N° Ordre ……. / FS / UMBB / 2012

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ M'HAMED BOUGARA DE BOUMERDES

Faculté des **S**ciences

Mémoire de Magister

Présenté par

M^{elle} HAMADOUCHE Samiya

Filière : Systèmes Informatiques et Ingénierie des Logiciels
Option : Spécification de Logiciels et Traitement de l'Information
(École Doctorale)

Étude de la sécurité d'un vérifieur de byte code et génération de tests de vulnérabilité

Devant le jury :

Mr. Mohamed AHMED NACER	Professeur USTHB	Président
Mme. Thouraya TEBIBEL	MCA ESI	Examinatrice
Mr. Ahmed AIT BOUZIAD	MCB UMBB	Examinateur
Mr. Mohamed MEZGHICHE	Professeur UMBB	Encadreur
Mr. Jean Louis LANET	Professeur UNILIM (France)	Co-encadreur

Année Universitaire : 2011/2012

Remerciements

Arrivée au terme de mon travail de Magister, il m'est particulièrement agréable d'exprimer ma gratitude et mes remerciements à tous ceux qui, par leurs connaissances, leur soutien et leurs conseils, m'ont aidé à sa réalisation.

Je tiens tout d'abord à remercier Monsieur Mohamed MEZGHICHE, Professeur à l'Université de Boumerdes, pour avoir accepté d'être mon encadreur, ainsi que pour m'avoir accueilli au sein de son laboratoire LIMOSE où il m'a assuré un cadre favorable de travail. Je le remercie pour son écoute attentive et ininterrompue, ses conseils avisés et sa rigueur scientifique qui m'ont été d'une aide précieuse pour avancer.

Mes plus grands remerciements s'adressent aussi à mon co-encadreur Monsieur Jean Louis LANET, Professeur à l'Université de Limoges, pour la confiance qu'il m'a accordé en me proposant ce passionnant sujet de recherche. Je le remercie profondément pour m'avoir fait découvrir le merveilleux monde des cartes à puce, pour sa disponibilité tout au long de ce travail, pour les précieuses discussions que j'ai eu l'occasion d'avoir avec lui et qui m'ont été si bénéfiques pour mener à bien mon travail.

J'adresse aussi mes vifs remerciements à Monsieur Mohamed AHMED NACER, Professeur à USTHB, de m'avoir fait l'honneur de présider mon jury de soutenance, à Madame Thouraya TEBIBEL, Maître de conférences à l'ESI, et Monsieur Ahmed AIT BOUZIAD, Maître de conférences à l'UMBB, d'avoir bien voulu examiner mon travail.

J'ai pu travailler dans un cadre particulièrement agréable, grâce à l'ensemble des membres de l'équipe Informatique de LIMOSE ainsi que mes collègues de post-graduation. Je les remercie tous pour leur bonne humeur, pour tous ces moments de rires et de détente inoubliables qu'on a pu partager. Et un merci spécial pour Sarah, Razika, Hamama, Warda et Soumia pour leur soutien amical.

Je tiens à remercier les membres de l'équipe SSD (Secure Smart Devices), du laboratoire Xlim de l'université de Limoges, avec qui j'ai eu grand plaisir à travailler, pour l'aide qu'ils m'ont porté d'une manière ou d'une autre afin de mener à bien ce travail de recherche. Plus particulièrement, un grand merci à Guillaume BOUFFARD pour sa collaboration et le temps qu'il a consacré pour répondre à mes questions et faire la relecture de ce présent mémoire.

Mes remerciements sincères, pleins de reconnaissance et de gratitude vont aussi à ma famille et tous mes amis qui, grâce à leur soutien moral, m'ont permis de surpasser les moments de doute et d'avancer droit vers mon objectif final.

À la mémoire de mon père
qui restera à jamais dans mes pensées,
et au fond de mon cœur,
lui qui m'a toujours soutenu
et tant attendu ce jour ...

A celle qui m'est la plus chère au monde,
qui a été à mes côtés dans tous les moments difficiles
par son amour et son encouragement,
que dieu te protège pour nous MAMAN

A mes très chères sœurs,
SABRINA, CHAHRAZED
et NARIMENE

Je dédie ce mémoire

Table des matières

Liste des Figures

Liste des Tableaux

Liste des Abréviations

AID	Application IDentifer
APDU	Application Protocol Data Unit
API	Application Programming Interface
ATR	Answer To Reset
CAD	Card Acceptance Device
CAP	Converted APplet
CISC	Complex Instruction Set Computer
COS	Card/Chip Operating System
CPU	Central Processing Unit
EEPROM	Electrecally, Erasable, Programmable Read Only Memory
GSM	Global System for Mobile Communications
HTTP	Hyper Text Transmission Protocol
ISO	International Standards Organization
JCRE	Java Card Runtime Environment
JCVM	Java Card Virtual Machine
JVM	Java Virtual Machine
MBT	Model-Based Testing
PIN	Personal Identification Number
PIX	Proprietary Identifier eXtension
PUK	PIN Unblocking Key
RAM	Random Access Memory
RID	Ressource IDentifier

RISC Reduced Instruction Set Computer

RMI Remote Method Invocation

ROM Read Only Memory

SIM Subscriber Identity Module

SUT System Under Test

SW Status Word

TAPDU Transport Application Protocol Data Unit

TLS Transport Layer Security

UML Unified Modeling Language

ملخص

تعتبر البطاقات الذكية أجهزة جد آمنة لتنفيذ التطبيقات و حفظ المعلومات. لذا يتوجب عليها مواجهة محاولات الدخول العدائية التي تقصد المهاجمة. و ذلك بتأمين سلامتها الخاصة من خلال احتواء عدة آليات للدفاع. في الواقع سلامة البطاقات الذكية "جافا كارد" تستند على حسن سير عمل مجموعة من الآليات المخصصة لهذا الغرض و التي تعمل بطريقة متكاملة. و لكن تعقيد هذه الآليات يجعل إنشاءها جد حساس. و منه يصبح التأكد من حسن إنشاء هذه الآليات شيئا أساسيا. لأن كل خطأ في هذا المستوى يمكن استغلاله لشن هجوم و ربما يؤدي إلى إضعاف النظام أو كشف بيانات سرية و حساسة موجودة داخل البطاقة. مع ذلك فان ضمان صحة النظام ليست دائما كافية. لذا يتوجب تحديد نقاط الضعف فيه بهدف تحسين سلامته و أمانه.

