont pour but d'isoler les applets malveillantes des autres applets et des ressources. Pour permettre le développement des cartes multi-applicatives, la norme Javacard a introduit un nouveau moyen pour les applets d'interagir directement. Une applet peut invoquer une méthode d'une autre applet à travers une interface de partage. Dans l'application du porte-monnaie électronique, l'applet purse a une interface de partage pour permettre aux applets loyalty de récupérer leurs transactions et les applets loyalty ont une interface de partage pour permettre aux loyalties partenaires de récupérer des points. Comme l'interaction entre applets est en dehors du champ d'application des fonctions de sécurité Java classiques, Java card a défini une nouvelle fonction de sécurité appelée applet firewall [11]. Cette fonction de sécurité contrôle que seules les méthodes des interfaces partagées peuvent être invoquées par les autres applets.

Nous faisons l'hypothèse que toutes les fonctions de sécurité Java et Javacard sont utilisées dans le porte-monnaie électronique. Cependant, ces fonctions ne couvrent pas toutes les menaces. Nous nous intéressons tout particulièrement aux menaces induites par les cartes multi-applicatives comme les interactions illicites d'applets. Un exemple d'interaction illicite dans le cas du porte-monnaie électronique est décrit sur le schéma suivant :

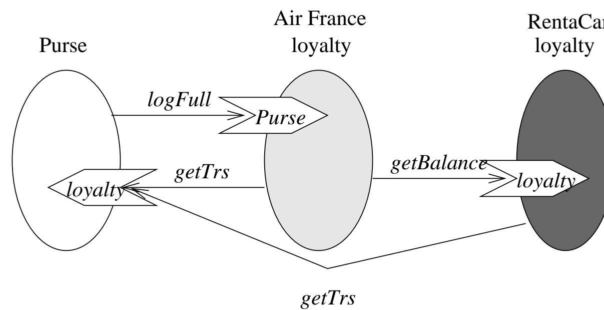

Fig. 2. Interactions d'applets

Un service “logfull” est proposé par l’applet purse aux applets loyalty : quand l’historique des transactions est plein, l’applet purse invoque la méthode $logfull$ des applets loyalty qui ont souscrit au service pour les prévenir que l’historique est plein et qu’ils devraient récupérer leurs transactions avant que certaines d’entre elles soient effacées et remplacées par de nouvelles. Nous supposons que l’applet Air France a souscrit au service logfull, mais pas l’applet RentaCar. Quand l’historique est plein, l’applet purse invoque donc la méthode $logfull$ de l’applet Air France. Dans cette méthode, l’applet Air France récupère ses transactions, mais veut également mettre à jour sa balance étendue qui contient ses points et tous les points qu’elle peut récupérer chez ses partenaires. Pour mettre à jour cette balance étendue, l’applet invoque la méthode *getbalance* de l’applet RentaCar. Dans ce cas de figure, l’applet RentaCar peut deviner que l’historique est plein quand l’applet Air France invoque sa méthode *getbalance* et donc récupérer ses transactions chez l’applet purse. Il y a une fuite d’information de l’applet Air France vers l’applet RentaCar et nous voulons pouvoir détecter de tels flux d’information illicites.

Ce comportement illicite ne serait pas interdit par le firewall car toutes les méthodes invoquées appartiennent bien à des interfaces de partage. Notre but est de fournir des techniques et des outils permettant à l’émetteur de cartes de vérifier que de nouvelles applets respectent bien certaines propriétés de sécurité définies sous la forme d’interactions autorisées. L’approche peut être décrite par le schéma suivant :

