# Intégration d'une politique de flot de contrôle dans un automate de sécurité

Guillaume Bouffard (guillaume.bouffard@xlim.fr)*
Mathieu Lassale (mathieu.lassale@etu.unilim.fr)*
Sergio Ona Domene (sergio.ona@etu.unilim.fr)*
Hanan Tadmori (hanan.tadmori@xlim.fr)*
Jean-Louis Lanet (jean-louis.lanet@unilim.fr)*

**Résumé :**
La carte à puce est un objet contenant des informations sensibles. Les attaques par injection de faute sont les plus difficiles à se prémunir. La surveillance du flot d'exécution par la machine virtuelle est un des mécanisme permettant de s'en protéger. Nous intégrons dans l'interpréteur Java Card un automate de sécurité permettant de garantir une politique de sécurité. Nous montrons comment une optimisation permet d'obtenir une solution efficace tant dans son utilisation mémoire que dans la surcharge de calcul.

**Mots Clés :** Attaques en faute, carte à puce, automate de sécurité, contre-mesure

## 1 Introduction

Les cartes à puce sont de petits objets sécurisés que nous utilisons au quotidien. Leurs applications sont nombreuses : carte bleue, passeport électronique, carte SIM, .... Ces objets contiennent donc des informations sensibles ou des programmes qui doivent être protégés. Depuis ses débuts, les cartes à puce sont sujettes à de nombreuses attaques matérielles et logicielles. Actuellement, les attaques les plus difficiles à contrer sont les attaques en fautes. Dû à son environnement d'exécution, les puces de ces cartes peuvent être soumises à des perturbations induites par des fautes extérieures. Ces fautes peuvent provenir d'une source naturelle (rayon cosmique), environnementale (rayon X, température), ou malveillante (injection précise de faute). Les conséquences peuvent être la modification dynamique des registres de la carte (pointeur d'instruction, le pointeur de pile, ...) ou l'altération de la mémoires inscriptibles contenant le code et les données d'un programme. Ces perturbations ont différents effets sur la carte, accès non autorisé à des données ou des services, par exemple.

La carte à puce est un système qui nécessite une alimentation externe pour fonctionner. C'est donc au lecteur de fournir cette source d'énergie. Il est alors possible que le lecteur soit malveillant, c'est le mécanisme le plus simple pour injecter des fautes, via des équipements spéciaux capables modifier les signaux transmis. De courtes variations de courant, peuvent provoquer des erreurs logicielles dans la carte à puce [AK98]. Ces modifications ont pour effet de perturber la mémoire ou de corrompre un programme offrant l'exécution arbitraire d'instructions.

*. Université de Limoges, 123 Avenue Albert Thomas, 87060 Limoges

Grâce à des procédés chimiques, la couche de silicium de la puce peut être rendu visible. Il est alors possible avec un laser, d'injecter des fautes dans la mémoire ou sur les bus de la carte. Les cellules de la mémoire persistante sont sensibles à la lumière, dû à l'effet photoélectrique. Un laser moderne peut cibler une zone très petite de la puce et injecter précisément une faute [Bar12]. Afin de se prévenir d'une injection de faute, il est nécessaire d'en connaître ses effets. [BOS03, Wag04] ont défini en détail un modèle de faute. Nous considérons dans ce travail qu'un attaquant peut modifier a un instant précis, pour une durée précise, un octet à la fois [VF10].

Ce travail s'inscrit dans le cadre du projet INOSSEM (INter Opérabilité Sécuritaire des Services EMbarqués) dont l'objectif est de fournir les moyens pour garantir interopérabilité sécuritaire entre des éléments de sécurité multi-applicatifs comme les cartes à puces. Afin de minimiser le coût d'une certification, il est impératif de pouvoir garantir un comportement identique d'une plate forme à une autre. Aujourd'hui, ceci est garanti au niveau fonctionnel grâce à une machine virtuelle (VM) Java Card et des API standardisées. Il est nécessaire pour obtenir cette interopérabilité pour la sécurité de définir une couche d'abstraction et nos travaux s'intéressent à fournir un service garantissant un mécanisme de détection de faute indépendant de la plateforme. Nous proposons donc dans cet article, un outil de ré-écriture prenant en entrée une application non sécurisée contre les fautes. Il réalise une analyse statique et nous tissons dans le code les appels nécessaires à la base de confiance représentée par l'interpréteur de la Java Card. Ce dernier est modifié afin de contrôler grâce à un automate représentant le flot de contrôle qu'aucune perturbation n'a eut lieu.

