UNIVERSITÉ DE LIMOGES
ÉCOLE DOCTORALE « Sciences et Ingénierie pour l'Information »
FACULTÉ DES SCIENCES ET TECHNIQUES

Thèse N° 29-2010

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE LIMOGES

Discipline / Spécialité : Informatique et Applications

présentée et soutenue par

Ahmadou Al Khary SÉRÉ

le 23 Septembre 2010

Tissage de contremesures pour machines virtuelles embarquées

Thèse dirigée par Jean-Louis Lanet,
Co-encadrée par Julien Iguchi-Cartigny

JURY

Rapporteurs :	
Mme Assia TRIA	Professeur à EMSE Gardannes
Mme. Isabelle SIMPLOT-RYL	Professeur à Université de Lille 1
Examinateurs :	
M. Pierre GIRARD	Ingénieur de recherche Gemalto
M. Aurélien FRANCILLON	PhD, ETH Zurich
M. Philippe GABORIT	Professeur à l'Université de Limoges
M. Jean-Louis LANET	Professeur à l'Université de Limoges
M. Julien IGUCHI-CARTIGNY	Maître de Conférences à l'Université de Limoges

Remerciements

Je tiens en premier lieu à adresser mes sincères remerciements, ainsi que ma profonde gratitude aux Professeurs Assia Tria et Isabelle Ryl, d'avoir trouver le temps de jouer leur rôle de rapporteur malgré un emploi du temps chargé, ainsi qu'à Pierre Girard, Aurélien Francillon et Philippe Gaborit d'avoir accepté d'être mes examinateurs.

Je tiens également à remercier mon directeur de thèse Jean-Louis Lanet pour la confiance qu'il m'a accordée, pour ses conseils toujours éclairés et avisés, ainsi que pour les «coups de fouet» qui m'ont permis d'avancer dans les moments difficiles.

Je tiens aussi à remercier Julien Iguchi-Cartigny d'avoir accepté d'encadrer cette thèse. Il a fait preuve d'une grande disponibilité, d'une grande écoute à mon égard, et de beaucoup de patience. Il m'a aussi apporté beaucoup en terme d'orientation de mes travaux.

Je remercie également ma fiancée Cécile de m'avoir soutenu dans les moments les plus difficiles ces dernières années.

Merci à tous les doctorants et membres de JIDE sans qui ces trois années de thèse n'auraient pas été les mêmes. Je pense tout particulièrement à Alex, Agnès, Amaury, Boussad, Céline, Cyril, Emma, Mickael, Nadir, Nassima, Pierre, Richard. Je n'oublierai pas les pauses café (très régulières) ainsi que les soirées que nous avons pu passer à JIDE.

A Hubert, et Laurent pour leur amitié.

Merci à tous mes amis qui m'ont soutenu hors du contexte de la thèse Didier, Gaelle, Josselin, Matsouma, Paul-Nana, Saminou, Vincent et beaucoup d'autres. J'en oublie certains, mais cela ne veut pas dire que je ne pense pas à eux au moment où j'écris ces mots.

Je remercie tout particulièrement Cricri, Sylvie, Sandrine, Julien, Clément, Matthieu d'avoir eu la patience de me relire. Ainsi que Frédéric pour sa disponibilité.

Je terminerai avec une très forte pensée à ma famille et plus particulièrement mon père Alassane et ma mère Assita, pour leur soutien, pour avoir cru en moi tout du long. Merci aussi à mon frère Ahmed et mes deux soeurs Zara et Malicka.

Resumé

Les cartes à puces sont de petits ordinateurs pouvant être transportés dans nos activités quotidiennes et qui bénéficient de capacités de calcul et de fonctions de sécurité (algorithmes cryptographiques). Elles sont utilisées dans divers domaines tels que la banque (carte de crédit ou de débit), les services du gouvernement (carte d'identité, passeport, permis de conduire), la téléphonie (carte SIM et USIM), la télévision à péage, etc.

La plateforme que nous avons retenue pour notre travail de recherche est la plateforme Java Card qui apporte des fonctionnalités multiapplicatives aux cartes à puce. Nous nous sommes intéressés aux attaques par injection de fautes qui utilisent des moyens physiques pour perturber la carte durant l'exécution des applications. Ces perturbations pouvant conduire à la modification des mémoires, à des erreurs du CPU, etc. Elles peuvent dans une certaine mesure permettre d'éviter certains tests importants pour la sécurité des applications et des données contenues dans la carte : la vérification du mot de passe de l'utilisateur ou celle de clefs cryptographiques. Elle constitue donc une catégorie d'attaque très puissante capable de porter atteinte à l'intégrité de la carte.

Notre objectif est de proposer des moyens de garantir que l'on peut détecter une injection de fautes causant la perturbation de la puce contenue dans la carte. Nous souhaitons que cette détection soit automatique tout en respectant les contraintes de ressources (mémoire et processeur) de la carte. Pour arriver à cet objectif, nous utilisons des annotations qui permettent au programmeur de marquer les méthodes et classes sensibles de son application, permettant à la machine virtuelle de passer dans un mode sécurisé lors de leur exécution. L'approche proposée est d'introduire des informations de sécurité dans le code et de modifier la machine virtuelle afin qu'elle les utilise pour détecter les attaques. Ces recherches ce sont focalisées sur le développement de nouveaux moyens de lutter contre les attaques par injection de fautes, en vérifiant l'intégrité du code ou celui du flot de contrôle.

Mot-clefs : *Carte à puce, java card, attaques par injection de fautes, intégrité du code, flot de contrôle*

Abstract

Smart Cards are small computers, small enough to be carried with us in our daily activities. They have computation capabilities and security functionalities (cryptographic computation), which allow their use in many domains like bank (credit or debit card), e-government (Identity card, and e-passport), telephony (SIM card, and USIM card), pay TV, and many more domains.

The platform that we choose, is Java Card that brings multi-applicative functionality to smart cards. We are concerned by fault attacks that use physical means to disturb the card activity during applications execution causing trouble that can lead to code or memory modification, to CPU glitches, etc. These consequences can allow bypassing (control flow modification) some crucial tests for the card security like cryptographic keys or pin code verification. Hence, they constitute a powerful class of attacks capable of lowering the card's integrity.

Our goal, is to propose some way to guaranty that we can detect that a fault attack occurs tampering the smart card chip. And we want to do this in an automatic way that is affordable in resources (memory and CPU) for the card. To achieve this goal, we use annotations that allow a programmer to tag sensitive methods or classes of his application. Allowing the virtual machine to execute them in a secured mode. The developed approach is to use security information introduce in application code and to modify the java virtual machine to make good use of them to detect the attack. These researches focus on proposing different mechanisms that can help to fight against fault attacks by checking the code integrity or the control flow integrity during runtime.

keywords : *smart card, java card, fault attacks, code integrity, control flow*

Table des matières

Liste des tableaux

Table des figures

Introduction

Introduction

Nous sommes aujourd'hui dans l'ère de la mobilité. Pour s'en rendre compte, il suffit de voir la quantité d'équipements mobiles qui nous entoure : ordinateurs portables, téléphones mobiles, assistants personnels, tablettes tactiles, etc. Autour de tous ces équipements, se développent des services qui nécessitent une authentification, l'outil de choix dans ce domaine est la carte à puce. La carte à puce est un ordinateur disposant de fonction de sécurité et grâce à sa petite taille, il peut être transporté facilement ce qui en fait un atout pour de nombreux domaines tels que la banque, les services du gouvernement, la téléphonie mobile, la télévision à péage, etc. Un individu peut ainsi facilement disposer d'une multitude de cartes à puce sous différents facteurs de forme (carte bancaire, carte d'identité, carte d'assurance maladie, passeport, carte de fidélité...).

L'élément primordial de la carte est la sécurité. En effet grâce à la cryptographie, elle peut offrir des services d'authentification [LH10 ; HL00 ; JP10], de signature de transaction [Sch91 ; Sha85]. Mais aussi parce qu'elle fait office de modèle de programmation robuste et sécurisé. Ainsi, une remise en cause de la sécurité des cartes pourrait avoir des conséquences désastreuses au vu des domaines dans lesquels elle s'applique : tant économiques, que pour la sécurité d'un état. Imaginons un instant que les sécurités contenues dans les cartes puissent être facilement contournées. On pourrait alors assister dans le domaine bancaire à des paiements non autorisés (récemment, un scientifique américain a montré qu'il était possible de faire croire à la carte qu'une transaction bancaire avait été autorisée alors même que le serveur de la banque n'avait pas été consulté). Dans le domaine de la téléphonie, on pourrait assister à l'utilisation frauduleuse des réseaux mobiles (ce qui constituerait un manque à gagner énorme pour les opérateurs). Une bonne partie des services d'identité se voit équiper de cartes à puce, si des passeports venaient à être falsifiés, il serait impossible de vérifier l'entrée sur le territoire de personnes dangereuses. Elle pourrait également être utilisée pour de l'espionnage. Nous pourrions citer une multitude d'autres situations où le contournement des mesures de sécurité de la carte pourrait avoir une conséquence néfaste.

Ces éléments permettent de comprendre pour quelles raisons, des personnes pourraient vouloir contourner la sécurité des cartes pour leurs profits personnels. Mais cela explique aussi pourquoi tant de moyens sont mis en œuvre pour s'assurer que les cartes restent sécurisées.

Cadre de la thèse

Les travaux de recherche réalisés dans le cadre de cette thèse portent sur la sécurité des applications qui s'exécutent sur la carte à puce. La sécurité de la carte à puce est un élément déterminant, nous l'avons vu, il peut avoir des conséquences plus ou moins importantes. Il est donc essentiel de pouvoir garantir la sécurité des données ainsi que celle des applications qui s'y trouvent.

La sécurité d'une cartes à puce peut être contournée de plusieurs manières, soit en prenant le matériel en défaut, soit en prenant l'applicatif en défaut. Au niveau matériel, il existe de nombreux mécanismes de protection. Ces mécanismes permettent de s'assurer que l'environnement dans lequel fonctionne la carte est sans danger, que la puce n'a pas été mise à nu, ou de s'assurer de l'intégrité des données manipulées. Au niveau logiciel, il existe également plusieurs techniques pour protéger la carte, qui vont passer par la protection des algorithmes cryptographiques qui se trouvent sur la carte ; le durcissement du système afin de détecter des dysfonctionnements ou l'utilisation de guide de programmation afin de créer des applications sécurisées pour la carte.

On retrouve dans la littérature beaucoup d'attaques et de contremesures pour les algorithmes cryptographiques (en temoigne le nombre de références sur le sujet parmi lesquels on peut citer [BDL97 ; BOS03 ; BE+06 ; GAD05 ; KKS99 ; Aum+03 ; KQ07 ; AG01]), or la sécurité du système peut être remise en question autrement qu'en attaquant les algorithmes cryptographiques, en attaquant par exemple directement les applications qui font appel à ces derniers. Il existe plusieurs types d'attaques, les attaques matérielles ainsi que les attaques logicielles. Ce travail de recherche va se focaliser sur les attaques matérielles, ayant des conséquences au niveau logiciel : les attaques par injection de fautes qui consiste à perturber la puce avec des moyens matériels (par laser, par rayonnement électromagnétique...) lors de son fonctionnement. L'injection de fautes peut causer la modification du code ainsi que des évitements de tests logiques.

Une application pour carte à puce est créée sur un ordinateur, puis chargée sur la carte avant d'être exécutée lorsque le besoin se présente. L'attaque peut avoir lieu à partir du moment où l'application est envoyée vers la carte jusqu'à son exécution au sein de celle-ci. Dans ce contexte on veut s'assurer que l'intégrité de l'application est préservée entre ces deux instants.

Dans le cadre de l'élaboration de techniques de protection pour la carte, mon travail s'est articulé autour de deux points essentiels : d'une part l'élaboration d'un modèle réaliste d'injection de fautes ainsi que les conséquences de ce modèle sur la carte et d'autre part le développement puis l'évaluation de techniques de protection logicielles contre les attaques par injection de fautes qui suivent ce modèle. L'optique de ces contremesures est de décharger l'application de code devant gérer la sécurité, pour le reporter sur le système. Car, les applications sont codées en Java, ce qui entraine une baisse des performances au niveau vitesse d'exécution ainsi qu'une augmentation considérable de la taille des applications. Ce sont deux types de ressources que nous souhaitons préserver au maximum sur une carte. De plus, mettre les protections au niveau du code des applications est une chose qui rend leur développement plus complexe, car en plus du code fonctionnel, il faut s'occuper de sa sécurité.

Organisation du mémoire

L'ensemble de ce document est organisé en deux parties contenant en tout six chapitres. Dans la première partie intitulée «domaines d'applications», j'expliquerais les différents éléments qui m'ont permis de comprendre le sujet de la thèse. Cette partie s'articule comme suit :

- Dans le chapitre 1, j'explique le fonctionnement de la carte à puce, de ses composants, du protocole de communication qu'elle utilise ainsi que des éléments qui font d'elle un équipement sécurisé.
- Dans le chapitre 2 , je parle de la plateforme Java Card pour laquelle, j'ai développé toutes les contremesures. j'explique le principe général de fonctionnement de la plateforme ainsi que les éléments de sécurité qui la caractérisent.
- Dans le chapitre 3, je parle des attaques, notamment de l'injection de fautes en donnant quelques exemples. Je parle également de l'importance d'avoir un modèle de fautes réaliste, ainsi que des conséquences du modèle de fautes sur les différents composants de la carte. De plus, j'aborde plus en détails la problématique à laquelle je me suis attaqué.
- Dans le chapitre 4, j'expose les différentes techniques de protections logicielles qui existent dans la littérature, j'y donne les définitions nécessaires à la compréhension de ces techniques. j'aborde des sujets tels que les techniques de vérification d'intégrité du code ainsi que du flot de contrôle des applications.

Dans la deuxième partie intitulée «contributions et évaluations», j'aborde le noeud du problème à savoir le travail réalisé pour répondre à la problématique qui est posée.

- Dans le chapitre 5, j'explique le fonctionnement des différentes techniques développées afin de détecter les modifications dues aux attaques par injection de fautes. Ces techniques sont basées sur l'intégrité du flot de contrôle et du code des applications, via une détection pendant l'exécution du code.
- Dans le chapitre 6, j'explique la façon dont j'ai procédé afin d'évaluer ces techniques, et je donne les résultats associés. Ces résultats sont obtenus en implantant les différents mécanismes sur une machine virtuelle, et sur un simulateur abstrait de machine virtuelle capable de reproduire le comportement des attaques sur la puce.

Première partie

Domaines d'études

Chapitre 1

La carte à puce

Sommaire

1.1 Qu'est ce que la carte à puce ?

Une carte à puce (voir la Fig. 1.1) est un équipement se présentant sous forme de carte plastique de petite taille, intégrant un circuit électronique qui permet d'effectuer des opérations (stocker, calculer, etc.) de façon sécurisée. C'est donc un ordinateur à part entière qui est équipé de fonctions de sécurité lui permettant de résister aux attaques qu'il peut subir. On peut donc la qualifier de carte intelligente (ou *smart card* en anglais), elle est à opposer à la carte à mémoire qui contient des fusibles destructibles électriquement au fur et à mesure de leur utilisation. De plus, contrairement

Fig. 1.1 – Une carte à puce

à cette dernière, la carte à puce possède sa propre puissance de calcul et sa propre capacité de stockage d'informations, tout cela de façon sécurisée. Tout ce qui concerne la carte à puce a fait l'objet d'une normalisation, auprès de l'Organisme International de Normalisation (ISO), portant le nom de ISO 7816 (voir [Sta87]).

1.2 Historique

Il y a bientôt quarante ans que le premier prototype de carte a été conçu ; cela n'empêche pas la carte d'être un objet méconnu du grand public. Elle a toujours été considérée comme étant un équipement électronique à la pointe de la technologie. En effet, depuis les années 70, les technologies intégrées à la carte à puce reflètent les dernières avancées en matière de microprocesseurs de même qu'en matière du nombre et de la diversité de ses applications.

- En 1968, deux inventeurs allemands Jürgen Dethloff et Helmut Gröttrup introduisent en premier un circuit intégré dans une carte plastique.
- En 1970, le chercheur japonais Kunikuta Arimura de l'Institut de Technologie Arimura dépose un brevet sur la carte à puce.
- Entre 1974 et 1978, le français Rolland Moreno dépose 47 brevets en tout sur la carte à puce parmi lesquels on peut citer [197 ; Mor77 ; Mor76 ; Mor78 ; Mor83] ; il est considéré comme étant l'inventeur de la carte à puce telle qu'on la connait aujourd'hui. En effet, il crée : une mémoire portative dotée de moyens inhibiteurs. Ces inhibiteurs (matériels ou logiciels) ont pour but de protéger l'accès à la mémoire aussi bien en lecture qu'en écriture. Sans ceux-ci, l'utilité de la carte est limitée en terme d'application. Si l'on prend l'exemple de la carte de

crédit, il est indispensable de protéger la lecture du code confidentiel ou le changement du numéro de compte inscrit à l'intérieur. La carte décrite dans le premier brevet de Rolland Moreno est d'ailleurs couplée à un lecteur par radiofréquence, comme le seront par la suite les cartes sans contact.

- En 1977, Derthloff dépose un brevet [197] pour une carte à mémoire portative dont les moyens inhibiteurs sont constitués par un microprocesseur. Ce brevet significatif permet le changement des fonctions de la carte par simple reprogrammation. Aujourd'hui la majorité des cartes à puce sont dotées d'un microprocesseur.
- En 1979, la première carte est créée et assemblée à Toulouse par Motorola pour Bull CP8 avec 1 ko de mémoire programmable et un microprocesseur à base de 6805
- En 1983 apparaissent les premières cartes téléphoniques.
- En 1984 le groupe d'intérêt économique carte bancaire adopte le prototype de carte bleue basé sur la technologie Bull CP8. Cela donne naissance à la carte bleue moderne (la version B0').
- Entre 1984 et 1987, les normes internationales de l'ISO sur la carte à puce avec contact voient le jour sous la référence 7816 [Sta87].
- En 1997, les premières cartes à puce multiplicatives voient le jour.

1.3 Les différents types de cartes à puce

Il existe plusieurs types de cartes à puce que l'on peut classer soit suivant les technologies utilisées en interne, soit suivant les possibilités de communication de ces cartes.

1.4 La carte à mémoire

Elle constitue le premier type de cartes à puce et représente encore la majorité des cartes vendues dans le monde en 1999.

Une telle carte possède une puce mémoire qui peut également être accompagnée d'une logique câblée non programmable (instruction programmée une fois pour toutes), cette carte ne possède pas de microprocesseur. Autrefois, la taille de la mémoire était limitée à quelques Kilo-octets alors qu'aujourd'hui certaines peuvent atteindre le Mégaoctet. Elle bénéficie d'une technologie simple ainsi que d'un faible coût de revient (environ 1 euro). Ce type de carte ne possède pas de capacité propre de calcul, elle est aisément duplicable, et est très fortement dépendante du lecteur.

1.5 La carte à microprocesseur

Ce type de carte constitue réellement ce que l'on peut appeler une smart card. En effet, la puce de cette carte embarque un microprocesseur, ce qui la rend autonome (c.-à-d. intelligente). Les dimensions de la puce sont au maximum de 25 mm2 pour sa surface et de 200 micromètres pour son épaisseur.

Précédemment, ces cartes étaient basées sur des processeurs Motorola 6805 ou l'Intel 8051, 8 bits CISC, avec une vitesse d'horloge externe de l'ordre de 5 MHz. Aujourd'hui, on dispose

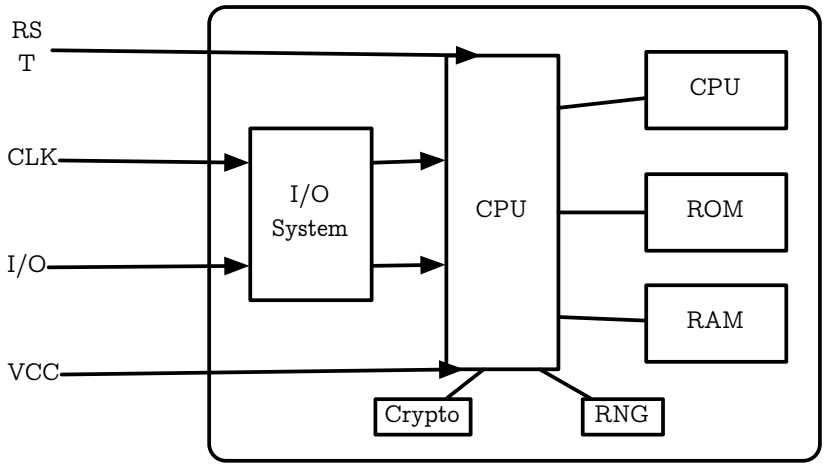

Fig. 1.2 – Architecture simplifiée d'une carte à microprocesseur

de processeurs plus puissants 32 bits avec des architectures CISC [1] ou RISC [2] travaillant à des fréquences internes comprises entre 5 et 40 MHz. Cependant, le standard ISO 7816 oblige à respecter une certaine plage de fréquences d'horloge afin d'assurer la compatibilité avec le matériel déjà en place (lecteurs). Pour cette raison, les fabricants utilisent des multiplicateurs de fréquence en interne afin d'effectuer les opérations dans la carte plus rapidement. Actuellement, il existe ainsi des cartes à microprocesseur de dernière génération RISC fonctionnant avec une horloge interne de 100 à 200 MHz en utilisant un multiplicateur x20 ou x40.

La puce possède éventuellement un coprocesseur cryptographique qui réalise les opérations de chiffrement et déchiffrement de façon matérielle ce qui permet d'améliorer les performances. Ce coprocesseur cryptographique est associé à un générateur de nombres aléatoires RNG (Random Number Generator).

L'avantage d'une telle carte réside dans le microprocesseur qui fournit la sécurité grâce au coprocesseur cryptographique et à d'autres mécanismes, en interdisant que les données soient accessibles de l'extérieur. Le coût de revient d'une carte à puce avec un tel degré de sécurité se situe ou dépasse largement les 1 euro.

En résumé, pour ce type de carte le très haut niveau de sécurité repose sur l'association entre l'électronique, l'informatique et la cryptographie.

1.6 Carte à puce et mémoire

Une particularité de la carte à puce provient des différents types de mémoire qu'elle utilise. La mémoire que l'on retrouve sur une carte joue un rôle prépondérant dans la sécurité de celle-ci, il me paraît donc important d'expliquer le rôle de chaque type de mémoires contenue dans la carte.

On retrouve trois types de mémoires dans la carte (voir Fig. 1.2) que l'on peut classer en fonction de leurs propriétés : une mémoire figée en usine (ROM), une mémoire persistante (EEPROM) et une mémoire volatile (RAM). L'encombrement de chaque type de mémoires utilisés dépend de la

1. Un microprocesseur à jeu d'instruction étendu, ou Complex instruction set computer
2. Reduced instruction set computer

technologie employée. La finesse de gravure des mémoires contenues sur carte à puce entraine pour le fabricant des contraintes au niveau du choix des capacités accordées à chaque type de mémoire afin d'obtenir un équipement ayant de bonnes performances.

À titre d'exemple, en fonction de la technologie utilisée, une cellule d'EEPROM occupe quatre fois plus d'espace qu'une cellule de ROM ; et une cellule de RAM occupe quatre fois plus d'espace qu'une cellule d'EEPROM.

1.6.1 La ROM (Read Only Memory) ou mémoire morte

La ROM constitue la mémoire programmée en usine, c'est une mémoire persistante et figée. Elle n'a donc pas besoin d'énergie pour sauvegarder l'information qu'elle contient et son contenu n'est pas modifiable. Elle sert à stocker le système d'exploitation de la carte (COS ou Card Operating System) ainsi que des données permanentes. Sa taille est le plus souvent de 32 ko, mais on peut trouver des cartes contenant 300 ko ou plus (sur les cartes modernes haut de gamme).

1.6.2 La mémoire volatile

La mémoire volatile est utilisée comme espace de stockage temporaire pour le calcul grâce à la vitesse de son temps d'accès. Elle possède un caractère non persistant et perd donc son contenu entre deux utilisations. Elle a la particularité de pouvoir être lue et écrite à l'infini. Il s'agit d'une mémoire RAM (Random Access Memory) comme celle que l'on peut trouver sur un ordinateur classique. Néanmoins dans le cas de la carte, sa capacité est seulement de 1 ko à 24 ko (sur les cartes modernes haut de gamme).

1.6.3 La mémoire persistante

La mémoire persistante a les mêmes caractéristiques que la ROM à ceci près qu'elle peut être modifiée. Elle est en général désignée par le terme de mémoire non volatile (NVM). Elle est chargée du stockage des informations qui peuvent évoluer dans le temps tels que les applications, les clefs cryptographiques et le code PIN de l'utilisateur ; c.-à-d. les informations qui doivent être conservées même lorsque la carte n'est plus sous tension.

Il n'existait encore récemment qu'un seul type de mémoire possédant ces caractéristiques : l'EEPROM (Electrical Erasable Programmable Read Only Memory). Aujourd'hui, on trouve des produits basés sur d'autres types de mémoire aux propriétés similaires comme la mémoire Flash ou la FeRAM. Dans la suite de cette section, je décris uniquement les technologies des mémoires EEPROM et Flash car elles sont les plus répandues.

L'EEPROM présente deux inconvénients :

- Sa durée de vie limitée en nombre de cycles d'écriture c.-à-d. son endurance qui est d'environ 100000 cycles et en temps de rétention de l'information (10 ans) ce qui nécessite de changer régulièrement la carte pour éviter des dysfonctionnements tels que la perte d'informations.
- sa lenteur d'accès c.-à-d. le phénomène de latence lors de l'écriture (car elle doit au préalable être effacée électriquement). Donc l'opération d'effacement/écriture est 1000 fois plus lente que dans le cas de la RAM.

La mémoire Flash, quant à elle, a le gros avantage de ne pas occuper beaucoup de place sur la puce. En revanche, elle possède une latence en écriture très supérieure à l'EEPROM et une durée de vie équivalente. Malgré cela, cette mémoire a permis l'élaboration de produits contenant très peu de ROM (ou pas du tout) et beaucoup de mémoire flash, ce qui permet de reprogrammer entièrement la carte. Le plus souvent, ce sont des cartes de développement destinées à la création de systèmes d'exploitation.

Les cartes à puce communes possèdent uniquement de la mémoire de type EEPROM avec des capacités variant de 16 ko à 256 ko.

1.7 Moyens de communication

Pour interagir avec un environnement extérieur la carte à puce dispose de fonctions de communication. Plusieurs technologies de communication furent mises au point pour réaliser des échanges avec le monde extérieur. Cependant, les cartes, en raison de leur conception extrêmement compacte, sont des ordinateurs passifs qui doivent être alimentés depuis l'extérieur pour pouvoir fonctionner (bien qu'il existe des systèmes d'alimentation mobiles assez compacts afin de pouvoir être embarqués dans la carte). Ainsi, les cartes ne communiquent pas directement avec d'autres ordinateurs, mais plutôt avec un lecteur de cartes à puce qui leur fournit l'énergie requise pour fonctionner. Donc le lecteur sert d'interface entre la carte et la machine hôte avec laquelle elle doit dialoguer. Tout d'abord j'aborde le sujet du type de technologie de communication utilisé puis celui des protocoles de communication.

1.7.1 La carte à contact

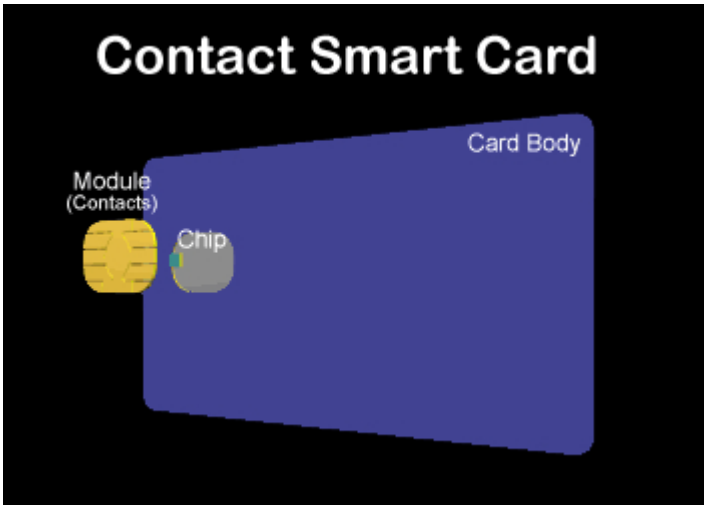

Fig. 1.3 – Carte à contact - source Gemalto

La cartes à contact (Fig. 1.1) communiquent par un microcontact relié à la puce (microchip) par des fils d'or. Cet assemblage se nomme micromodule (voir la Fig. 1.4)).

Il est placé dans le corps de la carte (dans la partie plastique) (voir la Fig. 1.3) et correspond à la partie intelligente de la carte. Ces cartes utilisent une communication série avec huit contacts définis

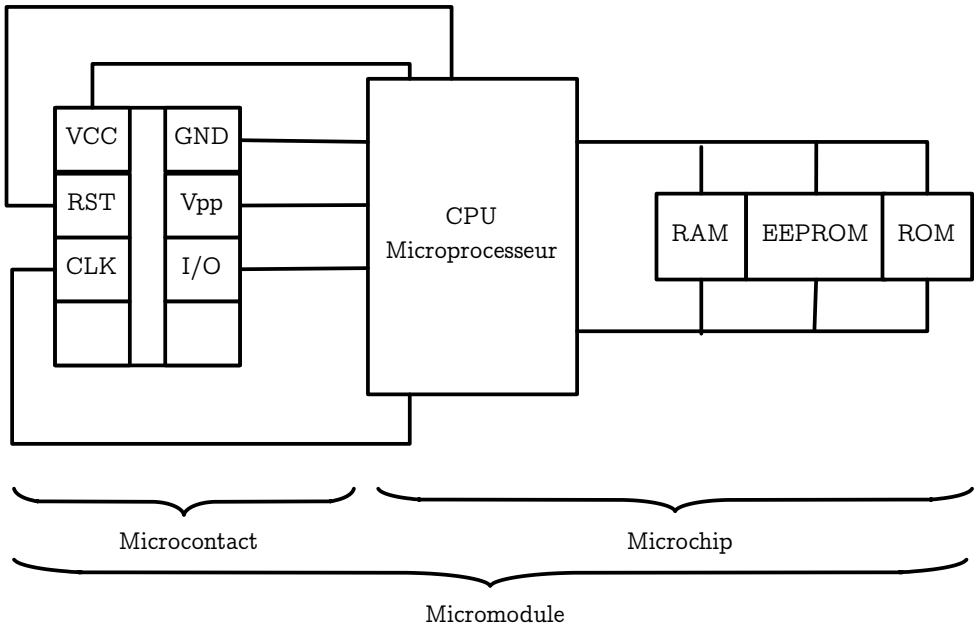

Fig. 1.4 – Le micromodule

dans le standard ISO 7816. En pratique, seulement cinq de ces huit contacts sont réellement utilisés pour réaliser la communication série. Le contact Vpp n'est plus utilisé puisqu'il servait uniquement à fournir une alimentation suffisante lors de la programmation de l'EEPROM ce qui constitue un problème de sécurité contre les attaques par canaux cachés. En revanche, les deux contacts inutilisés à l'origine sur la Fig. 1.4 qui étaient réservés à des usages futurs servent aujourd'hui à communiquer en USB (Universal Serial Bus). Les cartes utilisant cette technologie intègrent de fait la gestion de l'USB transformant le lecteur en adaptateur entre la carte et le PC. Cela permet d'avoir des lecteurs à faible coût. N'ayant pas d'alimentation électrique propre, la carte doit être insérée dans un lecteur qui lui fournit l'énergie nécessaire à son fonctionnement.

1.7.2 La carte sans contact

Les cartes sans contact communiquent grâce à une antenne reliée au microchip (voir la Fig. 1.5). Les problèmes rencontrés avec les cartes à contact ne se posent plus, car ces cartes n'ont pas besoin d'être insérées dans un quelconque lecteur pour fonctionner. L'énergie, nécessaire à la puce, provient soit d'un couplage capacitif qui consiste par exemple en une batterie ; soit d'un couplage inductif distant collecté par l'antenne. Le couplage inductif fonctionne sur le même principe que celui du transformateur à bobine, c.-à-d. qu'une bobine dans le lecteur induit un courant dans une autre bobine à savoir l'antenne de la carte. La fréquence de transmission est de l'ordre de quelques MHz. La puce est à même de transmettre, en changeant sa résistance, un signal qui est capté par le lecteur et interprété comme un signal de données.

Tous les problèmes ne sont pas réglés avec ces cartes. En effet, la portée entre la borne de lecture et la carte est limitée (environ 10 cm). On peut également citer le problème du temps de transaction entre cette même borne et le lecteur qui ne doit pas excéder les 200 millisecondes (ms) ce qui limite la taille des données échangées. De plus, on peut citer leur fort coût de fabrication en

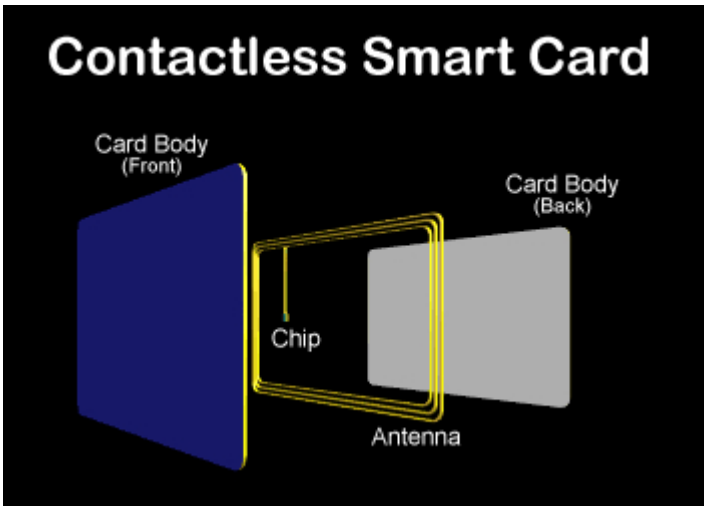

Fig. 1.5 – Carte à contact - source Gemalto

comparaison des cartes à contact.

Leur utilisation se fait pour les usages où l'insertion et le retrait dans un lecteur ne sont pas pratiques (Exemple de la carte sans contact pour régler le péage sur l'autoroute).

La carte à interface hybride

Une carte à interface hybride est simplement une carte combinant les technologies sans contact et celles avec contact. Elle dispose donc de deux possibilités de communication ce qui en fait la carte idéale.

1.7.3 Protocoles de communication

Une carte à puce classique communique avec le lecteur en mode semi-duplex, c.-à-d. qu'à l'instant t seule une des parties peut utiliser la ligne de communication : soit le lecteur, soit la carte, mais jamais les deux en même temps.

Cette manière de procéder nécessite d'avoir un canal de communication pour savoir qui est autorisé à émettre et qui est autorisé à recevoir des informations. Pour parvenir à cela il faut déterminer une relation de type client/serveur, où le client initie la communication et le serveur reçoit l'information. Si ces règles ne sont pas respectées, il peut alors se produire deux situations qui ont des effets gênants lors de la communication :

- Les deux parties émettent au même moment, ce qui crée des collisions et entraine des pertes de données.
- Les deux parties se mettent en attente d'un message au même moment et dans ce cas on a une situation de *deadlock*.