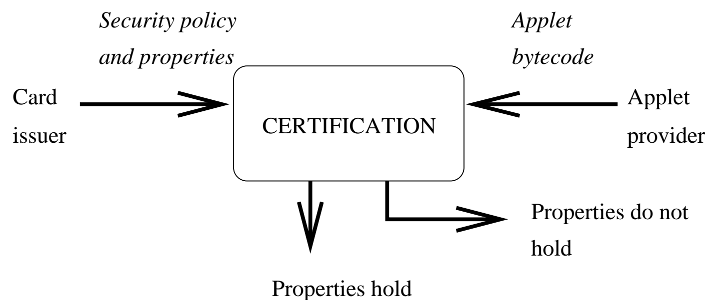

Fig. 3. Certification d’applet

Si le fournisseur d’applications veut charger une nouvelle applet sur une carte, il fournit le bytecode de cette applet. L’émetteur de carte a une politique de sécurité et des propriétés de sécurité qui doivent être satisfaites. Nous fournissons des techniques et des outils qui permettent de décider si les propriétés sont respectées par la nouvelle applet (ces techniques s’appliquent sur le byte code et utilisent le model checking). Si les propriétés sont respectées, l’applet peut être chargée sur la carte, sinon, elle est rejetée.

3 Politique de sécurité multi-applicative

Pour mettre en œuvre cette approche de certification d'applet, il nous faut d'abord choisir une politique de sécurité adaptée aux cartes multi-applicatives et des propriétés de sécurité associées.

3.1 Politique de sécurité

Nous proposons d'utiliser une politique de sécurité multiniveau [3] qui a été conçue pour les cartes à puces multi-applicatives. On associe à chaque fournisseur d'applet un niveau de sécurité et on considère des niveaux spéciaux pour les données partagées. Sur l'exemple du porte-monnaie électronique, nous avons un niveau pour chaque applet : AF pour Air France, P pour purse et RC pour RentaCar, et des niveaux pour les données partagées : $AF+RC$ pour les données partagées par Air France et RentaCar, $AF+P$ pour les données partagées par Air France et purse, etc. La relation entre les niveaux $\preceq$ est utilisée pour autoriser ou interdire des flux d'information entre applets. Dans la politique que nous considérons, $AF+P \preceq AF$ et $AF+P \preceq P$, cela signifie que les flux d'informations de $AF+P$ vers P ou AF sont autorisés. L'information partagée par Air France et Purse peut être reçue par les applets Air France et Purse. Pour modéliser le fait que les applets peuvent uniquement communiquer par le biais d'interfaces partagées, les flux d'information directs entre niveaux AF, P et RC sont interdits.

Les niveaux munis de la relation $\preceq$ forment un treillis, il y a un niveau minimal *public* et un niveau maximal *private*.

3.2 Propriétés de sécurité

Il faut maintenant définir les propriétés de sécurité qui doivent être respectées. Nous avons choisi le modèle des dépendances sûres [1] qui s'applique à des systèmes où des applications malveillantes pourraient communiquer des informations confidentielles à d'autres applications. Comme d'autres modèles de flux tels que la non-interférence [4], ce modèle assure que les dépendances entre les variables du système ne peuvent pas être exploitées pour établir des canaux de communication indirects. Nous appliquons ce modèle au porte-monnaie électronique : les interactions illicites seront détectées en contrôlant les dépendances entre les variables du système.

L'application de ce modèle nécessite trois étapes. Il faut tout d'abord ajouter des informations de niveaux aux variables du système. On peut ainsi exprimer la propriété qui doit être satisfaite : la valeur d'une variable de niveau l ne dépend que de valeurs de variables de niveau inférieur ou égal à l. La deuxième étape consiste à définir des conditions suffisantes pour que cette propriété soit vraie (conditions plus faciles à vérifier avec des outils de model checking). Finalement, on constate que ces conditions suffisantes ne dépendent pas de la valeur des variables du système, mais seulement de leur niveau. Il est donc suffisant de

les vérifier sur une "abstraction" du système où les valeurs des variables sont remplacées par leur niveau. Nous allons détailler ces trois étapes.

Nous distinguons trois types de variables : variables d'entrées qui ne sont pas calculées par le système, variables de sortie qui sont calculées par le système et sont directement observables et variables internes, inobservables. Nous devons tout d'abord associer un niveau de sécurité aux variables d'entrée et de sortie du système. La propriété à vérifier est alors que la valeur d'une variable de sortie de niveau l ne dépend que de valeurs de variables d'entrée de niveau inférieur ou égal à l (ordre défini par la relation $\preceq$). Cela signifie que toute paire d'exécutions du programme commençant avec les mêmes valeurs pour toutes les variables d'entrée de niveau inférieur ou égal à l doit calculer la même valeur pour les variables de sortie de niveau l. Cette propriété n'est pas facile à vérifier avec des outils de vérification, nous utilisons donc des conditions suffisantes qui sont plus facilement traitées par ces outils.