Cet article se déroulera dans l'ordre suivant. Tout d'abord, dans la section 2, nous allons étudier comment écrire un code protégé contre les attaques en faute. Ensuite, nous analyserons les moniteur de référence pour Java Card avec une analyse expérimentale dans la section 3. Enfin nous ouvrirons le sujet lors de la conclusion.

## 2 Écrire du code sécurisé

Dans un programme, une faute est la principale raison qui conduit à une défaillance du système. Afin d'éviter une telle erreur, les fautes doivent être détectées le plus tôt possible afin de corriger ou arrêter le(s) service(s) corrompu. Pour mettre en place un mécanisme efficace il est nécessaire de comprendre l'effet d'une faute sur le programme.

### *2.1 Les effets des fautes*

Un attaquant peut injecter de l'énergie dans une cellule mémoire afin de modifier son état. Suivant la technologie utilisée, le bloc mémoire va physiquement prendre la valeur `0x00` ou `0xFF`. Si la mémoire est chiffrée, la valeur physique stockée sera alors inconnue [1].

La faute peut entraîner de multiples erreurs. Si la cellule mémoire contient un programme, c'est l'intégrité de l'application qui en est affectée causant une potentielle désynchronisation du code [2]. La sémantique du programme en est perturbée. Si une donnée du programme mute, c'est le graphe de flot de contrôle (CFG) du programme qui en est affecté. Nous donnons ci-après quelques exemples d'attaques décrit dans le projet INOSSEM :

1. Elle dépendra des données chiffrées, de l'adresse où elles sont stockées et de la clef utilisée.
2. Les opérandes sont évaluées comme des instructions.

- `PERTURB_INVOKE_BYPASS` : cette attaque à pour effet d'empêcher l'appel d'une méthode sensible. Le CFG en est alors modifié.
- `PERTURB_CONDITION` : le but de cette attaque est de perturber le fonctionnement d'une condition afin d'exécuter une branche plutôt qu'une autre. Cette attaque peut être réalisé par plusieurs moyens. L'attaquant a modifié le code de telle manière qu'au lieu d'exécuter l'instruction `IFEQ`[3], il a exécute l'instruction `IFNEQ`. Il peut aussi perturber, avant ce test, l'instruction (de type `sload n`) poussant au sommet de la pile la variable devant être évaluée. En modifiant l'index de cet instruction, le programme va montée une autre donnée sur la pile. Modifiant ainsi la test effectué. Enfin, un autre moyen de perturber la condition est de modifier directement l'information présente au sommet de la pile d'évaluation.
- `PERTURB_CMP_DATA` : Cette attaque à pour but de forcer la non exécution d'une boucle de comparaison faite sur des données sensibles. Une telle boucle est présentée dans le Listing 2.

**Listing 1:** Boucle de comparaison

```
boolean res = true; //succès
  for(s=0 ; s<longueur ; ++s) res = res & (bufferA[s]==bufferB[s]);
return res;
```

Si les tableaux sont stockés dans des containers sécurisés, et que l'opération de comparaison est elle même sécurisée, on peut imaginer que l'attaque ne peut pas avoir d'impact. Cependant, dans la boucle `for`, un ET logique est fait entre le résultat de cette copie et un short poussé au sommet de pile. Si une perturbation à lieu sur les variables au sommet de la pile, alors le résultat est faussé et fait sortir de la boucle. Un attaquant peut aussi profiter d'une attaque de type `PERTURB_CONDITION` pour obtenir le même effet.