La communication entre la carte et le lecteur se fait suivant le modèle client/serveur où le lecteur joue le rôle de client et c'est d'ailleurs ce dernier qui fournit la carte en énergie.

La communication client/serveur se fait au travers d'une pile protocolaire que l'on retrouve dans la Fig. 1.6, où j'ai représenté les différentes couches de cette pile en m'alignant au mieux sur

le modèle OSI. Dans ce modèle, à chaque couche correspond un protocole de communication. Dans la suite de cette section je parle des deux premières couches à savoir la couche application et la couche transport.

A partir de maintenant, j'utiliserais indifféremment les termes CAD (Card Acceptance Device), IFD (InterFace Device), et lecteur de cartes à puce.

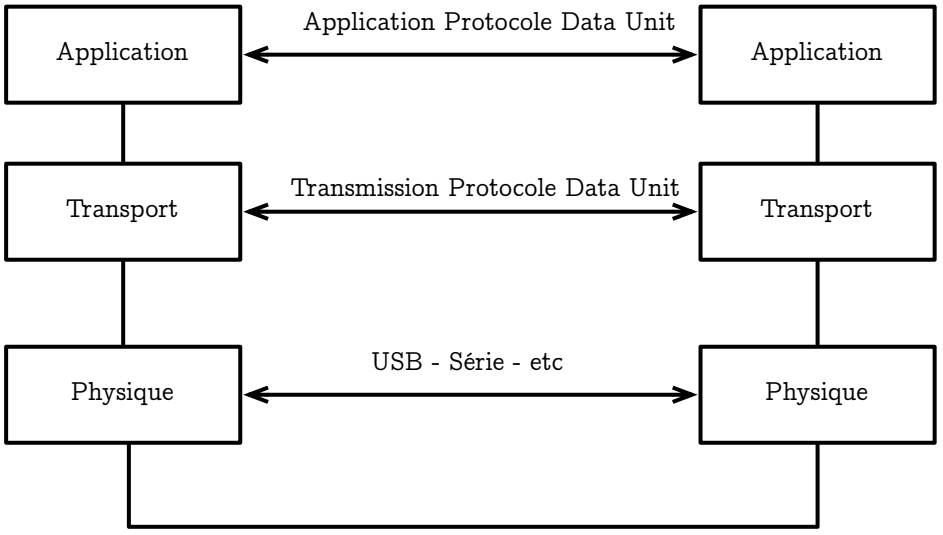

Fig. 1.6 – Pile de communication entre le lecteur et la carte

La couche physique : La couche physique est normalisée par l'ISO 7816-3 (qui définit les signaux électroniques et les protocoles de transmission). Elle définit notamment une fréquence d'horloge comprise entre 1 MHz et 5 MHz, ainsi qu'une vitesse pour les communications pouvant aller jusqu'à 115200 bauds.

La couche transport - TPDU : Le protocole utilisé pour le transport des données est le protocole TPDU (Transmission Protocol Data Unit). Tous les échanges de données pour ce protocole sont également définis dans l'ISO 7813-3. Ce protocole définit deux modes principaux de transmission qui sont :

- le T=0 qui utilise une transmission par octet ;
- le T=1 qui utilise une transmission par paquet.

Il y a aussi certaines cartes qui utilisent le T=14 qui est réservé aux protocoles propriétaires. On trouve également des cartes qui utilisent T=CL pour les communications sans contact.

Cette norme définit aussi la sélection du type de protocole si plusieurs protocoles sont disponibles ainsi que le format de ATR (*Answer To Reset*) qui est le message envoyé par la carte au lecteur juste après sa réinitialisation.

La couche application - APDU : Le protocole ici utilisé échange des données au format APDU (Application Protocol Data Unit) qu'on retrouve également normalisé dans l'ISO 7816-3. Il existe principalement deux types d'APDU :

Entête obligatoire				Corps optionnel		
CLA	INS	P1	P2	Lc	Champ de données	Le

Table 1.1 – APDU de commande

Corps optionnel	En-queue obligatoire	
Champ de données	SW1	SW2

Table 1.2 – APDU de réponse

– les commandes APDU qui sont émises par le CAD en direction de la carte ;
– les réponses APDU qui sont émises par la carte en direction du CAD.

Comme le modèle de communication entre le CAD et la carte est un modèle client/serveur où le CAD est le client et la carte le serveur, alors la carte est toujours en attente d'une requête en provenance du CAD sous forme d'une APDU de commande et retournera en conséquence toujours une APDU de réponse. En somme, une commande APDU sera toujours couplée avec une réponse APDU.

La commande APDU est constituée (voir le Tableau 1.1) d'un entête et d'un corps. L'entête obligatoire contient quatre champs correspondant à un octet ayant une signification précise :

– **CLA**, l'octet de classe qui identifie la catégorie de la commande et de la réponse APDU
– **INS**, l'octet d'instructions qui indique l'instruction à exécuter,
– **P1** et **P2**, deux octets contenant les paramètres de l'instruction ;

Le corps de la commande est variable et peut être omis :
– **Lc**, l'octet qui spécifie la taille du champ de données ;
– Le **Champ de données** qui contient les données à envoyer à la carte pour exécuter l'instruction spécifiée dans l'entête
– **Le**, l'octet correspondant au nombre d'octets attendu par le CAD pour le champ de données de la réponse APDU de la carte.

La réponse APDU est constituée des éléments du Tableau 1.2

– Un corps optionnel de taille variable
  – Le **champ de données** de taille **Le** déterminé dans la commande APDU correspondante.
– Une zone de fin obligatoire comportant deux octets :
  – **SW1** et **SW2** (*Status Word*), deux champs d'un octet précisant l'état de la carte après exécution de la commande APDU. Par exemple, 0x9000 signifie que l'exécution s'est déroulée jusqu'à son terme et avec succès.

Comme certaines parties des APDUs sont optionnelles, il y a quatre cas possibles d'échanges :
– cas 1 : Aucune donnée n'est échangée entre le CAD et la carte mis à part l'entête de commande APDU et l'en queue de la réponse APDU (voir le Tableau 1.3).
– Cas 2 : Aucune donnée (dans le champ de données de la commande APDU) n'est envoyée à la carte. Le corps de la commande contient le champ *Le* qui spécifie le nombre d'octets que la carte doit fournir en retour dans le champ de données de la réponse APDU correspondante (voir le Tableau 1.4).
– cas 3 : Des données de taille Lc octets sont fournies à la carte (dans le champ données de la commande) et la carte ne renvoie dans la réponse que son en-queue (voir le Tableau 1.5).
– cas 4 : Des données sont échangées dans le champ de données de la commande et de la réponse APDU (voir le Tableau 1.6).

CLA	INS	P1	P2

Table 1.3 – Cas 1 : Aucune commande envoyée, aucune réponse requise

CLA	INS	P1	P2	Le

Table 1.4 – Cas 2 : Aucune commande envoyée, une réponse requise

1.8 L'émission et le cycle de vie de la carte

L'objectif de cette section est de présenter les étapes de la vie d'une carte ainsi que les différents acteurs qui y jouent un rôle. En effet, chacun de ceux-ci peut avoir un rôle plus ou moins important dans la sécurité de la carte car pouvant être tour à tour commanditaire d'une évaluation, ou utilisateur (bien ou mal intentionné).

1.8.1 Les acteurs

On rencontre dans le monde de la carte à puce quatre acteurs majeurs qui sont :
- le fabricant de puces (Atmel, Infineon, etc) ou fondeurs ;
- le fabricant de cartes (Gemalto, Oberthur, etc) ;
- l'émetteur de la carte ou fournisseur de carte (*card issuer*) (banques, opérateurs téléphoniques, assurance maladie, etc) qui fournit la carte à l'utilisateur final ;
- le porteur de la carte qui en est l'utilisateur final (*end user*).

1.8.2 L'émission de la carte

La première partie du processus commence chez le fabricant de puces. C'est lui qui décide des technologies du microprocesseur, des différentes mémoires embarquées, de la vitesse d'horloge, etc. C'est également lui qui inscrit dans la mémoire ROM un système d'exploitation minimum et certaines données nécessaires au bon fonctionnement de la carte.

La seconde partie du processus se déroule chez le fabricant de cartes. En effet, celui ci reçoit des galettes de silicium contenant les puces qu'il découpe, teste, puis assemble avec le microcontact pour former le micromodule (dans le cas d'une carte à contact). Le corps de la carte en PVC, en ABS ou en polycarbonate reçoit ce micromodule, ce qui donne une carte qui peut débuter son cycle de vie.

1.8.3 Le cycle de vie de la carte

En général, le fabricant de cartes fournit à l'émetteur de la carte des applications qui vont être embarquées dans la carte. Au début du cycle de vie de la carte, il y a une phase de personnalisation qui débute par l'inscription en mémoire persistante des données communes aux applications et éventuellement par l'insertion d'autres caractéristiques de sécurité. Ensuite, il y aura :

CLA	INS	P1	P2	Lc	Données

Table 1.5 – Cas 3 : une commande envoyée, aucune réponse requise

CLA	INS	P1	P2	Lc	Données	Le

Table 1.6 – Cas 4 : une commande envoyée, une réponse requise

- Une étape de personnalisation au niveau électrique : cette étape consiste à inscrire en mémoire persistante des données relatives à chaque porteur de carte.
- Une étape de personnalisation au niveau graphique : cette étape consiste tout d'abord à faire la personnalisation du corps de carte suivant le désir de l'émetteur, par exemple son logo, et enfin à faire l'embossage (écriture en relief à la surface de la carte) de données relatives à chaque porteur de carte.

Les deux étapes précédentes sont faites à la demande de l'émetteur de la carte. La carte est alors utilisable et son cycle de vie se poursuit avec l'utilisateur. Enfin, Il se termine soit par son invalidation logique, soit par la saturation de sa mémoire, soit par la perte ou le vol, etc.

1.8.4 Carte à puce et sécurité

Cette section présente les différents niveaux de sécurité offerts par la carte à puce. C'est une étape obligatoire car nous en aurons besoin dans la suite lorsque je parlerai des attaques au chapitre 3.

Bien que souvent mise en doute par la presse, la carte à puce est pourtant le résultat de l'association des meilleures technologies en matière de sécurité. Même si, certains des acteurs en font une mauvaise intégration ou une utilisation dans des systèmes plus globaux pouvant favoriser la mise au point d'attaques, elle conserve encore son titre de périphérique le plus sécurisé au monde.

Il existe une vraie guerre entre les fabricants de produits de technologies de l'information et les attaquants. Les premiers essayant de rendre la vie plus difficile aux seconds en intégrant des protections dans leurs équipements. Et les seconds en mettant au point de nouvelles attaques pour contourner les sécurités mises au point par les premiers. Je vais maintenant expliquer quelles sont les raisons qui font de la carte à puce un équipement sécurisé.

Sécurité au niveau physique

Le premier rempart de sécurité dans une carte est constitué de son apparence. En effet, dans le cadre bancaire par exemple (*i.e.* carte bleue), aucun commerce n'accepterait un règlement via un banal morceau de plastique, ou même via un microcontact ISO 7816 relié à un ordinateur (une telle carte est décrite dans les références [Gue97 ; Gue98]). Les faussaires doivent donc être capables de reproduire les cartes avec les mêmes caractéristiques d'aspect. Arriver à un tel résultat est difficile, car les fabricants de cartes utilisent des techniques de conception particulières pour le corps des cartes à savoir : des technologies d'impression à diverses profondeurs dans le plastique composant le corps de la carte, des hologrammes (cas des cartes visa qui possèdent pour la plupart un hologramme d'aigle), et enfin un embossage des informations relatives au porteur.

Mais ces techniques ne suffisent malheureusement pas à protéger les commerçants inattentifs des faussaires. D'où l'utilité d'intégrer d'autres sécurités aussi bien logicielles que matérielles.

Nous verrons en chapitre 3 que certaines attaques nécessitent un accès direct au microprocesseur. En conséquence, afin que le retrait des couches protectrices ne soit pas aisé, différents mécanismes

d'encollage du micromodule ont été développés. Le matériau constitutif du corps de la carte peut aussi être d'origine différente selon la robustesse souhaitée. Ces diverses protections visent à mettre en échec les attaques chimiques d'extraction du micromodule. Un autre aspect des choses rendant ces attaques compliquées est la méthode de conception du microcontact qui rend la puce très sensible à l'extraction.

Tous ces mécanismes, bien que contournables par des personnes avec de hautes connaissances dans le domaine et avec du matériel de pointe, assurent une première défense assez compliquée à franchir pour le néophyte. Le but des fabricants de cartes étant de pousser l'attaquant à endommager la carte lors de l'ouverture, ce qui la rend inutilisable.

Il faut également rappeler que la norme ISO 7816 permet de garantir un certain niveau de sécurité au travers de la multitude de tests de caractérisation effectués. Ce qui m'amène à citer la norme ISO 7816-1 qui détaille les éléments suivants :

- la protection contre les ultraviolets et les rayons X ;
- le profil de surface des contacts ;
- la résistance mécanique des cartes et des contacts (au pliage et à la torsion) ;
- la résistance électrique des contacts ;
- l'interférence électromagnétique entre les pistes magnétiques éventuelles de la carte et des circuits intégrés ;
- l'exposition de la carte à des champs magnétiques ;
- l'exposition de la carte à une décharge d'électricité statique ;
- la dissipation thermique du circuit intégré.

Sécurité au niveau matériel

Le second rempart d'une carte à puce se compose du matériel. C'est la pierre angulaire de la sécurité de la carte. Car c'est autour de lui que sont construits les autres mécanismes de sécurité implantés dans la carte. S'il peut être pris en défaut, alors toute la sécurité de la carte peut être remise en question. D'où l'importance apportée par les constructeurs à la sécurité lors de la conception des puces équipant les cartes.

Cela commence par l'introduction d'un numéro de série unique pour chaque puce produite. Au fil du temps, le microprocesseur contenu dans la carte se complexifie avec le système d'exploitation, apportant ainsi plus de fonctionnalités. Puis, les constructeurs commencent à intégrer des mémoires programmables de type PROM (mémoire persistante) qui permettent en cas de problèmes après la délivrance des cartes de modifier le code des applications.

Comme je l'ai évoqué plus tôt, la réalisation de certaines attaques nécessite d'avoir un accès direct à la puce. Dès lors, les fabricants de puces se sont mis à intégrer des capteurs sur la carte afin de déceler leurs utilisation dans un environnement hostile. C'est à ce titre qu'on peut retrouver des détecteurs de lumière et autres rayonnements, des détecteurs de température, des détecteurs de tension ou de fréquence de fonctionnement.

Ces méthodes ont pour objectif de rendre la mise au point et la réussite des attaques le plus compliqué possible. Car si l'attaquant n'a pas les compétences ainsi que le matériel nécessaire, il verra ses attaques échouées. Par contre, un attaquant qualifié pourra passer outre les capteurs intégrés.

Parmi les attaques possibles, nous pouvons observer les émissions électromagnétiques de la puce. Afin d'empêcher ce type d'attaque, les fabricants ont dans un premier temps réalisé un blindage physique de leur composant en ajoutant une grille de protection à la puce. De cette façon, l'attaquant ne peut plus avoir accès à la puce, mais cette mesure a été contournée grâce aux avancées en matière de techniques de microélectronique. Cela a entrainé l'avènement de boucliers actifs (c.-à-d. la grille à une certaine tension) qui deviennent inactifs lorsqu'ils sont enlevés ce qui verrouille la puce. Mais à nouveau, il faut compter avec de nouvelles techniques permettant de contourner ces boucliers.

D'autres méthodes de protection ont été mises en place, par exemple le chiffrement des données sur les bus et dans les mémoires, la dissémination des bus sensibles dans différents niveaux de métallisation, la pose de leurres (c.-à-d. de pistes reliées à aucune autre), la redondance de certaines fonctionnalités matérielles et l'obfuscation de l'information.

On peut aussi citer l'existence d'un coprocesseur cryptographique qui accélère et sécurise les calculs cryptographiques et rend l'observation de ceux-ci beaucoup plus compliquée, ainsi que l'implantation d'un générateur de nombres aléatoires interne pour tirer les aléas nécessaires à tous les algorithmes cryptographiques.

Le dernier mécanisme que je vais citer, est celui du lissage de consommation en courant qui permet d'uniformiser la consommation électrique du circuit, quelles que soient les opérations qu'il effectue. Ce mécanisme permet de lutter contre les attaques qui consistent en l'analyse de la consommation électrique du circuit afin de déterminer son activité.

La sécurité au niveau logiciel

Le troisième rempart de la sécurité est assuré par l'applicatif et plus particulièrement par le système d'exploitation. Celui-ci utilise généralement une politique draconienne de contrôle d'accès aux données. L'accès à des données se fait selon des règles dépendantes du type d'opérations effectuées. Le système d'exploitation assure aussi l'intégrité des données par des mécanismes de vérification d'intégrité sur toute ou partie de la mémoire, mais également par des mécanismes de transaction et d'atomicité des opérations. C'est aussi lui qui supervise les entrées/sorties, en les chiffrant si besoin est. Il peut aussi implémenter des mécanismes de migration de code et de données dans les mémoires afin d'empêcher la localisation de ces dernières. Le système d'exploitation à également la possibilité de travailler à des fréquences différentes pour perturber les observations de l'attaquant, mais il peut également assurer qu'une suite d'opérations s'effectue avec un même nombre de cycles d'horloge indépendamment des données manipulées afin de masquer l'utilisation de données précises à un observateur. En somme, le système d'exploitation est un élément important qui permet de s'assurer que les applications, qui vont faire usage de ses fonctionnalités, s'exécutent dans un environnement sain.

La sécurité des applications est aussi le fruit de l'application de techniques de programmation sécurisées. Ainsi, tous les développeurs d'applications ou de systèmes d'exploitation pour carte à puce disposent de guides de programmation pour assurer la sécurité de leurs applications. Ces guides sont en général basés sur l'état de l'art des attaques, voire même en avance sur celui-ci et il contient des règles de programmation telles que la diminution du compteur d'essais du code PIN avant de faire la comparaison plutôt que l'inverse, les mécanismes de description d'état lors de l'échec d'une transaction, etc.

La sécurité de l'environnement

La sécurité de la carte à puce provient aussi de l'environnement de sa chaîne de production c.-à-d. de fabrication, d'assemblage, de tests, de transport entre les différents acteurs, etc. qui est physiquement sécurisée. Ainsi, un individu avec de mauvaises intentions ne doit pas pourvoir de s'introduire dans cette chaine de production en altérant le code ou le matériel, ni même pouvoir récupérer des échantillons (p. ex. dans des poubelles). En effet, un échantillon volé peut faciliter les techniques de rétro-ingénierie permettant de mener à bien la mise à nu du microprocesseur de la carte.

Pire encore, si un micromodule est dérobé, il permet d'avoir directement accès à la puce. Fort heureusement, les sites de fabrication des cartes sont en général sous haute surveillance de sorte à s'assurer qu'aucune carte ni aucun rebut ne sorte de la chaîne de production. De plus, ces sites sont régulièrement audités, aussi bien dans le cadre d'évaluation sécuritaire ou dans le cadre d'un audit interne, assurant ainsi un peu plus de sécurité.

1.9 Les applications de la carte à puce

Aujourd'hui, la carte à puce est utilisée dans différents secteurs [Hen01] :

- l'industrie des télécommunications avec, par exemple, les cartes téléphoniques prépayées, cartes USIM insérées dans les téléphones GSM, 3G ou 3G+,
- l'industrie bancaire et monétaire avec les cartes de crédit (Europay, MasterCard, Visa) et les porte-monnaies électroniques (Monéo),
- le secteur du gouvernement avec les cartes d'identité, les passeports, les permis de conduire,
- l'industrie audiovisuelle dans le cadre de la télévision à péage (carte à insérer dans les décodeurs numériques ou analogiques),
- l'industrie du transport avec les cartes sans contact pour les transports en commun ou pour les transports routiers avec le remplacement des chronotachygraphes par des cartes à puces,
- l'industrie du contrôle d'accès physique des personnes aux locaux utilise de plus en plus la carte sans contact.

D'autres applications existent aujourd'hui pour la carte à puce à savoir

- l'authentification (sur des sites internet),
- les applications de fidélisation (carte de fidélité).

On peut facilement imaginer d'autres applications de la carte à puce notamment grâce à internet et à la multiplication des e-services qui exigent de plus en plus une identification et une sécurisation des transactions. Ainsi, la carte à puce peut être utilisée comme terminal de stockage de clefs servant à l'authentification et à la sécurisation des communications dans le cadre :

- du e-commerce,
- de la banque à distance,
- du courrier électronique,
- du télétravail,
- etc.

En raison de son utilisation massive et de la diversité des domaines dans lesquels elle est utilisée, il est normal que la carte à puce soit aujourd'hui une cible privilégiée des faussaires. De cet état

de fait découle directement les avancées technologiques en terme de sécurité pour la carte à puce. En raison de cette diversité, il est nécessaire d'avoir une plateforme standard permettant de créer des applications de façon simple et sécurisée ; d'où l'apparition de plateformes ouvertes à base de machines virtuelles telles que Java Card ou MULTOS (voir [JG02 ; wik10]). Dans le chapitre 2, je m'attaquerai uniquement à la plateforme Java Card car c'est sur celle-ci que s'est portée mon travail.

Chapitre 2

La technologie Java Card

Sommaire

2.1 Présentation

La plateforme Java Card a pour but de faire fonctionner la technologie Java sur des équipements fortement contraints tels que les cartes à puce ou d'autres équipements avec peu de mémoire et peu de puissance de calcul. Une Java Card est donc une carte à puce sur laquelle on est capable de charger et d'exécuter des applications Java du type *applets* ou *servlets* (depuis la version 3.0 de la spécification). Contrairement aux cartes à puce traditionnelles, ce sont des cartes ouvertes, c.-à-d. que les programmes qui y sont contenus ne sont pas nécessairement fournis par l'émetteur de la carte.

La technologie Java Card peut être considérée comme étant une plateforme fournissant un environnement sécurisé pour cartes à puce, interopérable et multiapplicatif qui jouit des avantages du langage Java.

2.2 Historique

En 1996, un groupe d'ingénieurs de Shlumberger à Austin au Texas cherchent à simplifier le modèle de programmation existant pour cartes à puce tout en préservant sa sécurité. La raison est simple, à l'époque il y avait plusieurs plateformes de gestion d'applications pour cartes à puce. Et, pour créer des applications pour ces cartes à puce, il fallait connaître le modèle de programmation intrinsèque à la plateforme, ce qui ne permettait pas à tout programmeur d'être capable de produire une application pour la carte. De plus, les applications produites n'étaient absolument pas portables d'une plateforme à une autre. Fort de ce constat, ils se sont mis à réfléchir à une plateforme qui permettrait de simplifier le modèle de programmation et d'obtenir un code universel. Le seul langage qui permettait cette flexibilité à l'époque était le langage Java dont la philosophie était « écrire une fois, exécuter n'importe où », peu importe le système d'exploitation. Mais, en raison de la faible puissance des cartes à puce, il était aussi clair qu'il ne pouvait pas transposer le langage Java sans procéder à des coupes. Ils décident alors d'adapter la plateforme Java et de n'utiliser qu'un sous-ensemble de Java. Ainsi, la machine virtuelle Java communément appelée JVM [3] ainsi que le système de runtime [4] ne devait pas dépasser les 12 ko. C'est ainsi que naquit la première version de Java Card. Schlumberger devint alors la première entreprise à obtenir une Licence en proposant la première spécification de quatre pages.

En février 1997, Bull et Gemplus se joignent à Schlumberger pour cofonder le Java Card Forum [For] qui recommande les spécifications à JavaSoft (la division de *Sun* à qui appartient Java Card). Il a pour but de promouvoir des API [5] Java Card afin de permettre son adoption en masse comme standard pour l'industrie de la carte à puce.

Aujourd'hui, le Java Card Forum regroupe les fabricants de cartes, Sun et des utilisateurs. Dès lors, Sun rachète la société « Integrity Arts » afin de développer la technologie Java Card comme plateforme de la technologie Java pour les cartes à puce et les périphériques ayant de fortes contraintes matérielles et mémoires. Cela constitue une véritable aubaine pour Gemplus alors spécialisé dans le développement de machines virtuelles et de systèmes d'exploitation pour cartes à puce.

En novembre 1997, Sun Microsystems fournit la spécification 2.0 de Java Card qui consiste en un sous-ensemble du langage et de la machine virtuelle Java. Cette spécification définit les concepts de base de la programmation et des API très différents de ceux de la version de Schlumberger (version 1.0). Mais il n'y a encore rien sur le format des applets à charger sur la plateforme.

En mars 2001, sort la version 2.1 des spécifications Java Card, qui se compose de trois sous-ensembles :

- une spécification pour les APIs Java Card ;
- une spécification pour l'environnement d'exécution des applications ;
- une spécification pour la machine virtuelle ;

Cette version de la spécification apporte un remaniement des API tels que la cryptographie et la gestion des exceptions. L'environnement d'exécution des applets est standardisé. Mais le changement le plus significatif de cette version est la définition explicite de la machine virtuelle

3. pour Java Virtual Machine ou Machine Virtuelle Java
4. l'environnement d'exécution
5. Application Programming Interface ou interface de programmation est un ensemble de classes mises à disposition d'un programme par une bibliothèque

Java Card (JCVM) et du *capfile* (fichier binaire contenant l'application java compilée), permettant ainsi une vraie interopérabilité.

En juin 2002, Sun Microsystems publie la version 2.2 des spécifications dont les principales nouveautés sont la prise en charge des canaux logiques ainsi que l'invocation de méthodes à distance (RMI voir [Gro02]). Elles rajoutent quelques éléments sur l'installation et l'effacement des applications sur la carte et quelques nouveaux algorithmes cryptographiques.

Depuis cette version, les spécifications se poursuivent ; ainsi en octobre 2003, la version 2.2.1 est publiée (voir [Myc03a ; Myc09a ; Myc03b]), cette dernière corrige l'API de la version précédente et apporte quelques clarifications. Puis lui succède en mars 2006 la version 2.2.2 qui n'apporte pas de changement majeur mis à part quelques nouveaux algorithmes cryptographiques et une clarification des API. Après cette version arrive la branche 3.0 des spécifications qui apporte une révolution dans les spécifications, car elle introduit deux types de spécification Java Card :

- *Java Card 3 Classical Edition* qui est juste une évolution de la Java Card 2.2.2 faite pour fonctionner avec des cartes ayant de faibles ressources. Elle contient les fonctionnalités nécessaires aux supports des applets.
- *Java Card 3 Connected Edition* qui introduit de vraies nouveautés, car en plus des applets classiques, elle supporte un nouveau type d'application : les servlets. Les servlets sont des applications web qui nécessitent d'avoir un serveur web embarqué dans la carte. Elle nécessite donc des cartes avec des contraintes de ressources moins fortes (cartes modernes haut de gamme). Elle introduit un quatrième sous-ensemble dans la spécification celui des servlets. Ainsi pour cette édition, on a :
  - *Java Card Runtime Environment Specification, Connected Edition* [Myc09c] ;
  - *Java Card Application Programming Interface Specification, Connected Edition* [Myc09b] ;
  - *Java Card Virtual Machine Specification, Connected Edition* [Myc09f] ;
  - *Java Card Servlet Specification, Connected Edition* [Myc09d].

2.3 Qu'est-ce qu'une machine virtuelle ?

Lorsqu'une application Java (ou Java Card) est compilée, nous obtenons un fichier binaire au format class. Ce fichier binaire contient des instructions Java ou bytecodes qui servent à représenter le code de l'application. Cette conversion est faite, car Java est un langage interprété, c.-à-d. que le code n'est pas directement exécuté par le processeur du matériel sur lequel doit fonctionner l'application. En effet, la machine virtuelle peut être vue comme un processeur virtuel ayant un jeu d'instructions propre. Ce jeu d'instructions comporte des instructions arithmétiques, des instructions classiques, des instructions de branchement, des instructions logiques, etc.

Ce processeur virtuel ne se comporte pas exactement comme un processeur classique par le fait qu'il ne dispose d'aucun registre utilisateur. Les registres utilisateurs sont des registres que le programmeur peut utiliser directement (bien qu'il existe deux registres spécialisés qui sont le PC [6] et le SP [7]). ils sont remplacés par des variables locales numérotées à partir de zéro. Cette caractéristique peut paraitre étonnante au premier abord, mais elle est tout à fait logique : comme le bytecode Java doit être exécutable sur toutes les plateformes, il n'est pas possible de définir un nombre précis de registres, sans risquer de rencontrer un processeur n'en ayant pas autant.

6. pointeur de code
7. pointeur de pile

L'absence de registres implique l'utilisation intensive de la pile. En réalité, lorsque la JVM charge une classe afin de l'exécuter, elle crée un thread dédié. Chaque thread dispose de sa propre pile, celle-ci n'étant donc pas partagée entre plusieurs instances de classes en cours de traitement. Le processeur fait usage de la pile pour passer des arguments à des fonctions lors de leur appel, comme pour l'exécution d'opérations mathématiques. Le jeu d'instructions comporte donc un certain nombre de commandes permettant de gérer la pile.

Toute machine virtuelle comporte un élément central qui est chargé de l'interprétation du code des applications : l'interpréteur.

2.3.1 Comportement de l'interpréteur d'une machine virtuelle

Pour chaque méthode de l'application en cours d'exécution, l'interpréteur crée un espace mémoire dédié ou Frame [8]. Cet espace mémoire contient un certain nombre d'éléments qui sont nécessaires à l'interprétation des bytecodes. Ces éléments sont la pile des opérandes, ainsi que le tableau des variables locales (voir Fig. 2.1). Enfin, il réalise les opérations relatives à chacune des instructions d'une méthode en utilisant les éléments de la frame.

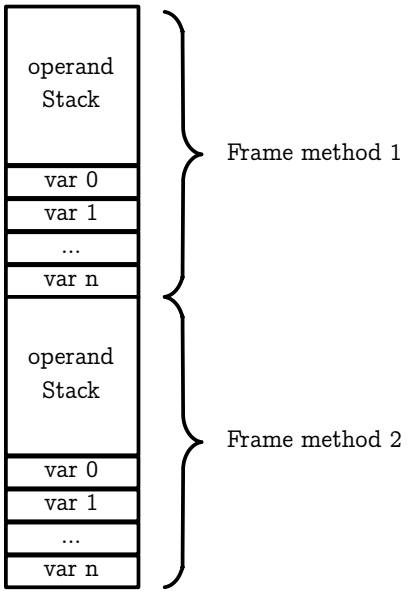

Fig. 2.1 – Représentation schématique de frames Java

2.4 Les avantages de la technologie Java Card

Les avantages de Java Card comme plateforme de gestion d'applications se situent d'une part au niveau des développeurs et d'autre part au niveau des fabricants de cartes. En effet, au niveau des développeurs d'applications la plateforme Java Card apporte les mêmes avantages que le Java par rapport aux autres langages de programmation existants.

8. la frame Java est un espace mémoire dédié à la méthode dans la mémoire vive de la carte

La plateforme Java Card a permis de vulgariser le modèle de développement d'applications pour cartes à puce pour les raisons qui suivent :

- Elle introduit un langage de programmation simple : facile à prendre en main, à compiler, à déboguer, et à apprendre. De plus, tous les développeurs Java sont potentiellement capables de concevoir des applications Java Card, ce qui constitue de nos jours un nombre non négligeable de personnes.
- Elle est orientée objet ce qui permet de créer, de manipuler des objets et de les faire fonctionner entre eux. Offrant ainsi une meilleure modularité ainsi que la possibilité de réutiliser simplement du code déjà écrit.
- Elle encapsule la complexité sous-jacente et les détails du système des cartes à puce qui les rendaient complexe et donc difficile à programmer.
- Elle présente une plateforme ouverte avec des API et un environnement d'exécution standardisé.
- Il existe des outils de développement puissants aussi bien libres que commerciaux, permettant de faciliter la conception et le chargement des applications au sein de la carte.

Tout comme Java, la plateforme Java Card est indépendante du matériel sur laquelle elle s'exécute. En effet, il suffit qu'une carte à puce soit certifiée Java Card pour exécuter une applet.

Elle a été conçue avec l'esprit de sécurité grâce à un typage fort, à une utilisation de référence à la place de pointeur, un pare-feu (firewall), et d'autres mécanismes de sécurité (voir la section 2.6). Tous ces mécanismes empêchent un programme d'avoir un comportement hostile en dehors de son domaine d'action.

Java Card est une plateforme pour cartes à puce capable de faire fonctionner plusieurs applications d'origines diverses. Ainsi, une fois la carte remise à son utilisateur, il est potentiellement possible d'ajouter ou de retirer des applications de la carte ; ce qui facilite la mise à jour des programmes sans avoir à changer les cartes. Par ailleurs, la communication entre deux applets est régie par des règles strictes contrôlées par le pare-feu. En effet, celui-ci empêche toute interaction entre deux applications, si cette interaction n'est pas explicitement stipulée.

Enfin, Java Card a été conçue autour de la norme ISO 7816, lui octroyant de pouvoir supporter les applications qui sont compatibles avec cette norme. Ainsi que d'être compatible avec toutes les cartes à puce Java, mais aussi avec tous les CAD existants. Malgré tous ces avantages, la plateforme n'est pas exempte de défauts. je reviendrai sur ce point par la suite.

2.5 Architecture de la plateforme

Java Card a été conçue pour les équipements fortement contraints tels que les cartes à puce. Les cartes les plus communes aujourd'hui ont des ressources modestes. Mais, elles laisserons progressivement leur place aux cartes modernes qui contiennent des microprocesseurs 32 bits avec des architectures RISC avec beaucoup plus d'EEPROM (500 ko) ou des Gigaoctets de Flash, et des interfaces de communication multiples pouvant fonctionner en USB [9] 2.0 et pouvant contenir plusieurs applications co-résidentes et indépendantes.

9. Universal Serial Bus est une norme décrivant une interface de communication entre deux équipements informatique

Ces nouvelles cartes permettent d'introduire des applications que nous n'aurions jamais cru voir sur ces équipements : stockage d'un film entier, encodage/décodage en temps réel d'une vidéo-conférence, serveur web embarqué. Il fallait donc préparer la plateforme à prendre le virage de ces nouveaux usages, d'où le remaniement de l'architecture Java Card introduite par la version 3.0 de la plateforme déjà existante tout en conservant une rétrocompatibilité avec les versions antérieures.

Du fait que la majorité des cartes d'aujourd'hui soient des cartes avec de faibles ressources, il parait logique de ne pas fermer la porte à ces équipements, et donc d'introduire deux versions de la spécification. L'une pour les cartes les plus communes (Classic Edition) et l'autre pour les cartes de nouvelle génération (Connected Edition). Chacune de ces deux éditions est compatible avec les applications écrites pour les éditions précédentes. Dans ce chapitre je parle plutôt de l'édition connectée puisqu'elle permet aussi de faire fonctionner les applications de l'édition classique.