في هذا العمل, الاهتمام منصب على عنصر أساسي لأمن "جافا كارد" : مدقق رمز البايت. اقترحنا منهجية لتحليل ضعف هذا العنصر من أجل اكتشاف نقاط الضعف الأمنية المحتملة في مختلف الإنشاءات الممكنة له. و تستند هذه المنهجية على نموذج وفقا للطريقة B. هذا النموذج أنجز انطلاقا من مواصفات مدقق رمز البايت من أجل توليد سلسلة اختبارات أمنية. هذه الأخيرة سيتم إرسالها للمدقق بهدف تحليل ردة فعله. تحصيل بيانات الاختبار هذه يرتكز على نفي و حل الالزامات.

الكلمات الأساسية : البطاقات الذكية, جافا كارد, مدقق رمز البايت, تحليل الضعف, الطريقة B, الاختبار المعتمد على النموذج.

Résumé

Les cartes à puce sont considérées comme étant des supports d'exécution d'applications et de stockage d'informations très sécurisés. Donc, elles doivent faire face aux accès malveillants en assurant leur propre sécurité par l'implémentation de plusieurs mécanismes de défense. En effet, la sécurité de Java Card repose sur le bon fonctionnement d'un ensemble de composants dédiés et qui opèrent de façon complémentaire. Mais vue la complexité de tels composants, leur développement s'avère très délicat. Il devient alors primordial de s'assurer qu'ils sont bien implémentés car toute erreur à ce niveau peut être exploitée pour mener une attaque et éventuellement conduire à une défaillance du système ou encore révéler des données sensibles de la carte. Cependant, garantir la conformité d'un système n'est pas toujours suffisante et il devient donc nécessaire de détecter ses vulnérabilités en vue d'améliorer le niveau de sécurité.

Dans ce présent travail, on s'est intéressé à un composant crucial pour la sécurité de la plateforme Java Card : le vérifieur de byte code. Nous avons proposé une approche d'analyse de vulnérabilité de ce composant dans le but de découvrir d'éventuelles failles de sécurité dans différentes implémentations de ce dernier. Cette approche se base sur un modèle formel B, construit à partir de la spécification du vérifieur, pour la génération de suites de test de sécurité qui seront soumises au vérifieur de byte code pour analyser sa réaction. Cette génération de données de test s'appuie sur la négation et la résolution de contraintes.

Mots-Clés : Carte à puce, Java Card, vérifieur de byte code, analyse de vulnérabilité, méthode B, test à partir de modèle

Abstract

Smart cards are very secured devices designed to execute applications and store confidential data. Thus, they have to face the hostile accesses by ensuring their own security through the implementation of several defense mechanisms. Indeed, Java Card security relies on the efficiency of a set of dedicated components operating in complementary way. The development of such components is very delicate due to their complexity. Therefore, it is essential to ensure that they are well implemented because any error at this level can be exploited to lead an attack and possibly results in the system's failure or the disclosure of the sensitive card's data. However, ensuring the system's conformity is not always sufficient. Therefor, it is necessary to detect its vulnerabilities in order to improve the security level.

In this present work, we are interested in a crucial component for the Java Card platform security : the byte code verifier. We proposed a vulnerability analysis approach of this component in order to discover possible security vulnerabilities in its different implementations. This approach is based on a formal B model, built from the verifier's specification, in order to generate security test suites. The latter, will be sent to the byte code verifier to analyze its reaction. This generation of test's data is based on the negation and the resolution of constraints.

Keywords : Smart card, Java Card, byte code verifier, vulnerability analysis, B method, model-based testing

Introduction

Contexte du Travail

Les cartes à puce ne sont pas de simples objets de stockage. Ce sont des ordinateurs aux capacités réduites offrant un niveau de sécurité élevé qui est à la base de l'instauration de la confiance nécessaire aux utilisateurs finaux. En effet, elles détiennent et manipulent souvent des informations hautement sensibles, telles que des données à caractère personnel, voire des données biométriques ou bancaires. De ce fait, elles sont devenues la cible d'attaques, menées par des personnes malveillantes voulant contourner les mécanismes de sécurité de la carte pour s'approprier des données qu'elles contiennent voire même prendre le contrôle du système.

La sécurité d'une carte à puce peut être contournée selon plusieurs méthodes qui peuvent être soit des attaques physiques (prenant le matériel en défaut en utilisant des outils physiques adaptés), soit des attaques logiques (prenant l'applicatif en défaut en se basant sur la découverte d'une faille et de son exploitation). Et plus récemment sont apparues des attaques dites combinées car elles associent les attaques physiques et logiques.

Les cartes à puce ouvertes telle que Java Card, plateforme retenue pour notre travail, offrent la possibilité de charger plusieurs programmes issus ou non du constructeur de la carte après sa délivrance. De ce fait, le modèle de sécurité reposant sur la confiance accordée seulement au logiciel de ce constructeur n'est plus valable. En effet, du moment où l'utilisateur a la possibilité de charger d'autres programmes sur la carte, il faudra s'assurer que ces programmes n'interféreront pas avec les programmes déjà existants et même ne remettent pas en cause la sécurité de la plateforme. Un tel niveau de sécurité est assuré par un ensemble de mécanismes de défense tels que le pare-feu et le vérifieur de byte code dans Java Card.

Problématique et Objectif du Travail

Le travail qu'on a réalisé s'inscrit dans le domaine de la sécurité des cartes à puce. Il porte sur l'étude de la sécurité de l'un des mécanismes de la défense implémentés dans la plateforme Java Card : le vérifieur de byte code qui est une composante cruciale pour la sécurité de cette plateforme. En effet, il s'assure de l'innocuité du code à charger dans la carte pour protéger les données des autres applications déjà existantes ainsi que de la machine virtuelle contre tout débordement de cette nouvelle application.

Vu la complexité du vérifieur de byte code d'un côté et les ressources limitées des cartes à puce d'un autre côté, le processus de vérification était établi hors carte. Cependant, plusieurs travaux [Cas02] [CBR02] [BCR03] [DG02] ont été menés dans le but d'embarquer ce composant de sécurité dans la carte afin d'augmenter encore le niveau de sécurité. Et effectivement, certains prototypes de cartes disposant d'un vérifieur de byte code embarqué commencent à arriver sur le marché. Certes, avec ce progrès dans l'architecture de sécurité de la carte, un nombre important d'attaques logiques ne peuvent plus avoir lieu. Cependant, d'autres restent encore possibles.

La chose commune à toutes les attaques logiques est le fait qu'elles reposent sur des faiblesses dans l'implémentation des composants responsables de la sécurité dans la carte. Donc plus des défauts d'implémentation sont découverts :

- plus ces attaques ont des chances pour aboutir à un succès (point de vue attaquant) ;
- plus la sécurité de la carte peut être compromise (point de vue constructeur de cartes).

Et comme la vérification de byte code est reconnue comme étant une tâche très complexe et coûteuse, toute faute d'implémentation du vérifieur et/ou l'implémentation d'un ensemble restreint de fonctionnalités peuvent constituer une faille de sécurité. Pour cela, il faudra étudier le niveau de sécurité de ce composant en se basant sur des suites de test pour le caractériser et faire ressortir ses faiblesses qui peuvent être exploitées pour mener d'autres attaques.