Les fautes présentées ont un effet à la fois sur la confidentialité et l'intégrité des biens d'un programme, perturbant fonctionnement attendu d'un service. Il est en effet difficile de s'en prémunir en ayant aucune connaissance des biens à protéger. Le développeur de l'application est le seul qui connaît le mieux les biens qu'il faut protéger. C'est donc à lui de définir comment protéger les biens de son programme. Pour cela, nous allons faire un état de l'art des contre-mesures existantes.

## 2.2 Contre mesures applicatives

Il y a deux grandes approches pour la détection de faute : l'analyse statique et l'analyse dynamique. La première consiste à examiner le code d'un programme pour déterminer des propriétés sans exécution du code. Cette technique a été très souvent utilisée pour valider des propriétés sur des applications dédiées aux cartes à puce. L'analyse dynamique consiste, quant à elle, à exécuter le programme afin d'analyser des comportement non souhaités. L'analyse statique a des avantages par rapport à l'analyse dynamique :

- L'analyse statique permet un recherche exhaustive et la validation de la propriété est réalisée pour toutes les traces d'exécution. Par opposition, une analyse dynamique n'est valide que pour les cas évalués.

3. Le test est validé si, et seulement si, la valeur testé est égale à 0. À l'instar du monde C, dans le monde Java binaire, la valeur 0 signifie faux, le reste interprété comme vrai.

– L'analyse est faite avant toute exécution ce qui est important dans le domaine de la sécurité où l'échec d'une fonction donne de l'information à l'attaquant.
– Aucun surcoût d'exécution.

Néanmoins, il est impossible statiquement de garantir certaines propriétés. Seule donc une vérification dynamique le permettre. Parfois, une approche mixte (statique et dynamique) est utilisée afin de valider une partie des propriétés.

Dans le monde de la carte, le développement de contre-mesures répond à une ou à une série d'attaques. Cela rend les contre-mesures actuelles souvent imparfaite face aux menaces existants. Il est en effet très difficile de développer une protection peu coûteuse, ayant une couverture maximale.

## 2.3 Approches statiques

Une approche statique garantit que chaque test est correctement effectué et que le CFG est inchangé. Afin de vérifier si une condition s'est correctement effectuée, l'usage d'une redondance dans la branche sensible est utilisée. Si une faute est injectée durant la réalisation du test `if`, un attaquant pourrait exécuter un branche spécifique sans contrôle. En pratique, injecter deux fautes simultanément est très difficile. Pour se prévenir d'une condition fautée, une double condition imbriquée est utilisée dans chaque branche sensible, vérifiant la présence des conditions. Un exemple est disponible dans le Listing 2.

**Listing 2:** Condition `if` de $2^{d}$ ordre

```
// condition est un booléen
if(condition) {
  if(condition) {
    // Opération sensible
  } else {/*Attaque détectée!*/}
} else {
  if(!condition) {
    // Accès non autorisé
  } else {/*Attaque détectée!*/}
}
```

**Listing 3:** Compteur d'état

```
short compteur_etat=0;
if(compteur_etat==0) {
  // Opération sensible 1
  compteur_etat++;
} else {/*Attaque détectée!*/}
/* ... */
if(compteur_etat==1) {
  // Opération sensible 2
  compteur_etat++;
} else  {/*Attaque détectée!*/}
```

Il va de soi que la double condition `if` ne garantit pas l'intégrité du CFG du programme. Pour s'en assurer, le développeur peut implémenter un compteur d'état, décrit dans le Listing 3. Grâce à cette méthode, chaque nœud, défini par le développeur, est vérifié durant la phase d'exécution. Si, pour un nœud exécuté, une valeur de compteur d'état est incorrecte, une attaque est alors détectée.

## 2.4 Approches mixtes

Contrairement aux approches statiques décrite précédemment, les contre-mesures mixtes sont basées sur les fonctionnalités proposées par le système afin garantir l'exécuter sans modification du programme. Les contre-mesures présentées dans cette partie nécessite une analyse statique. Les résultats seront réutilisés durant la phase d'exécution du programme offrant ainsi une approche peu coûteuse pour la carte.