2.5.1 Java Card 3 Classic Edition

L'édition classique conserve l'architecture initiale de la plateforme voir [Myc09e], car sur les cartes cibles (cartes communes d'aujourd'hui) en raison de la quantité de mémoire disponible, il est impossible d'intégrer le code contenant la gestion des chaînes de caractères, des nombres flottants simples et doubles précisions, ainsi que la gestion du multithreading. Et même si cela était possibile, il ne resterait plus assez de mémoire pour pouvoir installer des applications, ce qui irait à l'encontre de l'esprit d'une carte multiapplicative. Dans ce cas, la seule solution possible est d'implémenter un sous-ensemble de Java.

Dans cette architecture comme le montre la Fig. 2.2 le seul modèle d'application supporté est le modèle basé sur les applets que j'appellerai par la suite applets classiques.

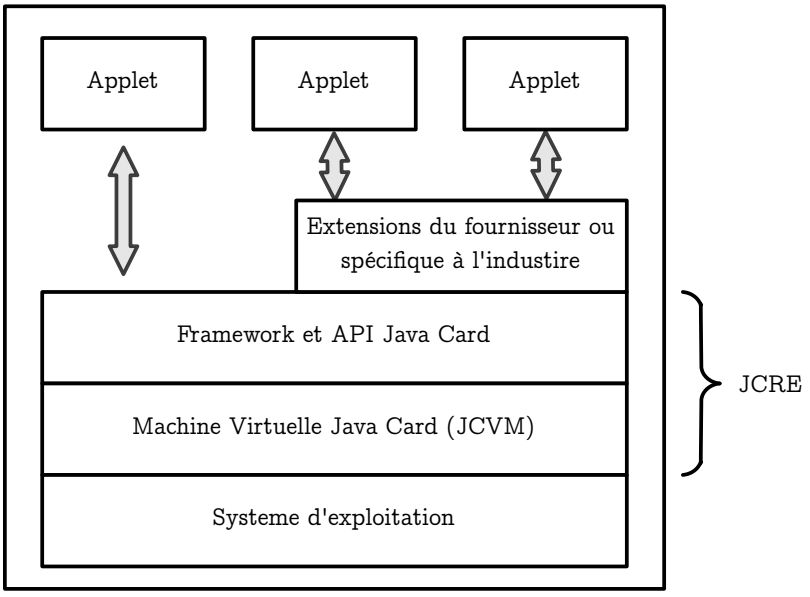

Fig. 2.2 – Architecture Java Card 3 Classic Edition

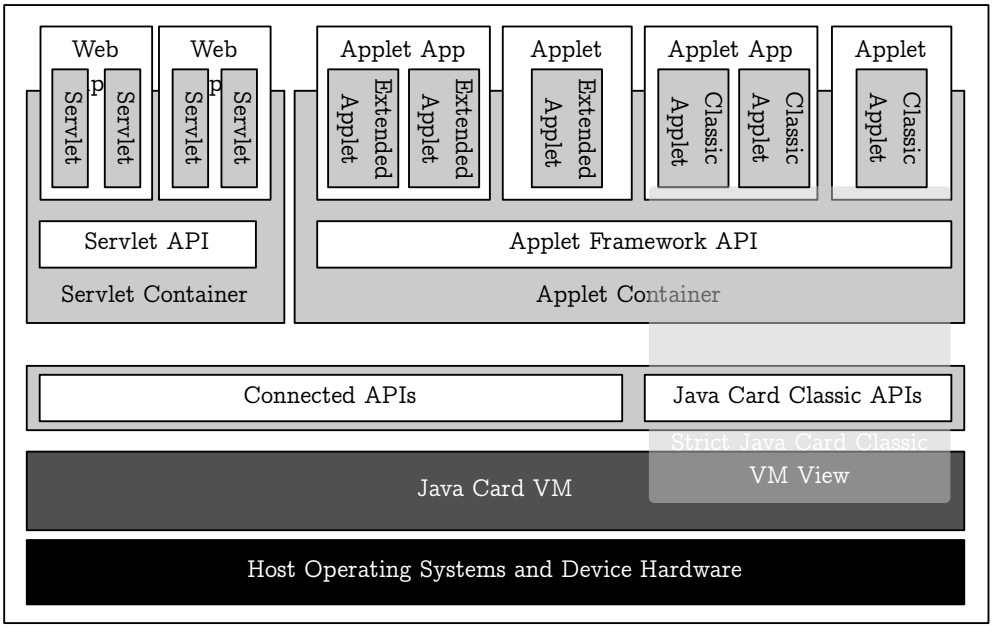

Fig. 2.3 – Architecture Java Card 3 Connected Edition

2.5.2 Java Card 3 Connected Edition

L'édition connectée de la plateforme introduit dans son architecture (voir la Fig. 2.3) une nouvelle machine virtuelle ainsi qu'un nouvel environnement d'exécution qui supporte maintenant trois modèles d'application en opposition à l'unique modèle proposé par l'édition classique des précédentes versions de la plateforme. Elle est pensée pour les cartes de nouvelle génération avec plus de ressources.

Ces trois modèles d'application sont :

- Le modèle d'application utilisant les applets classiques est basé sur celui hérité des précédentes versions de Java Card et à ce titre a les mêmes caractéristiques. Ces applications utilisent le modèle de communication avec la carte basé sur le protocole APDU.
- Le modèle d'application utilisant les applets étendues est aussi basé sur celui des applets, mais avec des améliorations parmi lesquelles on peut citer entre autre la gestion du multi-threading, du format de fichier class, la gestion des chaines de caractères, etc. Elle utilise aussi le protocole de communication APDU.
- Le modèle basé sur les applications Web utilise le modèle d'application basé sur la construction de servlets [10]. Il hérite des nouvelles API introduite avec cette édition de la plateforme. Et elle utilise le protocole HTTPS [11] pour effectuer les communications avec la carte.

2.5.3 Le langage, un sous-ensemble de Java

La quantité de ressources mémoires et processeurs disponible sur la carte étant faible, la plateforme Java Card ne contient qu'un sous-ensemble de Java. Cette section explique les différences

10. Une servlet est une application Java qui permet de créer dynamiquement des données au sein d'un serveur Web
11. Protocole de transfert sécurisé dédié au communication web

qui existent avec Java.

Les éléments de Java non supportés

Dans le cas de l'édition classique et donc des applets classiques, les éléments de Java non supportés sont :

- Le chargement dynamique de classes : les classes sont masquées dans la carte au moment de la fabrication ou téléchargées dans la carte après leur émission. Ainsi, les programmes s'exécutant sur la carte doivent uniquement faire appel aux classes déjà présentes dans celle-ci. Car il n'existe aucun mécanisme permettant de télécharger des applications pendant l'exécution ;
- le gestionnaire de sécurité ou *Security Manager* : il est directement implémenté dans la machine virtuelle. La classe *Java.lang.SecurityManager* n'existe pas au contraire de Java où elle fait office de gestionnaire de sécurité ;
- la finalisation : il n'y a aucune invocation automatique de la méthode *Finalize ()* ;
- les threads : la classe *Threads* n'est pas supportée ainsi qu'aucun des mots clefs liés à la gestion des threads ;
- le clonage d'objet : la classe *Object* n'implémente pas la méthode *clone ()* et il n'y a pas d'interface *cloneable* ;
- le contrôle d'accès : le contrôle d'accès existe bien tout comme en Java par contre avec quelques restrictions (voir [Myc09e] pour plus d'informations) ;
- les types énumérés : pas de types énumérés, ni de mot clef enum ;
- les arguments de taille variable : parce qu'elle nécessite que le compilateur génère un code qui crée un nouveau tableau d'objets à chaque fois qu'un tel argument est invoqué, ceci cause une allocation de mémoire implicite dans le tas de la Java Card ;
- les annotations visibles à l'exécution : car l'introspection de classe n'est pas supportée ;
- les types *char, double, float* et *long* ainsi que les tableaux de plus d'une dimension ;
- de façon générale aucune des classes des API principales de java n'est supportée entièrement. À titre d'exemple dans le paquetage *Java.lang* les classes supportées sont *Object* et *Throwable* ;
- le paquetage *Java.lang.System* n'est pas supporté. La plateforme fournit une classe *javacard.framework.JCSystem* qui apporte une interface vers les fonctionnalités du système ;
- les fichiers class : ne sont pas supportés en effet, le format des applications supporté est le fichier CAP (Converted APplets) qui est un fichier class dont toutes les éditions de liens sont déjà faites afin d'accélérer l'exécution.

Dans le cas de l'édition connectée et donc des applications web ainsi que des applets étendues, les éléments non supportés sont :

- toujours pas de prise en charge des nombres à virgules flottantes c.-à-d. les types float et doubles, par contre, tous les autres types de Java sont supportés ;
- pas de classloader définit par l'utilisateur car toute machine virtuelle Java Card 3.0 doit utiliser les classloaders de la plateforme qui ne peuvent pas être modifiés. Cette fonctionnalité a été supprimée pour des raisons de sécurité ;
- pas de groupes de threads ou des threads fonctionnant comme des démons. Les opérations sur les threads telles que le démarrage d'un thread ne s'applique qu'à des objets threads individuels. Si le développeur a besoin de démarrer un groupe de threads, il doit utiliser une collection d'objets contenant plusieurs objects threads ;

– Toujours pas de finalisation des instances de classes ;
– Pas d'erreur ou d'exception asynchrone : en effet, la gestion des erreurs sur la plateforme est très limitée. Il y a une classe d'erreurs qui est définie dans la spécification ; en dehors de ces erreurs la machine virtuelle doit soit s'arrêter, soit renvoyer l'erreur la plus proche possible de la super classe de l'erreur représentant la condition d'erreur à lever.

Caractéristiques Java supportées

Les caractéristiques Java supportées dans le cas de l'édition classique sont :

– les paquetages,
– la création dynamique d'objets,
– les méthodes virtuelles,
– les interfaces,
– les exceptions,
– la généricité,
– l'importation statique,
– les annotations invisibles à l'exécution,
– les types simples court : *boolean, short, byte*,
– les classes *Object* et *Throwable*,
– ainsi qu'en option le type *int* et le mécanisme de suppression d'objets.

En revanche, l'édition connectée supporte l'intégralité du langage Java :
– ainsi tous les types de données existants sont supportés en Java sauf les *double* et les *float* ;
– le multi-threading ;
– toutes les APIs Java (Java.lang, Java.Util, GCF, etc) ;
– le support en natif des fichier classes avec l'édition des liens qui se fait dans la carte.
– la destruction automatique des objets non utilisés ou *on demand Garbage Collector*

2.6 La sécurité dans Java Card

Il existe plusieurs types de mécanismes de sécurité fournis par Java Card 3, que l'on peut classifier en deux catégories principales :
– ceux qui sont intégrés au langage Java
– ceux qui sont intégrés à la plateforme

2.6.1 Sécurité du langage Java

Le fait que Java Card soit un sous ensemble de Java, il intègre tous les mécanismes de sécurité intrinsèques à ce langage de programmation à savoir :
– qu'il est fortement typé ce qui permet d'éviter les codes agressifs,
– il n'utilise pas de pointeur, il n'y a pas d'arithmétique sur les pointeurs,
– il sépare le langage système du langage applicatif ce qui permet d'éviter les attaques du système par le biais des applications
– il ne permet pas de conversion de type non prévu, il n'y a donc aucun moyen de falsifier un pointeur,

– l'opération de cast suit des règles strictes, cast implicite d'un sous-type vers un super-type, et cast explicite obligatoire d'un type vers un sous-type,
– au niveau du bytecode, il y a beaucoup de vérifications pour lutter contre les opérations de cast non prévues.

2.6.2 Sécurité de la plateforme

Le vérifieur de fichiers classes

Comme dans Java, la plateforme Java Card doit être capable de refuser un fichier classe. Cela signifie que l'implémentation de la plateforme doit supporter la vérification de fichiers classes. Le processus de vérification a lieu durant le chargement de l'application, et consiste à faire une interprétation abstraite du bytecode qui implique entre autre de vérifier le type des objets manipulés, et la consistance de la pile.

Avec la nouvelle version de la plateforme, cette vérification est implémentée en utilisant l'approche de vérification de type basée sur l'attribut *StackMapTable*. Cet attribut du fichier classe contient soit zéro, soit plusieurs états du type des variables locales et des données contenues dans la pile des opérandes aux points de jonction (endroit du code où il y a un transfert des flots de contrôle). Cet attribut est systématiquement ajouté au fichier classe par le compilateur. Il est considéré comme une preuve formelle de validité du code. Il est utilisé par le *vérifieur* lors du chargement de l'application afin qu'il puisse effectuer la validation du code de l'application. Chaque méthode a son propre attribut *StackMapTable*.

Le pare-feu : isolation de contexte et partage d'objets

Le pare-feu permet d'apporter une protection contre la fuite d'informations vers d'autres applications dues aux erreurs de programmation et de conception. Une application peut obtenir la référence d'un objet d'une zone publiquement accessible, par contre, si cet objet appartient à une application protégée par le pare-feu, la première application doit satisfaire un certain nombre de conditions avant de pouvoir accéder à cet objet. Le pare-feu apporte également une protection contre un code incorrect. En effet, si un code incorrect est chargé dans la carte, il protège les objets contre les accès provenant de ce code.

Le pare-feu partitionne le système en espaces protégés, appelés des contextes de groupe (voir la Fig. 2.4). Un contexte de groupe correspond à un paquetage Java. Quand une applet est créée le système de runtime de Java Card lui attribue un AID [12] (Applet IDentification) à partir duquel on peut facilement récupérer le nom du paquetage dans lequel elle est définie. Si deux applets sont des instances d'une classe appartenant au même paquetage, alors elles appartiennent au même contexte de groupe. En plus des contextes définis par les applications s'exécutant sur la carte, il y a un contexte spécial qui est le contexte du système, appelé contexte du JCRE [13] (Java Card Runtime Environment). Les applets appartenant à ce contexte peuvent accéder aux objets de n'importe quels autres contextes de la carte.

12. l'AID est une chaine de caractères définie dans l'ISO 7816 permettant d'identifier de façon unique une applet dans le système de fichiers de la classe
13. Environnement d'exécution des applications Java card

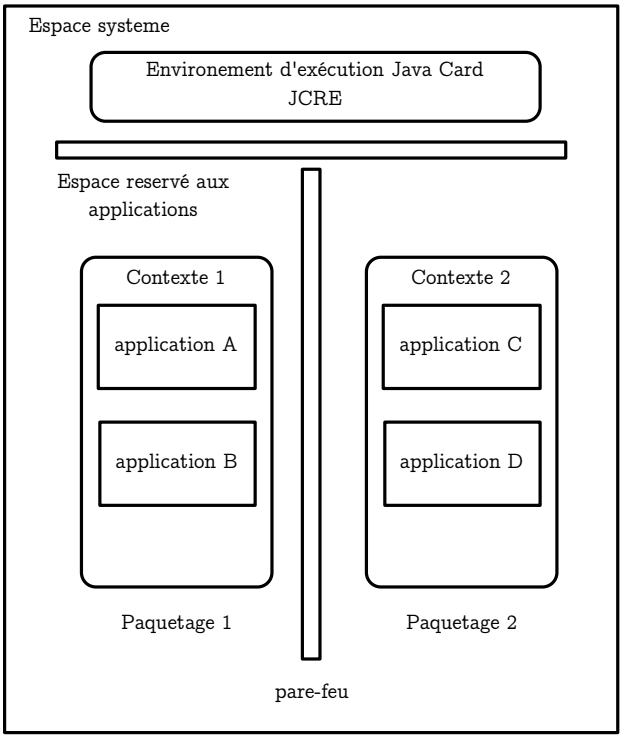

Fig. 2.4 – Firewall dans la Java Card 3.0

Chaque objet est associé à un contexte unique qui est le contexte de l'applet qui l'a créé. Les méthodes d'un objet s'exécutent dans le contexte de cet objet. C'est ce contexte qui permet de déterminer si l'accès à un autre objet est accordé ou pas. Le pare-feu isole les contextes en faisant en sorte qu'une méthode s'exécutant dans un contexte ne puisse pas accéder aux objets appartenant à un autre contexte.

Il y a deux cas où le pare-feu autorise l'accès d'une méthode à un autre contexte que le sien : dans le cas des objets appartenant au contexte système ou à travers les objets dits «*shareable*». Les objets appartenant au JCRE sont des objets qui peuvent être atteints depuis n'importe quel contexte (par exemple l'objet APDU). Les objets shareable permettent de faire du partage entre applets en donnant un accès limité aux objets d'autres contextes. En utilisant ce mécanisme, une applet peut donner un accès à une méthode appartenant à un objet en utilisant l'interface shareable qui est une interface qui étend javacard.framework.shareable.

La gestion des autorisations

L'édition connectée de Java Card contient un mécanisme de contrôle d'accès aux services de la plateforme ainsi qu'à ses services partagés (c.-à-d. les services dispensés par d'autres applications). Ce mécanisme est implémenté au sein même du code de la plateforme (voir l'exemple du Code 1).

Ainsi, lors du chargement de l'application un fichier définissant les permissions (fichier *policy*) de l'application est ajouté. Lors de l'exécution, si une application souhaite accéder à un service, le système récupère le contexte d'exécution de l'application (car chaque application s'exécute dans un contexte qui lui est propre) ; de celui-ci, il récupère les permissions définies pour l'application et vérifie si ces permissions autorisent ou non cet accès.

Code 1 Contrôle d'accès Java Card 3

```
public Cipher getInstance (String algoName) {
    AccessControler.checkPermission (
            new CryptoServicePermission(algoName))
    //instanciation code
}
```

L'isolation de code et le contrôle d'accés des paquetages du langage Java

L'isolation du code permet d'être certain qu'un code chargé avec une application n'interfère pas avec le code d'applications déjà présentes ou à venir. Le code chargé de cette manière ne peut pas modifier ou accéder de façon directe au code d'autres applications.

Le chargeur de classe (ou *classloader*) permet de séparer les classes appartenant à différentes applications et de charger/décharger simplement du code. Toutefois, au contraire de Java, la plateforme n'accepte pas les chargeurs de classes définis par l'utilisateur.

L'authentification

L'authentification est le mécanisme par lequel un utilisateur prouve son identité à la carte. L'authentification est un élément important pour la plateforme Java Card. En effet, c'est au travers de ce mécanisme que la carte peut décider quels sont les droits de l'utilisateur qui s'est authentifié sur la plateforme. La carte Java Card 3 connected edition base une partie de sa sécurité sur des rôles. Un rôle détermine si un utilisateur authentifié a la possibilité d'accéder à des ressources protégées, telles que les services basés sur les SIO [14], ainsi qu'aux ressources Web.

Sur la plateforme Java Card, les utilisateurs se subdivisent en deux catégories : le propriétaire de la carte (qui est l'utilisateur principal), et les autres utilisateurs (par exemple l'administrateur de la carte). Chaque utilisateur a une identité utilisateur différente sur la carte.

L'authentification sur la carte se fait grâce à des identificateurs ou *authenticators*. Un identificateur est un service spécialisé dans l'authentification qui peut utiliser plusieurs schémas d'authentification tels qu'un mot de passe, un code PIN [15], ou une information biométrique. Ce service peut être utilisé par une application ou par un conteneur Web.

Les communications sécurisées

Étant donné que sur la plateforme, il existe un nouveau protocole de communication : HTTP [Fie+99] (*HyperText Transmission Protocol*), celui-ci se doit d'être sécurisé contre les attaques provenant du réseau telles que l'injection de paquets ou l'écoute des transmissions. C'est pour cette raison que ce protocole est sécurisé via un algorithme cryptographique : TLS [Wea06] (*Transport Layer Security*). Ce protocole crée un tunnel sécurisé entre le client et le serveur, permettant de s'assurer que personne ne peut espionner les communications, ni les trafiquer.

14. SIO ou Shareable Interface Object : ce sont des objets Java Card qui implémentent l'interface Shareable voir sous-section 2.6.2
15. PIN : Personal Identification Number

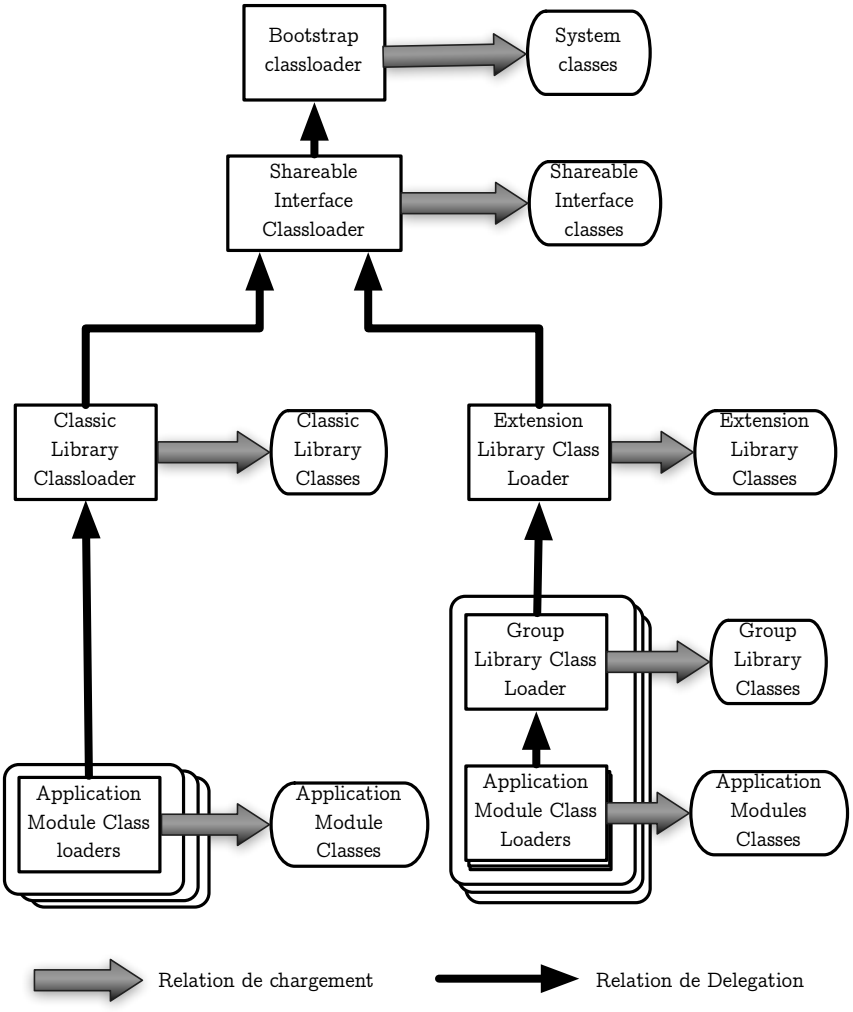

Fig. 2.5 – Le chargeur de classe dans la Java Card 3.0

Les annotations de sécurité

La spécification introduit des annotations de sécurité aux applications qui permettent de renforcer leur exécution par le système. Ce qui permet de se protéger contre les comportements inattendus de la plateforme (les attaques en fautes, ou les attaques par canaux cachés). Ce sont des annotations Java au niveau de la classe (Code 2), ou au niveau de la méthode (Code 3). Le plus petit élément que l'on peut marquer en utilisant cette technique est la méthode.

Code 2 Exemple d'utilisation des annotation de sécurité au niveau de la classe

```
@sensitiveClass (Sensitive.CONFIDENTIAL);
public class KeyImplementation implements Key
        public void setKey (byte[] value) {
                ...
        }
}
```

Code 3 Exemple d'utilisation des annotation de sécurité au niveau de la méthode

```
@SensitiveMethod (Sensitive.INTEGRITY);
public boolean verifyAccessCondition (AccessCondition ac){
....
}
```

Ces annotations peuvent être utilisées pour protéger le contenu de classes sensibles ou de méthodes sensibles. Pour ce faire, la machine virtuelle peut soit passer dans un mode d'exécution sécurisé (implémentation de mécanisme de vérification d'intégrité du code) soit renforcer le code à exécuter grâce à des outils externes. Il faut noter que la plateforme fournit ces annotations, mais laisse le fournisseur de carte ou le développeur du système d'exploitation implémenter les mécanismes qui rentrent en jeux pour ce qui est de la sécurisation du code.

2.7 Conclusion

Dans ce chapitre, nous avons vu quels étaient les avantages de la plateforme Java Card : à savoir une plateforme offrant de la portabilité aux applications pour cartes à puce, une simplicité de développement du fait de l'existence de nombreux environnements de développement et de la simplicité de l'apprentissage du langage Java, et de nombreuses fonctionnalités de sécurité. De plus, c'est la plateforme pour cartes à puce la plus déployée dans le monde. Toutes ces raisons m'ont convaincus de choisir cette plateforme pour mon travail de recherche.

Chapitre 3

Les attaques

Sommaire

3.1 Introduction

Au vue des domaines dans lesquels la carte à puce est utilisée, il est logique que des personnes avec des intentions peu louables souhaitent avoir accès aux informations secrètes stockées dans la carte. Dans un autre contexte, pour tester la résistance des périphériques aux attaques, les fabricants de matériel mettent au point de nouveaux moyens de contourner la sécurité des cartes afin de détecter les faiblesses de leurs produits et ainsi d'améliorer les protections qui leurs sont intégrées. En somme, une attaque consiste à utiliser les caractéristiques ou les particularités de la cible à attaquer afin de tenter de contourner les mécanismes de sécurité matériels ou logiciels qui lui sont intégrés. Un attaquant est un individu qui souhaite obtenir des informations par des moyens non conventionnels en appliquant une attaque sur la carte à puce.

Les attaques peuvent être logicielles ou matérielles. Les attaques logicielles sont des attaques permettant d'exploiter les logiciels de la plateforme pour être réalisées : comme exemple, on peut citer les attaques en dépassement de capacité, ou l'injection de code. Les attaques matérielles consistent à perturber par des moyens physiques les composants matériels de la carte à puce à attaquer : on peut citer les attaques par injection de fautes, ou les attaques par canaux cachés.

On peut classifier les attaques en faute en fonction de l'effet induit sur la carte. Les attaques peuvent être invasives, dans ce cas, elles portent atteinte à l'intégrité des composants. Une fois qu'une telle attaque est réalisée la carte devient inutilisable. Elle nécessite un haut degré de compétences en microélectronique ainsi qu'un équipement couteux et spécialisé. Les attaques non invasives, quant à elles, n'affectent pas l'intégrité des composants de la puce. Le matériel nécessaire pour effectuer une attaque de ce type est en général moins couteux (tel qu'un flash d'appareil photo, une bobine capable de générer un champ électromagnétique...) ainsi qu'un bon niveau de compétence en microélectronique. Cette deuxième catégorie s'appuie souvent sur les canaux cachés tels que la consommation énergétique de la puce lors de certaines opérations afin d'être mise au point.

3.2 Les attaques par injection de fautes

Cette partie concerne uniquement les attaques par injection de fautes (car tout mon travail s'articule autour de ce type d'attaques). L'attaque par injection de fautes consiste à appliquer une force extérieure sur le matériel afin d'induire des erreurs ou fautes dans le microprocesseur. Ces fautes permettent dans une certaine mesure à l'attaquant de pouvoir obtenir des informations sensibles comme les clefs cryptographiques ou d'effectuer des traitements normalement sécurisés par la réalisation d'un évitement du test d'un code PIN.

Ces attaques consistent à perturber de manière très brève l'environnement physique de la carte durant l'exécution. On peut ainsi faire des *glitches*[16] positifs ou négatifs, par exemple sur la tension d'alimentation. On peut aussi, à l'aide de laser, envoyer des impulsions lumineuses de courte durée et de différentes natures telles que des rayons infrarouges, de la lumière blanche, des faisceaux d'électrons... sur la puce. De façon générale, tous les moyens physiques capables d'effectuer un ajout d'énergie à la puce peuvent causer une faute (par exemple l'utilisation d'un champ électromagnétique).

En général, l'injection de fautes est réalisée sur plusieurs exécutions du même programme à des instants différents de l'exécution et cela en faisant varier la durée d'exposition de la puce à la perturbation pour pouvoir identifier la zone mémoire la plus intéressante à attaquer.

J'utiliserai indistinctement les termes d'attaque en faute et d'attaque par injection de fautes pour désigner la même chose. Le domaine des attaques par injection de fautes n'est pas nouveau, en témoigne la quantité de publications qui parle du sujet [BE+06 ; GK04 ; YMH02 ; Hem04 ; KQ07 ; GT04 ; PQ03 ; Ott05]

3.3 Quelques attaques par injection de fautes

3.3.1 Attaque électrique

Cette attaque consiste à faire brusquement varier la tension d'entrée de l'alimentation de la puce afin de perturber l'exécution de certaines opérations [Aum+03]. Cela est suffisant pour obtenir une faute exploitable. Cette attaque est rendue possible par le fait que la carte n'est pas auto alimentée,

16. brèves variations

l'énergie qu'elle utilise pour effectuer des opérations lui vient du CAD. De plus, ces variations peuvent être modifiées selon neuf critères parmi lesquels on peut citer la puissance, la forme du pic, la durée, etc.

3.3.2 Attaque par variation de fréquence de l'horloge

Cette attaque utilise la même faille de conception que dans le cas des attaques électriques. En effet, la carte tire sa fréquence d'horloge du CAD ; il est dès lors possible de faire varier celle-ci. En faisant varier la vitesse de l'horloge hors des limites autorisées par la carte, on peut induire des fautes au niveau du microprocesseur. Il est par exemple possible de faire doubler la fréquence de l'horloge afin de faire exécuter au microprocesseur une opération t2, avec des paramètres qui devraient normalement être utilisés pour une opération t1 qui lui est antérieure [KKS99 ; Tri+10].

3.3.3 Attaque optique ou par laser

En utilisant une source lumineuse concentrée sur la puce, il est possible d'apporter suffisamment d'énergie à une cellule mémoire pour lui faire changer son contenu. C'est en se basant sur cette propriété physique des mémoires que cette attaque est née. Il est montré en [SA03] que le matériel nécessaire à cette attaque n'est pas forcement coûteux c.-à-d. un flash d'appareil photo, ou un laser. Pour mener à bien cette attaque, la lumière a besoin de pouvoir toucher directement la puce, c'est pourquoi elle nécessite l'extraction des couches protectrices de cette dernière. Une fois l'attaque réalisée, il est impossible de réutiliser la carte. Ainsi, on peut la classer parmi les attaques invasives.

3.3.4 Attaque par perturbation électromagnétique

En créant un fort champ électromagnétique à proximité d'une cellule mémoire, les ions représentant l'état de cette cellule mémoire bougent, permettant ainsi de modifier le contenu de cette mémoire. Il est établi en [SQ02 ; SH07], que le courant de Foucault qui est créé à l'aide d'une bobine active assez puissante est capable de générer un champ électromagnétique suffisant pour perturber la mémoire de la carte. Ici, il n'est pas utile d'enlever les couches protectrices de la carte afin d'effectuer l'attaque ; en effet, il suffit de générer le champ électromagnétique à proximité de la carte. De ce fait, c'est une attaque non invasive.

Aujourd'hui, la majorité des cartes sont équipées de contremesures pour lutter contre l'injection de fautes par le courant électrique ou par la fréquence de l'horloge alors qu'il est encore difficile de lutter contre les attaques électromagnétiques ou optiques, car elles ajoutent de l'énergie directement à la surface de la puce. Ce sont donc les deux catégories d'attaques contre lesquelles il est le plus difficile de se protéger.

3.4 Conséquences de l'injection de fautes

On retrouve dans la littérature divers moyens d'effectuer de l'injection de fautes comme en témoigne la pléthore d'articles qui traitent du sujet (voir section 3.3).

Les conséquences de l'attaque par injection de fautes peuvent être plus ou moins graves. On peut citer des perturbations :

- Dans le *microprocesseur* : en effet, les perturbations peuvent entrainer des modifications dans les registres contenus dans le microprocesseur tels que le pointeur de pile (ou SP pour *Stack Pointer*), ou le pointeur de code (ou PC pour *Program Counter*). Une modification du PC peut par exemple faire pointer le code dans une zone mémoire différente de celle qui devait être exécutée.
- Dans les *différentes mémoires* : en effet, l'attaque peut aussi avoir lieu dans l'EEPROM ou la mémoire flash, par exemple où peuvent être stocké le code des applications, les clefs cryptographiques, ou même le code PIN de l'utilisateur. Dans la RAM, où l'on peut modifier la structure des objets temporaires manipulés. Le résultat obtenu peut être une inconsistance des données, par contre, notons que les données importantes peuvent être protégées par des techniques de signature.
- Sur les *bus* : la faute sert à modifier les données qui transitent sur le bus. C'est un moyen d'attaquer les informations contenues dans la carte, car les données sauvegardées ou manipulées circulent sur les bus entre les différents composants de la carte à puce. Cela peut dans certains cas faciliter l'attaque des données sauvegardées dans la carte.

En somme, une attaque en faute entraîne des modifications du code, des perturbations dans le flot de contrôle et ou de données, ou une modification des données. Mais la carte contient des mécanismes de protection suffisant pour assurer l'intégrité des données. En effet, des mécanismes de redondance des données, de chiffrage des bus et de la mémoire, de dissémination de l'information... garantissent que les données manipulées sont bien les données initialement sauvegardées dans la mémoire. Reste que l'on peut quand même altérer le code des applications qui utilisent ces données. Du fait de l'existence de ces mécanismes, on s'intéresse uniquement aux conséquences des attaques sur le code et sur le flot de contrôle.

3.5 Modèle de fautes

Pour se prémunir contre une attaque par injection de fautes, il faut savoir quels sont les éléments à même de pouvoir caractériser cette faute. Avoir un modèle d'attaque permet de déduire l'impact de la faute sur les applications à protéger. Les différents critères qui caractérisent une faute sont les suivants :

- La *fenêtre temporelle* pendant laquelle l'attaque s'effectue. En effet, une attaque réussie doit pouvoir être synchronisée avec une instruction ou alors avec la manipulation d'une donnée importante.
- La *localisation spatiale* de la faute à la surface de la puce. Si l'on veut effectuer une attaque efficace, il faut savoir à quel endroit de la carte on l'effectue. Il faut donc arriver à distinguer les différents types de mémoire.
- La *précision* est la quantité d'information modifiée par l'attaque. Car en fonction des propriétés physiques de l'attaque, et des protections intégrées à la puce, il est plus ou moins difficile d'affecter des zones mémoires très petites. Plus la carte contient de mécanismes de protections, et plus il est difficile de modifier une zone suffisamment petite et précise de la mémoire.
- Le *type de faute* est l'état de la zone mémoire modifiée après l'attaque. Ici, il faut savoir si l'attaque à la capacité de changer l'information vers une valeur donnée dans le cas des mémoires non chiffrées ou si elle est soumise à un aléa dans le cas des mémoires chiffrées (je

reviendrais sur ce point par la suite).

Grâce à ces différents critères, on peut en déduire les modèles de fautes cités dans le Tableau 3.1 (voir [BOS03 ; Wag04 ; Gir07 ; Ott05]). Pour les critères de fenêtre temporelle et de localisation, l'attaquant a trois niveaux de contrôle :

- *Total* : il a un contrôle total sur le critère.
- *Moyen* : il a un contrôle peu précis sur le critère.
- *Aucun* : il n'a aucun contrôle du critère.