L'objectif de notre travail est l'étude de la sécurité du vérifieur de byte code embarqué à travers une analyse de vulnérabilité. L'idée est de trouver des données d'entrée invalides, qui soient sciemment fausses, et qui seront transmises au vérifieur pour analyser sa réaction. Donc l'acceptation d'une donnée invalide signifie qu'une vulnérabilité est détectée.

Pour ce faire, on a mis en œuvre une approche originale [BHLS11] [HLM11] de test à partir d'un modèle formel du vérifieur de byte code. Ce modèle sera construit à partir de la spécification informelle de ce dernier. Par la suite, il sera dérivé en plusieurs modèles avec fautes qui vont servir à la génération des données de test invalides en se basant sur une résolution de contraintes. Le fait que l'approche soit basée uniquement sur la spécification fait d'elle une approche assez générique car elle n'est pas spécifique à une implémentation donnée. De ce fait, cette approche peut être utilisée pour analyser la vulnérabilité de toute implémentation de vérifieur embarqué conforme à cette spécification.

Organisation du Mémoire

Le présent document s'articule autours de sept chapitres regroupés dans deux parties. En raison de la diversité des domaines liés à notre travail, la première partie réunit cinq chapitres. Chacun d'entre eux traite d'un domaine d'étude.
On a commencé par introduire le domaine des cartes à puce dans le ***chapitre1*** en présentant les différents concepts de base (typologie, standards, protocoles, etc).
Par la suite, on a présenté la plateforme Java Card dans le ***chapitre2*** en mettant l'accent sur l'aspect sécurité (attaques et mécanismes de défense).
Et comme le vérifieur de byte code est notre composant cible, on a réservé le ***chapitre3*** pour une présentation plus approfondie de ce composant, tout en reprenant la problématique et notre objectif derrière ce travail en détail.
Le ***chapitre4*** regroupe quant à lui un état de l'art sur les approches de test, partant du test logiciel en général et arrivant au test à partir de modèle. Ceci dans le but de positionner notre travail par rapport à ce qui existe dans la littérature.
Cette partie est clôturée avec le ***chapitre5*** qui aborde les méthodes formelles et leur utilisation pour la modélisation et le test. Et spécialement la méthode B, sur laquelle notre choix a porté.

Ainsi cette première partie englobe les briques de base nécessaires pour aborder la seconde partie qui est consacrée, quant à elle, à notre contribution pour répondre à la problématique posée. Cette deuxième partie est scindée en deux chapitres.
Dans le ***chapitre6***, on a exposé notre approche d'analyse de vulnérabilité en détaillant chacune des étapes la constituant.
Enfin, le ***chapitre7*** concerne la mise en pratique de notre proposition d'approche en présentant les détails de son implémentation ainsi que les résultats expérimentaux qui ont pu être obtenus.

Pour finir, une conclusion générale reprendra notre contribution à travers ce travail et ses apports ainsi que quelques perspectives de recherche.

Première partie

Domaines d'étude

CHAPITRE 1

La Carte à Puce

Sommaire

1.1 Qu'est ce qu'une carte à puce ?

La carte à puce est un support électronique, portable et sécurisé pour conserver des données personnelles. Pratiquement, c'est une carte en plastique de taille très réduite (figure 1.1), quelques centimètres de côté et moins d'un millimètre d'épaisseur, incorporant un circuit électronique (la puce) capable de stocker et traiter des informations, généralement sensibles, de façon très sécurisée. Bien que les ressources d'une carte à puce soient toujours limitées, en termes d'espace mémoire et capacité de calcul, mais dans ses dernières générations, elle est capable d'exécuter du code et d'héberger diverses applications se rapprochant ainsi d'un ordinateur personnel .

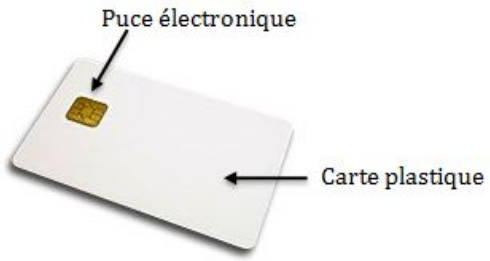

Fig 1.1 – Une carte à puce

1.2 Historique

La carte à puce actuelle a connu une grande évolution et un passage à l'échelle qui fait d'elle un support électronique à la pointe de la technologie. L'idée de base était de regrouper toutes les fonctionnalités sur un seul circuit électronique tout en mettant en œuvre les dispositifs nécessaires à la protection des données qui sont manipulées. Le tableau suivant (tableau 1.1) résume les dates les plus importantes dans l'histoire de l'évolution des cartes à puce.

Année	Évènement
1968	* Les deux Allemands "Jürgen Dethloff" et "Helmut Gröltrup" introduisent l'idée d'incorporer un circuit intégré dans une carte en plastique.
1970	* Au Japon, "Kunitaka Arimura" dépose un brevet sur la carte à puce.
1974-1979	* Le Français "Roland Moreno" dépose 47 brevets (dans 11 pays) sur la carte à puce. Son invention a été distinguée des autres travaux par le fait qu'elle incorpore des moyens de protection (matériels et/ou logiciels) de la mémoire pour restreindre l'accès en lecture et en écriture.
1977	En Allemagne, "Derthloff" dépose un brevet où il introduit un microprocesseur comme moyen de protection de la carte à mémoire.
1979	* En France, le groupe "Bull" crée la première carte programmable à microprocesseur (nommée CP8).
1983-1984	* Arrivée des premières cartes téléphoniques et cartes bancaires en France et en Allemagne .
1987	* Introduction des premières normes ISO, en Europe, régissant le domaine des cartes à puce.
1991	* Lancement du réseau GSM et des premières cartes SIM (Subscriber Identity Module).

1994	* Lancement de la première carte à puce sans contact (carte MIFARE de la société Mikron).
1997	* Apparition des cartes multi-applicatives.

Tab 1.1 – Historique de la carte à puce

1.3 Typologie des cartes à puce

Les différents type des cartes à puce existant peuvent être classés en deux grandes catégories en fonction de leurs architectures internes ou leurs modes de communication avec le monde extérieur. Chacune de ces catégories peut être scindée elle même en sous-catégories qui sont résumées dans la figure 1.2 et présentées dans ce qui suit.