Pour s'assurer de l'intégrité du code, Prevost *et al.* ont breveté en 2004 [PS04] une méthode où, pour chaque bloc de base du programme, une somme de contrôle est calculée. Le programme est ensuite installé sur la carte avec les sommes de contrôle associées.

Durant l'exécution, la carte vérifie la somme de contrôle de chaque basic bloc exécuté. Si la somme est incorrecte, un comportement anormale est alors détecté.

En 2010, dans son mémoire de thèse [AKS10], Séré décrit trois contre-mesures basées sur les champs de bits, les basic blocs et la vérification du chemin d'exécution pour protéger la carte des attaques par injection de faute. La carte va alors vérifier dynamiquement le CFG de l'application courante. Comme les calculs sont effectués statiquement, ses contre-mesures ont une empreinte faible sur l'exécution de la carte.

Activer plusieurs contre-mesures systèmes en même temps est très coûteux pour la carte. De plus, elles ne sont pas toutes utiles tout au long de l'exécution. Pour n'exécuter que les contre-mesures nécessaires, Barbu *et al.* ont présentés en 2012 [BAG13], le principe d'activer les contre-mesures à la volée par le développeur.

# 3 Un moniteur de référence pour Java Card

La nature dynamique de l'attaque empêche toute solution statique. Par exemple, vérifier par un contrôle d'intégrité avant exécution que le programme stocké est bien celui qui a été chargé est particulièrement inefficace. Si l'attaquant modifie les instructions sur le bus, le code est intègre mais l'exécution ne l'est pas. Vérifier l'intégrité de chaque instruction avant chaque exécution garantira effectivement l'intégrité de l'exécution, cependant le coût est tellement prohibitif que la VM sera inutilisable. Une bonne contre mesure est toujours un dosage entre capacité de détection et surcoût d'exécution.

L'utilisation de moniteur d'exécution pour surveiller une application est assez ancienne [And72]. Toutes les activités de l'application sont surveillées soit par des mécanismes matériels, soit de manière logiciels. Il doit garantir au moins trois propriétés :

- Toutes les opérations de base relevant des propriétés de sécurité doivent être évaluées par le moniteur de référence,
- L'intégrité du moniteur doit être garantie et généralement il fait partie de la TCB (*Trusted Computing Base*),
- La correction du moniteur doit être assurée. Pour cela, il doit être le plus petit possible afin d'être analysé et testé ou mieux, prouvé.

Les moniteurs de référence intègrent un mécanisme pour filtrer les évènements relevant de la sécurité et un mécanisme pour évaluer ces évènements.

## *3.1 Les automates de sécurité*

Schneider définit dans [Sch00] un automate de sécurité, basé sur un automate de Büchi, comme un triplet (Q, $q_0$, $\delta$) où Q est un ensemble d'état, $q_0$ l'état initial et $\delta$ un ensemble de fonction de transition $\delta : (\mathrm{Q} \times \mathrm{I}) \rightarrow 2^Q$. L'ensemble S représente les symboles d'entrée, l'ensemble des évènements liés à la propriété de sécurité. L'automate évalue une séquence de symboles d'entrée $s_1$, $s_2$, ... et chaque symbole est évalué un-à-un. Pour chaque action valide, l'état est modifié en partant de l'état initial $\{s_0\}$. À chaque évènement $s_i$, l'automate change d'état Q' in $\cup_{q \in Q'} \delta(s_i, q)$. Si l'automate accepte la transition, alors il change d'état sinon il arrête l'exécution du programme. Schneider montre que ce mécanisme permet de vérifier toute propriété de sûreté contrôle d'accès..., mais pas les propriétés de *liveness* comme la disponibilité, les flux d'informations...