Modèle de fautes	Timing	Localisation	Précision	Type de faute
Precise bit error	total	total	bit	set (0) ou reset (1)
Precise byte error	total	total	octets	0x00 ou reset (0xFF) ou aléatoire
Unknown byte error	moyen	moyen	octets	set(0x00) ou reset (0xFF) ou valeur aléatoire
Unknown error	aucun	aucun	variable	aléatoire

Table 3.1 – Les différents modèles de fautes

Dans ce tableau, les modèles sont cités par ordre décroissant de puissance de l'attaquant, c.-à-d. que le modèle de fautes "precise bit error" est plus compliqué à mettre en oeuvre que le modèle "unknown error". Au cas où, un des critères précédents n'a pas la valeur adéquate, la carte irait lever une exception sur la plateforme, ce qui entrainerait son verrouillage ou alors elle retournerait une information inexploitable à l'attaquant. Une attaque ayant une précision peu élevée peut toucher plusieurs octets, et très probablement altérer la structure interne des objets, ce qui peut causer des problèmes dans le système d'exploitation de la carte s'accompagnant de l'arrêt brutal de l'interpréteur de la machine virtuelle et par la même occasion, l'arrêt de l'exécution d'un programme.

Le laser est la méthode la plus utilisée dans les laboratoires pour réaliser une attaque par injection de fautes parce qu'il permet d'avoir une bonne précision sur la zone mémoire ciblée. En réalité, lorsque le laser est utilisé pour injecter une faute, il rajoute de l'énergie au niveau physique dans la mémoire de sorte que la zone mémoire visée passe à la valeur 0x00 ou à la valeur 0xFF en fonction de la plateforme sur laquelle on se trouve. Il existe deux comportements bien distincts au niveau logique, dépendant du type de mémoires dont est équipée la carte à puce :

- Dans le cas d'une mémoire chiffrée, le chiffrage étant fonction d'une clef de chiffrement, de la donnée à chiffrer ainsi que de son adresse mémoire, la valeur physique 0x00 ou 0xFF injectée se transforme en une valeur inconnue du fait de la fonction de chiffrement utilisée. Etant donné que cette fonction de chiffrement est une propriété du fournisseur de la mémoire, elle est une inconnue ; cette situation sera traduite en terme de modélisation par le fait que la valeur obtenue après déchiffrement correspond à une valeur aléatoire.
- Dans le cadre d'une mémoire non chiffrée, le phénomène est plus simple à expliquer en effet, la valeur physique injectée 0x00 ou 0xFF est la valeur logique dont la plateforme se sert puisqu'il n'y a aucune opération de déchiffrement à faire.

Le modèle de fautes étudié lors de mon travail de recherche est le modèle "precise byte error" ; dans lequel l'attaquant a la capacité d'influencer n'importe quel octet qu'il souhaite, au moment exact où il le souhaite, en ayant la possibilité d'injecter la valeur 0x00 - 0xFF (cas des mémoires

non chiffrées) ou alors une valeur aléatoire qu'il ne maîtrise pas (cas des mémoires chiffrées). Le choix de ce modèle paraît réaliste dans le contexte actuel, car dire que l'attaquant a la capacité de toucher un bit de façon précise aujourd'hui est une hypothèse forte qui suppose des cartes ne disposant pas de l'ensemble des protections matérielles dont elles disposent. De plus, dans le cadre de mémoire chiffrée avoir un impact sur un seul bit revient à avoir un impact sur un octet entier lors du déchiffrement. Il ne faut pas non plus partir sur un modèle de fautes trop permissif tel que le dernier modèle de fautes cité, car, il suppose un attaquant peu puissant or avec les progrès réalisés en physique et en microélectronique, les attaquants ont des moyens élevés. Ces éléments ont conforté mon choix, ce qui permet également de protéger les applications contre les attaques utilisant un modèle de fautes moins puissant.

3.6 Problématique

L'injection de fautes peut toucher la puce en différents endroits de la carte (voir section 3.4), mais elle peut aussi avoir lieu à différentes étapes du cycle de vie de l'application. En effet, la Fig. 3.1, montre schématiquement ce qui se passe durant la création d'une application jusqu'à son stockage dans la mémoire volatile de la carte. L'application est créée sur le serveur, puis est chargée dans la carte, où elle sera stockée, sélectionnée et exécutée par l'interpréteur de la machine virtuelle. Le protocole utilisé lors de l'étape de chargement (Global Platform) permet de s'assurer que l'application qui est envoyée dans la carte reste la même (c.-à-d. que son code reste identique) entre le moment du chargement et le moment de l'arrivée dans le buffer (mémoire tampon) d'entrée sortie de la carte. Car ce protocole dispose de mécanismes de contrôle d'intégrité.

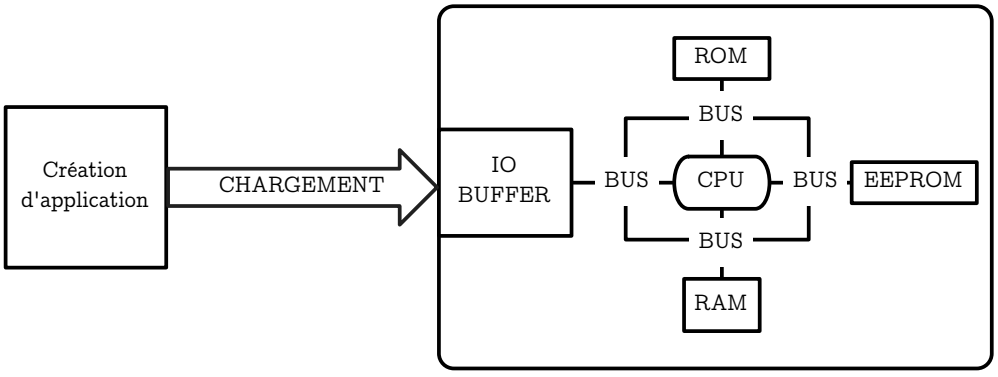

Fig. 3.1 – Chargement d'une applet

L'injection de fautes peut avoir lieu entre le moment du chargement et l'exécution de l'application dans la machine virtuelle ; mais, du fait de l'utilisation de Global Platform, je ne considère pas les attaques pouvant survenir lors du chargement de l'application. Je m'intéresserai uniquement à celles qui ont lieu dans la carte et plus précisément, à partir de l'arrivée dans le buffer d'entrée sortie de celle-ci. L'injection de fautes a des effets divers sur l'exécution des applications (voir section 3.4), mais les données contenues dans la carte étaient bien protégées (voir section 3.4). Ces deux hypothèses permettent de limiter le champ d'étude à la vérification des effets des fautes sur le code des applications : modification de l'intégrité des applications.

En conclusion, la problématique à laquelle je m'attaque est celle selon laquelle nous voulons nous assurer que :

> "*le code de l'application qui a été chargé dans la carte, est exactement le même à chaque fois qu'il est utilisé par la machine virtuelle. Cela sans utiliser de mécanismes classiques de contrôle d'intégrité à base de checksums.*"

En général, le contrôle d'intégrité se fait avant ou après l'exécution du code. Or l'attaque en faute peut survenir pendant l'exécution du code. Du coup le mécanisme de vérification d'intégrité ne décèlerait pas de modification car dans le premier cas, la détection interviendrait après la vérification. Dans le second cas, la carte pourrait retourner de l'information à l'attaquant avant que la vérification ne soit faite. De plus, ces mécanismes sont souvent très couteux en terme de ressources processeurs (voir la sous-section 4.5.2).

Chapitre 4

Etat de l'art des contremesures

Sommaire

4.1 Introduction

Dans le domaine de la sécurité informatique, lorsqu'une faille est découverte dans un système, la communauté scientifique ou industrielle met tout en œuvre pour fournir des mécanismes permettant de détecter son exploitation ou en la comblant. En faisant le rapprochement avec notre domaine d'activité, la faille serait l'attaque par injection de fautes, le système à protéger est la carte à puce, et dans l'ordre des choses il existe des mécanismes visant à enrayer la mise au point des attaques par injection de fautes ou à les détecter.

Les protections peuvent être matérielles, dans ce cas elles sont déjà intégrées à la puce (voir sous-section 1.8.4) par le fabricant de la puce elle même, ou alors elles peuvent être logicielles et dans ce cas chacun des acteurs participant au cycle de vie de la carte, excepté le porteur de la carte, est à même d'intégrer de la sécurité dans le puce à des niveaux plus ou moins bas du système (système d'exploitation, machines virtuelles ou applications). Il est plus aisé de retrouver dans la littérature

des protections pour lutter contre les attaques en faute sur les algorithmes cryptographiques, mais plus difficile d'en trouver portant sur le code des applications. Ce chapitre va se concentrer sur les précédents en matière de contremesures logicielles permettant de lutter contre les attaques par injection de fautes sur le code des applications.

4.2 Illustration

Un exemple concret va être utilisé afin d'illustrer l'ensemble des techniques de protection évoquées dans ce chapitre. Cet exemple sera une application servant à assurer la gestion d'un porte-monnaie électronique. C'est une applet simple utilisant l'authentification de l'utilisateur par code pin. En effet, une fois insérée dans le lecteur l'applet va demander à l'utilisateur de rentrer son code pin, une fois celui-ci vérifié et l'utilisateur authentifié, l'opération qu'il a requise est exécutée (validation du code PIN, débit, crédit, vérification du montant disponible...). Plus précisément, la méthode considérée est celle de «débit», dont un condensé du code source se trouve dans le Code 4 (voir la section A.1 pour la totalité du code source) et sa représentation en bytecode qui se trouve au niveau du Code 5 (voir la section A.2 pour la totalité du bytecode correspondant). Cette méthode vérifie que le code PIN de l'utilisateur a bien été validé et effectue l'opération de diminution du solde du porte-monnaie.

Code 4 Extrait d'une méthode de débit provenant d'une applet de portemonnaie électronique

```
private void debit(APDU apdu) {

        // access authentication
    if ( pin.isValidated() ) {
                ...

                balance = (short) (balance - debitAmount);

        } else {

                ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);

        }
} // end of debit method
```

Code 5 Extrait de la représentation en bytecode de l'opération de débit

```
  0 aload_0
  1 getfield #4 <fr/xlim/ssd/wallet/Wallet.pin>
  4 invokevirtual #18 <javacard/framework/OwnerPIN.isValidated>
  7 ifeq 98 (+91)
       ...
 84 aload_0
 85 getfield #20 <fr/xlim/ssd/wallet/Wallet.balance>
 88 iload 5
 90 isub
 91 i2s
 92 putfield #20 <fr/xlim/ssd/wallet/Wallet.balance>
 95 goto 104 (+9)
 98 sipush 25345
101 invokestatic #13 <javacard/framework/ISOException.throwIt>
104 return
```

4.3 Définitions importantes

Afin de bien comprendre les méthodes de protection utilisées dans ce chapitre, il faut d'abord s'assurer que les termes clefs sous-jacents sont bien définis. Ainsi, cette section a pour but de définir ces notions clefs qui pour la plupart proviennent du domaine de la compilation ou de la théorie des graphes.

4.3.1 Flot de contrôle

En programmation un flot de contrôle désigne l'ordre dans lequel sont exécutées les instructions, ou les fonctions d'un programme. Ce flot de contrôle peut être modifié ou interrompu par divers type d'instructions pouvant entrainer la séparation du flot en zéro, un ou plusieurs chemins. Ce type d'instructions dans le cas de Java en général et de Java Card en particulier peut se subdiviser en plusieurs catégories :

- *Branchement inconditionnel* ou *saut* (voir Fig. 4.2.a) : c'est le cas lorsque l'exécution se poursuit à une instruction différente.
- *Branchement conditionnel* (voir Fig. 4.2.b) : c'est le cas lorsqu'il y a une condition qui doit être satisfaite afin qu'un groupe d'instructions puisse être exécuté. Dans ce cas, il y a, au maximum, une alternative d'exécution.
- *Branchement conditionnel à choix multiple* (voir Fig. 4.2.d) : ce sont des branchements conditionnels un peu particulier car il peut y avoir plus d'une alternative d'exécution.
- *Boucles* (voir Fig. 4.2.c) : correspondent à l'exécution zéro ou plusieurs fois d'un groupe d'instructions jusqu'à ce qu'une condition soit satisfaite.
- *Arrêt inconditionnel :* constitue des instructions capables d'arrêter le cours de l'exécution d'un programme telles que les retours de fonction ou les levées d'exception.

4.3.2 Blocs élémentaires

Un bloc élémentaire se compose de l'enchaînement d'instructions successives ne contenant aucune rupture du flot de contrôle et aucune cible d'instructions de branchement ; le bloc élémentaire a un point d'entrée et un point de sortie, ainsi, si un bloc est exécuté, chacune des instructions le composant sera exécutée sans interruption. Le point d'entrée du bloc peut être atteint de plusieurs autres blocs. Le point de sortie du bloc pourra être toute instruction capable de rompre le flot de contrôle (voir sous-section 4.3.1). La Fig. 5.3 montre le premier bloc élémentaire de la méthode de débit (voir Code 5) qui va de la ligne 0 avec l'instruction *aload_0* jusqu'à la ligne 7 avec l'instruction *ifeq*.

```
00: aload_0;
01: getfield 4;
04: invokevirtual 18;
07: ifeq 98 (+91);
```

Fig. 4.1 – Le premier bloc élémentaire de la méthode débit (voir Code 4)

4.3.3 Graphe de flots de contrôle ou CFG

En informatique, un graphe de flots de contrôle est un graphe représentant l'ensemble des chemins que peut emprunter l'interpréteur lors de l'exécution d'un programme. L'exemple de la Fig. 4.2 montre un graphe de flots de contrôle. Les sommets du graphe se composent de blocs élémentaires ; les arrêtes du graphe se composent des relations entre l'instruction terminant un bloc et l'instruction débutant un autre bloc (par exemple si l'instruction qui débute un bloc est la cible du branchement de l'instruction qui termine un autre bloc alors il existe une arrête entre ces deux blocs).

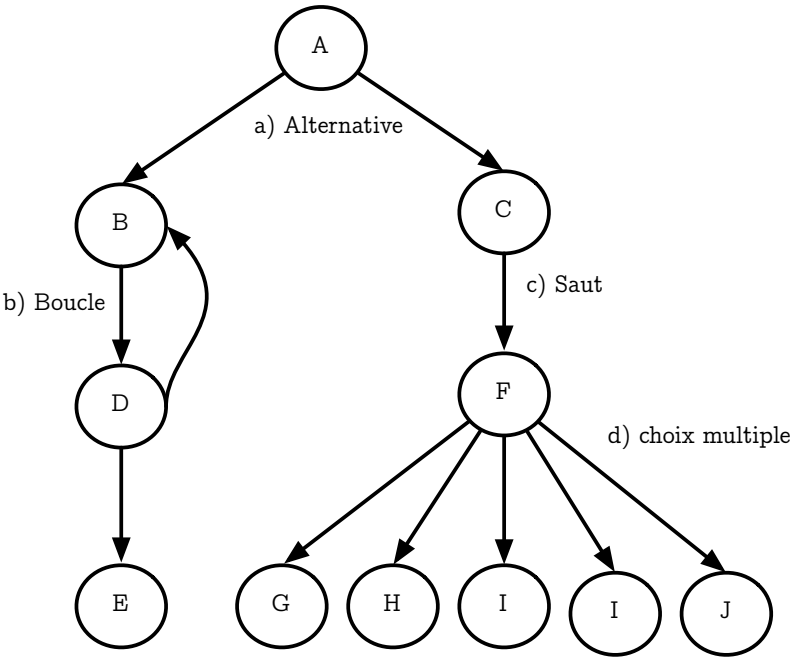

Fig. 4.2 – Un graphe de flots de contrôle

4.3.4 Tissage de code

Le tissage de code correspond à l'insertion automatique de fragment de code dans un code déjà existant.

4.4 Intégrité du flot de contrôle

Une des conséquences de l'attaque par injection de fautes va être la modification du flot de contrôle de l'application. Avec comme intérêt d'éviter les tests logiques comme la vérification du code PIN ou la vérification des clefs cryptographiques pouvant être contenues dans la carte.

4.4.1 Exemple d'attaque en faute

Dans l'exemple de la Fig. 4.3 qui reprend la méthode utilisée en section 4.2, nous constatons qu'il est possible grâce à une injection de fautes de réaliser un évitement du test du code PIN. Cette

attaque peut être transposée au cas de la vérification d'une clef cryptographique. En changeant l'instruction *ifeq* qui correspond à l'octet 0x60, en une instruction *nop* qui correspond à l'octet 0x00, le résultat obtenu est celui de la Fig. 4.4.

Bytecode	Octets	Source Java
00 : aload_0	00 : 18	private void debit (APDU apdu) {
01 : getfield #4	01 : 83 00 04	
04 : invokevirtual #18	04 : 8B 00 23	
07 : ifeq 59	07 : 60 00 3B	if (pin.isValidated()) {
10 : ...	10 : ...	// make the debit operation
...	...	} else {
59 : sipush 25345	59 : 13 63 01	ISOException.throwIt (
63 : invokestatic #13	63 : 8D 00 0D	SW_PIN_VERIFICATION_REQUIRED);
66 : return	66 : 7A	}

Fig. 4.3 – Représentation de la méthode avant l'attaque

Bytecode	Octets	Source Java
00 : aload_0	00 : 18	private void debit (APDU apdu) {
01 : getfield #4	01 : 83 00 04	
04 : invokevirtual #18	04 : 8B 00 23	
07 : nop	07 : 00	~~if (pin.isValidated())~~ {
08 : nop	08 : 00	
09 : pop	09 : 3B	
10 : ...	10 : ...	// make the debit operation
...	...	} else {
59 : sipush 25345	59 : 13 63 01	ISOException.throwIt (
63 : invokestatic #13	63 : 8D 00 0D	SW_PIN_VERIFICATION_REQUIRED);
66 : return	66 : 7A	}

Fig. 4.4 – Représentation de la méthode après l'attaque

Nous constatons alors après l'attaque que la partie haute de l'adresse qui correspond à l'instruction 0x00 (*nop*) est alors interprétée comme étant une instruction et plus comme étant un opérande ; il en est de même pour la partie basse de l'adresse 0x3B qui va être interprétée comme l'instruction *pop* et qui va quant à elle remettre la pile des opérandes dans le même état que s'il n'y avait pas eu d'attaque. Ainsi, la machine virtuelle continuera d'interpréter le code comme s'il ne s'était rien passé. Nous constatons donc que la valeur du code PIN entrée par l'attaquant importe peu car elle n'est jamais vérifiée ; il est donc possible d'effectuer l'opération de débit et même de lever une exception qui intervient trop tard puisque la carte a déjà retourné le résultat erroné ; et cela uniquement grâce à une attaque par injection de fautes réussie et correctement synchronisée avec l'exécution d'une instruction précise.

4.4.2 Méthode AGL

La première méthode de protection exposée est une méthode permettant de s'assurer de l'intégrité du flux de contrôle lors de l'exécution d'une application. Cette méthode a été proposée en

[AGL03], je l'appellerai «AGL» (du nom des auteurs Akkar, Goubin et Ly) pour les besoins de l'exposer, car les auteurs ne lui ont pas donné de nom. L'objectif de cette protection est de vérifier que l'exécution du programme se déroule bien selon un chemin autorisé par le code. Ce chemin aura été au préalable calculé et sauvegardé avant l'exécution.

C'est une protection exclusivement applicative parce qu'elle se base uniquement sur la modification du code afin de le rendre résistant aux attaques par injection de fautes. Elle utilise plusieurs notions comme le *tissage de code* qui consiste à réécrire des fragments du programme, des *étiquettes* ou *tags* ou *marqueurs* qui correspondent à un marquage de certaines parties du code jugées sensibles, un *préprocesseur* qui va traiter les étiquettes afin de fournir un nouveau code contenant les mécanismes de sécurité.

Principe de la méthode

Sur le principe, cette méthode repose sur le calcul d'un historique de chemins que le système peut emprunter durant l'exécution du code. En effet, le développeur (programmeur) de l'application va marquer les points du code par lesquels, il faut que le système passe pour arriver à une ressource donnée, pour ce faire, il devra utiliser des tags ; ces tags peuvent être de différentes natures : les *starting points*, les *flags*, les *check points*, les *race declarations*. Une fois le code marqué, il est analysé par le préprocesseur, qui transforme ces étiquettes en code pour le système de détection (voir Fig. 4.5).

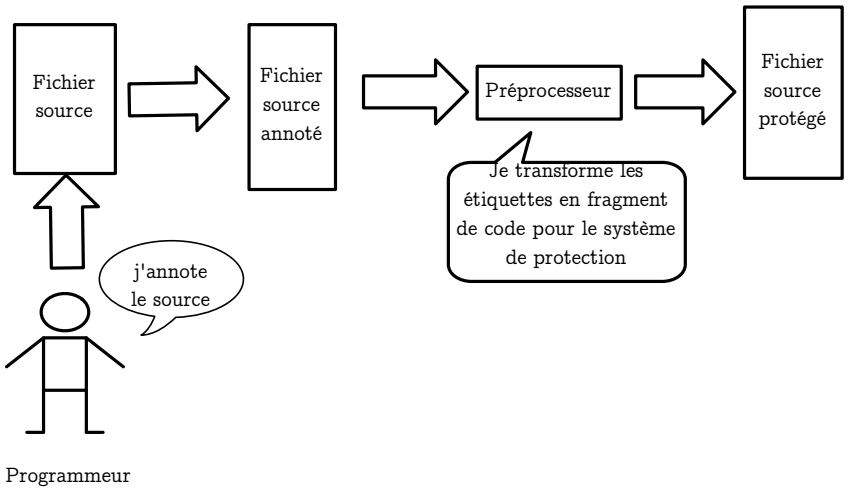

Fig. 4.5 – Fonctionnement de la méthode AGL

Ce système d'étiquettes se trouve au centre de la méthode de protection, nous allons donc voir la signification de ces différentes étiquettes ainsi que les tags correspondant :

- L'étiquette *starting point* désigne le moment de l'interprétation du code où le système de détection doit commencer à enregistrer l'historique. Elle sera remplacée par un fragment de code permettant la réinitialisation des structures de données nécessaires au système de détection
- L'étiquette *flag* désigne un point de passage obligatoire c.-à-d. un point du code par lequel l'interpréteur doit passer. Une telle étiquette sera remplacée par un fragment de code faisant la mise à jour des structures de données du système de détection.

– L'étiquette *check-point* désigne un endroit du code où le système de détection doit effectuer la vérification entre l'historique de chemin et le chemin réellement emprunté lors de l'exécution du code. Cette étiquette sera remplacée par un fragment de code permettant d'effectuer les différentes vérifications qui s'imposent.
– L'étiquette *race declaration* désigne une famille d'exécution, elle est accompagnée d'une condition permettant de déterminer si l'exécution de cette famille doit être autorisée ou non. Si l'exécution d'une famille est autorisée, alors elle est représentée grâce aux étiquettes de type *flag*. Pour chacune de ces étiquettes le programmeur doit préciser si l'exécution doit ou ne doit pas passer par elle ou si seule une famille d'exécution donnée peut le faire. Cette étiquette sera remplacée par un fragment de code permettant d'activer ou de désactiver la famille d'exécution concernée.

Bilan de la méthode

Selon les auteurs, la méthode de protection AGL a l'avantage du taux de couverture. En effet, durant leur expérimentation 80% des attaques par injection de fautes réalisées et pouvant mener à un résultat erroné ont été détectées. Par contre, comme cette méthode est purement applicative, cela signifie que l'intégralité du système de détection (structure de données, ainsi que code de détection) se retrouve inséré dans le code de l'application, ce qui augmente de manière considérable l'occupation mémoire de l'application. De plus, cette méthode s'applique directement au code interprété (bytecode ou code source Java), elle est donc plus lente à exécuter qu'une méthode de protection qui agirait en natif (côté système). De plus, du fait que le code soit totalement à la merci de l'attaquant, le système de protection peut par la même occasion être attaqué : il serait possible d'éviter le test logique en remplaçant les étiquettes de type *check-points*.

En conclusion, on se retrouve avec une méthode qui consomme beaucoup d'espace en terme de mémoire non volatile (EEPROM), volatile (RAM) et qui nécessite un processeur relativement puissant. Or nous savons que ce sont des ressources rares et coûteuses quand il s'agit de la carte à puce. Ce qui fait qu'une telle protection ne remplit pas forcément les standards en terme de consommation de ressource mémoire et processeur, lui permettant d'être utilisée dans des conditions réelles.

4.4.3 La méthode CFI (Control flow integrity)

Les auteurs de [AEL05] offrent un moyen de détecter les détournements du flot de contrôle pouvant avoir lieu lors de l'exécution d'une application. Pour ce faire, ils vont également utiliser un système d'étiquettes, et de tissage de code, afin d'arriver à faire de la détection de sauts non autorisés lors de la rupture du flot de contrôle. Sur le principe cette technique de protection va se baser sur le graphe de flots de contrôle de l'application, afin de s'assurer que les transferts de flots de contrôle se font bien selon ce qui est prévu dans ce graphe. Ainsi, si une attaque par injection de fautes parvient à perturber les instructions permettant d'obtenir un transfert du flot de contrôle vers un autre endroit du code que celui prévu initialement, ce mécanisme de protection le décèle.

Principe de la méthode

Elle va consister 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 contremesure proposent l'utilisation de deux directives principales : une directive ID et une directive ID-checks.

- La directive ID consiste en un label qui va correspondre à l'adresse de l'instruction vers laquelle va se faire le saut. Ainsi, chaque instruction qui correspond à l'adresse d'un saut va se voir affecter un ID qui doit être unique.
- La directive ID-Checks consiste en un label qui va être transformé en un code de vérification des IDs précédemment calculées. En effet, chaque instruction de transfert de flot de contrôle va être étiquetée grâce à ce type de directive. Ainsi, le code qui va remplacer ces directives va vérifier pendant le transfert de flot de contrôle que l'ID correspondant au transfert est bien un de ceux qui ont été enregistrés pour cette ID-Checks. Si ce n'est pas le cas, c'est qu'il y a probablement eu une perturbation du flot de contrôle donc l'exécution du code peut s'arrêter.

Pour que cette méthode fonctionne, l'auteur pose un certain nombre d'hypothèses :

1. l'ID doit être unique.
2. Le programme ne doit pas être capable de modifier le code en mémoire durant l'exécution afin d'éviter qu'un attaquant ne puisse contourner le mécanisme de vérification en écrasant un ID-check
3. Il ne doit pas être possible au programme d'exécuter des données comme s'il s'agissait d'instructions. Autrement un attaquant pourrait faire en sorte de forcer l'exécution d'un code qui aurait exactement le même ID que celui attendu.

Bilan de la méthode

Le problème majeur que pose cette méthode est le fait que dans notre cas, si une attaque est réussie, il est potentiellement possible de modifier le code durant son exécution, et même de faire en sorte qu'une donnée soit interprétée comme étant du code. Ce qui ne permet pas de garantir les deux dernières hypothèses proposées par les auteurs de cette contremesure. Cette dernière est donc dans certains cas inefficace face aux attaques en faute que l'on souhaite détecter. De plus, c'est une contremesure purement applicative, elle pose donc les problèmes de ressources mémoires et processeurs déjà évoqués pour ce type de protection.

4.5 Intégrité du code

De façon générale, les mécanismes de vérification d'intégrité contenus dans la carte reposent sur des valeurs de contrôle. J'aborde dans cette section la méthode qu'on retrouve en [PS06]. Cette contremesure se charge de vérifier l'intégrité du flux de contrôle au cours de l'exécution de l'application.

4.5.1 Principe de la méthode

Le principe de la méthode revient à subdiviser le code en blocs élémentaires (voir la sous-section 4.3.2). Une fois tous les blocs obtenus, on calcule une valeur de contrôle pour chacun des

blocs qu'on sauvegarde d'une façon ou d'une autre puis elle sera chargée avec le code dans la carte. Les valeurs de contrôle sont calculées grâce à des fonctions de hachage comme le MD-5 [Riv92] ou SHA-1 [EJ01]. Une fois dans la carte, la machine virtuelle lors de l'interprétation du code va se charger de recalculer ces valeurs de contrôle et de les comparer avec celles déjà sauvegardées hors de la carte, si cette comparaison échoue alors le code a probablement été modifié. Cette méthode est subdivisée en deux parties, l'une se passant sur le serveur (PC) et l'autre au sein même de la carte. La partie du mécanisme de protection se trouvant hors de la carte a pour objectif d'effectuer un calcul des blocs élémentaires ainsi que leur valeur de contrôle.

Hors de la carte

Sur l'ordinateur qui sert à développer l'application, et sur une méthode donnée, il faut dans un premier temps calculer tous les blocs élémentaires qui la composent. Une fois obtenu, il faut calculer une valeur de contrôle correspondant à chacun des blocs trouvés. Pour se faire, les auteurs utilisent une fonction de hachage sur les éléments constituant le bloc. Les fonctions de hachage retenues sont le MD-5 ou le SHA-1, qui produisent respectivement des valeurs de contrôle de taille 128 bits (16 octets) et 160 bits (20 octets) quelle que soit la taille du bloc de données sur laquelle on l'applique. Une fois ces valeurs de contrôle calculées, elles sont sauvegardées avec l'application et chargées en même temps que celle-ci. Le diagramme de la Fig. 4.6 résume parfaitement l'étape qui se fait hors de la carte.

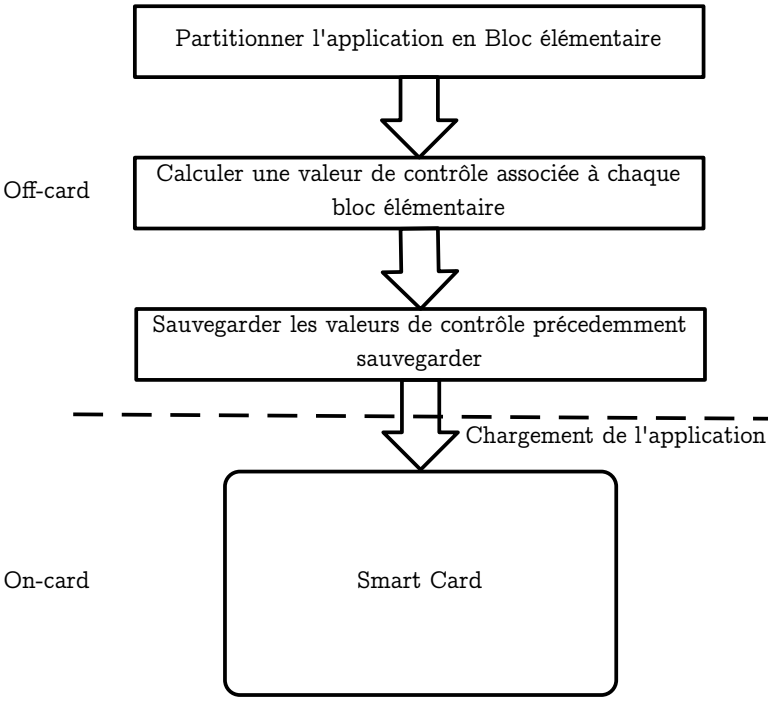

Fig. 4.6 – Diagramme de calcul des blocs élémentaires hors de la carte

Dans la carte

Dans la carte l'interpréteur de la machine virtuelle doit être modifié afin de pouvoir calculer et comparer les valeurs de contrôle pendant qu'il interprète le code. Le pseudocode de l'Algorithm 1 explique les étapes qui sont suivies par l'interpréteur pour faire la détection. Le premier bloc élémentaire de l'application est récupéré pour former le bloc courant. Puis tant qu'il reste des blocs à exécuter, la valeur de contrôle du bloc courant est récupérée. Puis celle du bloc en cours d'interprétation est calculée. L'interpréteur va ensuite vérifier si les deux valeurs de contrôle sont différentes. Si elles le sont alors l'interpréteur pourra traiter une erreur, sinon le bloc suivant devient le bloc courant et l'opération de vérification peut recommencer pour celui-ci.

Algorithm 1 Algorithme de vérification de block élémentaires

```
1: currentBB ← firstBasicBlock ()
2: while currentBB ≠ NULL do
3:    temp ← computeCheckValue (curentBB)
4:    currentBB.setCheckValue(temp)
5:    temporaryBB ← getCorrespondingSavedBasicBlock (currentBB)
6:    if currentBB.getCheckValue() ≠ temporaryBB.getCheckValue() then
7:       throwError ()
8:       break
9:    end if
10:   currentBB ← getNextBasicBlock ()
11: end while
```

4.5.2 Bilan de la méthode

L'avantage de cette méthode est son taux de couverture, car elle permet de détecter n'importe quel changement intervenant sur les éléments qui rentrent en compte pour obtenir chacune des valeurs de contrôle. Par contre, elle a deux inconvénients majeurs au vu de la plateforme ciblée (la carte à puce). En effet, quelle que soit la taille du bloc à signer, on obtient une valeur de contrôle constante (16 octets ou 20 octets). Celle-ci peut dans certains cas être supérieure à la taille du bloc. C'est le cas des blocs contenant une seule ou plusieurs instructions de taille inférieure à 16 octets ou 20 octets en fonction de la fonction de hachage choisie. Dans l'exemple de la fonction de débit, nous constatons que la majorité des blocs sont de tailles inférieures à 20 octets, ce qui entraine une perte au niveau de l'espace de stockage utilisé par l'application. Il faut aussi considérer les plateformes ne disposant pas de coprocesseur cryptographique, en effet sur celles-ci exécuter les opérations nécessaires au calcul d'un MD5 ou d'un SHA-1 serait couteux en terme de temps d'exécution. On peut l'affirmer, car pour effectuer un MD5 (voir [Riv92]), il est nécessaire d'effectuer au moins 496 opérations pour avoir un hash et pour effectuer un SHA-1 il faut effectuer beaucoup plus d'opérations, car il y a beaucoup plus d'étapes pour l'effectuer (voir [EJ01]). Cela constitue un surplus considérable de traitement à faire par le processeur. Pour ces raisons, j'estime que cette méthode est trop coûteuse aussi bien en terme d'occupation mémoire qu'en terme de ralentissement de l'exécution du code.

Deuxième partie

Contributions et évaluations

Chapitre 5

Contributions

Sommaire

5.1 Introduction

Les contremesures proposées dans le chapitre précédent ne respectaient pas les exigences en terme de processeur et de mémoire qui sont imposées par la carte à puce. De plus, on retrouve aisément dans la littérature des contremesures pour lutter contre les attaques par injection de fautes afin de protéger les algorithmes cryptographiques (voir [BDL97 ; Aum+03 ; KKS99 ; BOS03]). Il en est autrement pour les techniques plus généralistes comme celle que j'ai mis au point et que j'expose dans ce chapitre. Ces techniques ont également toutes fait l'objet de publication que l'on peut retrouver aux références [Ser+09 ; SICL09a ; SICL09b].

De façon générale, l'approche (voir la Fig. 5.1) que j'ai choisi est une approche indépendante du type de traitement effectué par la puce : on insère des informations dans le code de l'application à l'aide d'annotations de sécurité qui vont servir à déterminer quelles méthodes de l'application sont à protéger, ainsi que le type de sécurité utilisé pour y parvenir. Un module traite ces annotations afin de les transformer en informations pouvant être différentes en fonction du type de sécurité demandé. Elles sont utilisées par l'interpréteur de la machine virtuelle qui passe dans un mode appelé «*mode sécurisé*» pour interpréter la méthode courante.