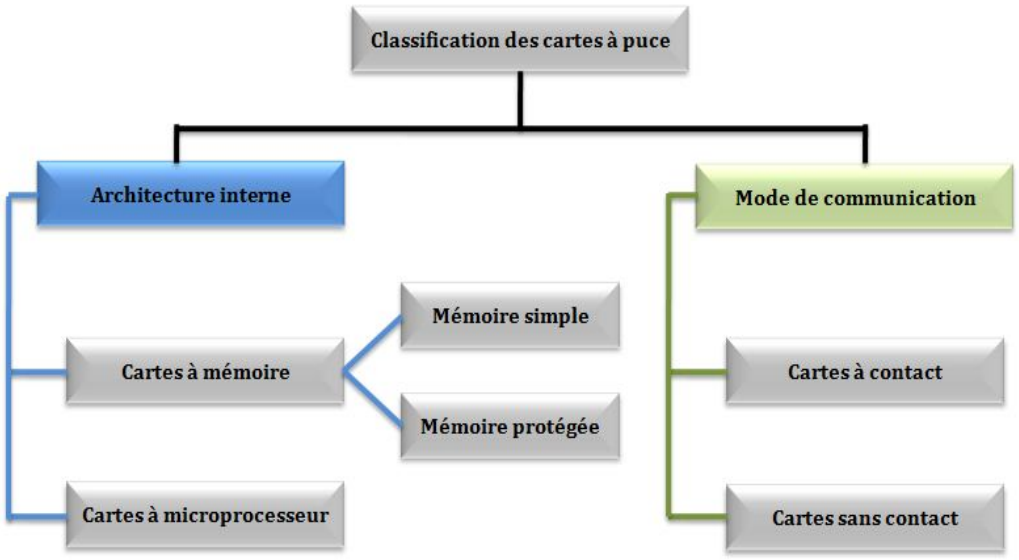

Fig 1.2 – Typologie des cartes à puce

1.3.1 Selon la technologie interne

Carte à mémoire

Elle constitue la famille la plus ancienne des cartes à puce. Ne contenant que de la mémoire, de telles cartes sont limitées à des applications relativement simples. Cette catégorie de cartes peut être scindée à son tour en deux sous-catégories [Tav07] :

- *Les cartes à mémoire simple* ne contiennent qu'une zone mémoire et le minimum de logique nécessaire pour pouvoir y accéder via les signaux électriques (figure 1.3). Le niveau de sécurité offert par ces cartes est très faible et ne peut reposer que sur des artifices relativement simples.
- *Les cartes à mémoire protégée* associent de la mémoire, dont certaines zones sont accessibles seulement en lecture ou seulement en phase de personnalisation de la carte, et de la logique

permettant l'exécution d'automates simples allant jusqu'à la présentation de mots de passe ou autres succédanés de codes PIN. Il s'agit typiquement des télécartes contenant un compteur d'unités, un numéro de série et une donnée secrète permettant d'authentifier la carte.

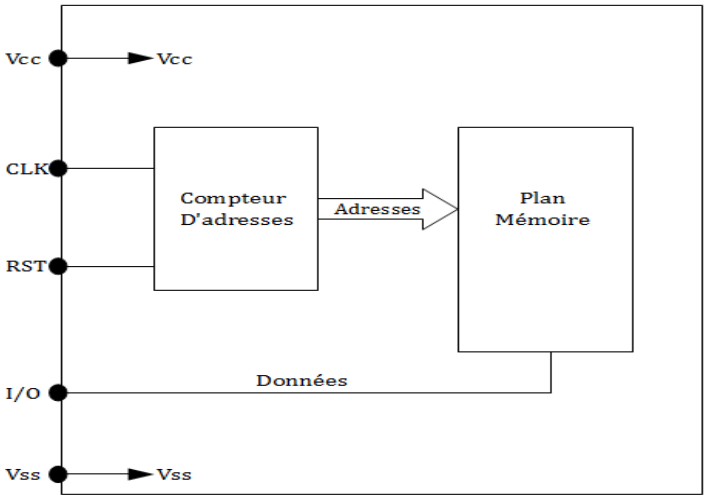

Fig 1.3 – Architecture interne générale d'une carte à mémoire simple [Tav07]

Bien que les cartes à mémoire protégée soient plus sûres que celles à mémoire simple, elles ne permettent pas la mise en place des applications les plus complexes que sont les cartes bancaires, les cartes SIM ou bien encore les cartes de décryptage TV. Il faut en effet pour cela que la carte dispose d'une "intelligence" locale que n'ont pas les cartes à mémoire.

Carte à microprocesseur

De telles cartes sont dites "intelligentes" (*Smart Cards* en Anglais) car elles ne se limitent pas à de la simple mémoire associée à de la logique, mais plutôt elles renferment un microcontrôleur complet. Ce dernier, se présente typiquement sous la forme d'un rectangle de silicium dont la surface est inférieure à 25 mm^2. D'une part, cette taille est imposée par les contraintes de flexion induite par le support en plastique. Et d'autre part, cette dimension limitée réalise un compromis entre sécurité physique et complexité du composant.

En effet, c'est l'association en un seul circuit [Tav07] (Figure 1.4) :

- d'un *microprocesseur* : de 8, 16 ou 32 bits avec des architectures CISC (Complex Instruction Set Computer) ou RISC (Reduced Instruction Set Computer) travaillant à des fréquences internes allant de 5 à 40 MHZ.
- d'une *mémoire morte (ROM)* : C'est une mémoire persistante. Elle est programmée en usine de façon figée. Donc son contenu (qui est constitué du système d'exploitation ainsi que les données permanentes) n'est pas modifiable. Sa taille varie de 32 Ko jusqu'à 256 Ko et même plus pour les cartes haut de gamme.
- d'une *mémoire vive (RAM)* : Du fait qu'elle est volatile, elle est utilisée comme espace temporaire pour modifier et stocker les données. Sa taille va de 1 Ko à 24 Ko (pour les cartes haut de

gamme).

- d'une *mémoire EEPROM* : C'est une mémoire persistante comme la ROM, mais son contenu peut être modifiable. Elle contient les données qui peuvent évoluer dans le temps sans être perdues. Sa taille varie de 16 Ko à 256 Ko (Pour les cartes haut de gamme)
- d'une *interface d'entrée/sortie série* : Pour les échanges de données.
- de toute la logique nécessaire pour faire fonctionner l'ensemble .