Dans la carte à puce Java Card, l'interpréteur permet de garantir les propriétés du moniteur. L'application ne peut pas accéder au moniteur (isolation via l'interpréteur),

tous les évènements du langage sont évalués par l'interpréteur et ce dernier est certifié par une évaluation Critère Commun [ANS10]. Cependant, [And72] montre que le surcoût d'un interpréteur est trop élevé car il nécessite l'évaluation de chaque instruction machine.

### 3.2 Politique de sécurité CFI

Abadi présente dans [ABEL05] l'application au renforcement d'une politique de sécurité appelée CFI (CFG *Integrity*) garantissant à l'aide d'un moniteur de référence l'exécution correct du programme. Il nécessite une ré-écriture du code binaire et une vérification dynamique. Il note que les attaques n'affectant pas le CFG pourront éviter ce mécanisme. Le graphe peut être déterminé par une analyse statique. La méthode consiste à instrumenter le code grâce à des étiquettes qui vont être transformées en code de vérification des sauts. Pour ce faire, les concepteurs de cette contre-mesure proposent d'insérer des motifs uniques de destination des sauts et des instructions de vérification de ces motifs. Chaque instruction de transfert du CFG est étiqueté et le code va vérifier pendant le transfert du CFG que le motif correspondant au transfert est bien un de ceux qui ont été enregistrés pour cette vérification. Si ce n'est pas le cas, c'est qu'il y a probablement eu une perturbation du CFG donc l'exécution du code doit s'arrêter. Cette méthode s'attache au flot de contrôle initial mais ne permettrait pas de détecter l'attaque `PERTURB_CONDITION` en cas de modification des instructions, ni si la donnée évaluée est modifiée : les deux étiquettes de destination étant valides.

La politique de sécurité CFI peut être détectée par un automate reconnaissant des expressions régulières. En effet, elle ne dépend que des évènements passés et courants et jamais du futur, c'est bien une propriété de sûreté. Le Listing 4 est extrait du protocole de paiement sur internet présentée par Gemalto [GVLP10], composé d'une initialisation d'un tableau par une boucle et de deux appels de méthode `update()` pour initialiser le pin code et `register()` pour valider l'applet auprès du système.

**Listing 4:** Constructeur de l'application de paiement

```
protected Protocolpayment (byte[] buffer, short offset, byte length) {
   A[0] = 0;  // Initialisation du tableau A
   for (byte j = 0; j < buffer[(byte)(offset+12)]; j++)
      D[j] = 0; // Initialisation du tableau D
   // Création du PIN
   pin = new OwnerPIN((byte) NB_ESSAIES_MAX, (byte) LONGUEUR_PIN);
   pin.update(myPin, (short) 0, (byte) LONGUEUR_PIN); // Initialisation
   register();  // Enregistrement de l'instance
 }
```

L'ensemble $S$ est donc composé des éléments du langage permettant d'exprimer la politique CFI, c'est-à-dire les instructions binaires de déroutement du CFG, `ifeq`, `ifne`, `goto`, `invoke`, `return`, ... ainsi qu'une instruction fictive `join` marquant chaque instruction ayant un label. Dans cet exemple, le nombre de tours de boucle dépend d'une variable passée en paramètre, il est donc inconnu a priori et ne peut être calculé statiquement. Il est nécessaire d'utiliser une expression régulière pour exprimer la trace. Le CFG de ce fragment de programme est donné Figure 1. Le premier se termine par un `goto`, la fin du second bloc précède un label donc un `join` et le dernier s'achève par un `return`. Les blocs contiennent des appels à des méthodes, le premier est le constructeur de la super classe. Dans le quatrième bloc, il y a le constructeur de `OwnerPIN` suivi de la méthode `update`