Il est facile de calculer pour chaque variable du programme l'ensemble des variables dont elle dépend syntaxiquement. Comme les valeurs de ces variables déterminent les valeurs calculées par le programme, il est suffisant de montrer qu'une variable de niveau l ne dépend syntaxiquement que de variables de niveau inférieur ou égal à l.

Cependant, cette propriété peut faire intervenir des niveaux de variables internes. Comme il n'est pas toujours possible d'associer un niveau à toutes les variables, nous définissons la notion de niveau calculé d'une variable. Le niveau calculé d'une variable d'entrée est son niveau de sécurité. Sinon, le niveau calculé est la borne supérieure des niveaux calculés des variables dont elle dépend syntaxiquement. Pour vérifier la sécurité, il est alors suffisant de montrer que, dans tout état du programme, le niveau calculé d'une variable de sortie est inférieur ou égal à son niveau de sécurité.

Cette dernière condition ne fait pas intervenir la valeur des variables du système, et comme nous voulons utiliser des model checkers pour la vérification des propriétés de sécurité, il est important de réduire la taille des domaines de valeur pour éviter des problèmes d'explosion combinatoire. Pour vérifier la sécurité, il est suffisant de montrer que la propriété précédente est vérifiée dans tout état d'une abstraction du programme où les valeurs des variables ont été remplacées par leur niveau calculé.

4 Certification d'applet

4.1 Technique d'analyse globale

Deux questions se posent pour être en mesure d'appliquer l'approche en pratique : quelles sont les variables auxquelles on est capable d'associer un niveau de sécurité? et quel est le programme que nous considérons? (une méthode d'une applet, toutes les méthodes d'une applet, toutes les méthodes de toutes les applets de la carte?)

Nous sommes capables d'associer des niveaux de sécurité aux attributs d'une applet et aux invocations de méthodes entre applets. Par défaut, nous associons

le niveau AF (resp. P et RC) à tous les attributs des applets Air France (resp. Purse et RentaCar). Comme Air France peut invoquer les méthodes *getbalance* ou *debit* de RentaCar, nous associons le niveau de sécurité $RC + AF$ à ces interactions. De la même façon, comme l'applet Purse peut invoquer la méthode $logfull$ d'Air France, nous associons le niveau $AF + P$ à cette interaction. Enfin, nous associons le niveau $P + AF$ (resp. $P + RC$) à l'invocation de la méthode $getTransaction$ de l'applet Purse par l'applet Air France (resp. par l'applet RentaCar).

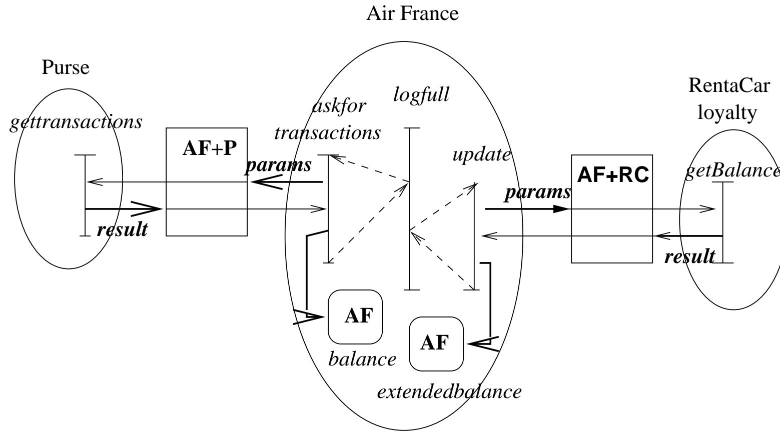

Fig. 4. Vérification "Assume-Guarantee"