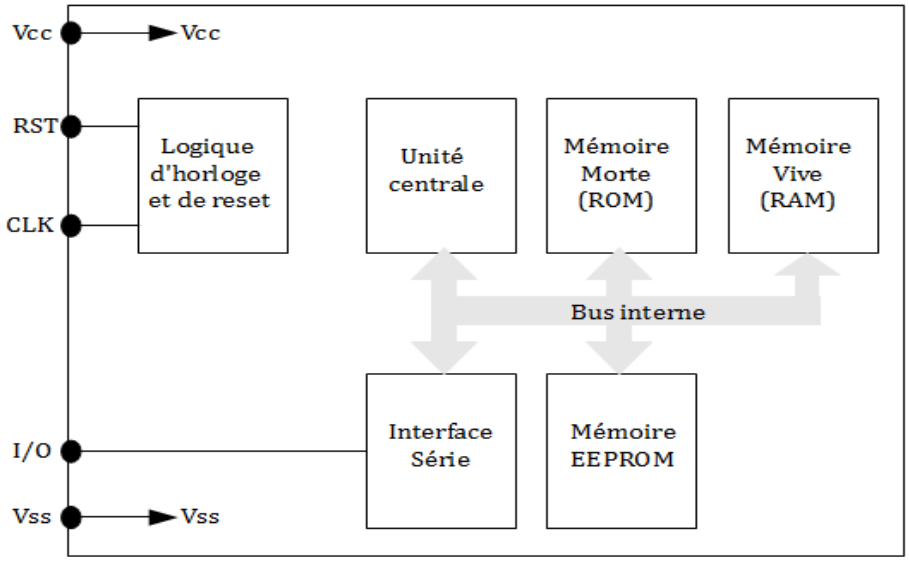

Fig 1.4 – Architecture interne générale d'une carte à microprocesseur [Tav07]

Compte tenu des soucis sécuritaires de plus en plus forts des utilisateurs d'applications à base de cartes à puce, le microcontrôleur se trouve parfois associé à un processeur spécialisé ou cryptoprocesseur chargé de réaliser des calculs cryptographiques, nécessitant une puissance de calcul relativement importante, destinés à sécuriser l'application.
Les cartes à microprocesseur conviennent aux applications les plus sensibles où le degré de sécurité des données est le facteur prédominant : contrôle d'accès sécurisé, carte bancaire, télécommunication, etc. Dans tout le reste du document, l'utilisation du terme *"carte à puce"* fera référence à ce type de cartes *i.e.* les cartes à microprocesseur.

1.3.2 Selon le mode de communication

Carte à contact

Pour communiquer avec le monde extérieur, la carte à contact doit être insérée dans un lecteur (appelé aussi CAD pour Card Acceptance Device) afin d'utiliser les points de contact présents sur sa surface. Les contacts (figure 1.5) sont en nombre de huit (C1-C8) et dont les caractéristiques sont définies dans la norme ISO 7816-2. Le tableau suivant (tableau 1.2) en résume de l'utilité de chacun des contacts

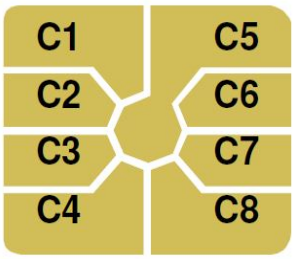

Fig 1.5 – Numérotation et position des contacts selon la nome ISO 7816-2

Contact	Appellation	Utilisation
C1	Vcc	tension d'alimentation positive de la carte
C2	RST	commande de remise à zéro
C3	CLK	horloge fournie à la carte
C5	GND	masse électrique
C6	Vss	tension de programmation (n'est plus utilisée)
C7	I/O	entrée/sortie de données (bidirectionnelle)
C4 et C8	RFU	à l'origine, contacts réservés pour utilisation future mais actuellement ils servent à communiquer en USB (Universal Serial Bus)

Tab 1.2 – Utilisation des contacts selon la norme ISO 7816-2

Carte sans contact

La communication s'établit à travers une interface radio fonctionnant par induction grâce à une antenne imprimée ou intégrée dans la carte à puce. Cette carte n'utilise pas de contact physique avec le lecteur. Plusieurs technologies existent, mais de façon générale, la carte sans contact contient une puce électronique avec un émetteur hyperfréquence et une antenne intégrée dans le plastique (figure 1.6). La carte doit être assez près du lecteur, entre 3 et 10 centimètres. Certaines cartes sans contact utilisent les ondes radio et peuvent donc fonctionner à plus grande distance. Cette carte est un peu plus complexe car elle doit intégrer un régulateur de tension, un modulateur/démodulateur ainsi qu'un mécanisme d'anti-collision, un générateur d'horloge et bien entendu une antenne [Lan06]. Les cartes sans contact sont privilégiées dans le domaine du transport ainsi que pour le contrôle d'accès aux bâtiments, domaines dans lesquels les transactions doivent être faites à une vitesse assez élevée.

Carte hybride

Les technologies étant différentes, une puce ne peut pas être à la fois avec et sans contact. De ce fait les cartes hybrides embarquent deux puces, la première reliée aux contacts, la deuxième à l'antenne (figure 1.7). Ces cartes sont le meilleur compromis car elles offrent les avantages des deux types de

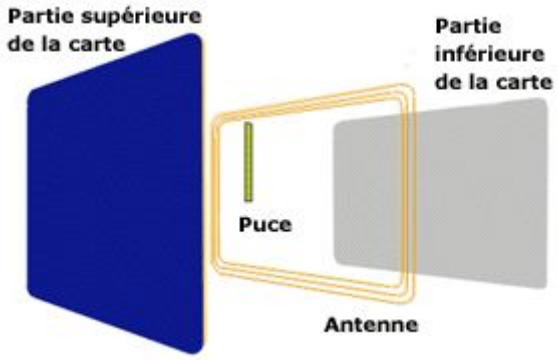

Fig 1.6 – Carte à puce sans contact

carte à puce mais leur prix est beaucoup plus élevé.

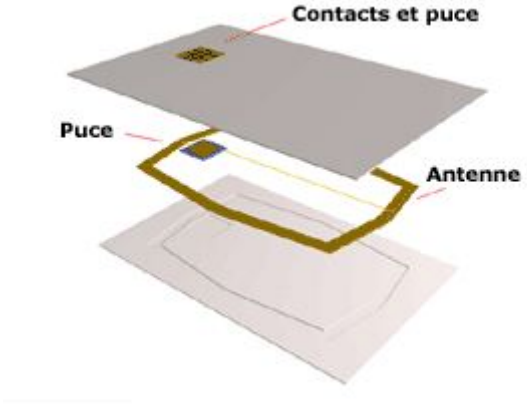

Fig 1.7 – Carte à puce hybride

1.4 Systèmes d'exploitation pour cartes à puce

Les cartes à microprocesseur sont supportées par un puissant système d'exploitation appelé "COS" (*Chip Operating System* ou *Card Operating System*). Cependant, vu les ressources limitées des cartes à puce, leur système d'exploitation doit être le plus léger possible, tout en étant riche, afin qu'il puisse être embarqué dans sa mémoire. A ce niveau là, trois sous-familles de cartes peuvent être distinguées [Tav07] :

1.4.1 Les cartes à puce *"spécifiques"*

Dans lesquelles de contenu de la majeur partie du système d'exploitation est définie par le fabriquant de la carte. En respectant les normes définies dans les standards, ce dernier peut définir ces propres instructions propriétaires, ses propres fichiers avec le contenu de son choix. Un tel développement conduit à écrire le programme *i.e.* le système d'exploitation, qui est ensuite programmé par masque dans le microcontrôleur pour produire des cartes "sur mesure". Ce type de cartes est réservé aux grands

groupes industriels, commerciaux ou financiers puisque cette approche impose de produire la carte en très grande série afin d'amortir les coûts d'étude et de développement qui sont très élevés.