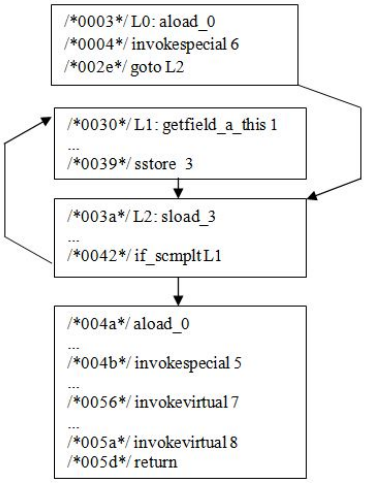


**Figure 1:** CFG du constructeur de l'applet

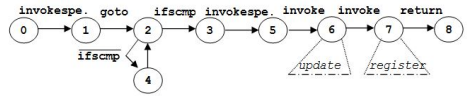


**Figure 2:** Automate du constructeur de l'applet

puis finalement la méthode `register`.

Chaque méthode invoquée aura à son tour, son propre CFG. La trace $T$, reconnue pour cette méthode, sera donc l'expression (`invokespecial goto ifscmpt`$^*$ `join invokespecial invoke`$^2$ `return`). L'automate reconnaissant ce langage est représenté Figure 2.

Cette technique permet de se prémunir contre les modifications du CFG mais ne peut rien contre les modification à l'intérieur des blocs de base. Dans les contre-mesures applicatives, on utilise souvent des compteurs d'étape comme décrit dans la section 2.3. Pour automatiser ce processus, il suffit d'inclure un appel à une méthode avant et après afin d'obliger le moniteur à s'assurer que les pas ont été réalisés, c'est l'objectif de la méthode `setState()`. Ceci force l'insertion d'un `invoke` dans le code binaire.

L'attaque `PERTURB_INVOKE_BYPASS` est contrée par cette contre mesure. Pour `PERTURB_CONDITION` et `PERTURB_CMP_DATA`, seule une partie des chemins d'attaque est couverte. Ce mécanisme ne permet pas de détecter une attaque par modification de la valeur au sommet de la pile qui peut être dû à une attaque directe ou bien à une attaque sur l'instruction ayant amené l'information au sommet de la pile. Il faut ajouter une contre-mesure système pour que l'attaque soit entièrement couverte. Un calcul d'intégrité de la pile, ou une redondance de stockage est nécessaire, différentes techniques ont déjà été présentée dans la littérature [Bar12].

# 4 Évaluation de l'approche

La difficulté est de proposer une implémentation efficace, permettant de détecter les chemins d'attaque, avec un coût acceptable pour un composant aussi restreint qu'une carte à puce. Il faut donc minimiser le nombre d'évènements à surveiller afin d'éviter de rajouter un surcoût à la boucle d'interprétation. La construction de l'automate spécifiant la politique de sécurité doit être réalisé automatiquement et statiquement. L'approche combine

donc une analyse statique pour optimiser l'exécution, une analyse lors du chargement pour construire le CFG et une analyse dynamique minimale pour contrôler l'exécution.

### 4.1 Implémentation

Un appel/retour de fonction en Java est coûteux car il consiste à construire une nouvelle *frame* en mettant à jour des adresses de retour, le nombre de locales, le nombre d'arguments et le contexte de sécurité courant. Il doit être aussi garantie que le sommet de pile de dépassera pas la limite de la mémoire allouée à la pile (quelques centaines d'octets). L'opération de retour faisant l'opération inverse. Nous avons noté expérimentalement qu'un transfert au sommet de la pile (opération simple) coûte de l'ordre de 1$\mu$s, quand un appel/retour de *frame* avoisine les 30 $\mu$s. L'appel à une méthode comme `setstate()` est forcément coûteux. Il faut donc remplacer l'instruction `invokestatic` par un simple `goto` sur l'instruction suivante qui se traduira par un état et une transition supplémentaires dans l'automate. Ce traitement est réalisé en dehors de la carte par un outil d'optimisation.