Nous avons décidé d'analyser une invocation de méthode à la fois. Nous proposons une discipline "assume-guarantee" qui permet de vérifier des méthodes localement sur chaque applet, même si la méthode invoque des méthodes d'autres applets par le biais des interfaces de partage. Par exemple, la méthode $logfull$ de l'applet Air France invoque la méthode *getbalance* de RentaCar, nous allons analyser chaque méthode séparément. Nous vérifions que, dans la méthode $logfull$ de l'applet Air France, le niveau des paramètres de la méthode *getbalance* est inférieur au niveau de cette interaction (i.e. $RC{+}AF$). Et nous supposons que $RC + AF$ est le niveau du résultat de l'invocation de cette méthode. Quand nous analysons la méthode *getbalance* dans l'applet RentaCar, nous contrôlons que le niveau du résultat est inférieur au niveau de l'interaction et nous supposons que les paramètres ont le niveau $RC + AF$.

Nous adoptons la même discipline pour les attributs à l'intérieur d'une applet. Quand un attribut est lu, nous supposons que son niveau est le niveau de sécurité qu'on lui a attribué. Quand l'attribut est modifié, nous vérifions que le nouveau niveau est inférieur au niveau de sécurité de l'attribut. Cette discipline assume-guarantee à l'intérieur d'une applet permet de ne contrôler qu'un ensemble de méthodes à la fois (et non toutes en même temps).

4.2 Technique d'analyse locale

Nous construisons un modèle SMV à partir du byte code de chaque méthode. Nous ne détaillerons pas la syntaxe des spécifications SMV (voir [8]), mais il est utile pour la suite de savoir qu'une variable SMV peut être définie de deux manières différentes: soit part une équation (`var := expr;`), soit par une initialisation et une expression donnant sa valeur à l'état suivant pour chaque transition (`init(var) := val; next(var) := expr;`).

Notre méthode pour vérifier les propriétés de sécurité sur le byte code de l'application est basée sur trois éléments :

- abstraction : nous abstrayons toutes les valeurs de variables par les niveaux calculés ;
- condition suffisante : nous vérifions un invariant qui est une condition suffisante de la propriété de sécurité ;
- model checking : nous vérifions cet invariant par model checking.

Nous illustrons notre technique sur une version simplifiée de la méthode $logfull()$ de l'applet Air France. Cette méthode invoque directement la méthode *getbalance* de l'applet RentaCar et met à jour l'attribut *ExtendedBalance*.

```
Method void logfull()
   0 aload_0
   1 invokespecial 108 <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 Le byte code de la méthode $logfull()$ est modélisé par un module [8] qui contient les variables suivantes :

- *pc*: compteur de programme ;
- $mem[i]$: tableau représentant les emplacements mémoire ;
- $stck[i]$: tableau représentant la pile d'opérandes ;
- sP: pointeur de pile ;
- *ByteCode*: nom de l'instruction courante.

Les valeurs des variables sont abstraites par des niveaux. Les niveaux sont définis dans un module appelé *Levels* de telle façon qu'un niveau est représenté par un booléen. Ainsi le type des variables abstraites est booléen ou tableau de booléens. La valeur du compteur de programme n'est pas abstraite, elle donne le séquencement des instructions. De même, la valeur du pointeur de pile est conservée, elle donne l'indice du premier emplacement libre.

```
L: levels;
pc : -1..9;
mem : array 0..1 of boolean;
stck : array 0..1 of boolean;
sP : -1..1;
ByteCode : {invoke_108, load_0, return, nop, store_1, dup,
            load_1, getfield_220,op, putfield_220};
```

L'exécution du byte code commence à la ligne 0 du programme. A l'instant initial, la pile est vide, le niveau des paramètres est stocké à l'emplacement 0 de la mémoire, il est égal au niveau de l'interaction que nous analysons (i.e. niveau $AF + P$) qui est codé comme la conjonction des niveaux $L.AF$ et $L.P$.