1.4.2 Les cartes à puce *"personnalisables"*

Elles contiennent un système d'exploitation non modifiable, programmé par le fabriquant. ce système supporte un certain nombre de commandes, définies dans les standards, mais bien souvent complétées par des commandes propriétaires crées par le fabriquant de la carte et adaptées aux différentes applications pouvant être visées par la carte. Lorsqu'elles sont fournies par le fabriquant, certaines parties de la mémoire sont accessibles afin de définir le comportement global de la carte, les propriétés des fichiers, etc. C'est le stade de la *"personnalisation"*. Ceci revient au développement de l'application visée par la carte. Cette personnalisation est d'autant plus souple que le fabriquant aura prévu de nombreux fichiers de types différents et un jeu d'instruction riche. Une fois cette phase de personnalisation achevée, il y a une opération de "verrouillage" du contenu de la carte afin de le rendre impossible à modifier par la suite. A ce niveau là, la carte peut être mise en circulation afin de faire fonctionner l'application.

1.4.3 Les cartes à puce à *"Système ouvert"*

Pour ces cartes, le développement des applications est plus simple car il se fait en langage évolué (Java, Basic, C) et non pas en langage machine. Ainsi le programme obtenu est traduit en un code intermédiaire qui sera chargé dans la carte puis exécuté par le microcontrôleur qui est doté d'un interpréteur du code intermédiaire (figure 1.8). Dans un tel schéma, l'évolution du code est désormais possible (à l'inverse des deux types de système précédents où ce programme contenait à la fois le système opératoire et l'application rendant ainsi toute évolution du code difficile voire impossible). Ceci revient à dire que les cartes à système ouvert autorisent le chargement des applications après délivrance de la carte et peuvent même héberger plusieurs applications différentes (dans le cas des cartes multi-applicatives). Il existe sur le marché plusieurs Systèmes ouverts et qui sont totalement incompatibles entre eux. Parmi les systèmes existants, on va présenter : MULTOS, Basic Card et Java Card.

MULTOS

C'est un système d'exploitation mlti-applicatif (MULTOS) [1]. Il a été proposé initialement par Mondex et MasterCard organismes dont le centre d'intérêt premier est le payement électronique. MULTOS est donc très bien adapté à toutes les opérations de ce type et s'avère même, dans ce cas, être un système très sûr grâce à l'utilisation d'algorithmes de cryptage et de signature à clé publique. MULTOS s'exécute sur le microcontrôleur de la carte à puce. A chaque fois qu'une application est envoyée vers la carte, il vérifie sa validité moyennant divers systèmes sécuritaires prévus pour cette fin. Ensuite, il alloue, par l'utilisation de firewalls spéciaux, au programme une zone mémoire isolée,

1. `http://www.multos.com`

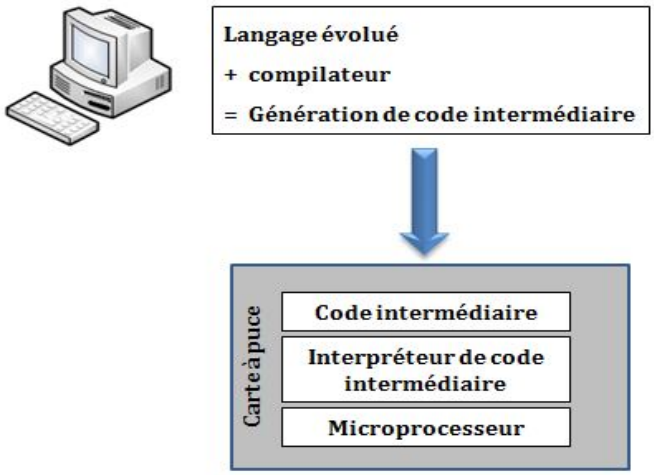

Fig 1.8 – Principe de programmation d'une carte à puce à système ouvert

protégée des autres applications résidant dans la même carte. Les cartes MULTOS se programment au moyen d'un langage spécifique appelé MEL (MULTOS Executable Language). Cependant, il y a la possibilité de développer des applications en C ou en Java et qui seront ensuite traduites en MEL afin d'ouvrir un peu plus le système.

Basic Card

C'est une carte à système ouvert, proposée par ZeitControl [2], facilement programmable en langage Basic. En intégrant diverses solutions cryptographiques, selon les versions des cartes, elles offrent un niveau élevé de sécurité. L'outil de développement nécessaire et complet pour la réalisation des applications pour ces cartes est gratuit. Cet outil intègre un simulateur de carte permettant de tester l'application développée en absence de la carte et du lecteur. Cependant, la Basic Card n'est pas supportée par les fabricants de carte.

Java Card

C'est une carte multi-applicative sur laquelle il est possible de charger et d'exécuter des programmes écrits en Java. Et comme cette plateforme est au centre du présent travail, le chapitre suivant (2) est dédié pour y donner une présentation détaillée.

1.5 Principaux standards de normalisation

Les cartes à puces sont très standardisées car elles doivent être utilisables avec la gamme la plus large possible de lecteurs dans le monde entier. C'est la raison pour laquelle les caractéristiques des cartes à puce ont été fixées par des règles reconnues universellement qui appartiennent à une famille de standards et protocoles internationaux. À ce propos, la normalisation ne concerne pas que la seule

2. `http://www.basiccard.com`

puce, mais aussi les dimensions physiques de la carte. La normalisation concerne essentiellement trois types de paramètres différents :

- *paramètres physiques* qui indiquent la taille de la carte et la position de la puce et de ses contacts ;
- *paramètres électriques* qui précisent les tensions d'alimentation et niveaux électriques mis en œuvre ainsi que le brochage de la puce sur la carte ;
- *paramètres logiciels* qui définissent le mode de dialogue avec la carte, les commandes qu'elle peut interpréter et son comportement face à ces dernières.

Cela s'est traduit par la publication d'un certain nombre de normes internationales parmi lesquelles celles appartenant au standard ISO (ce sont les premières normes définies pour les cartes à puce et encore les plus utilisées). Le tableau ci-dessous (tableau 1.3) en résume quelque unes.