La VM doit être adaptée pour d'une part, construire à la volée, lors de la phase d'édition de lien, l'automate du CFG, puis, dans l'interpréteur, pour évaluer les instructions modifiant le flot de contrôle. Chaque `frame` dispose de sa variable `currentState`, initialisée, lors de la construction de la `frame`, à zéro. L'automate `SM` est stocké dans une matrice ayant une ligne par état et plusieurs colonnes. Une colonne mémorise le code de l'instruction. Le Listing 5 montre comment l'automate s'assure que l'instruction décodée est bien celle stockée. Cela permet de se prémunir contre les fautes temporaires, ligne 2. Si l'évaluation de la condition est vraie, le programme Java pointe la destination, ligne 4, sinon il s'assure que la prochaine instruction appartient bien au bloc suivant. Lorsque l'automate refuse les transitions, il exécute la macro `ACTION_BLOCK` destinée, soit à bloquer le cycle de vie de l'application, soit à rendre la carte muette.

**Listing 5:** Fonction de transfert pour le byte code `if_scmplt`

```
1 int16 BC_if_scmplt(void) {
2     if (SM[frame->currentState][INS] != *vm_pc)          return ACTION_BLOCK;
3     vm_sp -= 2;
4     if (vm_sp[0].i < vm_sp[1].i)                        return BC_goto();
5     if (SM[frame->currentState][NEXT] != state(vm_pc)) return ACTION_BLOCK;
6     vm_pc += 2;
7     return ACTION_NONE; }
```

La fonction `state(vm_pc)` permet de s'assurer que le `jpc` (le pointeur d'instruction Java) est compatible avec l'état courant : il appartient au domaine de cet état.

Le second outil développé est l'analyseur statique permettant, à l'utilisateur, de spécifier une politique de sécurité complémentaire par ajout de points de vérification supplémentaires. Ces points sont composés d'arc et de nœud que l'utilisateur peut relier à sa guise, et une phase de vérification s'assurant ensuite de la cohérence de l'automate ainsi généré par rapport au CFG. Ensuite, l'outil insère dans le code final (binaire) des instructions `goto` afin que l'analyseur embarqué le considère comme la fin d'un état de l'automate.

Si le code n'est pas destiné à être embarqué dans une carte à puce ayant un interpréteur gérant les automates de sécurité, alors il génère uniquement des appels à l'API INOSSEM qui génère l'automate de sécurité. Cette dernière option est utilisable si la carte n'intègre

pas l'API INOSSEM. Le code de gestion de l'automate est tissé automatiquement dans le code source au travers des points insérés par l'utilisateur au travers l'interface graphique.

### 4.2 Métriques

La transformation de la VM intervient à deux endroits lors du chargement et lors de l'exécution. Le surcoût de notre implémentation doit être évalué en terme d'occupation mémoire ROM, RAM et EEPROM. La RAM étant la ressource la plus critique, nous nous focaliserons essentiellement sur celle ci. La structure de la *frame* Java Card a été modifiée en ajoutant un octet, afin d'inclure la valeur courante de l'état de l'automate. Cet octet est évidement dupliqué à chaque appel de méthode. Le coût est donc d'un octet fois la profondeur maximum de la pile. La seconde mémoire importante est l'EEPROM elle contient la matrice `SM` pour chaque méthode. Il s'agit d'un tableau à deux dimensions. Une entrée particulière est allouée pour gérer les instructions à destination multiples `tableswitch`, `lookupswitch`, ... Nous n'avons pas optimisé cette structure afin de conserver un accès en $O(1)$. Pour toutes applications installées (API, Applet ROMizées, ...) lors de la fabrication, cette table sera stocké dans la zone ROM. La consommation de mémoire RAM est quasi nulle, seule l'EEPROM sera utilisée pour les applications chargée après émission de la carte.

La seconde métrique concerne le surcoût d'exécution. Chaque instruction Java Card se décompose en deux cycle : *prefetch* et *execute*. Le *prefetch* est fixe et sur le processeur ARM7 et coûte 0.96$\mu$s. L'*execute* coûte, pour l'instruction `if_scmplt`, 0.615$\mu$s. La modification de la VM augmente le temps de traitement de 0.332$\mu$s. l'instruction complète passe donc de 1.575$\mu$s à 1.907 $\mu$s, soit un surcout de 19%. Seules 45 instructions ont un surcoût (celles évaluées par l'automate) sur les 184 instructions disponibles. Pour l'exemple donné par le constructeur de l'application de paiement, seules 7 instructions sur les 93 ont un surcoût.