```
init(pc):= 0; init(sP):= 1; init(mem[0]):= L.AF & L.P;
for (i=0; i< 2; i=i+1) {init(stck[i]) := L.AF & L.P; }
```

La boucle de contrôle définit la valeur du compteur de programme et de l'instruction courante. C'est une traduction presque directe du byte code java. Quand *pc* est égal à -1, l'exécution est terminée et l'instruction courante est nop qui ne fait rien. Comme dans [5], chaque instruction que nous considérons représente plusieurs instructions du byte code Java. Par exemple, comme on ne s'intéresse pas au type de mémoire et d'emplacements mémoires, l'instruction load_0 représente les instructions Java (aload_i, iload_i, lload_i,...). De façon similaire, l'instruction *op* modélise toutes les opérations binaires (iadd, ladd, iand, ior, ...).

```
(next(pc), ByteCode) :=
 switch(pc) {
  -1: (-1, nop);
  0 : (pc+1, load_0 );
  1 : (pc+1, invoke_108 );
  2 : (pc+1, store_1 );
  3 : (pc+1, load_0 );
  4 : (pc+1, dup );
  5 : (pc+1, getfield_220 );
  6 : (pc+1, load_1 );
  7 : (pc+1, op );
  8 : (pc+1, putfield_220 );
  9 : (-1, return);  };
```

La partie suivante du modèle SMV décrit l'effet des instructions sur les variables. Les instructions calculent les niveaux de chaque variable. L'instruction *load* met le niveau d'un emplacement mémoire au sommet de la pile, l'instruction *store* enlève le sommet de la pile et stocke ce niveau dans un emplacement mémoire, l'instruction *dup* duplique le sommet de pile. L'instruction *op* calcule le maximum des niveaux des deux emplacements en sommet de pile. Le maximum de deux niveaux $l1$ et $l2$ est modélisé par la disjonction des deux

niveaux $l1 \vee l2$. L'instruction *invoke* dépile les paramètres et empile le résultat de l'invocation. D'après la discipline assume-guarantee, on suppose que le niveau du résultat de l'invocation de la méthode *getbalance* est $L.AF \wedge L.RC$. L'instruction *getfield* empile le niveau de l'attribut *ExtendedBalance* qui est $L.AF$. Finalement, l'instruction *putfield* dépile le niveau de l'attribut *ExtendedBalance*.

```
switch(ByteCode) {
 nop :;
 load_0 : {next(stck[sP]):= mem[0];next(sP):=sP-1;}
 load_1 : {next(stck[sP]):= mem[1];next(sP):=sP-1;}
 store_1 : {next(mem[1]):=stck[sP+1] ;next(sP):=sP+1;}
 dup : {next(stck[sP]):= stck[sP+1]; next(sP):=sP-1;}
 op : {next(stck[sP+2]):=(stck[sP+1]|stck[sP+2]);
       next(sP):=sP+1};
 invoke_108 : {next(stck[sP]):=L.AF & L.RC;next(sP):= sP+1;}
 getfield_220 : {next(stck[sP+1]):=L.AF;}
 putfield_220 : {next(sP):=sP+2;}
 return : ;  }
```

Invariant Nous avons expliqué comment calculer un niveau pour chaque variable et quels niveaux de sécurité associer aux attributs et aux interactions. L'invariant que nous vérifions alors est que le niveau calculé des variables que nous voulons contrôler est toujours inférieur ou égal au niveau autorisé.

Pour la méthode logfull, nous allons nous intéresser à deux propriétés : une pour vérifier que l'interaction entre *logfull* et *getbalance* est correcte, l'autre pour vérifier que *logfull* utilise correctement l'attribut *ExtendedBalance*. La propriété *Smethod_108* signifie que, à chaque fois que l'instruction courante est l'invocation de la méthode *getbalance*, le niveau des paramètres transmis (le sommet de pile) est inférieur au niveau de l'interaction $AF + RC$. La propriété *Sfield_220* signifie que, à chaque fois que l'instruction courante est la modification de l'attribut *ExtendedBalance*, le niveau de la nouvelle valeur (le sommet de pile) est inférieur au niveau de l'attribut AF.

Comme la méthode *logfull* ne renvoie aucun résultat, il n'est pas nécessaire de vérifier la propriété *Sresult* qui signifie que, à chaque fois que la méthode est finie, le niveau de la valeur de retour (le sommet de pile) est inférieur ou égal au niveau de l'interaction $AF + P$.