Norme ISO	Titre officiel de la norme
Normes relatives aux cartes en général	
	Cartes d'identification :
ISO 7810	- Caractéristiques physiques
ISO 7811-1	- Techniques d'embossage
ISO 7811-2	- Techniques d'enregistrement magnétique
ISO 7811-3	- Emplacement des caractères embossés sur les cartes de type ID 1
ISO 7811-4	- Position des pistes magnétiques à lecture seule - Pistes 1 et 2
ISO 7811-5	- Position des pistes magnétiques à lecture/écriture - Piste 3
ISO 7812-1	- Identification de l'émetteur - système de numérotation
ISO 7813	- Cartes pour transactions financières
Normes relatives aux cartes à puce avec contact	
	Cartes d'identification - Cartes à circuits intégrés avec contacts :
ISO 7816-1	- Caractéristiques physiques
ISO 7816-2	- Dimension et position des contacts
ISO 7816-3	- Signaux électroniques et protocoles de transmission
ISO 7816-4	- Commandes inter-industries
ISO 7816-5	- Enregistrement des fournisseurs d'applications
ISO 7816-8	- Commandes pour des opérations de sécurité

ISO 7816-9	- Commandes pour la gestion des cartes
ISO 7816-10	- Signaux électroniques et réponse à la mise à zéro des cartes synchrones
ISO 7816-11	- Vérifications personnelles par méthodes biométriques
ISO 7816-15	- Application des informations cryptographiques
Normes relatives aux cartes à puce sans contact	
	Cartes d'identification - Cartes à circuits intégrés sans contacts :
ISO 14443-1	- Caractéristiques physiques
ISO 14443-2	- Interface radio fréquence et des signaux de communication
ISO 14443-3	- Initialisation et anticollision
ISO 14443-4	- Protocole de transmission

Tab 1.3 – Principales normes relatives aux cartes à puce [Tav07]

En plus de la large gamme des normes de la famille ISO, il existe d'autres standards relatifs à des domaines d'application spécifiques [Zhi00] :

- **GSM** pour *"Global System for Mobile Communications"* défini par l'ESTI (European Telecommunications Standards Institute) est une spécification pour un système de téléphonie mobile terrestre international. Au départ, cette norme avait pour objectif de couvrir quelques pays d'Europe centrale, puis elle s'est développée pour devenir un standard international pour la téléphonie mobile.
- **EMV** défini par **E**uropay, **M**astercard et **V**isa est un ensemble de spécifications techniques, basées sur le standard ISO 7816, des cartes à puce pour l'industrie bancaire. Leur but était de définir un minimum de fonctionnalités nécessaires pour supporter l'interopérabilité internationale entre les cartes, les terminaux et les périphériques en rapport avec les cartes.
- **PC/SC** défini par le groupe de travail PC/SC, un consortium industriel qui regroupe les principaux acteurs dans l'industrie des cartes à puce (Apple, Bull, Gemalto, Hewlett Packard, Infineon, Intel, Microsoft et Toshiba). Les spécifications PC/SC définissent une architecture globale pour l'utilisation des cartes à puce sur les systèmes des ordinateurs personnels (PC). La standardisation a porté sur trois parties :
  - l'interfaçage du lecteur au PC ;
  - une API de haut niveau pour accéder aux fonctionnalités de la carte ;
  - les mécanismes autorisant de multiples applications à partager les même ressources tels que la carte ou le lecteur.

1.6 Communication de la carte à puce avec son environnement

Pour que la carte à puce communique avec le monde extérieur, elle doit être placée dans un lecteur (carte avec contact) ou à proximité de ce dernier (carte sans contact). Le lecteur est connecté lui même à un autre ordinateur (hôte). fournissant ainsi l'énergie nécessaire au fonctionnement de la carte.

Modèle de communication

Les données peuvent être envoyées "de la carte" ou "vers la carte" via la même ligne de communication (contact C7- I/O). Il s'agit d'une communication en half-duplex : les deux partie (carte/hôte) peuvent envoyer des données mais pas simultanément. Les données circulant entre la carte et l'hôte sont des paquets de données spéciaux appelés APDUs *(Application Protocol Data Units)* défini dans la norme ISO 7816-4. Chaque APDU est soit une *"commande APDU"* qui spécifie une requête, soit une *"réponse APDU"* retournant le résultat d'exécution d'une commande. Les échanges d'APDU se font suivant un modèle client/serveur. Dans ce mode de fonctionnement, la carte joue le rôle passif (serveur), elle n'initialise pas la communication mais se contente de répondre aux commandes APDUs envoyées par l'hôte (client). Les commandes et réponses APDUs sont échangées alternativement entre la carte et l'hôte (figure 1.9).

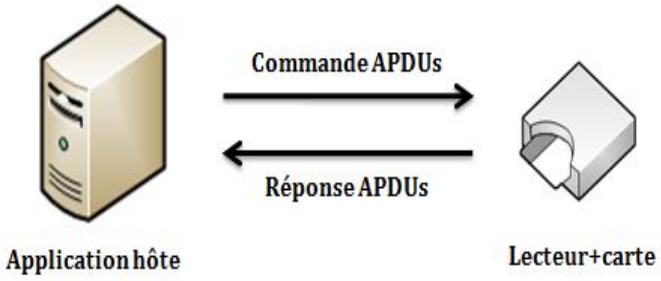

Fig 1.9 – Modèle de communication d'une carte à puce

ATR

Juste après le signal de remise à zéro lors de la réinitialisation de la carte, elle envoie un message spécial à l'hôte afin de se synchroniser pour les échanges futurs. C'est un paquet de données appelé ATR *(Answer To Reset)* défini par la norme ISO 7816-3 et qui contient la description des paramètres de transmission entre autre : protocole de transport supporté par la carte, vitesse de transmission des données, paramètres matériels de la carte, etc.

Protocole APDU

C'est un protocole de niveau applicatif (la carte d'un côté et l'application de l'hôte d'un autre côté), il est défini par la norme ISO 7816-4. Comme on a précisé plus haut, un message APDU peut prendre

deux formes différentes mais utilisées comme une paire pour chaque échange :

- **Commande APDU** : envoyée par l'application hôte vers la carte, sa structure comprend (figure 1.10) :
  - Un en-tête obligatoire comprenant les champs suivants :
    - CLA : indique la catégorie de la commande APDU correspondant généralement à un domaine d'application donné (par exemple, CLA= A0h pour les commandes des cartes SIM)
    - INS : indique l'instruction à exécuter
    - P1, P2 : les paramètres de l'instruction
  - Un corps dont la longueur est variable et qui comprend :
    - LC : la taille du champ de données
    - Données : contient les données à envoyer à la carte pour exécuter l'instruction définie dans l'en-tête (INS)
    - Le : correspond au nombre d'octets attendus par l'application hôte pour le champ "données" de la réponse
- **Réponse APDU** : envoyée par la carte vers l'application hôte, sa structure comprend (figure 1.10) :
  - Un corps optionnel contenant :
    - Données : contient les données à envoyer par la carte et dont la taille est déterminée par le champs "Le" de la commande.
  - Une zone de fin obligatoire :
    - SW1 et SW2 (Status Word) : c'est un code sur deux octets indiquant l'état de la carte après exécution de la commande. Si cette dernière s'est correctement déroulée, alors SW1=90h et SW2=00h. Sinon, les deux champs SW1 et SW2 serviront à coder le type d'erreur qui s'est produite.