## 5 Conclusion

Nous avons cherché à résoudre une partie du problème de l'interopérabilité sécuritaire entre différentes plateforme de carte à puce. Nous avons proposé une solution à base d'automate de sécurité implémentant une politique CFI. Nous avons adapté une VM Java Card pour intégrer les fonctions de transition de l'automate. Ce choix permet d'optimiser l'empreinte mémoire et surcoût d'exécution. Nous avons proposé une interface graphique pour personnaliser la politique de sécurité par l'ajout d'étapes dans les blocs de base. Notre analyseur statique tisse le code adéquat dans l'application suivant trois niveaux : coût minimal pour l'application (intégration d'instructions `goto`), appels à l'API INOSSEM et intégration complète dans le binaire de l'automate et de la gestion des transitions.

Nous pensons qu'il est possible d'intégrer d'autres politiques, de plus haut niveau, permettant, entre autre, de garantir des séquences d'appels. À ce moment là, l'automate doit prendre en compte non plus la commande courante mais une séquence de commandes dans la session en cours. L'objectif ne sera plus de garantir une exécution sous hypothèse de faute mais une politique de sécurité plus classique.

## 6 Remerciements

Ce travail est partiellement financé par le projet INOSSEM (PIA-FSN2-Technologie de sécurité et résilience des réseaux).

## Références

[ABEL05] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *12th ACM conference on Computer and communications security*, pages 340–353, USA, 2005.

[AK98] R. Anderson and M. Kuhn. Low Cost Attacks on Tamper Resistant Devices. In *Security Protocols*, volume 1361 of *Lecture Notes in Computer Science*, pages 125–136, Berlin / Heidelberg, April 1998. Springer.

[AKS10] A. Al Khary Sere. *Tissage de contremesures pour machines virtuelles embarquées.* PhD thesis, Université de Limoges, 123 Avenue Albert Thomas, 87100 Limoges Cedex, 2010.

[And72] J. P. Anderson. Computer Security technology planning study. Technical report, Deputy for Command and Management System, USA, 1972.

[ANS10] ANSSI. *Protection Profile (U)SIM Java Card Platform Protection Profile - Basic Configuration, ANSSI-CC-PP-2010/04 12/07/2010.* ANSSI, 2010.

[BAG13] G. Barbu, P. Andouard, and C. Giraud. Dynamic Fault Injection Countermeasure A New Conception of Java Card Security. In *Smart Card Research and Advanced Applications*, number 7771 in Lecture Notes in Computer Science, pages 16–30. Springer, 2013.

[Bar12] G. Barbu. *On the security of Java Card platforms against hardware attacks.* PhD thesis, Grant-funded PhD with Oberthur Technologies and Télécom ParisTech, 2012.

[BOS03] J. Blömer, M. Otto, and J.-P. Seifert. A new CRT-RSA algorithm secure against bellcore attacks. In *10th conference on Computer and communications security*, pages 311–320, 2003.

[GVLP10] P. Girard, K. Villegas, J.-L. Lanet, and A. Plateaux. A new payment protocol over the Internet. In *Risks and Security of Internet and Systems (CRiSIS), 2010 Fifth International Conference on*, pages 1–6, 2010.

[PS04] S. Prevost and K. Sachdeva. Application code integrity check during virtual machine runtime, August 2004.

[Sch00] B. Schneider. Enforceable security policies. *Information and System Security*, 3(1) :30–50, February 2000.

[VF10] E. Vetillard and A. Ferrari. Combined Attacks and Countermeasures. In *Smart Card Research and Advanced Application*, volume 6035 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 2010.

[Wag04] D. Wagner. Cryptanalysis of a provably secure CRT-RSA algorithm. In *11th ACM conference on Computer and communications security*, pages 92–97, 2004.