Les informations sont ajoutées au fichier binaire sous forme de composants additionnels. En effet, la spécification de Java Card permet d'ajouter des composants additionnels aussi bien au

niveau de la classe qu'au niveau de la méthode. Dans la spécification (voir [Myc09f]), un composant additionnel a la forme que l'on retrouve dans le Code 6.

Code 6 composant additionnel

```
custom_component_info {
        u2 attribute_name_index;
        u4 attribute_length;
        u1 info[attribute_length];
}
```

Dans cette notation du type structure de données du langage C, u1 équivaut à un octet, u2 à deux octets, u3 à trois octets, u4 à quatre octets, etc. Avec *attribute_name_index* qui désigne une valeur sur deux octets dans le *constant pool* de la classe ; *attribute_length* correspond à la taille de la zone *info* du composant ; la zone *info* correspond à la donnée essentielle du composant. Ce nouveau composant ainsi créé est ajouté au niveau de la méthode à protéger. Il est précisé dans la spécification de la machine virtuelle que lorsqu'elle rencontre un composant non standard (c.-à-d. non prédéfini dans la spécification) elle doit l'accepter et l'ignorer de façon silencieuse dans le cas où elle ne le reconnait pas. j'ai modifié une machine virtuelle, afin qu'elle puisse faire bon usage des informations introduites dans le fichier binaire (voir chapitre 6). Ce travail est réalisé à l'aide d'une API de manipulation de classfile telle que ASM Asssembly ou BCEL (Byte Code Engeneering Language), notre choix s'est porté sur BCEL, pour une question de simplicité d'utilisation et de documentation.

Cette approche est à la base de chacune des méthodes de protection utilisées, elle est essentielle en ce qui concerne la sécurité et les performances :

- **Pour la sécurité** : le modèle de fautes choisi impose qu'une attaque synchronisée et simultanée soit effectuée à la fois dans la zone de stockage (soit la mémoire EEPROM, soit la mémoire flash) ainsi que dans la ROM de la carte. En effet, dans l'approche que J'utilise, le système de détection est divisé en deux parties. Une partie dans l'applicatif (mémoire de stockage) de la carte et l'autre partie dans la machine virtuelle (ROM). La ROM est par définition une zone mémoire non modifiable qui bénéficie de surcroit de mécanismes de vérification d'intégrité qui permettent de détecter les altérations dont elle pourrait faire l'objet. Ainsi, tout changement dans cette zone de la mémoire pourra être détecté. Même si l'attaquant parvenait à modifier la ROM, il faudrait que l'attaque modifie aux mêmes moments la zone EEPROM correspondante, ce qui est hors de portée de l'attaquant contre lequel on veut se défendre (voir la section 3.5).
- **Pour les performances** : chacune des ressources critiques de la plateforme est préservée :
  - *Mémoire* : Toutes les informations nécessaires à la détection se trouvent au niveau de l'application. De ce fait, il faut juste choisir le format de l'information à sauvegarder de façon minimale pour optimiser l'augmentation de l'espace nécessaire au stockage d'une application.
  - *Processeur* : Le code fonctionnel c.-à-d le code correspondant à la détection des modifications du code est du code natif. Il est directement exécuté par la plateforme sans interprétation préalable ce qui permet d'améliorer la vitesse d'exécution du processus de détection.

Dans ce chapitre, je vais uniquement parler des protections logicielles qui servent à lutter contre les attaques par injection de fautes qui suivent le modèle de fautes choisi (section 3.5). je n'aborde

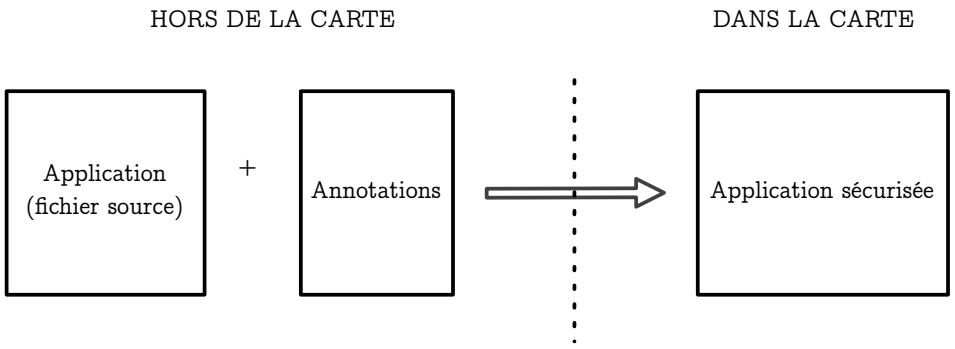

Fig. 5.1 – Schéma de l'approche utilisée

pas la sécurité des données car celles-ci sont protégées à l'aide de mécanismes classiques de vérifications d'intégrité tels que des techniques de signature, de la redondance de données, etc. Il s'agit donc de protéger l'intégrité du code et de protéger le flux de contrôle contre les modifications dues à ces attaques pouvant intervenir lors de l'exécution des applications. Dans un premier temps, j'explique comment les techniques mises au point permettent de détecter les changements intervenants sur la zone mémoire qui contient les octets correspondant aux instructions de l'application. Ensuite, je m'intéresse à une méthode permettant de protéger le pointeur de code de la machine virtuelle lors de l'exécution d'une application.

5.2 Intégrité du code des applications

5.2.1 Méthode du champ de bits

Problématique

Cette contremesure a pour objectif de déceler le décalage pouvant se produire lorsqu'une instruction est remplacée par une autre instruction. Dans ce cas précis, trois situations peuvent se poser :

1. L'instruction courante est remplacée par une instruction avec un nombre inférieur d'opérandes. Cela se produit par exemple lorsque l'instruction *invokevirtual* qui a deux opérandes (et qui permet d'invoquer l'instance d'une méthode) est remplacée par l'instruction *ifeq* qui dispose d'un seul opérande qui correspond à l'adresse du saut à effectuer (qui permet de récupérer la valeur qui est en haut de la pile et qui la compare avec zéro). Dans ce cas précis, l'instruction *ifeq* étant plus petite (en terme de nombre d'octets), elle est interprétée comme une instruction et le premier paramètre de l'instruction invokevirtual est interprété comme paramètre de l'instruction *ifeq*. Le deuxième paramètre par contre est interprété comme étant une instruction, entrainant par la même une augmentation du nombre d'instructions qui composent la méthode.
2. L'instruction courante est remplacée par une instruction ayant un nombre supérieur d'opérandes ; cela se produit par exemple dans le cas où l'instruction *iload_0* qui a zéro opérande (et qui charge la première variable locale qui est un entier) se retrouve remplacée par l'instruction

getfield qui a un seul opérande (et qui récupère un champ d'un objet donné en paramètre). Dans ce cas de figure, on a une diminution du nombre d'instructions, car l'instruction *getfield* utilise l'octet qui correspond à l'instruction suivante comme étant son opérande.

3. L'instruction courante est remplacée par une instruction ayant le même nombre d'opérandes ; cela se produit lorsque l'instruction *iadd* qui n'a pas d'opérandes (instruction qui prend les deux valeurs au dessus de la pile qui les additionne et qui remet le résultat sur la pile des opérandes) est remplacée par l'instruction *isub* qui n'a pas non plus d'opérande (instruction qui récupère les deux valeurs qui sont au-dessus de la pile, qui les soustrait, et qui retourne la valeur entière qui résulte de cette soustraction). Dans cet exemple, aucune détection de la modification n'est faite, car l'instruction est remplacée par une instruction identique (ou équivalente).

Principe de la méthode

Le but recherché ici est de faire en sorte que les décalages que l'on retrouve en 1, et 2 (voir la sous-section 5.2.1) soient détectés par la machine virtuelle lorsqu'elle exécute le code. Pour parvenir à ce résultat, on procéde en plusieurs étapes : une étape hors de la carte et une autre à l'intérieur de la carte.

Hors de la carte : La première étape est de marquer la méthode à protéger avec l'annotation que l'on retrouve dans le Code 7. Grâce à cela, la méthode est marquée comme devant être protégée en intégrité par la méthode du champ de bits.

Code 7 Operation de débit provenant d'une applet de porte-monnaie électronique

```
@SensitiveType{
        sensitivity=SensitiveValue.INTEGRITY
        proprietaryValue=''FieldOfBit''
}
private void debit(APDU apdu) {

        // access authentication
    if ( pin.isValidated() ) {
                ...
                balance = (short) (balance - debitAmount);

    } else {

                ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);

    }
} // end of debit method
```

Le code ainsi produit est ensuite analysé par un outil que j'ai conçu. Cet outil a pour but de donner une signification à chaque octet interprété par la machine virtuelle.

Une méthode est représentée par une suite d'octets qui représente pour chacun un code d'opération (opcode) ou un opérande. Cette propriété est utilisée afin de générer un marquage du code qui correspond à la signification de l'octet correspondant (opcode ou opérande). L'octet correspondant à un opcode est marqué comme étant exécutable ; celui correspondant à un opérande comme étant un octet en lecture. On obtient ainsi pour chaque méthode protégée l'information qui permet de faire correspondre à chaque octet le type de comportement attaché.

On utilise une convention afin de coder l'information obtenue, cette convention doit être choisie de sorte qu'on puisse gagner de l'espace de stockage. La convention est que tout opcode est marqué par un bit de valeur 1 et tout opérande par un bit de valeur 0 ou inversement ; d'où le nom de la méthode de protection utilisée.

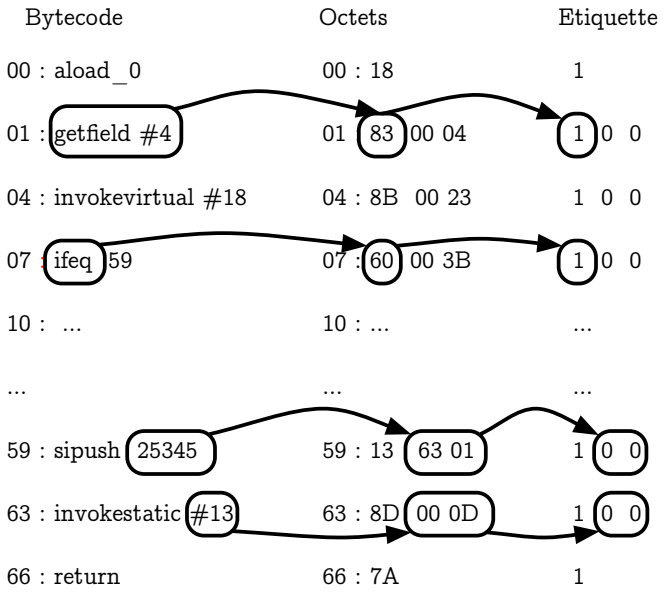

Fig. 5.2 – Exemple d'utilisation de la convention de marquage

L'exemple de la Fig. 5.2 a été obtenu en appliquant la convention précédemment énoncée et nous voyons que l'instruction aload_0 correspondant à l'octet 0x18 est marquée avec le bit 1 et l'instruction ifeq correspondant à l'octet 0x60 est marquée par le bit 1. Par contre, les octets 0x00 (partie haute) et 0x3B (partie basse) correspondant à l'adresse du saut vont être marqués par deux bits 0. le même principe s'applique à toute la méthode. Une fois l'opération de marquage effectuée, on obtient un tableau de bits contenant pour chaque octet de la méthode sa signification. Ce tableau est sauvegardé dans le fichier binaire de l'application sous forme de composants additionnels (voir la section 5.1).

Code 8 composant additionnel champ de bit

```
field_of_bits{
        u1 attribute_name_index;
        u4 attribute_length
        u1 field_of_bit_info[attribute_length]
}
```

La signification des deux premiers éléments de cette structure se trouve à la section 5.1 ; l'élément field_of_bit_info contient sous forme d'octets les bits qui composent le champ de bits de la méthode concernée. On effectue une conversion du tableau de bits obtenu vers un tableau d'octets en suivant le Code 9, cette fonction prend en entrée un tableau t d'octets contenant des 1 et des 0 à chaque indice et retourne un autre tableau d'octets t' contenant à chacun de ses indices la compression du tableau t de sorte que la taille du tableau t soit divisée par 8. Ainsi pour l'exemple

de la Fig. 5.2, on a en entrée le tableau formé des éléments $\{1, 1, 0, 0, 1, 0, 0, 1, ...0, 1, 0, 0, 1, 0, 0, 1\}$, et en sortie le tableau avec les éléments $\{197, ..., 73\}$.

Code 9 Conversion tableaux d'octets vers champ de bits en Java

```
public static ArrayList<Integer> coding (int[] tabToCode) {
        int counterOfByte = 8;
        int result = 0x00;
        int i = 0;
        ArrayList<Integer> tabResult = new ArrayList<Integer>();

        for (; i < tabToCode.length; i++) {
                if (tabToCode[i] == 1) {
                        result = ( result | (0x80 >> (i % SIZE_OF_UNIT))) & 0xFF;
                        counterOfByte--;
                } else {
                        counterOfByte--;
                }


                if ((counterOfByte == 0)) {
                        tabResult.add( result );
                        counterOfByte = 8;
                        result = 0;
                }
        }

        if ( ((int) i / (int) SIZE_OF_UNIT == (int) tabToCode.length / (int)
            SIZE_OF_UNIT)
                && counterOfByte < SIZE_OF_UNIT) {
                        tabResult.add( result );
        }

        return (tabResult);
}
```

Dans la carte : Le but est de vérifier, lorsque le PC se déplace, que l'octet interprété correspond à un opérande ou a un opcode. Pour parvenir à cet objectif, il faut modifier la machine virtuelle afin qu'elle utilise le composant additionnel créé. Dans chaque méthode protégée figure un composant supplémentaire. La première vérification que la machine virtuelle effectue est de vérifier que ce composant existe. Si c'est le cas alors, cette méthode est à protéger, sinon elle sera traitée de façon normale. Ensuite, l'interpréteur de la machine virtuelle vérifie que la méthode à exécuter est bien la méthode de champ de bit, car plusieurs mécanismes de protection sont proposés : si c'est le cas, alors la machine virtuelle passe en mode sécurisé avec la contremesure champ de bits, sinon elle fera un traitement normal de la méthode. Enfin, elle peut lancer la boucle principale d'interprétation du code dans laquelle elle vérifie au fur et à mesure l'octet courant :

1. Si l'octet courant est un opcode et si le bit correspondant au PC courant dans le tableau d'octet est égal à 1, l'interpréteur peut passer à l'octet suivant qui devient l'octet courant. Sinon, le code a été modifié et l'interpréteur peut s'arrêter en levant une exception au niveau de la plateforme.
2. Si l'octet courant est un opérande et si le bit correspondant au PC courant dans le tableau d'octet est égal à 0, l'interpréteur peut passer à l'octet suivant qui devient l'octet courant. Sinon, le code a été modifié et l'interpréteur peut s'arrêter en levant une exception au niveau de la plateforme.

3. Ces deux opérations de vérification sont répétées pour chaque octet jusqu'à ce que la méthode se termine soit par une levée d'exception soit par une instruction de type *return*.

Discussion sur le taux de détection

La technique du champ de bits permet de détecter les cas 1 et 2, par contre, elle est impuissante par rapport à une attaque permettant d'effectuer le remplacement d'une instruction par une autre instruction ayant le même nombre d'opérandes et le même effet sur la pile. Le «même effet sur la pile» signifie que l'instruction retire le même nombre d'éléments sur la pile, et dépose le même nombre d'éléments sur la pile. De plus, le type des éléments sur la pile avant et après l'instruction est respecté. Dans un souci de simplification, j'appellerai ces instructions des «*instructions indistinctes*». Dans ce cas précis, la machine virtuelle ne détecte pas de modifications et le mécanisme de détection non plus. L'explication est que le mécanisme de détection développé permet de faire la détection uniquement s'il y a un changement de la taille (en nombre d'instructions) de la méthode. Pour ce qui est de la machine virtuelle, l'instruction a exactement le même effet que l'instruction qu'elle remplace. Il faut donc déterminer quelles sont les instructions indistinctes de l'ensemble du jeu d'instructions Java Card.

L'analyse des opcodes Java Card permet de distinguer les opcodes indistincts. je vais classer ceux-ci en groupes que je vais appeler "*groupes indistincts*". Un groupe indistinct se compose d'instructions indistinctes du même type c.-à-d. d'instructions ayant la même taille et le même effet sur la pile. Si on se base uniquement sur le nombre d'opérandes et sur la consommation de mots sur la pile des opérandes, on obtient les 6 groupes du Code 10.

Code 10 Les groupes indistincts

```
< nop, return, breakpoint >

< aconst_null, iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3,iconst_4, iconst_5
    , iload_0, iload_1, iload_2, iload_3, aload_0, aload_1, aload_2, aload_3 >

< iaload, aaload, baload, saload, iadd, isub,imul, idiv, irem, ishl, ishr, iushr,
    iand, ior, ixor >

< istore, istore_0, istore_1, istore_2, istore_3, astore_0, astore_1, astore_2,
    astore_3, pop, ireturn, areturn >

< iastore, aastore, sastore >

< ineg, i2s, arraylength, athrow >
```

Maintenant que les groupes sont déterminés et qu'on dispose de leur taille, on détermine la probabilité qu'un opcode appartenant à un groupe soit remplacé par un autre opcode du même groupe (sachant qu'il peut être remplacé par lui même). Pour pouvoir parvenir au résultat, on détermine par statistiques le nombre de fois qu'un élément appartenant à un groupe donné apparait dans les applications. Cela permet de calculer la probabilité d'apparition d'un groupe donné (voir l'équation 5.1).

$$P(A)_i = \frac{nombreApparitionGroupe_i}{nombreInstructionsVérifiées} \tag{5.1}$$

Il y a équiprobabilité si le phénomène étudié est le nombre de chances qu'une instruction soit remplacée par une autre instruction ayant un sens dans le jeu d'instructions Java Card. Etant donné

que le nombre d'instructions disponible en Java Card est de 187, alors on obtient la probabilité $P_B$de l'équation 5.2.

$$P_B = \frac{1}{nombreInstructionsJavaCard} = \frac{1}{187} \tag{5.2}$$

Maintenant, pour obtenir la probabilité P_i qu'une instruction appartenant à un groupe indistinct soit remplacée par une autre instruction du même groupe indistinct, il faut pondérer le résultat précédent par la taille du groupe, ce qui donne l'opération de l'équation 5.3.

$$P_i = tailleGroupe_i \times \frac{1}{187} \tag{5.3}$$

On peut maintenant obtenir la probabilité P_C de remplacer une instruction indistincte par une autre instruction indistincte grâce à l'équation 5.4.

$$P_C = \sum_{i=0}^{n} P_i \tag{5.4}$$

Afin de pouvoir se rendre compte de ce qui se passe dans la réalité, on détermine le pourcentage d'apparitions des instructions indistinctes dans le code d'un échantillon d'application Java Card. Le nombre d'applications Java Card que j'ai étudiées se porte à une quinzaine, avec des champs d'action divers tels que la banque, les communications réseau, le calcul arithmétique, la signature numérique, etc. Afin d'avoir le taux de réussite d'une attaque qui ne sera pas détectée par le mécanisme de protection on se reporte à l'équation 5.5. Si on applique cette méthode sur un échantillon de 21 fichiers classes appartenant à des applications Java Card de type applet et contenant 8304 instructions, on obtient un taux de réussite d'attaque d'environ 4%.

$$P = \sum_{i=0}^{n} P(A)_i \times P_i \tag{5.5}$$

On peut encore réduire ces groupes en fonction du type des éléments sur la pile avant et après les instructions. On obtient alors les 8 groupes du Code 11, ce qui donne un taux de réussite d'attaque d'environ 3% avec les mêmes paramètres que précédemment. Ces résultats montrent que cette protection n'est pas infaillible, mais aussi qu'elle reste efficace, car les chances pour qu'une instruction soit remplacée par une autre instruction indistincte, sont très faibles. De plus, elle offre un moyen de détection immédiat de l'injection d'une faute.

5.2.2 Méthode de l'intégrité des blocs élémentaires

La méthode de protection étudiée précédemment ne permet pas de détecter tous les changements pouvant survenir lors d'une attaque en faute qui suit notre modèle. Partant de ce constat, j'ai décidé de créer une nouvelle technique de protection qui permettrait de détecter n'importe quel changement d'octets dans la zone qui correspond à la représentation de la méthode. Cette contremesure se base sur la subdivision du code en blocs élémentaires et sur le calcul de valeurs de contrôle pour chacun de ces blocs puis sur la vérification de ces informations au sein de la carte.

Code 11 Les groupes indistincts 2

```
< nop, return, breakpoint >
< iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3,iconst_4, iconst_5>
< iload_0, iload_1, iload_2, iload_3>
< aload_0, aload_1, aload_2, aload_3 >
< aaload, baload, saload >
< iadd, isub,imul, idiv, irem, ishl, ishr, iushr, iand, ior, ixor >
< istore_0, istore_1, istore_2, istore_3, astore_0, astore_1, astore_2, astore_3 >
< aastore, sastore >
```

Principe de la méthode

On calcule chaque bloc élémentaire qui compose une méthode à protéger. Un élément peut jouer sur la façon dont sont calculés les blocs élémentaires. En effet, la recherche des débuts et des fins de blocs, que j'appellerai des «*leaders*», dépend du fait qu'on veuille ou non prendre en considération les instructions qui sont capables de lever des exceptions. Dans un premier temps, il faut déterminer les leaders d'une méthode que l'on souhaite protéger avec cette contremesure, je traite les deux cas.

Cas des instructions qui ne lèvent pas d'exceptions : Afin de déterminer l'ensemble des leaders d'une méthode, on s'appuie sur les règles suivantes :

- La première instruction de la méthode est un leader.
- Chaque instruction qui correspond à la cible d'une instruction de branchement inconditionnel, tels que les cibles des bytecodes *goto*, *goto_w*, *jsr*, *jsr_w*, *ret*, est un leader.
- Chaque instruction qui est la cible d'une instruction de branchement conditionnel, tel que *ifeq*, *iflt*, *ifle*, *ifgt*, *ifge*, *ifnull*, *ifnonnull*, *if_icmpgt*, *if_cmple*,*if_cmpge*, *if_acmpeq*, *if_acmpne*, *lcmp*, *fcmpl*, *fcmpg*, *dcmpl*, *dcmpg* est un leader.
- Chaque instruction qui est une cible d'une instruction de branchement conditionnel multiple tel que *tableswitch* ou *lookupswitch* est un leader.
- Chaque instruction qui suit immédiatement une instruction de saut conditionnel ou inconditionnel ou une instruction de type *<T>return* (*ireturn*, *lreturn*, *freturn*, *dreturn*, *areturn*, et *return*), ou une instruction de saut conditionnel multiple est un leader.

Maintenant, chaque leader individuel donne naissance à un nouveau bloc élémentaire correspondant à toutes les instructions jusqu'au prochain bloc ou jusqu'à la fin du bytecode. En outre, les instructions d'invocation de méthode telles que *invokevirtual*, *invokespecial*, *invokestatic*, *invokeinterface* appartiennent aux blocs dans lesquels elles apparaissent. Si on applique ce principe à la représentation en octets de la méthode débit que l'on retrouve à la section A.2, on obtient les blocs élémentaires de la Fig. 5.3 qui sont au nombre de 11 et de taille variable.

Cas des instructions qui lèvent des exceptions En Java ou Java Card, durant l'exécution du bytecode, trois situations peuvent entrainer la levée d'une exception :

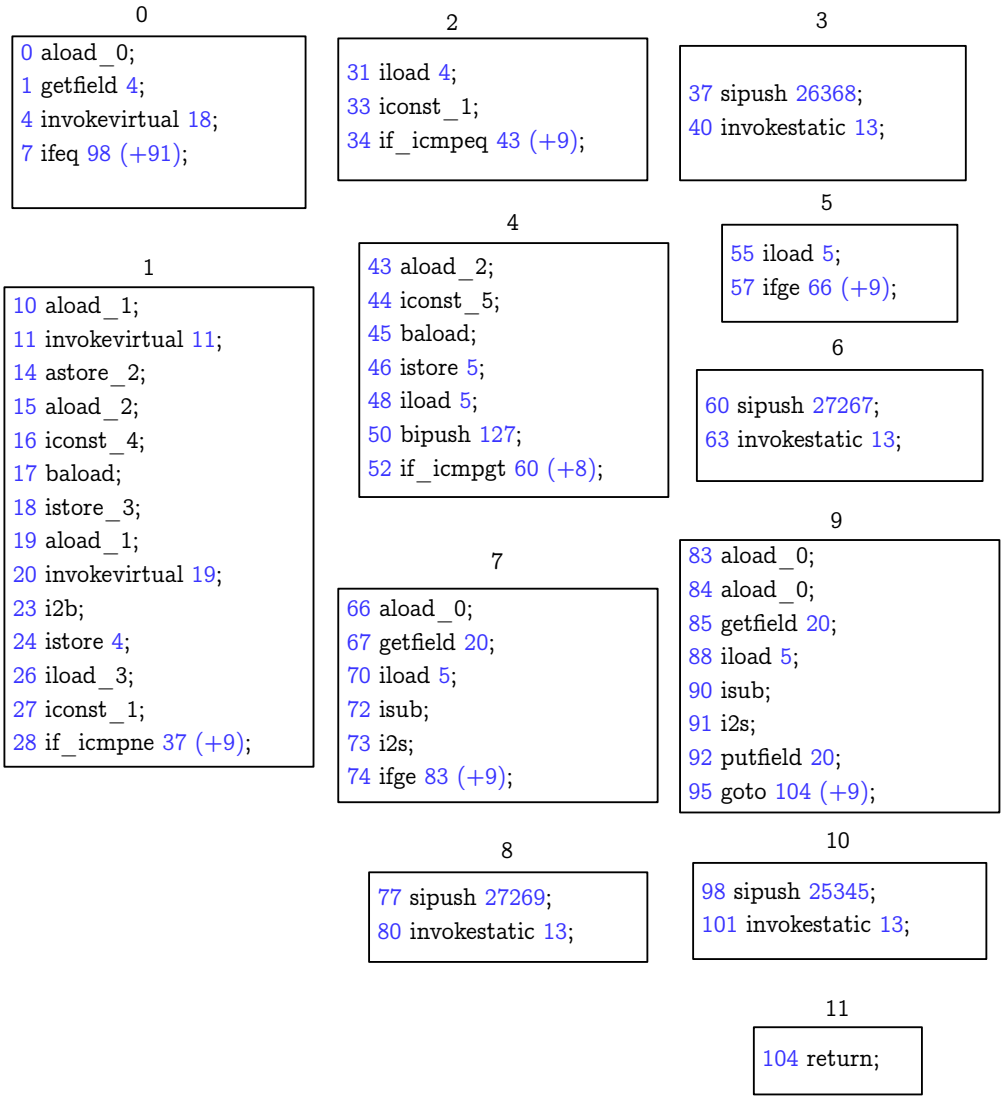

Fig. 5.3 – Bloc élémentaire de la méthode débit (voir section A.1)

- La machine virtuelle lève une instance d'une sous-classe *VirtualMachineError* dans le cas d'une erreur interne ou d'une limitation de ressource pour arrêter l'exécution,
- Une exception est explicitement levée par l'instruction *athrow*,
- Une exception est implicitement levée par une instruction de la machine virtuelle.

La machine virtuelle contient 187 instructions qui peuvent être utilisées pour implémenter un programme Java Card au format bytecode. Parmi elles, il existe une seule instruction capable de lever explicitement une exception : l'instruction athrow. Il y a vingt huit autres instructions qui peuvent implicitement lever une exception, ces instructions sont : *aaload*, *aastore*, *anewarray*, *arraylength*, *balaod*, *bastore*, *checkcast*, *getfield* , *getstatic* , *iaload*, *iastore*, *idiv*, *instanceof*, *invokeinterface* , *invokespecial* , *invokestatic* , *invokevirtual* , *irem*, *ldc*, *ldc_w*, *monitorenter*, *monitorexit*, *multinewarray*, *new*, *newarray*, *putfield* , *putsatic*, *saload* et *sastore*.

Dans le cas où on prend en compte les instructions pouvant lever des exceptions, alors elles peuvent également terminer un bloc élémentaire. Il faut donc ajouter aux règles précédemment

établies, les règles suivantes :

- Chaque instruction qui suit immédiatement toute instruction qui peut lever explicitement une exception est un leader
- Chaque instruction qui peut lever implicitement une exception est un leader

Deux possibilités permettent maintenant de déterminer l'ensemble des blocs élémentaires d'une méthode. Mais lorsqu'on choisit l'approche à utiliser pour le calcul des blocs, si l'on considère toutes les instructions qui peuvent lever une exception implicitement ou explicitement. On produit un plus grand nombre de blocs, donc on occupe plus d'espace dans la mémoire de stockage de la carte. Pour les besoins de l'exposé, je vais uniquement considérer le cas des blocs sans prise en charge des exceptions, cela ne change en rien le principe du mécanisme de protection. De plus, lors de l'implémentation de ce mécanisme de protection, j'ai utilisé chacune des deux approches précédentes (avec ou sans gestion d'exceptions).

Hors de la carte Tout d'abord, le programmeur détermine la méthode qu'il veut protéger et il précise le type de mécanisme qu'il veut utiliser pour la protéger. Pour ce faire, il utilise une annotation de sécurité dont on peut voir un exemple d'utilisation sur la méthode débit du Code 12. Dans cet exemple, elle est marquée comme devant être vérifiée en intégrité par le mécanisme de protection des blocs élémentaires ("BasicBlock") .

Code 12 Une opération de débit provenant d'une applet de porte-monnaie électronique

```
@SensitiveType{
        sensitivity=SensitiveValue.INTEGRITY
        proprietaryValue=''BasicBlock''
}
private void debit(APDU apdu) {

        // access authentication
    if ( pin.isValidated() ) {
                ...
                balance = (short) (balance - debitAmount);

    } else {

                ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);

    }
} // end of debit method
```

Un module calcule maintenant les blocs élémentaires grâce à la méthode que j'ai exposée précédemment. Par la suite, ce module transforme les annotations présentes dans le code en composants additionnels (voir Code 13). On conserve ainsi pour chaque bloc élémentaire calculé, la valeur du PC qui correspond au début du bloc, la valeur du PC qui correspond à la fin du bloc, ainsi que la valeur de contrôle pour le bloc.

La valeur de contrôle du bloc est calculée avec une fonction qui se sert de tous les éléments qui constituent le bloc c.-à-d. de tous les octets le composant. La fonction que j'ai retenue afin d'effectuer ce calcul est la fonction f (voir l'équation 5.6) qui prend en entrée tous les octets d'un bloc précis, indexé par I et qui retourne un octet.

$$f : (x_i)_{i \in I} \mapsto \bigoplus_{i \in I} x_i \text{ , avec } \bigoplus \text{ étant l'opérateur XOR} \tag{5.6}$$

Code 13 composant additionnel champ de bit

```
Basic_bloc{
        u1 attribute_name_index;
        u4 attribute_length;
        basic_bloc elements[attribute_length];
}

basic_bloc_element{
        u1 begin_pc;
        u1 end_pc;
        u1 check_value
}
```

J'ai choisi l'opérateur XOR parce que les fonctions de hachage classiques comme le SHA-1 ou le MD5 nécessitent beaucoup de ressources processeur et mémoire. De plus, avec le modèle de fautes retenu, c.-à-d. la modification d'un octet précis du code, il est possible grâce à l'opération XOR de déterminer s'il y a eu une modification. Par contre, si le changement a eu lieu dans plusieurs octets du bloc, l'opération de XOR devient insuffisante. Mais, ce cas couvre des attaques qui peuvent arriver uniquement au-delà des conditions définies par le modèle de fautes choisi (voir la section 3.5).

Afin d'illustrer le calcul de la valeur de contrôle, reprenons le fragment de code de la Fig. 5.4 tiré de l'exemple d'attaque sur l'opération de débit. Dans cette figure, on peut distinguer le bloc allant du PC 00 ("PC début" sur le schéma) au PC 09 ("PC fin" sur le schéma), on y retrouve également le calcul de la valeur de contrôle du bloc.

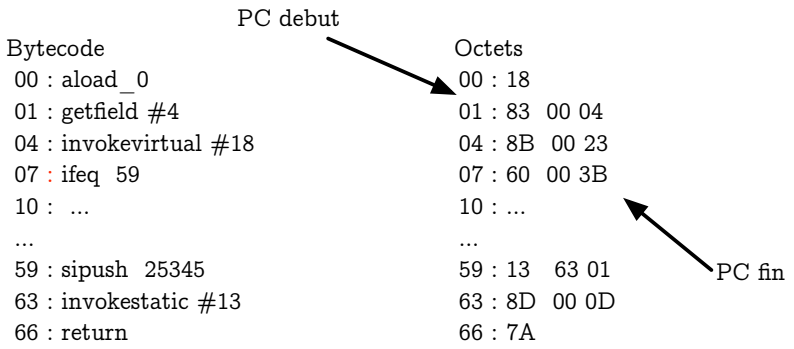

Fig. 5.4 – Calcul d'un bloc élémentaire et de sa valeur de contrôle

Une fois que le calcul des blocs et de leur valeur de contrôle est terminé (pour les méthodes à protéger), les composants additionnels créés sont ajoutés au classfile afin d'être chargés dans la carte avec l'application.

Dans la carte Une fois l'application stockée dans la carte, et pendant l'exécution d'une méthode protégée par le mécanisme des blocs élémentaires, la machine virtuelle effectue un certain nombre de tâches afin de s'assurer à l'aide des informations qui ont été ajoutées hors de la carte que le code

n'a pas été modifié. La machine virtuelle Java Card doit être modifiée afin de traiter les informations contenues dans les composants additionnels qui sont insérées dans l'application. Ainsi, pour une méthode donnée l'interpréteur de la machine virtuelle vérifie certains points, selon l'algorithme suivant :

1. Dans un premier temps, elle récupère le premier bloc élémentaire dans le tableau sauvegardé. Ce bloc devient le bloc courant à traiter.
2. Puis pour ce bloc, elle vérifie que le PC de début du bloc correspond bien au même PC de début que celui sauvegardé pour le bloc courant. Si ce test échoue, alors c'est le code qui a été modifié et la machine virtuelle arrête son interprétation du code. Sinon, elle passe à l'étape suivante
3. Au fur et à mesure de l'interprétation, elle applique l'opérateur XOR entre chacun des octets qui composent le bloc.
4. Une fois la fin du bloc rencontrée dans le code, elle vérifie que l'instruction qui s'y trouve correspond bien à une instruction pouvant terminer un bloc. Si ce test échoue, alors le code de l'application a changé. La machine virtuelle arrête alors son exécution. Sinon, elle passe à l'étape suivante.
5. La fin du bloc ayant été atteinte, alors le calcul de la valeur de contrôle s'arrête et le résultat obtenu est comparé avec celui sauvegardé pour le bloc correspondant. Si cette comparaison échoue, c'est que le code de l'application a changé, la machine virtuelle arrête donc l'exécution. Sinon, elle continue à l'étape suivante.
6. Le bloc suivant devient le bloc courant, et les étapes de 2 à 6 sont répétées jusqu'à ce qu'il n'y ait plus de blocs élémentaires à traiter.