```
Smethod_108 :
  assert G (ByteCode=invoke_108 ->(stck[sP+1] -> L.AF & L.RC));
Sfield_220  :
  assert G (ByteCode=putfield_220 ->(stck[sP+1] -> L.AF));
Sresult     :
  assert G (ByteCode=return -> (stck[sP+1] -> L.AF & L.P));
```

4.3 Exemple d'analyse

Une fois que nous avons le modèle abstrait et l'invariant, on utilise SMV [8] pour vérifier que l'invariant est satisfait par le modèle. Si la propriété n'est pas satisfaite, le model checker produit un contre-exemple qui représente une exécution du byte code amenant à un état où la propriété est fausse.

Un problème de sécurité va être détecté en vérifiant la propriété $Smethod_108$ de la méthode $logfull$. En effet, l'interaction $logfull$ entre le purse et Air France a le niveau $AF + P$. Le canal *getbalance* a le niveau $AF + RC$ et on détecte que l'invocation de la méthode *getbalance* dépend de l'invocation de la méthode $logfull$. Il y a donc une dépendance illicite d'une variable de niveau $AF + P$ vers une variable de niveau $AF + RC$. Une solution possible à ce problème est d'interdire l'appel aux méthodes d'autres Loyalty dans l'exécution du logfull.

5 Conclusion

Dans cet article, nous avons présenté une approche pour la certification d'applets chargées sur une Javacard. Les contrôles de sécurité que nous proposons sont complémentaires des fonctions de sécurité déjà présentes sur la carte. Le firewall contrôle l'interaction entre deux applets, tandis que notre analyse a une vue plus globale et permet de détecter des flux d'information illicites entre plusieurs applets.

Au niveau des performances, nous n'avons pas rencontré de problèmes pour l'instant. La vérification d'une methode est faite par SMV en quelques secondes, nous avons estimé le nombre d'analyses à faire à une trentaine sur les 250 méthodes du porte-monnaie.

5.1 Comparaison avec d'autres travaux

Beaucoup de travaux ont pour thème la sécurité et Java, mais à notre connaissance, ils ne s'intéressent pas à des propriétés de sécurité telles que nous les entendons dans cet article (interactions illicites). Ils étudient plutôt des propriétés telles que le typage correct, pas de débordement de la pile, etc. Une exception est [6] qui propose une vérification des flots de contrôle, mais ne traite pas les flots de données.

Parmi ces travaux, on peut distinguer deux types d'approches selon le type de technique utilisée pour la vérification. La plupart des approches sont basées sur l'analyse statique et proposent notamment des systèmes de type [2, 10]. Une approche utilise le model checking (SMV) pour spécifier du byte code Java [9] mais pour vérifier des propriétés différentes.

5.2 Perspectives

Les perspectives incluent l'automatisation de la production du modèle SMV à partir du byte code Java. L'automatisation complète est cependant difficilement réalisable : une interaction avec l'utilisateur sera nécessaire pour définir les niveaux et les attribuer aux attributs et aux interactions entre applets.

Un autre problème intéressant est l'analyse des résultats. Quand SMV produit un contre-exemple pour une propriété de sécurité, il faut pouvoir interpréter ce contre-exemple au niveau applicatif.

References

1. P. Bieber and F. Cuppens. A Logical View of Secure Dependencies. *Journal of Computer Security*, 1(1):99–129, 1992.
2. 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.
3. Pierre Girard. Which security policy for multiapplication smart cards? In *USENIX workshop on smartcard technology*, 1999.
4. J. Goguen and J. Meseguer. Unwinding and Inference Control. In *IEEE Symposium on Security and Privacy*, Oakland, 1984.
5. 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.
6. 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.
7. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.
8. K.L. McMillan. *The SMV language*. Cadence Berkeley Labs, 1999.
9. 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.
10. Raymie Stata and Martin Abadi. A type system for Java bytecode subroutines. In *Proc. 25th Symposium on Principles of Programming Languages*, 1998.
11. Sun Microsystems. *Java Card 2.1 Realtime Environment (JCRE) Specification*, February 1999.