Conclusion

Cette méthode a pour elle deux avantages. Le premier est qu'elle permet de détecter n'importe quel changement d'octets qui aurait lieu dans le code de l'application (modification d'opcode ou d'opérande). De plus, cette contremesure permet de générer une valeur de contrôle de taille constante d'un seul octet, indépendamment de la taille du bloc élémentaire pour lequel on souhaite obtenir cette valeur. Cela permet un gain d'espace important.

A titre d'exemple, on constate que la méthode de débit utilisée jusqu'à présent à une taille de 104 octets, avec les blocs élémentaires obtenus au nombre de 11, il faut 33 octets pour sauvegarder les informations nécessaires à la détection, ce qui correspond à 34,32 %. Ce qui n'est pas énorme lorsqu'on sait que dans le classfile la zone contenant les octets qui définissent les méthodes ne représente pas grand-chose au vu de la totalité des informations qui y sont contenues (entre autres le *constant pool*, la table des variables locales...).

Tout cela s'obtient sans sacrifier la qualité de détection du mécanisme. En outre, les vérifications sont effectuées en code natif, ce qui permet d'avoir une perte de performance minimale en terme de vitesse d'exécution pour la plateforme sur laquelle elles s'exécutent.

5.2.3 Méthode de la compression du bytecode

Introduction

Cette contremesure s'attaque également à la sécurité du code de l'application. En effet dans cette section, nous verrons en quoi la compression du bytecode permet d'améliorer la résistance des applications face à la modification du tableau d'octets qui représente une méthode, sachant que ces modifications se font en respectant le modèle de fautes choisi. En utilisant cette méthode, nous voulions appliquer des codes correcteurs [17] à la partie opcode des instructions afin de détecter tout changement et cela sans augmenter leur taille. Pour pouvoir atteindre cet objectif, il fallait réduire le nombre d'instructions Java Card.

Le but de la compression employée ici est de réduire le jeu d'instructions (nombre d'opcodes) nécessaires au codage des applications Java Card. Actuellement, pour exprimer un programme Java Card, la spécification de Sun propose un ensemble E d'instructions de taille $n = 187$. Toutes les instructions i de cet ensemble sont nécessaires pour coder l'ensemble des possibilités qu'offre une application Java Card. De ce fait, il faut 8 bits afin de pouvoir coder un opcode, donc moins on a d'instructions, moins on utilise de bits pour l'exprimer. Cela permet d'utiliser le ou les bits ainsi libérés pour pouvoir faire de la détection grâce aux codes correcteurs. Ainsi, le moyen que nous avons trouvé pour réduire cet ensemble E sans perdre la possibilité d'exprimer une application Java Card est d'effectuer de la compression de code afin d'obtenir de nouvelles instructions. La compression de code est un domaine très important dans le monde de l'embarqué (voir [CM99 ; Yos+97 ; LW98 ; Kim+10]). Car dans ce domaine on est limité en mémoire, la compression est donc utile afin d'optimiser cette ressource. Elle se fait en général au détriment des performances en terme de vitesse d'exécution. Je vais maintenant expliquer comment on peut réduire le jeu d'instructions Java Card sans perte de performance.

En observant attentivement le jeu d'instructions Java Card, il est possible de se rendre compte que leur distribution au sein des applications n'est pas anodine. En effet, on peut constater qu'il y a un certain nombre d'instructions qui apparaissent systématiquement les unes à la suite des autres. Ces instructions vont former ce que j'appellerai des *motifs* ou des groupes d'instructions indissociables. Une fois un tel groupe repéré, celui-ci donne naissance à une nouvelle instruction et permet de faire disparaitre les instructions qui forment ce groupe indivisible. La partie principale de cette méthode de protection est constituée par l'algorithme de recherche de motifs. On recherche donc dans le plus grand nombre d'applications possibles ces motifs. Une fois ces motifs obtenus, on les transforme en nouvelles instructions.

Algorithme de compression

Afin de parvenir à compresser le code de façon optimale, j'ai utilisé le même algorithme de factorisation du code que les auteurs des articles [Cla+00 ; BG02], que j'ai adapté pour nos besoins. Cet algorithme se décompose en plusieurs étapes que je vais décrire par la suite.

Le concept de cette méthode est donc de repérer dans le code les opérations récurrentes du bytecode afin de les factoriser pour en faire des instructions. Ainsi, chacune de ces séquences de

17. Un code correcteur est une technique de codage basée sur la redondance. Elle est destinée à détecter et à corriger les erreurs de transmission d'une information. Un exemple de code correcteur très populaire est le code de hamming [Ham50]

bytecode peut être vue comme un motif, chaque motif ayant le même effet sémantique que les instructions qu'il remplace. Factoriser un programme en utilisant les motifs produits, conduit à un programme ayant une taille réduite, où chacune des occurrences d'un motif est remplacée par une nouvelle instruction (ou macro) représentant les instructions qui forment ce motif. L'ensemble de tous les motifs remplaçables forme ce que l'on appellera un *groupe d'occurrences*.

Cette opération de factorisation se fait en trois étapes. La première est la génération des motifs afin de chercher les séquences d'instructions récurrentes. La seconde étape est de purger les motifs qui ne peuvent pas être mis sous forme de macros de la liste générée. Enfin, la dernière étape est de produire un bytecode factorisé en remplaçant les séquences d'instructions récurrentes par les motifs qui leur correspondent tout en respectant la liste de motifs générés.

Génération des motifs Afin de trouver les motifs qui vont permettre de factoriser les différents groupes, il faut créer un ensemble maximal de groupes d'occurrences. Premièrement, un groupe de taille 1 est créé pour chaque ensemble d'instructions équivalentes c.-à-d. qu'on recherche toutes les instructions qui apparaissent plus d'une fois dans le code. Ces groupes sont itérativement agrandis soit en les allongeant, soit en les divisant. Pour ce faire, si toutes les instructions qui suivent la dernière instruction d'un groupe sont équivalentes alors le groupe devient plus grand d'un élément. Sinon, un nouveau groupe est créé pour chaque ensemble d'instructions équivalentes qui suivent la dernière instruction du groupe. Par exemple, on crée un groupe de taille 1 avec l'instruction *<aload_0>* parce que cette instruction apparait plus d'une fois dans les applications. Si chacune de ces occurrences est suivie par une instruction *<getfield 2568>* alors le groupe de taille 1 devient un groupe de taille deux contenant les instructions *<aload_0, getfiled 2568>* et ainsi de suite.

Un groupe d'occurrences qui a été allongé peut couvrir partiellement une autre occurrence de ce même groupe, en ce sens où certaines de ses occurrences peuvent couvrir partiellement d'autres occurrences. Chaque fois que cela se produit, la dernière occurrence traitée est supprimée avant de vérifier l'occurrence qui suit, permettant ainsi d'enlever le plus petit nombre possible d'occurrences du groupe. Une fois que tous les groupes d'occurrences sont trouvés, il faut supprimer ceux qui couvrent totalement d'autres groupes.

Elagage des motifs On réduit les groupes d'occurrences de sorte que les instructions problématiques (instruction que l'on ne peut pas factoriser) n'apparaissent pas dans les motifs, faisant ainsi en sorte qu'aucune instruction de branchement ne soit dans les bornes des groupes trouvés. Cela se fait en deux étapes. Dans la première, on traite les instructions qu'on ne peut pas factoriser et les branchements vers des instructions que l'on appelle branchement sortant (correspondant aux instructions de branchement conditionnel ou inconditionnel). Puis dans la seconde, les branchements provenant d'instructions de branchements que l'on appelle branchement entrant (les cibles des instructions de branchement).

Les instructions *tableswitch*, *lookupswitch*, *jsr* et *ret* sont considérées comme étant des instructions qu'on ne peut pas factoriser. Les instructions de type switch par exemple ont des contraintes d'alignement qui les rendent difficiles à placer dans les motifs, et les instructions de saut vers les sous-routines induisent des problèmes de flot de contrôle intraprocéduraux. De plus, aucun de ces types d'instructions n'apparait très fréquemment dans le code, on considère donc qu'ils ne valent pas le coût du supplément de complexité qui serait nécessaire à leur gestion.

Les instructions qu'on ne peut pas factoriser sont donc enlevées des motifs calculés en les séparant en deux nouveaux motifs, divisant par la même occasion le groupe d'occurrences. De même, quand un branchement sortant quitte un motif, l'instruction de branchement correspondante est enlevée de ce motif, créant deux nouveaux motifs. Afin de réduire la complexité de cette étape, on n'autorise pas les branchements arrières (cas de boucles) dans les motifs. La grande majorité des branchements sont des branchements avant, causés par les instructions *if* et *break*. En vérifiant les branchements de façon ascendante, on prévient le fait que la subdivision d'un motif puisse rendre illégal un autre motif déjà vérifié.

Ensuite, les branchements entrants sont vérifiés. À contrario des branchements sortants, qui apparaissent dans chaque occurrence d'un motif, les branchements entrants peuvent être le résultat d'un branchement unique et non partagé par un groupe entier. De ce fait, on enlève uniquement l'occurrence qui a un branchement entrant. L'alternative pourrait être de subdiviser le motif entre l'instruction cible du branchement et celle qui la précède. Par contre, cela implique des vérifications supplémentaires pour s'assurer que les branchements intra-motifs ne sont pas affectés. La première instruction d'un gestionnaire d'exception, ou les premières et dernières instructions d'une région du code, où les exceptions sont récupérées et traitées de la même façon que les cibles de branchements entrants. Après avoir effectué ces réductions, les groupes totalement couverts par d'autres groupes ainsi que les groupes de taille 1 sont supprimés.

Maintenant qu'on dispose d'une première liste de motifs, on procède à d'autres ajustements, on considère dans cette liste de motifs que deux motifs ayant exactement la même taille et les mêmes instructions apparaissant exactement dans le même ordre sont identiques. Ainsi, les motifs <*sipush 26368*, *invokestatic 13*> et <*sipush 27267*, *invokestatic 13*> sont identiques. On fusionne ainsi tous les motifs du même type en un seul nouveau motif et toutes les autres occurrences de ce motif seront supprimées. On considère également que les instructions : de types <*x*>*store_*<*n*>, <*x*>*load_*<*n*>, <*x*>*const_*<*n*>, etc. sont identiques. Ainsi les motifs <*aload_0*, *getfield 20*, *iload_0*> et <*aload_1*, *getfield 20*, *iload_1*> sont identiques et vont donner le motifs <*aload <x>*, *getfield <x>*, *iload <x>*> (où <x> correspond à l'opérande de l'instruction, par exemple *aload_0* devient *aload 0*). On fusionne les motifs du même type ainsi détectés et on supprime toutes les autres occurrences qui leur correspondent.

Application des motifs Une fois tous les motifs obtenus, on crée de nouvelles instructions Java Card que l'on appele macro-instructions avec un comportement identique aux instructions factorisées. Par exemple, le motif <*sipush 26368*, *invokestatic 13*> devient l'instruction *sipushandinvokevirtual 26368 13* et le motif <*aload_0*, *getfield 20*, *iload_0*> devient l'instruction *aloadgetfieldandiload 0 20 0*. Les macros sont générées en sélectionnant les groupes d'occurrences qui permettent de réduire au maximum la taille des applications et en continuant jusqu'à ce qu'il n'y ait plus de macro-instructions disponibles ou de groupes qui permettent de réduire la taille des applications. Ensuite, la dernière étape consiste à réécrire le code des applications en tenant compte des nouvelles instructions créées. Cette conversion doit se faire lors du chargement de l'application après toutes les vérifications sur le fichier class.

Chargement de l'application dans la carte

Lors du chargement de l'application dans la carte, les nouvelles instructions vont être écrites en code natif et masquées dans la mémoire morte de la carte. À ce moment deux cas peuvent se

produire :

- La méthode marquée pour être protégée à l'aide de la méthode de compression du bytecode ne peut pas être interprétée grâce au nouveau jeu d'instructions produit, car il y a certaines instructions dans cette méthode qui ne font pas partie du nouveau jeu d'instructions créé. Dans ce cas précis, elle est exécutée avec le jeu d'instructions Java Card classique.
- La méthode marquée pour être protégée à l'aide de la méthode de compression du bytecode peut être entièrement remplacée par les instructions appartenant au nouveau jeu d'instructions créé. Et dans ce cas précis, elle est compressée puis exécutée avec ce jeu d'instructions.

Grâce à ce mécanisme, on s'assure de pouvoir exécuter tous les types d'applications sachant qu'il y a certaines méthodes qui ne seront pas protégées, car elles ne sont pas compressibles. Mais si cela arrive, on peut activer un autre mécanisme de sécurité comme celui de la vérification d'intégrité vu dans la sous-section 5.2.2.

Discussion sur le modèle de fautes et la méthode de compression

Le but initial de cette méthode de contremesure était d'appliquer un code correcteur aux opcodes afin de permettre une détection de leurs modifications. L'idée d'appliquer des codes correcteurs d'erreurs afin d'effectuer la détection n'est pas utile dans le cadre du modèle de fautes choisi. En effet, le fait de réduire le jeu d'instructions Java Card suffit à rendre le code plus résistant à une modification ; car l'attaquant a moins de chance de tomber sur un mot de code ayant du sens pour la machine virtuelle. En effet, le modèle de fautes implique que potentiellement lorsqu'une injection de fautes réussit à modifier un opcode, elle le fait de façon équiprobable avec une chance de $\frac{1}{256}$. Ainsi, lors d'une attaque en faute réussie qui suit notre modèle de fautes, la probabilité que l'on retombe sur une instruction qui ait du sens pour la machine virtuelle est calculée par l'équation 5.7. Le pourcentage obtenu est directement fonction de la taille du jeu d'instructions. Ainsi plus le nombre d'instructions est petit, moins on a de chance de tomber sur une instruction qui a du sens d'un point de vue de la machine virtuelle Java Card. En réduisant la taille du jeu d'instructions Java Card, on réduit le nombre de bits nécessaire au codage des opcodes. Par exemple, prenons les trois derniers bits utilisés pour appliquer un contrôle de parité, on aurait les cinq bits restants qui serviraient à coder l'opcode en lui-même et toujours le même nombre d'instructions dans le jeu d'instructions Java Card (sauf que leurs codages seraient différents). En raison du fait que le nombre d'instructions disponibles reste le même alors la probabilité calculée à l'équation 5.7 est strictement la même que si on s'était limité à la réduction du jeu d'instructions.

$$P = \frac{1}{255} \times tailledujeud'instructions = \frac{187}{255} \approx 0.72 \tag{5.7}$$

Pour aller plus loin, considérons les 8 groupes indistincts d'opcodes du Code 11. Lorsqu'une instruction appartenant à un de ces groupes est remplacée par une autre du même groupe, la machine virtuelle ne le voit pas. Ainsi, tout autre changement hors des instructions appartenant à ces groupes entraine un arrêt de l'interprétation du code et une levée d'exception. La probabilité pour que suivant le modèle de fautes choisi l'attaque remplace une instruction par une autre instruction ayant du sens pour la machine virtuelle est calculée par l'équation 5.8.

$$P = \sum_{i=0}^{8} tailledugroupe_i \times \frac{1}{256} = \frac{42}{256} \approx 0.16 \tag{5.8}$$

Étant donné que ces groupes sont encore réduits, voir pour certains inexistants grâce au nouveau jeu d'instructions introduit dans la machine virtuelle, la probabilité pour que le remplacement ait lieu est presque nulle.

5.3 Intégrité du flux de contrôle

Avec la précédente catégorie de mécanismes de protection, je me suis attaqué à la protection du code des applications, c.-à-d. que j'ai mis en oeuvre un ensemble de contremesures qui permettent de s'assurer que les octets composant le code des applications Java Card n'étaient pas modifiés durant son exécution. Il reste quand même un aspect des attaques à prendre en compte. En effet, qu'est-ce qui se passe si au lieu de toucher directement le code, l'injection de fautes perturbe les registres du microprocesseur comme le pointeur de code ou le pointeur de pile. Je m'intéresse ici aux fautes qui ont lieu sur le pointeur de code.

5.3.1 Méthode de vérification du chemin emprunté

Principe de la méthode

Cette méthode se base sur le fait que lors de l'interprétation du code d'une application, la machine virtuelle passe par un certain nombre de points de celui-ci. Cela donne un ensemble de points de passage que l'on appelle des «*chemins*». Il est possible de calculer de façon statique l'ensemble des chemins que peut emprunter l'interpréteur lors de l'exécution d'un code. Ainsi, pour une méthode donnée à protéger, afin de calculer cet ensemble de chemins, on détermine tout d'abord le graphe de flots de contrôle de la méthode, une fois ce graphe calculé, on lui applique un algorithme de recherche de chemins.

Une fois ces chemins obtenus, on constate que ceux pouvant mener à une ressource prédéterminée dans le cadre des applications Java Card ne sont pas nombreux. De façon pratique, on obtient un ou deux chemins sauf cas exceptionnel. Cela est dû au modèle de programmation des applets Java Card.

Création du graphe de flots de contrôle Un graphe de flots de contrôle est composé de noeuds ou sommets et d'arêtes. Les sommets du graphe vont correspondre aux blocs élémentaires. La première étape dans l'élaboration d'un graphe de flux de contrôle, pour une méthode donnée, est le calcul des blocs élémentaires (voir sous-section 5.2.2). Une fois ces blocs obtenus, on les chaine entre eux en fonction des liens qui peuvent exister. Soit $u \in V$ et $v \in V$ avec V correspondant à l'ensemble des sommets (blocs élémentaires) du graphe et A correspondant au CFG de la méthode. Pour calculer le graphe de flots de contrôle d'une méthode, il faut utiliser les règles suivantes :

1. Une arête est créée, si le sommet v suit immédiatement le sommet u dans le bytecode et si u ne se termine pas par un branchement inconditionnel.
2. Une arête (u, v) est créée, si la dernière instruction de u est une instruction de branchement conditionnel ou inconditionnel menant à la première instruction de v.
3. Une arête est créée entre chaque bloc élémentaire qui se termine par une instruction de type *tableswitch* ou *lookupswitch* et chaque instruction qui est définie comme étant une cible de ces instructions.

4. Pour chaque appel à une sous-routine, deux arêtes sont créées. Une du bloc élémentaire se terminant par un *jsr* ou un *jsr_w* vers le bloc cible et l'autre depuis le bloc élémentaire contenant l'instruction *ret* correspondante.
5. Il faut en plus créer une arête (u, v) du bloc contenant toutes les instructions qui peuvent lever une exception vers l'entrée du bloc élémentaire qui gère la levée d'exception, et qui couvre la région du code où apparait l'instruction.
6. Il faut aussi créer une arête du bloc contenant toutes instructions qui peut lever une exception vers un bloc élémentaire qui représente une fin anormale de la méthode, c.-à-d. lorsqu'une exception est levée sans être traitée par aucun gestionnaire d'exception.

Hors de la carte Le développeur d'application marque la méthode qu'il souhaite protéger et définit le mécanisme de protection qu'il souhaite utiliser pour la protéger (voir Code 14). Dans cet exemple, la méthode est marquée comme devant être protégée en intégrité par le mécanisme "*path check*" (pour détection de chemins).

Code 14 Extrait d'une opération de débit provenant d'une applet de porte-monnaie électronique

```
@SensitiveType{
        sensitivity=SensitiveValue.INTEGRITY
        proprietaryValue=''PathCheck''
}
private void debit(APDU apdu) {

        // access authentication
    if ( pin.isValidated() ) {
                ...
                balance = (short) (balance - debitAmount);

    } else {

                ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);

    }
} //   of debit method
```

Une fois les méthodes marquées et le programme compilé, un outil se charge dans un premier temps d'élaborer le CFG des méthodes marquées. Une fois les CFG obtenues, il détermine les chemins qui partent du sommet initial vers chacun des sommets du graphe. Partons d'un exemple, pour cela reprenons la représentation en bytecode de la méthode de débit (voir la section A.2), et appliquons-lui l'opération de calcul du graphe. La première étape de l'opération est de calculer les blocs élémentaires en appliquant la méthode vue en sous-section 5.2.2, on obtient alors les blocs de la Fig. 5.3. La seconde étape est de lier les blocs élémentaires entre eux grâce aux règles énoncées dans la sous-section 5.3.1. Le résultat de cette liaison donne le graphe de flux de contrôle de la Fig. 5.5

Dans un second temps, les annotations sont transformées en composant additionnel (voir Code 15). Je vais maintenant commenter la structure de données retenue pour ce composant additionnel. Les deux premiers éléments sont toujours les mêmes (voir section 5.1). Ensuite, on a un tableau qui contient pour chacun des sommets du graphe (bloc élémentaire), la valeur du PC de début de ce bloc, le *number_of_bit* est le nombre de bits utilisé pour coder le chemin qui conduit à ce noeud, et enfin le *path* est un tableau contenant les octets qui contiennent le chemin. La taille

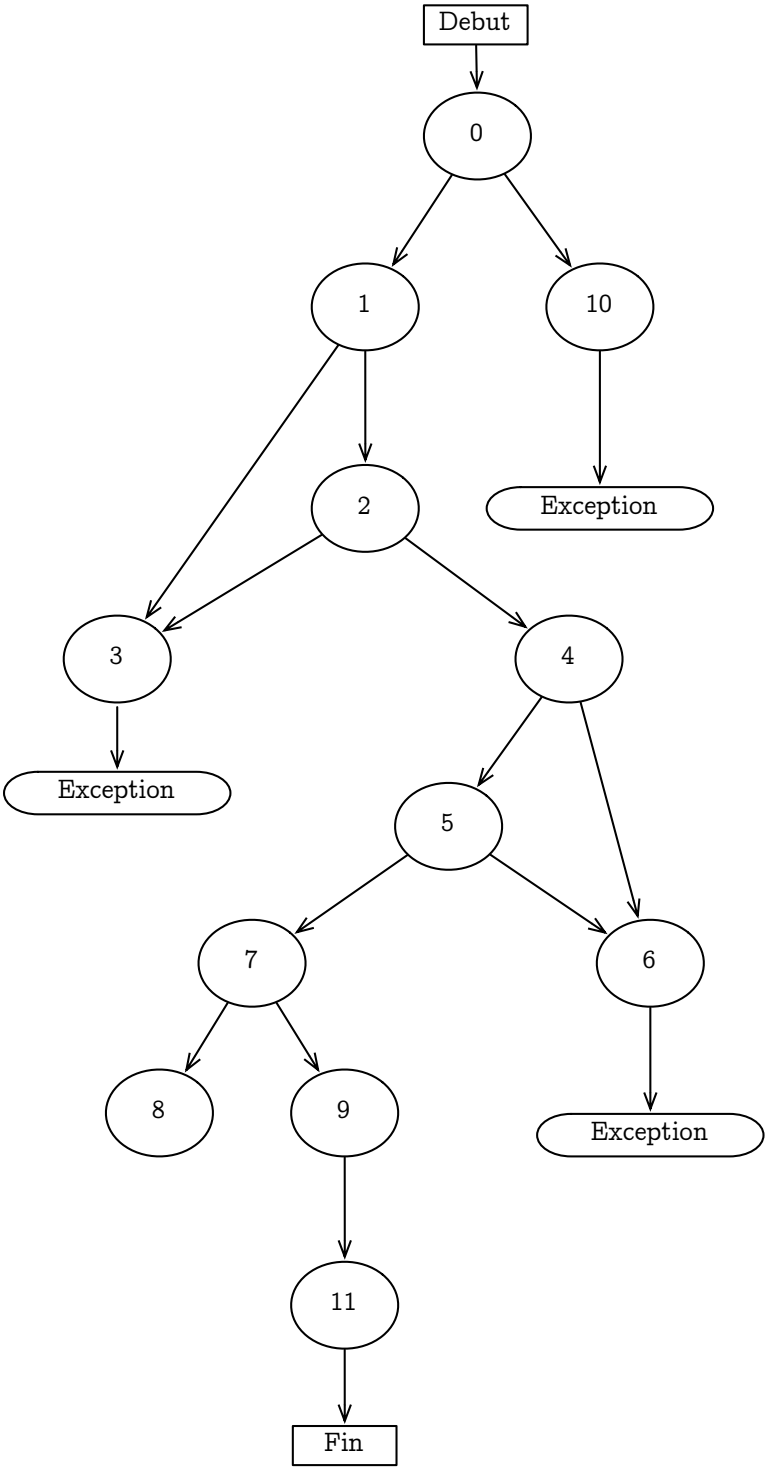

Fig. 5.5 – Le graphe de flot de contrôle de la méthode débit (voir section A.1)

de ce tableau est obtenu grâce à la division entière : $\frac{numberofbit}{8}$; si le reste de cette division est nul alors cette opération donne le résultat, sinon il faut utiliser la formule $\frac{numberofbit}{8} + 1$.

Code 15 composant additionnel permettant de mémoriser les chemins

```
path{
        u1 attribute_name_index;
        u2 number_of_path;
        path_element list_of_path[number_of_path]
}

path_element{
        u1 Pc;
        u2 number_of_bit;
        u1 path[size_of_path];
}
```

On utilise certaines règles afin d'établir une convention qui permet de coder les chemins potentiels du graphe. En effet, lorsqu'une instruction de rupture de flot de contrôle est rencontrée, il peut se produire plusieurs évènements :

1. Dans le cas où l'on se trouve au début de la méthode, l'arête qui part du sommet du graphe vers le premier bloc élémentaire est codée avec deux bits 01, cela permet d'éviter cette valeur soit modifiée par 00 ou 11.
2. Dans le cas d'un branchement conditionnel, l'interprétation continue à partir de l'instruction qui suit l'instruction de branchement ou alors elle se poursuit à partir de l'instruction qui est la cible du branchement. L'arête qui correspond au cas où l'interprétation se poursuivrait directement à l'instruction qui suit le branchement sera marquée avec le bit 0 et celle qui correspond au cas où il y aurait un saut vers la cible de l'instruction de branchement sera marquée avec le bit 1.
3. Dans le cas d'un branchement inconditionnel, l'interprétation continuera systématiquement à partir de l'instruction cible du saut. Dans ce cas précis, on code l'arête qui correspond au saut vers la cible avec les bits 01.
4. Il reste le cas des instructions de branchements multiples (tableswitch et lookupswitch). Ce sont des cas particuliers, car le codage sera dépendant du nombre de branchements possibles. En effet, on code chaque branchement avec autant de bits 0 ou de bits 1 que nécessaire, c.-à-d. que si on a quatre sauts possibles (voir la Fig. 5.6), alors chacune des arêtes correspondantes à ces sauts est codée comme dans le Tableau 4. On constate ainsi qu'il faut trois bits afin de mémoriser une instruction de branchements multiples à 4 branches. Les instructions à choix multiples vont entrainer une légère modification du composant additionnel. En effet, il faudra rajouter un composant additionnel *path_element* dont les éléments sont :
   - *number_of_cond_jump* : le nombre d'instructions de type switch de la méthode,
   - *tab_pc_of_cond_jump* : un tableau contenant la liste des PC qui correspondent à des instructions de types switch,
   - *tab_number_of_cond_jump* : un tableau contenant pour chaque instruction de switch le nombre de sauts possibles.

   Pour cette raison, il faut éviter, dans une application, d'utiliser la gestion de flot de contrôle à base d'instructions de type switch. Car cela peut augmenter considérablement la taille du composant additionnel. Il est à noter que les applications Java Card utilisent très peu ce type d'instructions.

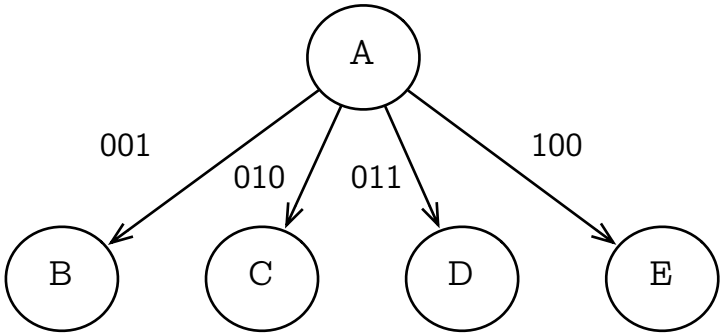

Fig. 5.6 – Le graphe de flot de contrôle de la méthode débit (voir section A.1)

Arêtes	Codages
A → B	001
A → C	010
A → D	011
A → E	100

Table 5.1 – Codage de l'instruction de branchements multiples de la Fig. 5.6

Code 16 composant additionnel permettant de mémoriser les instructions de type switch

```
switch{
    u1 attribute_name_index;
    u2 number_of_cond_jump;
    u2 tab_pc_of_cond_jump [number_of_cond_jump];
    u2 tab_number_of_cond_jump [number_of_cond_jump];
}
```

Si ces règles sont correctement appliquées, on obtient le graphe de la Fig. 5.7. Dans cette figure, nous voyons le codage appliqué à chacune des arêtes du graphe.

Les chemins ainsi obtenus seront des listes formées de suite de bits 0 et 1. On se focalise sur le sommet 9 qui contient l'opération fondamentale de cette méthode c.-à-d. que dans le bloc élémentaire correspondant à ce sommet, le porte-monnaie de l'utilisateur est bien décrémenté (opération de soustraction exécutée) et la carte retourne bien le statut de réussite de transaction. On obtient une liste contenant un seul chemin qui correspond à la suite de sommets : $\{0, 1, 2, 4, 5, 7, 9\}$. Cette suite de sommets est transformée en une suite de bits : $\{01, 0, 0, 1, 0, 1, 1\}$. Cette opération est répétée pour chacun des sommets du graphe, les chemins ainsi obtenus sont sauvegardés dans le composant additionnel de la méthode.

L'application peut maintenant être chargée dans la carte. La deuxième phase du mécanisme de détection peut maintenant être déclenchée, cette phase est celle qui se déroule dans la carte.

Dans la carte : Lorsque la machine virtuelle exécute une méthode, la première vérification effectuée est celle de la présence de l'annotation dans la zone de définition de la méthode. Si cette annotation est présente alors elle vérifie que le mécanisme en cours d'exécution est bien le mécanisme de contrôle des chemins. Si c'est bien le cas alors la machine virtuelle exécute le code en activant le système de détection correspondant. Elle calcule le chemin emprunté lors de l'interprétation du bytecode et vérifie que celui-ci correspond à un des chemins stockés hors de la carte. Pour ce faire, elle s'appuie sur les règles suivantes :

1. Si l'interpréteur de la machine virtuelle rencontre une instruction de branchement conditionnel, en fonction du cas :
   - L'interprétation se poursuit à l'instruction qui suit immédiatement ce branchement, alors elle mémorise le bit 1.
   - L'interprétation se poursuit à l'instruction cible du branchement, alors elle mémorise le bit 0.
2. Si l'interpréteur de la machine virtuelle rencontre une instruction de branchement inconditionnel, alors l'interprétation se poursuit à l'instruction cible du branchement et, dans ce cas précis, elle mémorise les deux bits 0 et 1 pour le chemin.
3. Si l'interpréteur de la machine virtuelle rencontre une instruction de branchement multiple, alors en fonction du branchement choisi elle mémorisera une valeur en accord avec les règles vues pour les branchements conditionnels à choix multiples au point 4 de la page 71.
4. La machine virtuelle aura ainsi calculé un chemin jusqu'à un bloc élémentaire donné. Ainsi, avant de commencer l'exécution de ce bloc, elle compare le chemin calculé avec celui enregistré pour le PC sur lequel on se trouve.
   - Si le chemin appartient à la liste des chemins sauvegardée pour ce bloc, alors il peut être exécuté.
   - Si le chemin n'appartient pas à la liste des chemins sauvegardée pour ce bloc, c'est que le chemin emprunté lors de l'exécution n'est pas un chemin légal. Alors, la machine virtuelle peut arrêter l'interprétation du bytecode.

   Avec cette façon de procéder on s'assure que la carte ne retournera rien s'il y a eu une perturbation du flot de contrôle (du PC). Car les blocs ne sont exécutés qu'une fois que le chemin emprunté a été vérifié.

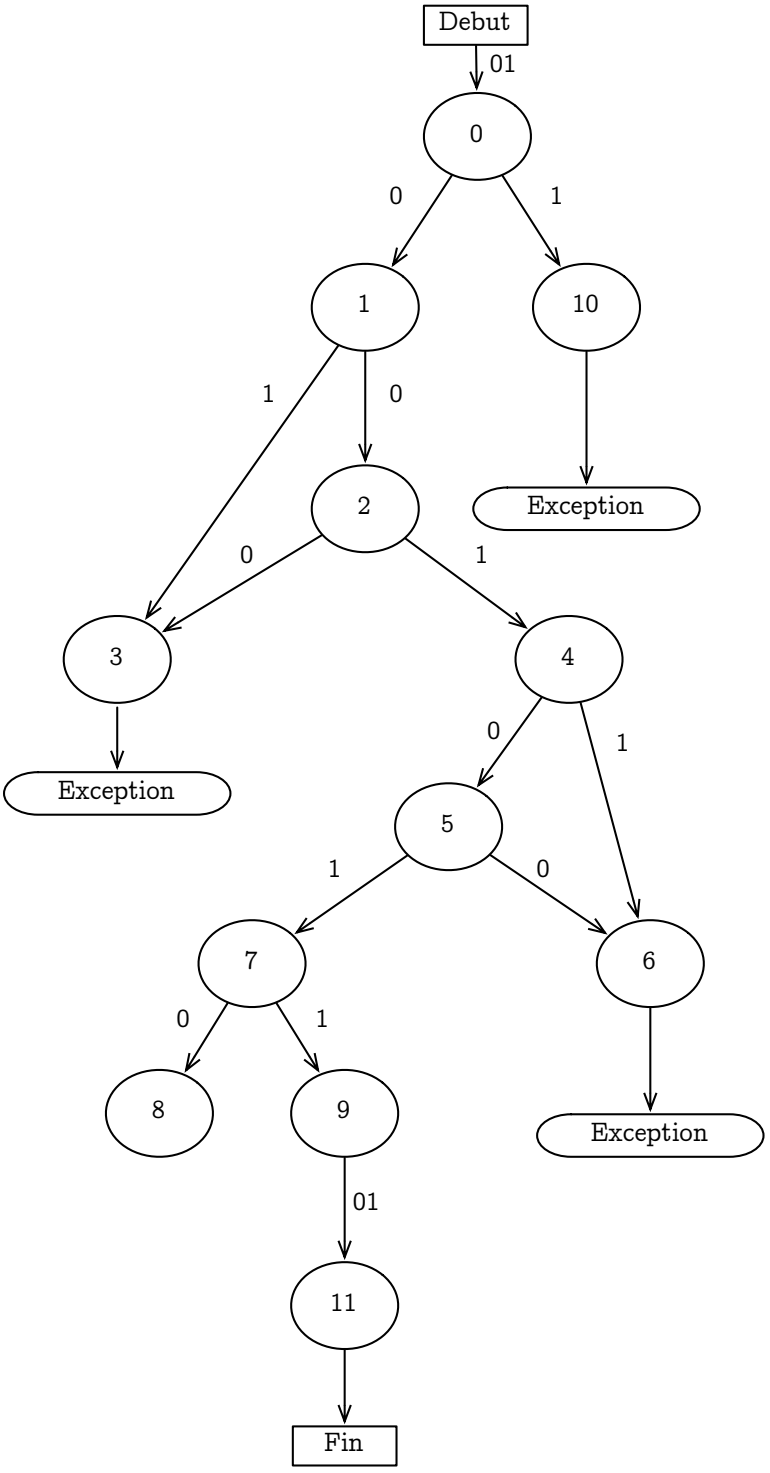

Fig. 5.7 – Le graphe de flot de contrôle de la méthode débit (voir section A.1) avec le codage des chemins

Conclusion

Avec cette méthode, j'ai utilisé le graphe de flot de contrôle comme référence afin de calculer les chemins potentiels que pouvait prendre la machine virtuelle lors de l'interprétation du code. J'ai réutilisé ces informations au sein du code de la machine virtuelle afin de détecter les déviations qui pourraient se produire dans le code si l'injection de fautes perturbait le pointeur de code du programme. Les résultats en terme de vitesse d'exécution sont un peu en retrait, car on a une augmentation plus importante de la vitesse d'exécution des applications par rapport aux autres mécanismes, sans toutefois dépasser les 8%. Par contre en terme d'augmentation de la taille des applications, on reste en dessous de 10% (pour tout le code protégé), ce qui reste acceptable pour la carte.

5.4 Comportement de la machine virtuelle lors d'une détection

Pendant tout ce chapitre, lorsqu'on détectait une attaque à l'aide d'un mécanisme, je proposais systématiquement d'arrêter l'exécution du programme. Ceci peut constituer une information utile à l'attaquant. Car le fait d'arrêter l'exécution du programme permet d'informer l'attaquant que l'injection de fautes qu'il vient de réaliser a réussi. Ce qui peut par exemple lui permettre de récolter certaines informations sur l'exécution de l'application ; de pouvoir faire de l'écoute sur les canaux cachés comme l'écoute de la consommation de courant (DPA [18][CCD00 ; JPS05], SPA [19][MS00 ; Man03], HODPA [20] [Gie+10]) ou d'émissions électromagnétiques (EMA [21][QS01]). Ainsi, il peut affiner son attaque en utilisant un autre modèle de fautes (le modèle de la double faute par exemple). Il serait alors intéressant de créer un leurre qui peut consister en une exécution normale, mais où les données manipulées seraient des données qui n'auraient aucun rapport avec les vraies données. Ainsi, l'attaquant ne saura pas si son injection de fautes a réussi ou non. Cela constitue un moyen de rendre plus difficile l'analyse par canaux cachés.

5.5 Conclusion

Ce chapitre a permis d'apporter une réponse à la problématique de la section 3.6 : à savoir «comment s'assurer que le code chargé sur la carte n'a pas subi de modifications entre le moment du chargement, jusqu'à son exécution». La réponse que j'apporte se résume en plusieurs mécanismes de protection qui permettent de détecter, durant l'exécution d'une application, qu'il n'y a pas eu de modification. Les contremesures se subdivisent en deux grandes parties :

– La vérification de l'intégrité du code avec des mécanismes d'intégrité innovants et différents des mécanismes classiques à base d'algorithmes cryptographiques. Le mécanisme du champ de bits qui permet d'effectuer une détection à l'instruction près d'une modification pouvant avoir eu lieu. Celui de la vérification des blocs élémentaires qui effectue une vérification en certains points du programme. Celui de la compression du bytecode qui permet de réduire

18. Differential Power Analysis
19. Simple Power Analysis
20. High Order Differential Power Analysis
21. ElectroMagnetic Analysis

la vulnérabilité du code par rapport à notre modèle de fautes et d'améliorer sensiblement le taux de détection du mécanisme du champ de bits, car elle réduit le nombre d'instructions indistinctes.

– La vérification du flot de contrôle de l'application avec un mécanisme de vérification des chemins qui permet de s'assurer que le code s'exécute bien selon un chemin connu et calculé lors de la création de l'application.

De plus, l'approche hybride (une partie du côté de l'application et l'autre dans le système) des mécanismes de protections développée, assure de meilleures performances en terme de ressources, ainsi qu'une meilleure couverture en terme de sécurité, car elle est plus difficile à attaquer. Elle constitue un bon compromis entre le taux de détection des fautes et les performances en terme de ressources processeurs et mémoires.

Chapitre 6

Evaluation et résultats

Sommaire

6.1 Introduction

Ce chapitre s'articule autour de l'évaluation des contremesures développées. On classifie les résultats obtenus en deux catégories : la *détection des attaques en faute*, ainsi que l'*utilisation des ressources* de la carte. La catégorie détection des attaques en faute va permettre de juger de l'efficacité des contremesures. Tandis que la catégorie utilisation des ressources va permettre aux industriels de déterminer le coût de mise en oeuvre de ces contremesures dans une carte. J'expliciterai tout d'abord le contexte d'évaluation des contremesures (outils utilisés), ensuite j'expliquerai comment s'est déroulée la collecte des métriques pour chacune des catégories de résultats.

6.2 Byte Code Engeneering Language

L'API BCEL est une API de manipulation de fichiers class (voir [Prone]) qui donne une abstraction de la façon de les lire et de les écrire. Elle est open source et elle comporte principalement trois éléments :

1. Un paquetage Java contient des classes qui reflète le format du fichier class. Il ne permet pas de modifier le bytecode de l'application. Ces classes doivent être utilisées pour lire les éléments d'un fichier class par exemple. Elles sont spécialement utiles quand il s'agit d'analyser des programmes Java dont on ne dispose pas des sources. La principale structure de données de cette classe est la classe "*JavaClass*" qui contient les méthodes, les champs, le *contant pool* de la classe, etc.
2. Un paquetage Java contenant des classes permettant de modifier ou de générer des objets de types "*JavaClass*" ou "*Method*". Il peut être utilisé afin d'insérer ou d'enlever des informations du fichier class, ou tout simplement pour implémenter le moteur du génération de code d'un compilateur Java.
3. Différents exemples de codes et des outils comme un visualiseur de fichier class, un convertisseur de fichier class en HTML, etc.

Pratiquement tous les outils qui ont été développés pendant cette thèse l'ont été avec cette API.

6.3 Analyseur Statique

L'un des premier outil que j'ai mis au point durant mon travail est un analyseur statique, il permet de faire l'analyse d'un fichier class et d'y ajouter les composants additionnels. Cet outil prend donc en entrée un fichier class contenant des annotations et fournit en sortie un fichier class contenant des composants additionnels. Pour ce faire, BCEL contient une classe capable de créer des attributs non prédéfinis exactement comme prévu par la spécification de Sun. Il va permettre d'obtenir les résultats en terme d'occupations mémoire qu'on retrouve dans la sous-section 6.8.2.

6.4 Interpréteur abstrait de bytecode

L'autre outil que j'ai mis au point durant mon travail est un interpréteur abstrait de bytecode Java. Cet outil a pour rôle de simuler le comportement de l'interpréteur de la machine virtuelle en implantant les éléments principaux du dispositif de frame d'une méthode à savoir la table des variables locales, et la pile des opérandes. Cet outil a été développé en Java à l'aide de BCEL.

Une implémentation naïve de l'interpréteur de la machine virtuelle ne fait aucune vérification supplémentaire sur le bytecode. Elle va allouer l'espace nécessaire aux éléments de la frame, puis elle va interpréter des bytecodes en réalisant les opérations nécessaires pour chacune d'entre elles comme l'instruction *iadd* qui prend les deux éléments de type entier au sommet de la pile ; qui réalise leur addition et qui dépose le résultat au sommet de la pile pour l'instruction suivante. L'interpréteur fait ensuite avancer le pointeur de code à l'opcode suivant. Ainsi, sur une telle implémentation il est possible en modifiant le contenu du pointeur de pile d'utiliser une valeur appartenant à la frame de la méthode qui précède. Il est également possible de réaliser des sauts hors des méthodes et

potentiellement d'aller continuer l'exécution du code dans une autre méthode de la classe. Les tests qui permettent d'empêcher ces problèmes d'arriver, sont évalués par le vérificateur de bytecode, mais les attaques par injection de fautes ont la capacité de modifier le code après cette vérification.

L'interpréteur abstrait va prendre en entrée un fichier class, il va l'interpréter et le sauvegarder dans un objet *JavaClass*. Il va exécuter de façon abstraite une méthode c.-à-d. en simulant le déplacement du pointeur de code entre les différentes instructions, et la frame Java. Cette frame va contenir la pile des opérandes ainsi que la table des variables locales de la méthode qui est en cours d'interprétation. Ainsi, deux structures de données vont être nécessaires à cet outil : une pile (voir Code 17) et une table des variables locales (Code 18).

Code 17 Representation d'une pile en Java

```
class Stack {
  private int maxStackSize;
  private int top
  private ArrayList<Integer> stackElement;

  public Stack () {
    maxStackSize = -1;
    ArrayList<Integer> stackElement = null;
  }

  public Stack (int max) {
    maxStackSize = max;
    stackElement = new ArrayList<Integer> (max);
  }

  /*
  * Tous les outils nécéssaires à la gestion d'une pile
  * boolean empty (), void push (int elt), int pop (), etc.
  */
  ...
}
```

Code 18 Table des variables locales

```
class Locale {
  private int maxLocale;
  private ArrayList<Integer> localeElements;

  public Locale () {
   maxLocale = -1;
    ArrayList<Integer> localeElements = null;
  }

  public Locale (int max) {
    maxLocale = max;
    localeElements = new ArrayList<Integer> (max);
  }

  ...
}
```

Il fonctionne à la façon d'un interpréteur classique de machine virtuelle dont le comportement est expliqué dans la sous-section 2.3.1. Il y a une différence fondamentale entre un interpréteur classique et l'interpréteur abstrait. Cette différence se situe au niveau de l'interprétation des opcodes. En effet, dans l'interpréteur classique de la machine virtuelle les instructions sont réellement exécutées, tandis que dans l'interpréteur abstrait, une abstraction de l'exécution est faite.

L'interprétation abstraite du bytecode va consister à suivre le graphe d'appel de l'application, à maintenir le pointeur de code ainsi que la pile des opérandes pour chaque méthode d'une application donnée. L'interpréteur va créer pour chaque méthode une pile des opérandes ayant une taille maximale. La hauteur de cette pile sera diminuée et augmentée à mesure de l'interprétation et en fonction du nombre d'éléments que chaque instruction prend et dépose sur cette pile. L'interprétation se limite à ces deux opérations, c.-à-d. qu'on ne va pas réellement faire l'opération relative à l'instruction courante. Par exemple lors d'une instruction d'addition *iadd*, on diminue la hauteur de la pile, puis on l'augmente, mais on ne réalise pas l'addition.

En plus du comportement normal de l'interpréteur, j'ai rajouté certaines vérifications (que j'appellerai des contremesures systèmes) qui correspondent à une version défensive de la machine virtuelle (voir [Ler03]). Parmi elles, on peut citer :

- La vérification qu'une instruction *nop* (qui correspond à l'octet 0x00) n'apparait pas dans le code. Car dans le cas où la carte dispose de mémoire non chiffrée, nous avons vu qu'elles étaient les conséquences d'autoriser le *nop* dans le code de l'application (voir sous-section 4.4.1). Ainsi, si cette instruction est rencontrée dans un code, alors l'interprétation est immédiatement arrêtée. On peut supprimer cet opcode, car il n'est pas nécessaire de l'utiliser afin d'avoir un bytecode fonctionnel.
- La vérification de l'overflow et de l'underflow de la pile. Ce qui permet d'éviter les débordements de mémoire qui peuvent causer l'utilisation d'une variable locale au lieu d'un élément de la pile. Ou d'utiliser une référence à un objet manipulé par une méthode précédente et non nettoyée à la sortie de celle-ci.
- La vérification que lorsqu'une instruction utilise une variable locale, celle-ci existe bien dans la frame courante. Ce qui permet de s'assurer qu'on ne va pas utiliser une variable locale appartenant à une autre méthode, voir manipuler une adresse de retour, le contenu de la pile précédente avant l'appel, etc.
- La vérification que lorsqu'une instruction de branchement (quelle que soit sa nature) est rencontrée, le branchement se fait bien au sein même de la méthode et pas hors de celle-ci.
- La vérification que lorsqu'une instruction utilise une entrée du constant pool de la classe courante, alors cette entrée existe vraiment.

L'interpréteur abstrait permet également de simuler une injection de fautes. Pour cela, on dispose d'une classe *FaultModel* (voir le Code 20), qui va permettre de déterminer les paramètres de l'attaque, à savoir, le type de mémoire utilisé (chiffréé ou non chiffréé), la quantité de données que l'on souhaite modifier en une seule fois, le nombre de modifications que l'on souhaite effectuer. Ainsi, si l'on souhaite paramétrer une attaque suivant le modèle de fautes, on va construire un objet *FaultModel* en utilisant le constructeur FaultModel (true, true, 1), qui va permettre de modifier un octet précis du code dans le tableau d'octets représentant une méthode et de lui faire prendre une valeur aléatoire.

Cet interpréteur permet aussi de manipuler le code en activant à la demande une ou plusieurs contremesures systèmes grâce à l'objet *Countermeasures* (voir le Code 20)). Cet objet permet d'activer les contremesures systèmes ainsi que les contremesures que j'ai mises au point (voir le chapitre 5).

Ces éléments vont permettre de simuler le comportement de l'interpréteur face aux attaques en faute sur le code, mais aussi de pouvoir en déduire la résistance des contremesures face à ces attaques.

Code 19 Classe *faultModel* permettant de paramétrer le modèle de fautes

```
class FaultModel {
  private boolean faultType;// true= octet, false= bit
  private boolean cypherMemory; // true= cypher memory model, false = non cypher
      memory model
  private int numberOfbyte;// number of byte to modify

  public void FaultModel (boolean type, boolean memory, int fault) {
     ...
  }
  ...
}
```

Code 20 Classe *Countermeasures* permettant de paramétrer les contremesures

```
class Countermeasures {

  /* naturals countermeasures */
  private boolean nop;
  private boolean stackCheck;
  private boolean localVarCheck;
  private boolean jumpCheck;
  private boolean constantPoolCheck;

  /* elaborate countermeasures */
  private boolean fieldOfBit;
  private boolean basicBlock;
  private boolean pathCheck;
  private boolean Compression;
  ...
}
```

6.5 Générateur d'applications mutantes

6.5.1 Qu'est ce qu'une application mutante ?

Une application mutante est une application Java Card qui a subi une modification due à une attaque (en faute par exemple) et qui continue d'avoir du sens pour la machine virtuelle lors de l'interprétation. C.-à-d. que cette application ne cause aucun plantage de la machine virtuelle et qu'elle arrive à passer outre les mesures de sécurité qui lui sont intégrées. L'attaque détaillée dans la sous-section 4.4.1 est un parfait exemple de mutant dangereux pour la sécurité de la carte. La génération et la détection de mutant est un nouveau champ de recherche introduit par [BTG10 ; VF10].

6.5.2 Générateur de mutants

L'interpréteur abstrait permet de réaliser des attaques par injection de fautes à la demande en visant les octets voulus. Un mutant va correspondre à une injection de fautes sur un octet qui n'est détectée par aucune des protections systèmes. Le générateur de mutant est en fait une automatisation du processus d'attaque sur chaque octet composant une application.

Le générateur de mutants permet de modifier chaque octet du tableau d'octets composant la méthode en leur faisant prendre toutes les valeurs possibles entre 0x00 et 0xFF, en vérifiant l'impact de ces changements sur l'interprétation du code. À savoir s'il y a eu ou non une détection. S'il n'y a pas eu de détection alors le mutant correspondant à cette attaque est créé. Prenons la suite d'instructions qui se composent des quatre octets suivants : «A», «B», «C», et «D». Tout d'abord, on modifie l'octet A en lui faisant prendre les valeurs 0x00, 0x01, 0x02, 0x03..., 0xFF en excluant sa valeur initiale. On lance l'interprétation après chacune des modifications de l'octet A ; s'il n'y a pas de détection, le mutant correspondant est enregistré. Quand toutes les valeurs de l'octet A auront été testées, on passe à l'octet B en remettant A à sa valeur initiale. Lorsqu'on passe à l'octet C, on remet les valeurs initiales des octets A et B, ainsi de suite jusqu'aux derniers octets de la méthode. En somme, lorsqu'on traite l'octet d'indice i dans le tableau tous les octets allant du premier indice du tableau à l'indice $i-1$ sont remis à leurs valeurs initiales. Ainsi pour une méthode donnée, le nombre d'attaques réalisées va correspondre à $nombredinstructions \times 2$ (dans le cas de mémoires non chiffrées) ou $nombredinstructions \times 254$ (dans le cas de mémoires chiffrées).

Cette façon de procéder permet de générer tous les mutants possibles d'une application. Le générateur de mutant est donc un outil d'analyse de vulnérabilité des applications.

6.6 La plateforme de test

L'évaluation s'est déroulée sur une machine virtuelle Simple RTJ (voir la section 6.7), qui a été portée sur la carte d'évaluation d'Embest AT91EB40A (voir la Fig. 6.1) équipée d'un processeur AT91R4008 de type ARM7TDMI. Elle a une vitesse d'horloge interne par défaut de 32,768 MHZ, pouvant être au choix divisée par 2, par 4 ou par 8 en fonction des besoins. Au niveau mémoire, elle dispose de 2 Mb de flash et 256 kbits de RAM. Enfin, elle peut fonctionner au choix avec un

adressage mémoire 16 ou 32 bits. Elle offre aussi la possibilité d'ajuster la vitesse de fonctionnement des mémoires afin de simuler les mémoires lentes. Ses caractéristiques de processeur, et de mémoire font d'elle un périphérique très proche des cartes à puce communes d'aujourd'hui.

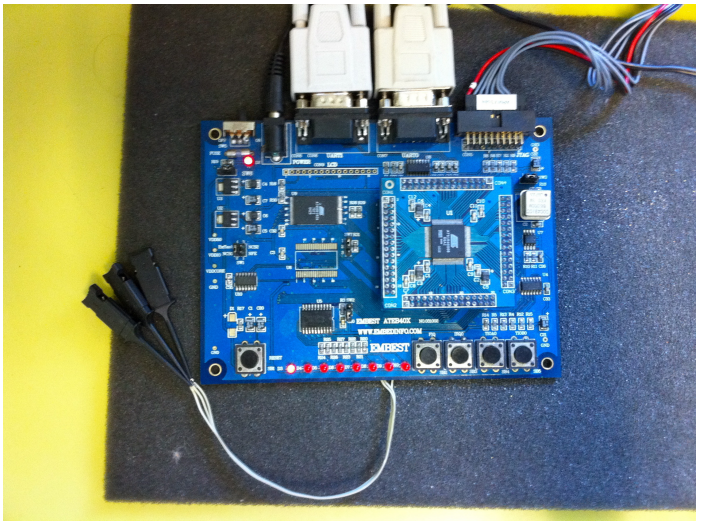

Fig. 6.1 – Carte d'evaluation AT91EB40A

6.7 Simple Real Time Java (SRTJ)

6.7.1 Présentation de SimpleRTJ

SimpleRTJ [Srt] est une machine virtuelle Java optimisée et conçue pour de petits objets embarqués basés sur des architectures processeur 8, 16 ou 32 bits avec une faible empreinte mémoire. Elle supporte le multithreading, les ramasse-miettes et les modèles mémoire linéaires (64Kb/16Mb) et banked (64kb). Elle utilise une technique d'édition de liens hors carte tout comme Java Card 2.x. Elle impose d'embarquer toutes les bibliothèques sur la plateforme ce qui permet de réduire le coût d'exécution et l'occupation mémoire à l'intérieur de la carte. Comme cette édition des liens est faite hors de la carte, le fichier binaire obtenu est quasi identique au fichier cap utilisé par la carte à puce.

6.7.2 Structure de SimpleRTJ

Type de données manipulées

SimpleRTJ a été conçu de façon modulaire et gère les types de données standards de Java, cette modularité permet avec des paramètres de compilation de la machine virtuelle de supprimer certains types de données qui ne sont pas désirés comme la manipulation des nombres à virgule flottante. Cette remarque est importante, car afin de pouvoir être le plus proche possible de Java Card, il est nécessaire de supprimer les nombres à virgule flottante, car ils ne sont pas supportés. Outre cela, SimpleRTJ manipule les types de données suivants :

– Les types primitifs tels que les nombres entiers non signés (uint8, uint16 et uint32), et les booléens,

– Les références permettant de stocker les références d'instance de classe et de tableau. Sans oublier qu'elle supporte les types composés tels que les tableaux de types primitifs (valeur et booléens) et les tableaux de références.

Modèle mémoire

SimpleRTJ utilise deux espaces de stockage, la mémoire volatile dans laquelle l'exécution des applications est faite, et la mémoire persistante où sont stockés le code de la machine virtuelle, le code de démarrage et les applications. Ces deux espaces permettent donc de stocker d'une part les données nécessaires au fonctionnement de la machine virtuelle et des applications et d'autre part les structures de données nécessaires à l'exécution des applications (voir la Fig. 6.2). Le modèle mémoire est très proche de celui que l'on peut retrouver dans une carte à puce de type Java Card standard. Ce qui conforte le choix d'utiliser cette implémentation de machine virtuelle.

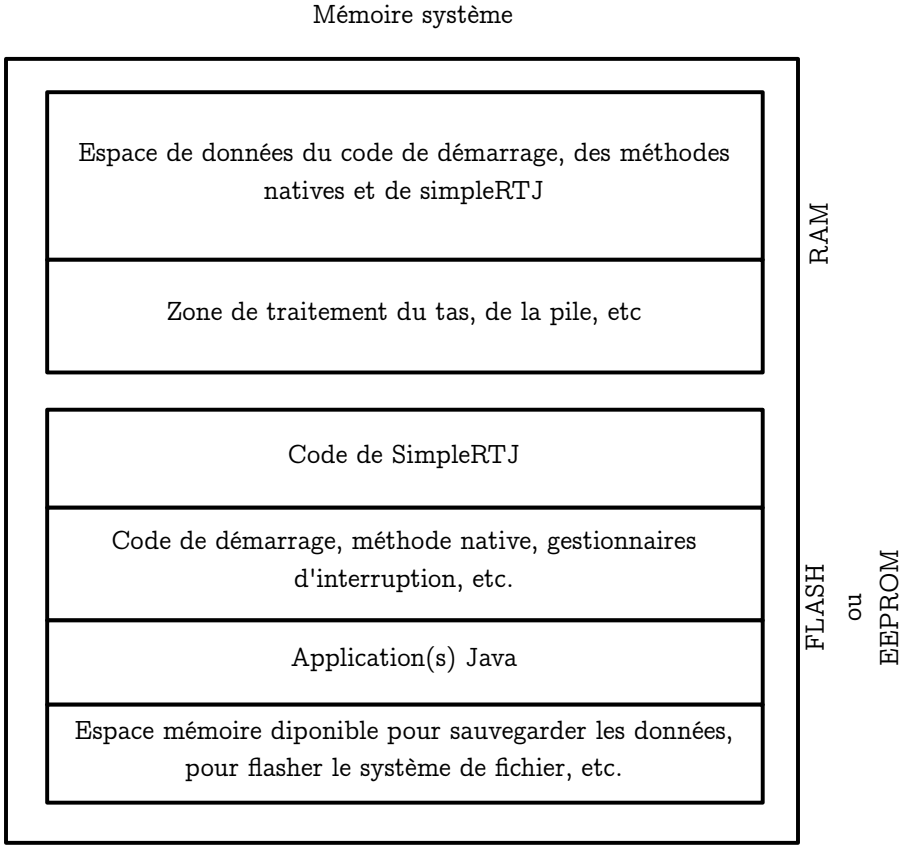

Fig. 6.2 – Modèle mémoire de simpleRTJ

6.7.3 Modifications de SimpleRTJ

Les principaux fichiers de simpleRTJ modifiés sont les suivants :

– *javavm.h* : C'est le fichier qui contient les différentes structures de données nécessaires à la machine virtuelle

– *j_common.h* : C'est le fichier qui contient les variables globales.

- *jvm.h* : C'est le fichier qui contient les structures des données internes de simpleRTJ. C'est aussi dans ce fichier que j'ai intégré les structures de données contenant les informations des différents composants additionnels. Pour chacune des contremesures existantes, j'ai modifié la structure de données correspondant à la définition d'une méthode (*method_T* voir le Code 21), pour y ajouter les données des composants additionnels.
- *j_vm.c* : C'est le fichier qui contient le code de l'interpréteur. La majorité des modifications ont été apportées à ce code. En effet, la boucle principale d'interprétation des bytecodes d'une méthode se trouve dans ce fichier et correspond à la fonction *run*. C'est donc cette méthode que j'ai modifiée pour y introduire mes contremesures.

Code 21 Structure de données correspondant à la méthode dans simpleRTJ

```
struct method_T
{
    class_t *class_ptr;/* pointer to class containing this method */
    uint16  flags;     /* method flags */
    uint16  locals;    /* size of locals */
    uint16  hash;      /* hashcode calc. over meth. signature and name */
    uint16  nargs;     /* number of arguments this method takes (in 32-bit words) */
    uint16  blen_idx;  /* bytecodes length or index to native method lookup table */
    uint16  unused;    /* just to make header size multiple of four */
};
```

Dans la section suivante, j'explique à travers un exemple le type de modification apporté à l'interpréteur de la machine virtuelle dans le cas des blocs élémentaires.

Cas des blocs élémentaires

Dans le cas de la contremesure basée sur l'intégrité des blocs élémentaires, on ajoute une structure de blocs élémentaires, qui va contenir le PC de début, le PC de fin du bloc, ainsi que la valeur de contrôle correspondant à ce bloc. on ajoute à la structure *method_T*, un tableau contenant l'ensemble des blocs appartenant à la méthode en cours. les modifications apportées se trouvent dans le Code 22.

Code 22 Structure de données : blocs élémentaires

```
struct basicbloc_T
{
    uint8 begin;
    uint8 end;
    uint8  xor;
}

struct method_T
{
    class_t *class_ptr;/* pointer to class containing this method */
    uint16  flags;     /* method flags */
    uint16  locals;    /* size of locals */
    uint16  hash;      /* hashcode calc. over meth. signature and name */
    uint16  nargs;     /* number of arguments this method takes (in 32-bit words) */
    uint16  blen_idx;  /* bytecodes length or index to native method lookup table */
    basicbloc_T *bb    /* the field of bit for the current method */
};
```

J'ai également modifié le code de la fonction *run* (voir le Code 23). Cette fonction est celle qui contient la boucle principale d'interprétation du bytecode d'une méthode donnée.

Code 23 Modifications relatives aux blocs élémentaires (simpleRTJ)

```
bool run (method_t *method)
{
   /*
  Do the initialisation stuff
  */
  ...
  while (true) {
    bc_item = NULL;
    /* Add by Ahmadou SERE*/
    vm_pc_sav = vm_pc;/*Saving the pc position*/
    xor_value+= *vm_pc;

    /*End Add AAS*/
    handler = bytecode_table[*vm_pc];

    vm_pc++;
    /* Add by Ahmadou SERE*/
    vm_pc_secure++;
    /*End Add AAS*/
    bc_action = handler();

    /* Add by Ahmadou SERE*/

    vm_pc_secure+= (vm_pc - vm_pc_sav);
    for ( cpt=1; cpt<= (vm_pc - vm_pc_sav) ;  cpt++ ) {
            xor_value += vm_pc_sav [cpt];
    }

    if (exec_method->bb != NULL) {
      /*verifying the bloc */
      if (vm_pc_secure - (vm_pc - vm_pc_sav) == current_bloc.end &&
                belong_to (vm_pc_sav) ) {
        /*
         * if we have reach the end of the block, verifying that block end
         * is an control flow instruction before continuing
         */
         if (xor_value != current_bloc.xor) {
            xor_value = 0;
            return false;
                }
         } else {
           return false;
         }
    }
    /*
    do the stuff related to instruction handle results
    like throwing an exception, invoking another method,
    etc. and verifying that we have reach the end of the
    method,
    */

  /*End Add AAS*/
  ...
  }
  return false;
}
```

6.8 Résultats

6.8.1 Génération de mutants

Méthodologie d'évaluation

La procédure qui m'a permis d'évaluer les contremesures consiste à utiliser le générateur de mutants dans plusieurs modes de fonctionnement qui sont les modes :

- Sans protections : le code est interprété sans aucune vérification.
- Avec les protections systèmes : le code est interprété en vérifiant que les branchements se font bien toujours à l'intérieur d'une méthode, que les variables locales utilisées appartiennent bien à la frame courante.
- Avec la vérification de la pile : le code est exécuté avec les protections systèmes en vérifiant à chaque instruction qu'il n'y a pas de dépassement de pile.
- Avec les contremesures activées : le code est interprété avec une des contremesures au choix : champs de bits, blocs élémentaires, vérification de chemins.

Cette procédure permet d'obtenir le nombre d'applications mutantes générées dans chaque modes de fonctionnement et d'observer la variation du nombre de mutants généré en fonction des contremesures. Cela permet de juger l'efficacité de celles-ci.

Réduction du nombre de mutants

Les résultats des différentes contremesures sur une partie des applications testées sont consignés dans le Tableau 6.1 qui contient le nombre de mutants détecté en fonction de différents modes d'exécution. La colonne "protection PT" correspond à une contremesure en cours de dépôt de brevet qui n'est par conséquent pas exposé dans ce manuscrit. Les trois dernières colonnes du tableau à savoir : Champs de bits, Blocs élémentaires et Chemins, correspondent aux résultats obtenus lorsqu'on active les contremesures développées.

Les applications prisent en exemple sont :

- Wallet qui contient 470 instructions et une seule classe
- Utilities qui contient 2344 instructions et plusieurs classes
- TeaApplet qui contient 1500 instructions et une seule classe

Applications	Sans protections	Protections Systèmes	vérifications de la pile	Protection PT	Champ de bits	Blocs élémentaires	Chemins
Wallet	440	434	54	23	18	0	37
Utilities	2740	2226	374	169	157	0	230
TeaApplet	1944	1916	382	140	95	0	163

Table 6.1 – Nombre d'applications mutantes en fonction des contremesures

Ce tableau montre l'efficacité de chaque contremesure par rapport à la détection des applications mutantes. La contremesure la plus efficace est celle des blocs élémentaires qui permet de détecter de façon systématique l'apparition d'un mutant. Car cette technique utilise une valeur de contrôle qui correspond à l'ensemble des octets composant le bloc.

La technique du champ de bits échoue à détecter les modifications dans le cadre des instructions indistinctes, par contre la détection des modifications est quasi immédiate. Cela s'explique par le fait que lorsqu'une instruction est remplacée par une instruction indistincte, il n'y a aucun changement du nombre d'opérandes de la méthode

La technique des chemins donne aussi de bons résultats en ce qui concerne la détection, car en cas d'attaque causant une rupture du flot de contrôle la détection est systématique. Dans ce cas précis, la détection se fait uniquement lorsque l'attaque perturbe le flux de contrôle ce qui n'est pas toujours le cas. Cette technique permet surtout de s'assurer que les tests cruciaux comme la vérification des clefs cryptographiques et celle des mots de passe utilisateurs, ne sont pas évités durant l'exécution.

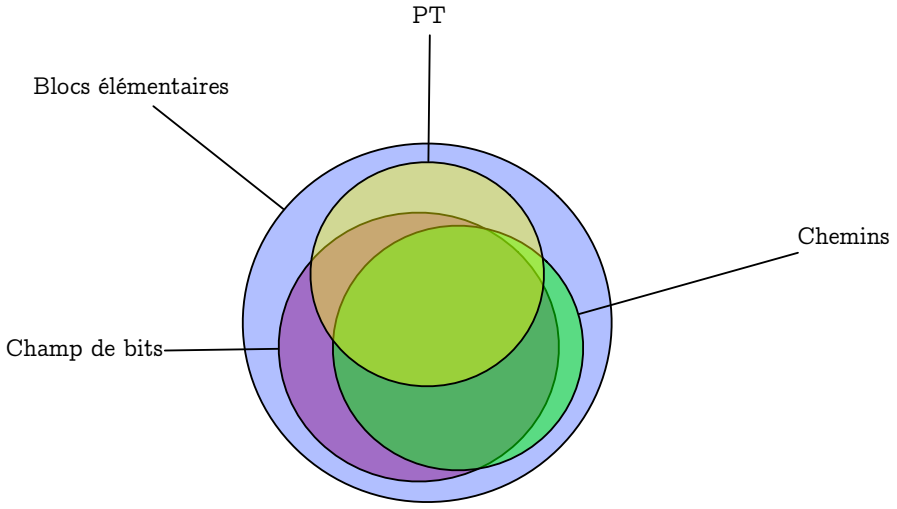

Fig. 6.3 – Repartitions de la détection des mutants entre contremesures

Le diagramme de la Fig. 6.3 montre comment se repartit la détection des mutants en fonction des contremesures. L'intersection entre deux cercles montre qu'il y a des mutants détectés par les deux contremesures qui représentent ces cercles. On peut constater que la contremesure des blocs élémentaires détecte tous les mutants y compris ceux qui sont détectés par les autres contremesures, de fait, elle englobe toutes les autres contremesures. La contremesure du champ de bits détecte une bonne partie des mutants détectés par la contremesure PT, et ceux détectés par la contremesure de vérification des chemins. On constate également que (1) environ 43 % des modifications par des instructions indistinctes (cas non détecté par la technique du champ de bits) sont détectées, par la contremesure basée sur les chemins. (2) De même, 76 % des attaques modifiant des instructions qui n'influencent pas le flux de contrôle (cas non détecté par la vérification des chemins) sont détectées par la technique du champ de bits. Ainsi la proportion des modifications du cas (2) détecté est plus importante que celle du cas (1). On peut donc dire qu'elles sont complémentaires.

On voit que les contremesures proposées permettent de fortement atténuer le nombre de mutants générés par une attaque. En effet, dans le cas général c.-à-d. pour l'ensemble des 20 applications

testées dont je dispose, on obtient une réduction du nombre de mutants de 91,6 % dans le cas de la vérification des chemins, de 92,8 % dans le cadre du champ de bit, et de 100 % dans le cas des blocs élémentaires. On a donc un très fort taux de détection des attaques.

Latences

La latence est le nombre d'instructions exécuté entre le moment où l'injection de fautes a lieu et le moment de sa détection. C'est une mesure très importante en ce qui concerne l'évaluation des mécanismes de sécurité dans la carte. En effet, si une latence élevée est enregistrée, il se peut que des instructions problématiques soient exécutées durant l'intervalle de temps qui sépare l'attaque de la détection. Comme l'exécution d'une ou plusieurs instructions qui peuvent modifier des objets persistants tels que les instructions de type *putfield*, *putstatic* et les instructions de type *invoke*.

Lorsqu'une instruction modifie un champ appartenant à un objet persistant, le champ en question sera manipulé lors des futures utilisations de la carte ou du moins jusqu'à ce qu'il soit à nouveau modifié. Pour cette raison, une contremesure efficace doit détecter un maximum de mutants, mais elle doit aussi avoir la latence la plus faible possible.

Dans le cas où des instructions problématiques apparaissent avant la détection, le générateur de mutants permet de voir la ou les instructions correspondantes en sauvegardant des informations facilitant l'analyse, avec :

- Dans le cas d'une instruction de type *invoke* des informations telles que le nom de la méthode, sa signature, la classe à laquelle elle appartient.
- Dans le cas d'une instruction de modification de champ d'un objet persistant des informations telles que le champ, le type du champ, l'objet auquel appartient le champ

Ces informations permettent de juger de la pertinence de mettre en place des protections purement applicatives pour gérer ces cas particuliers, telles que de la redondance de données ou de calculs.

Les latences enregistrées dans le Tableau 6.2 vont correspondre aux moyennes des latences pour chaque mode d'exécution du générateur de mutants. Dans le cas où aucune protection n'est activée, aucune latence n'est enregistré car aucune détection n'est faite.

Applications	Sans protections	Protections Systèmes	vérifications de la pile	Protection PT	Champ de bits	Blocs élémentaires	Chemins
Wallet	-	2	2,91	2,92	2,42	2,72	3
Utilities	-	106,77	14,4	13,56	8,61	10,53	13,06
TeaApplet	-	56,86	9,22	8,42	5,82	7,66	11,72

Table 6.2 – Latence en fonctions des contremesures

Dans le cas général (sur toutes les applications dont je dispose), les contremesures développées offrent une latence moyenne très basse : celle du champ de bits est de 5.5, celle des blocs élémentaires est de 6.97, et celle des chemins de 9.26. Si on les oppose à celles obtenues avec les contremesures systèmes dont la latence moyenne la plus basse est de 20,78 (protection système avec vérification

de pile) et la plus haute de 133,10 (protection système sans vérification de pile), cela confirme l'efficacité des contremesures développées.

6.8.2 Occupation mémoire et vitesse d'exécution

Méthodologie d'évaluation

Les résultats obtenus consignent les informations obtenues lors de l'évaluation sur la carte d'évaluation. En effet, on retrouve pour chacune des contremesures l'augmentation en terme de pourcentage du temps d'exécution (correspondant à la colonne de vitesse d'exécution), celle de la taille de l'application (colonne EEPROM), ainsi que celle de la taille de l'interpréteur après les modifications pour chacune des contremesures (colonne ROM). J'ai calculé le temps d'exécution en utilisant un analyseur logique (Fig. 6.4) connecté à la carte d'évaluation. Cet analyseur est précisément connecté à deux ports de sortie programmables et permet de récupérer un signal lorsque de l'énergie transite au sein de ces ports. Pour réaliser la mesure du temps d'exécution, on installe la machine virtuelle originale sur la carte d'évaluation, et on exécute un programme normal (sans aucune de mes contremesures). Au tout début et à la fin de l'interprétation, on ajoute une instruction qui active le premier port et on récupère les signaux au niveau de l'analyseur logique. Puis on exécute le programme un grand nombre de fois (1000 fois) pour obtenir la moyenne en terme de temps d'exécution. on répète les mêmes opérations respectivement pour chacune des contremesures avec la machine virtuelle modifiée. En faisant la différence entre le temps d'exécution obtenu pendant la simulation avec les contremesures activées et le temps obtenu pour une simulation normale, on obtient l'augmentation de la vitesse d'exécution. Ensuite, on calcule le rapport entre cette différence et le temps mis pour une simulation normale. Cela permet d'obtenir le ratio en terme d'augmentation du temps d'exécution.

Fig. 6.4 – Analyseur Logique

Résultats Obtenus

Les résultats consignés dans le Tableau 6.3 sont des moyennes obtenues sur plusieurs simulations et sur différentes applications. En appliquant la méthodologie vue précédemment.

Protection	Vitesse d'exécution	EEPROM	ROM
Champ de bits	$\approx +3\%$	$\approx +3\%$	+1%
Bloc élémentaire	$\approx +5\%$	Variable (0 à 5%)	+1%
chemins	$\approx +8\%$	Variable (0 à 10%)	+1%

Table 6.3 – Occupation mémoire et vitesse d'exécution

Dans le cas du champ de bits, on constate que la diminution de la vitesse d'exécution est très faible, cela est dû au fait qu'il n'y a qu'une simple comparaison à réaliser entre deux valeurs. Pour ce qui est de l'augmentation de la taille du fichier class, elle est fixe, car elle est directement fonction de la taille du code, cette valeur est faible à cause de la compression qui est appliquée sur le champ de bits. Cette compression influe aussi sur le temps d'exécution, car il y a une opération de masquage à faire afin de récupérer la valeur du champ de bits qui correspond au PC courant. Les performances en terme de temps d'exécution pourraient être améliorées. Pour cela, il faudrait ne pas utiliser de compression. Par contre, ce gain de performance se ferait au détriment de l'augmentation de la taille du code.

Dans le cas des blocs élémentaires, on constate que l'augmentation de la vitesse d'exécution est proche de ce que l'on peut avoir pour le mécanisme du champ de bits. Car pour les blocs élémentaires, le nombre d'opérations requises lorsque ce mécanisme est activé n'est pas élevé (calculs des valeurs de contrôle, ainsi que leur comparaison). En ce qui concerne la taille de l'application, l'augmentation est ici variable et dépend directement du nombre de blocs élémentaires de l'application.

Dans le cas des chemins, le temps d'exécution des applications lorsque la contremesure est activée est plus important que dans les deux cas précédents, car on a une opération de recherche dans la table contenant tous les chemins, ainsi que le calcul du chemin en lui-même pendant l'exécution, ainsi qu'une opération de comparaison de tous les éléments appartenant au chemin calculé. Pour ce qui est de l'augmentation de la taille de l'application, elle est aussi variable, car elle dépend du nombre de chemins contenus dans l'application.

Dans les trois cas, on constate que l'augmentation de la taille de l'interpréteur est de l'ordre de 1 %, ce qui montre bien le peu de modifications dont il fait l'objet. La raison pour laquelle j'ai insisté sur la taille de l'EEPROM et sur la vitesse d'exécution lors de la collecte des métriques est que ce sont les ressources les plus critiques. Il faut noter que ces métriques sont données à titre indicatif pour montrer que les contremesures développées sont très économes en terme de ressources même lorsqu'elles sont activées pour toutes les méthodes d'une application. Car il n'est pas tout le temps utile de protéger tout le code, mais uniquement les méthodes critiques de celle-ci. De fait, l'impact des contremesures proposées sur les performances dépend du nombre de fois où elles sont activées pour une application donnée.

6.9 Conclusion

Ce chapitre a permis d'aborder le sujet de l'évaluation des mécanismes de protection mis au point durant mon travail de thèse. Afin de pouvoir obtenir les métriques du chapitre, j'ai eu à mettre au point plusieurs outils. Un analyseur statique qui permet de créer des fichiers class contenant des composants additionnels. Un interpréteur abstrait qui a permis de faire une simulation des attaques

par injection de fautes et d'obtenir les taux de détection des mécanismes. Un générateur de mutants qui a permis d'évaluer la capacité de détection des contremesures. L'évaluation a aussi été l'occasion de prendre en main certains outils tels que BCEL, et d'être confronté aux problématiques de programmation dans le domaine de l'embarqué : contraintes de ressources mémoires et processeurs. En terme de détection, chacune des contremesures a des qualités et des faiblesses. Le choix de l'une ou l'autre dépend fortement des contraintes de vitesse d'exécution, d'occupation mémoire, et des besoins de sécurité des plateformes sur lesquelles elles doivent s'exécuter. L'association d'une faible latence, d'un fort taux de détection et d'une utilisation minimale des ressources fait des contremesures développées des protections idéales pour lutter contre les attaques par injection de fautes.

Troisième partie

Conclusions et perspectives

Conclusions et perspectives

Conclusion

Les attaques par injection de fautes sont une catégorie d'attaques très puissantes, qui permet l'obtention de secrets contenus dans la carte à savoir les clefs cryptographiques. Elles permettent aussi d'accéder à différents traitements protégés, tels que faire un débit alors qu'il n'était pas autorisé, de falsifier un document d'identité (passeport, carte d'identité, permis de conduire...). De la façon dont je l'expose dans ce manuscrit, il peut paraitre simple de réaliser une telle attaque, mais dans la réalité, réaliser une attaque sur les composants modernes n'est pas une chose simple à faire, car ces composants sont en général munis de mécanismes de protections matériels rendant très compliqué la mise au point d'une attaque. Afin de pouvoir réaliser une attaque par injection de fautes réussie, il faut de bonnes connaissances du composant que l'on veut attaquer et disposer du matériel adéquat pour pouvoir faire l'analyse des résultats ce qui n'est pas donné aux communs des mortels. Malgré toutes ces restrictions, un attaquant avec suffisamment de connaissances, de moyens et de motivation peut arriver à contourner ces sécurités matérielles et réussir une attaque par injection de faute.

De fait, il est important de bénéficier de protections logicielles qui permettent de détecter, les perturbations dont peut faire l'objet la carte qui est attaquée. D'où l'utilité de mon travail. Mes travaux de recherches ont permis de mettre au point des techniques de protection logicielle adaptées aux cartes à puce. Ces techniques ne se focalisent pas sur un algorithme en particulier et sont donc applicables à n'importe quel type de programme. Elles sont adaptées parce qu'elles fournissent des moyens simples de pouvoir sécuriser les applications développées sur la plateforme Java Card. Les contremesures que j'ai pu exposer dans ce mémoire ont les avantages suivants :

- Elles sont respectueuses des ressources de la plateforme sur laquelle elles s'exécutent (voir le Tableau 6.3), parce qu'elles ont été pensées dès le départ pour ne pas consommer trop de ressources processeur et mémoire. Je peux donc affirmer qu'elles sont peu coûteuses.
- Elles sont respectueuses des spécifications Java Card. En effet, j'utilise une approche qui permet d'obtenir des fichiers class en totale adéquation avec la spécification Java Card telle que fournie par Sun car elles utilisent le principe des composants additionnels.
- Elles sont simples à utiliser pour le développeur d'application (la section A.4 illustre bien le problème), car il n'a pas à se soucier des mécanismes de sécurité de la plateforme qui peuvent être parfois complexes. Il n'a pas à suivre de guide de programmation lui imposant un style de codage en particulier, la seule chose qu'il a à faire c'est de marquer les méthodes qu'il souhaite protéger. Ces règles de programmation pouvant conduire très souvent à considérablement augmenter la taille finale des applications.

- Elles nécessitent peu de travail pour être adaptées à l'interpréteur de la machine virtuelle. En effet, il faut juste modifier la boucle principale d'interprétation du code afin de les intégrer à la machine virtuelle.
- Elles permettent d'avoir de la portabilité entre plateformes. Car aujourd'hui il existe plusieurs fournisseurs de machines virtuellse et chacun d'eux conçoit la machine virtuelle à sa façon, ce qui entraine, une hétérogénéité au niveau des mécanismes de sécurité, car chaque machine virtuelle a ses particularités. Ainsi si sur une carte on a une machine virtuelle provenant d'un fournisseur,on aura certaines règles de sécurité à suivre qui sont probablement différentes chez un autre fournisseur.
- Elles permettent également de se protéger contre les attaques logicielles qui ont le même comportement que les attaques en fautes décrites dans ce mémoire.

Ce travail m'a permis de comprendre que la notion de sécurité dans un système est un compromis entre performance et qualité de la détection. Ceci est encore plus vrai dans le domaine des cartes à puce où les ressources sont très limitées. Et où les exigences en terme de performance sont très fortes. Dans ce contexte, il est très difficile de concevoir des mécanismes de sécurité efficaces, mais cela n'est pas impossible. C'est également un travail en perpétuel évolution, car de nouvelles failles sont fréquemment trouvées.

Perspectives

Notre travail a eu pour but de mettre au point des contremesures pour cartes à puce et de les évaluer. Pour mesurer leur performance, je me suis servis d'une machine virtuelle Java portée sur une plateforme très proche de la carte à puce. Le choix de cette machine virtuelle est dû au fait qu'il n'existe actuellement pas de machine virtuelle Java card disponible dans le domaine public que nous pourrions modifier à volonté. Donc un des travaux que nous avons entrepris est de porter l'API Java card à cette implémentation de machine virtuelle. Cela permettrait de faire des tests de performance sur de vraies applications Java Card afin de valider nos résultats. Mais encore mieux, nous disposerions d'une plateforme libre pour tous, ce qui permettrait de faciliter la vie des chercheurs qui souhaitent bénéficier d'une telle plateforme.

Le générateur de mutants permet de révéler certaines modifications qui pourraient être dangereuses pour le code et donc de déterminer les faiblesses d'une application. Ces faiblesses permettraient de déterminer un guide de bonnes pratiques utile à la création d'une application Java Card. Ce guide devant contenir les pièges qui pourraient conduire à un code moins sécurisé et expliquant les façons de les éviter.

L'objectif de compression était d'utiliser un code correcteur afin de détecter le changement (voir la sous-section 5.2.3).J'ai aussi expliquer que le modèle de faute actuel ne permettait pas d'utiliser un tel code, car si on modifie un octet c'est potentiellement tous les bits qui sont modifiés. Or la force d'un code correcteur est d'être capable de détecter un changement dans un petit nombre de bits. Cette idée prend tout son sens si on passe sur un modèle de fautes avec la modification d'un bit précis. Car dans ce cas, le code correcteur permettrait au pire de faire une détection de l'erreur et au mieux de faire une correction de l'erreur. Or, avec les avancées en matière d'électronique, il n'est pas impossible qu'un attaquant puisse aisément modifier un bit précis. C'est pour cette raison qu'il faut continuer sur cette piste afin aussi de régler les problèmes de la méthode de compression à savoir les méthodes non compressibles.

Annexe A

Annexes

A.1 Le code java de la méthode de débit de l'application porte-monnaie électronique

```
private void debit(APDU apdu) {

        // access authentication
    if ( pin.isValidated() ) {
                byte[] buffer = apdu.getBuffer();

        byte numBytes =
          (byte)(buffer[ISO7816.OFFSET_LC]);

                byte byteRead =
                (byte)(apdu.setIncomingAndReceive());

                if ( ( numBytes != 1 ) || (byteRead != 1) )
                ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

                // get debit amount
                byte debitAmount = buffer[ISO7816.OFFSET_CDATA];

            // check debit amount
                if ( ( debitAmount > MAX_TRANSACTION_AMOUNT)
                        || ( debitAmount < 0 ) )
                        ISOException.throwIt(SW_INVALID_TRANSACTION_AMOUNT);

         // check the new balance
                if ( (short)( balance - debitAmount ) < (short)0 )
                        ISOException.throwIt(SW_NEGATIVE_BALANCE);

                balance = (short) (balance - debitAmount);

        } else {

                ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
```

```
        }
} // end of debit method
```

A.2 Le bytecode de la méthode de débit de l'application porte-monnaie électronique

```
 0 aload_0
 1 getfield #4 <fr/xlim/ssd/wallet/Wallet.pin>
 4 invokevirtual #18 <javacard/framework/OwnerPIN.isValidated>
 7 ifeq 98 (+91)
10 aload_1
11 invokevirtual #11 <javacard/framework/APDU.getBuffer>
14 astore_2
15 aload_2
16 iconst_4
17 baload
18 istore_3
19 aload_1
20 invokevirtual #19 <javacard/framework/APDU.setIncomingAndReceive>
23 i2b
24 istore 4
26 iload_3
27 iconst_1
28 if_icmpne 37 (+9)
31 iload 4
33 iconst_1
34 if_icmpeq 43 (+9)
37 sipush 26368
40 invokestatic #13 <javacard/framework/ISOException.throwIt>
43 aload_2
44 iconst_5
45 baload
46 istore 5
48 iload 5
50 bipush 127
52 if_icmpgt 60 (+8)
55 iload 5
57 ifge 66 (+9)
60 sipush 27267
63 invokestatic #13 <javacard/framework/ISOException.throwIt>
66 aload_0
67 getfield #20 <fr/xlim/ssd/wallet/Wallet.balance>
70 iload 5
72 isub
73 i2s
74 ifge 83 (+9)
77 sipush 27269
80 invokestatic #13 <javacard/framework/ISOException.throwIt>
83 aload_0
84 aload_0
85 getfield #20 <fr/xlim/ssd/wallet/Wallet.balance>
88 iload 5
90 isub
91 i2s
```

```
 92 putfield #20 <fr/xlim/ssd/wallet/Wallet.balance>
 95 goto 104 (+9)
 98 sipush 25345
101 invokestatic #13 <javacard/framework/ISOException.throwIt>
104 return
```

A.3 Les différents composants additionnels

A.3.1 Composant additionnel champ de bits méthode débit

```
index dans le constant pool = 467;
taille = 14;
champs de bits = {201 51 249 185 100 158 169 73 50 228 156 185 36 128;}
```

Composant additionnel bloc élémentaire méthode débit

```
index dans le constant pool = 678;
taille= 9;
basic_bloc_info ={
 BB#0 Begin = 0 End = 7 XOR = -89
 BB#1 Begin = 10 End = 28 XOR = 105
 BB#2 Begin = 31 End = 34 XOR = -118
 BB#3 Begin = 37 End = 52 XOR = 59
 BB#4 Begin = 55 End = 57 XOR = -116
 BB#5 Begin = 60 End = 74 XOR = -68
 BB#6 Begin = 77 End = 95 XOR = 10
 BB#7 Begin = 98 End = 101 XOR = -53
 BB#8 Begin = 104 End = 104 XOR = -53
}
```

A.3.2 Composant additionnel chemin méthode process

```
index dans le constant pool = 655;
taille= 13;
basic_bloc_info ={
 PC = 0   nombre de bit utile = 2  chemin={0x04}
 PC = 10  nombre de bit utile = 3  chemin={0x04}
 PC = 31  nombre de bit utile = 4  chemin={0x04}
 PC = 37  nombre de bit utile = 4  chemin={0x05}
 PC = 37  nombre de bit utile = 5  chemin={0x40}
 PC = 43  nombre de bit utile = 5  chemin={0x48}
 PC = 55  nombre de bit utile = 6  chemin={0x48}
 PC = 60  nombre de bit utile = 6  chemin={0x4C}
 PC = 66  nombre de bit utile = 7  chemin={0x4A}
 PC = 77  nombre de bit utile = 8  chemin={0x4A}
 PC = 83  nombre de bit utile = 8  chemin={0x4B}
 PC = 98  nombre de bit utile = 4  chemin={0x07}
 PC = 104 nombre de bit utile = 10 chemin={0x13D}
}
```

A.4 Une méthode protégée par des contremesures applicatives

La méthode qui suit correspond à la méthode de vérification du PIN d'une applet. Cette méthode est non protégée.

```
boolean verify (byte[] buffer, short ofs, byte len)
{
    // No comparison if PIN is blocked
    if (triesLeft < 0)
      return false ;
    // Main comparison
    for(short i=0; i < len; i++)
      if (buffer[ofs+i] != pin[i])
      {
        triesLeft-- ;
        authenticated[0] = false ;
        return false ;
      }
    // Comparison is successful
    triesLeft = maxTries ;
    authenticated[0] = true ;
    return true ;
}
```

La méthode qui suit c'est la même méthode implémentant un panel de contremesures applicatives pour luter contre le détournement du flux de contrôle en utilisant un compteur (*stepCounter*) qui permet de vérifier le passage de l'application par certains points du code, ou encore de vérifier qu'une attaque n'a pas modifié la valeur des booléens, en les codant de façon spécifique (*BOOL_TRUE* et *BOOL_FALSE*). C'est sans appel, le code est plus complexe et demande plus de temps à être développé.

```
public final static short BOOL_TRUE  = (short)0x5a5a ;
public final static short BOOL_FALSE = (short)0xa5a5 ;
short equal = BOOL_TRUE ;

boolean verify(byte[] buffer, short ofs, byte len)
{


    // Initializes the step counter
    short stepCounter = INITIAL_COUNTER ;
    // First checks the integrity of the variable
    byte tl = triesLeft ;
    stepCounter++ ;
    if (tl != (short)(~triesLeftBackup))
      takeCountermeasure() ;
    stepCounter++ ;
    // No comparison if PIN is blocked
    if (tl < 0)
      return false ;
    stepCounter++ ;
    // First decrements the number of remaining tries
    JCSystem.beginTransaction() ;
```

```
      triesLeft = --tl ;
      stepCounter++ ;
      triesLeftBackup++ ;
    JCSystem.commitTransaction() ;
    stepCounter++ ;
    // Verifies the new value
    if (triesLeft != (short)(~triesLeftBackup))
      takeCountermeasure() ;
    stepCounter++ ;
    // Main comparison
    short equal = BOOL_TRUE ;
    stepCounter++ ;
    for(short i=0; i < len; i++)
      equal = equal && (buffer[ofs+i] != pin[i]) ;
    stepCounter++ ;
    if (equal == BOOL_TRUE)
    {
      // Comparison is successful
      // Reset the remaining tries to the max
      stepCounter++ ;
      JCSystem.beginTransaction() ;
        triesLeft = maxTries ;
        triesLeftBackup = (byte)(~maxTries) ;
      JCSystem.commitTransaction() ;
      stepCounter++ ;
      // Verifies the new value
      if ( (triesLeft != (short)(~triesLeftBackup)) ||
           (triesLeft != triesLeftBackup) )
        takeCountermeasure() ;
      stepCounter++ ;
      authenticated[0] = true ;
      if (stepCounter == (short)(INITIAL_VALUE+11) )
        return true ;
    } else {
      // Comparison failed
      stepCounter++ ;
      authenticated[0] = false ;
      if (stepCounter == (short)(INITIAL_VALUE+9) )
        return false ;
    }
    // Should have returned at this point
    takeCountermeasure() ;
}
```

Avec la contremesure des chemins, on obtient le même résultat en rajoutant juste l'annotation qui va bien. On s'évite ainsi bien des tracas pour sécuriser son code.

```
@SensitiveType{
        sensitivity=SensitiveValue.INTEGRITY
        proprietaryValue=''PathCheck''
}
boolean verify (byte[] buffer, short ofs, byte len)
{
    // No comparison if PIN is blocked
    if (triesLeft < 0)
      return false ;
    // Main comparison
```

```
    for(short i=0; i < len; i++)
      if (buffer[ofs+i] != pin[i])
      {
        triesLeft--;
        authenticated[0] = false;
        return false;
      }
    // Comparison is successful
    triesLeft = maxTries;
    authenticated[0] = true;
    return true;
}
```

Bibliographie

[197] "IDENTIFICATION SYSTEM". Brev. US Patent 3,641,316. 1972.

[AEL05] M. Abadi, M.B.U. Erlingsson et J. Ligatti. "Control-Flow Integrity". Dans : *CCS'05 : proceedings of the 12th ACM Conference on Computer and Communications Security : November 7-11, 2005, Alexandria, Virginia, USA*. Citeseer. 2005, p. 340.

[AG01] M.L. Akkar et C. Giraud. "An implementation of DES and AES, secure against some attacks". Dans : *Cryptographic Hardware and Embedded Systems, CHES 2001*. Springer. 2001, p. 309–318.

[AGL03] M.L. Akkar, L. Goubin et O. Ly. "Automatic Integration of Counter-Measures Against Fault Injection Attacks". Dans : *Pre-print found at http ://www. labri. fr/-Perso/ly/index. htm* (2003).

[Aum+03] C. Aumuller et al. "Fault attacks on RSA with CRT : Concrete results and practical countermeasures". Dans : *Lecture Notes in Computer Science* (2003), p. 260–275.

[BDL97] D. Boneh, R.A. DeMillo et R.J. Lipton. "On the importance of checking cryptographic protocols for faults". Dans : *Lecture Notes in Computer Science* 1233 (1997), p. 37–51.

[BE+06] H. Bar-El et al. "The sorcerer's apprentice guide to fault attacks". Dans : *Proceedings of the IEEE* 94.2 (2006), p. 370–382.

[BG02] G. Bizzotto et G. Grimaud. "Practical Java Card bytecode compression". Dans : *Proceedings of RENPAR14/ASF/SYMPA*. Citeseer. 2002.

[BOS03] J. Blomer, M. Otto et J.P. Seifert. "A new CRT-RSA algorithm secure against Bellcore attacks". Dans : *Proceedings of the 10th ACM conference on Computer and communications security*. ACM New York, NY, USA. 2003, p. 311–320.

[BTG10] G. Barbu, H. Thiebeauld et V. Guerin. "Attacks on Java Card 3.0 Combining Fault and Logical Attacks". Dans : *Smart Card Research and Advanced Application, Cardis 2010* LNCS 6035 (2010), p. 148–163.

[CCD00] C. Clavier, J.S. Coron et N. Dabbous. "Differential power analysis in the presence of hardware countermeasures". Dans : *Cryptographic Hardware and Embedded Systems ?CHES 2000*. Springer. 2000, p. 13–48.

[Cla+00] L.R. Clausen et al. "Java bytecode compression for low-end embedded systems". Dans : *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22.3 (2000), p. 489.

[CM99] K.D. Cooper et N. McIntosh. “Enhanced code compression for embedded RISC processors”. Dans : *ACM SIGPLAN Notices* 34.5 (1999), p. 139–149.

[EJ01] D. Eastlake 3rd et P. Jones. *The Secure Hash Algorithm 1 (SHA1)*. United States, 2001. url : `http://portal.acm.org/ft_gateway.cfm?id=RFC3174&type=txt&coll=GUIDE&dl=GUIDE&CFID=92476858&CFTOKEN=23222905`.

[Fie+99] R. Fielding et al. *Hypertext transfer protocol–HTTP/1.1*. 1999.

[For] Java Card Forum. *Java Card Forum*. `http://www.javacardforum.org`.

[GAD05] KO GADELLAA. “Fault Attacks on Java Card (Masters Thesis)”. Master Thesis. Universidade de Eindhoven, 2005.

[Gie+10] B. Gierlichs et al. “Revisiting Higher-Order DPA Attacks”. Dans : *Topics in Cryptology-CT-RSA 2010* (2010), p. 221–234.

[Gir07] Christophe Giraud. “Attaques de cryptosystèmes embarqués et contre-mesures associées”. Thèse de doct. Université de Versailles Saint-Quentin-en-Yvelines, 2007.

[GK04] C. Giraud et E.W. Knudsen. “Fault attacks on signature schemes”. Dans : *Information Security and Privacy*. Springer. 2004, p. 478–491.

[Gro02] William Grosso. *Java RMI*. O’Reilly & Associates, 2002.

[GT04] C. Giraud et H. Thiebeauld. “A survey on fault attacks”. Dans : *Smart Card Research and Advanced Applications VI* (2004), p. 159–176.

[Gue97] Patrick Gueulle. *Cartes magnétiques et PC*. Ed. Techniques et Scientifiques Françaises, 1997.

[Gue98] Patrick Guelle. *Cartes à puces et PC*. Dunod, 1998.

[Ham50] R.W. Hamming. “Error detecting and error correcting codes”. Dans : *Bell System Technical Journal* 29.2 (1950), p. 147–160.

[Hem04] L. Hemme. “A differential fault attack against early rounds of (triple-) DES”. Dans : *Cryptographic Hardware and Embedded Systems-CHES 2004* (2004), p. 170–217.

[Hen01] Mike Hendry. *Smart Card Security and Applications*. Artech House, 2001.

[HL00] M.S. Hwang et L.H. Li. “A new remote user authentication scheme using smart cards”. Dans : *IEEE Transactions on Consumer Electronics* 46.1 (2000), p. 28–30.

[JG02] T.M. Jurgensen et S.B. Guthery. *Smart cards : the developer’s toolkit*. Prentice Hall PTR, 2002.

[JP10] B. Jacobs et E. Poll. “Biometrics and Smart Cards in Identity Management”. Dans : (2010).

[JPS05] M. Joye, P. Paillier et B. Schoenmakers. “On second-order differential power analysis”. Dans : *Cryptographic Hardware and Embedded Systems–CHES 2005* (2005), p. 293–308.

[Kim+10] M.J. Kim et al. “Design and Performance Evaluation of Binary Code Packing for Protecting Embedded Software against Reverse Engineering”. Dans : *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE. 2010, p. 80–86.

[KKS99] O. Kommerling, M. Khun et P Street. “Design principles for temper-resistant smartcard processor”. Dans : *USENIX Workshop on Smartcard Technology Proceedings*. Sous la dir. d’USENIX Association. Chicago Illinois, 1999.

[KQ07] C. Kim et J.J. Quisquater. “Fault attacks for CRT based RSA : New attacks, new results, and new countermeasures”. Dans : *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems* (2007), p. 215–228.

[Ler03] X. Leroy. “Java bytecode verification : algorithms and formalizations”. Dans : *Journal of Automated Reasoning* 30.3 (2003), p. 235–269.

[LH10] C.T. Li et M.S. Hwang. “An efficient biometrics-based remote user authentication scheme using smart cards”. Dans : *Journal of Network and Computer Applications* 33.1 (2010), p. 1–5.

[LW98] H. Lekatsas et W. Wolf. “Code compression for embedded systems”. Dans : *Proceedings of the 35th annual Design Automation Conference*. ACM. 1998, p. 516–521.

[Man03] S. Mangard. “A simple power-analysis (SPA) attack on implementations of the AES key expansion”. Dans : *Information Security and Cryptology ?ICISC 2002* (2003), p. 343–358.

[Mor76] R. Moreno. “Methods of data storage and data storage systems”. Brev. US Patent 3,971,916. 1976.

[Mor77] R. Moreno. “Data-transfer system”. Brev. US Patent 4,007,355. 1977.

[Mor78] R. Moreno. “Systems for storing and transferring data”. Brev. US Patent 4,092,524. 1978.

[Mor83] R.C.D. Moreno. “Method and apparatus for electrically connecting a removable article, in particular a portable electronic card”. Brev. US Patent 4,404,464. 1983.

[MS00] R. Mayer-Sommer. “Smartly analyzing the simplicity and the power of simple power analysis on smartcards”. Dans : *Cryptographic Hardware and Embedded Systems - CHES 2000*. Springer. 2000, p. 78–92.

[Myc03a] Sun Mycrosystems. *Java CardTM 2.2.1 Application Programming Interface*. Sun Microsystems. 2003.

[Myc03b] Sun Mycrosystems. *Java CardTM 2.2.1 Virtual Machine (JCVM) Specification*. Sun Microsystems. 2003.

[Myc09a] Sun Mycrosystems. *Java CardTM 2.2.1 Runtime Environment (JCRE) Specification*. Sun Microsystems. 2009.

[Myc09b] Sun Mycrosystems. *Java CardTM 3.0.1 Application Programming Interface Specification, Connected Edition*. Sun Microsystems. 2009.

[Myc09c] Sun Mycrosystems. *Java CardTM 3.0.1 Runtime Environment (JCRE) Specification, Connected Edition*. Sun Microsystems. 2009.

[Myc09d] Sun Mycrosystems. *Java CardTM 3.0.1 Servlet Specification, Connected Edition*. Sun Microsystems. 2009.

[Myc09e] Sun Mycrosystems. *Java CardTM 3.0.1 Specification, Classic Edition*. Sun Microsystems. 2009.

[Myc09f] Sun Mycrosystems. *Java CardTM 3.0.1 Virtual Machine (JCVM) Specification, Connected Edition*. Sun Microsystems. 2009.

[Ott05] M. Otto. "Fault attacks and countermeasures". Thèse de doct. University of Paderborn, 2005.

[PQ03] G. Piret et J.J. Quisquater. "A differential fault attack technique against SPN structures, with application to the AES and Khazad". Dans : *Cryptographic Hardware and Embedded Systems-CHES 2003* (2003), p. 77–88.

[Prone] Apache Jakarta Project. *Byte Code Engeneering Language (BCEL)*. June. url : `http://jakarta.apache.org/bcel/`.

[PS06] Sylvain Prevost et Kapil Sachdeva. "Application code integrity check during virtual machine runtime". Brev. 20060047955. 2006. url : `http://www.freepatentsonline.com/y2006/0047955.html`.

[QS01] J.J. Quisquater et D. Samyde. "Electromagnetic analysis (EMA) : Measures and counter-measures for smart cards". Dans : *Smart Card Programming and Security* (2001), p. 200–210.

[Riv92] R. Rivest. *The MD5 Message-Digest Algorithm*. United States, 1992. url : `http://portal.acm.org/ft_gateway.cfm?id=RFC1321&type=txt&coll=GUIDE&dl=GUIDE&CFID=92476661&CFTOKEN=41626815`.

[SA03] S.P. Skorobogatov et R.J. Anderson. "Optical fault induction attacks". Dans : *Lecture notes in computer science* (2003), p. 2–12.

[Sch91] C.P. Schnorr. "Efficient signature generation by smart cards". Dans : *Journal of cryptology* 4.3 (1991), p. 161–174.

[Ser+09] A.A. Sere et al. "Cartes à puce : Attaques et contremesures." Dans : *Proceedings of Majecstic'09*. 2009.

[SH07] J.M. Schmidt et M. Hutter. "Optical and em fault-attacks on crt-based rsa : Concrete results". Dans : *Proceedings of the Austrochip*. Citeseer. 2007, p. 61–67.

[Sha85] A. Shamir. "Identity-based cryptosystems and signature schemes". Dans : *Advances in cryptology*. Springer. 1985, p. 47–53.

[SICL09a] A.A. Sere, J. Iguchi-Cartigny et J.L. Lanet. "Automatic detection of fault attack and countermeasures". Dans : *Proceedings of the 4th Workshop on Embedded Systems Security*. ACM. 2009, p. 1–7.

[SICL09b] A.A. Sere, J. Iguchi-Cartigny et J.L. Lanet. "Évaluation de mécanismes de détection d'attaques en fautes sur le tas statique d'une Java Card". Dans : *Proceedings of CryptoPuce'09*. 2009.

[SQ02] D. Samyde et J.J. Quisquater. "Eddy current for for magnetic analysis with active sensor". Dans : *Smart Card Programming and Security (E-Smart)* (2002).

[Srt] RTJ Computing, 2010. url : `http://www.rtjcom.com/main.php?p=home`.

[Sta87] International Organisation for Standardization. *ISO7816*. Rap. tech. ISO, 1987.

[Tri+10] A. Tria et al. "When Clocks Fail : On Critical Paths and Clock Faults". Dans : *Smart Card Research and Advanced Application* (2010), p. 182–193.

[VF10] E. Vetillard et A. Ferrari. “Combined Attacks and Countermeasures”. Dans : *Smart Card Research and Advanced Application, Cardis 2010* LNCS 6035 (2010), p. 133–147.

[Wag04] D. Wagner. “Cryptanalysis of a provably secure CRT-RSA algorithm”. Dans : *Proceedings of the 11th ACM conference on Computer and communications security*. ACM New York, NY, USA. 2004, p. 92–97.

[Wea06] A.C. Weaver. “Secure sockets layer”. Dans : *Computer* 39.4 (2006), p. 88–90.

[wik10] wikipedia. *MULTOS*. 2010. url : `http://en.wikipedia.org/wiki/MULTOS`.

[YMH02] S.M. Yen, S. Moon et J.C. Ha. “Hardware fault attack on RSA with CRT revisited”. Dans : *Proceedings of the 5th international conference on Information security and cryptology*. Springer-Verlag. 2002, p. 374–388.

[Yos+97] Y. Yoshida et al. “An object code compression approach to embedded processors”. Dans : *Proceedings of the 1997 international symposium on Low power electronics and design*. ACM. 1997, p. 265–268.