      Dans la suite du document (section 6.3.5), on reviendra plus en détails sur ces codes renvoyés par la carte et leur signification qui constitue un point très important dans l'une des étapes de notre solution.

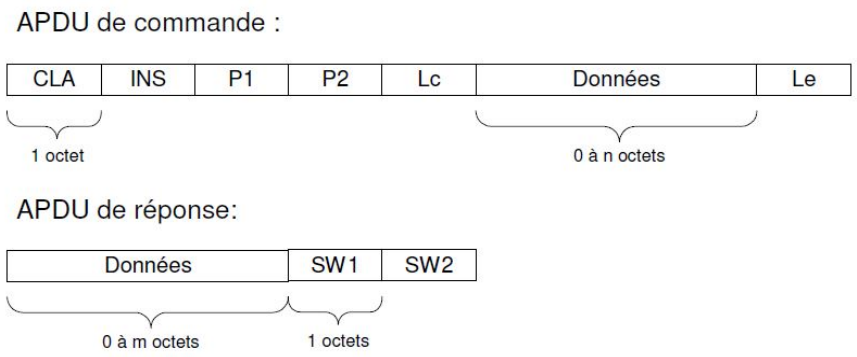

Fig 1.10 – Structure des APDUs

Protocole TAPDU

C'est le protocole utilisé pour le transport des APDUs de la couche application. Il est défini par la norme ISO 7816-3. Les deux principaux modes de transmission par ce protocole sont :

- T=0 qui utilise une transmission *"orientée octet"*
- T=1 qui utilise une transmission *"orientée paquet"*

1.7 Domaines d'application

La carte à puce a désormais pris une grande place dans notre vie quotidienne, du fait qu'elle est utilisée dans plusieurs domaines où ses capacités de stockage sécurisé des informations sont mises en valeurs. Parmi ces applications on peut citer :

- **Applications de payement :**
  - carte bancaire ;
  - carte pré-payée ;
  - porte monnaie électronique (e-purse) ;
  - titres de transport en commun ;
  - télévision à péage, etc.
- **Applications d'identification :**
  - dans une entreprise : contrôle d'accès au locaux, contrôle horaire ;
  - sur un réseau téléphonique : carte SIM ;
  - dans le secteur du gouvernement : carte d'identité, passeport, carte de santé, etc.
- **Applications de sécurisation :** la carte est utilisée comme un support de stockage des clefs cryptographiques servant à l'authentification et la sécurisation des communications :
  - contrôle d'accès logique à un serveur par authentification ;
  - protection des messages et documents numériques par signature électronique et chiffrement des données ;
  - sécurité des transactions sensibles, etc.

Les deux figurent ci-dessous (figures 1.11 et 1.12) résument les derniers chiffres du marché des cartes à puce, publiés dans le communiqué de presse d'EuroSmart (Association internationale représentant les professionnels de l'industrie des cartes à puce)[Eur10]

Domaines	Systèmes d'exploitation		2010	2010 vs 2009 en %	2011	2011 vs 2010 en %
	dédiés	ouverts				
Télécom	SIM, Télécartes	JavaCArd(450)	4000	+18	4500	+13
Banquaire, Fidélité	B0', EMV	MultOS(50) JavaCard	880	+17	1010	+15
Identité, Santé	Vitale (Scot 400, IGEA)	MultOS	190	+19	225	+18
Transport		JavaCard	65	+63	80	+23
TV à péage		JavaCard	110	+10	125	+14
Sécurité, Contrôle d'accès		JavaCard	75	+7	80	+7
Total			5320	+18	6020	+13

Fig 1.11 – Marché des cartes à puce et systèmes d'exploitation associés en 2010 et prévision pour 2011 (en millions d'unités) [Eur10]

Domaines	2010	2010 vs 2009 en %	2011	2011 vs 2010 en %
Télécom	0	-	15	-
Banquaire, Fidélité	175	+46	225	+29
Identité, Santé	100	+33	125	+25
Transport	65	+63	80	+23
Sécurité, Contrôle d'accès	30	-	30	-
Total	370	+40	475	+28

Fig 1.12 – Marché des cartes sans contact en 2010 et prévision pour 2011 (en millions d'unités) [Eur10]

CHAPITRE 2

La Plateforme Java Card

Sommaire

2.1 Présentation de Java Card

La plateforme Java Card a pour but de faire fonctionner la technologie Java sur des équipements fortement contraints tels que les cartes à puce (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*, terme qu'on utilisera dans la suite du document pour désigner une application Java Card , ou *servlets* depuis la version 3.0 de la spécification. Contrairement aux cartes à puce traditionnelles, ce sont des cartes ouvertes, *i.e.* que les programmes qui y sont contenus

ne sont pas nécessairement fournis par l'émetteur de la carte et donc peuvent être chargés après la délivrance de celle-ci.

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

2.2 Historique et évolution des versions

En Novembre 1996, un groupe d'ingénieurs du centre de production de Schlumberger à Austin au Texas a cherché à simplifier la programmation des cartes à puce tout en préservant la sécurité. Le langage de programmation Java apparaît comme la solution afin de simplifier le modèle de programmation des cartes à puce et d'obtenir un code portable indépendamment du fabriquant de la carte. Mais, en raison des ressources limitées des cartes à puce, il fallait adapter la plateforme Java et de n'utiliser qu'un sous-ensemble du langage. Schlumberger devint alors la première entreprise à acquérir une licence en proposant la spécification Java Card 1.0. [Zhi00]

En Février 1997, Bull et Gemplus se joignent à Schlumberger pour co-fonder le Java Card Forum [1]. Le but de ce consortium industriel est d'identifier et de résoudre les problèmes de la technologie Java Card en proposant des spécifications à JavaSoft (la division de Sun Microsystems à qui appartient Java Card). Il a également pour objectif de promouvoir des APIs (Application Programming Interfaces) Java Card afin de permettre son adoption par l'industrie de la carte à puce. Aujourd'hui, le Java Card Forum regroupe les fabricants de cartes, Oracle et des utilisateurs.[Zhi00]

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 rien n'a été précisé concernant le format des applets à charger sur la plateforme.

En mars 1999, 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 APIs telles que celles de 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 Java Card (JCVM pour Java Card Virtual Machine) et du fichier CAP (format binaire représentant des fichiers *.class* convertis, voir section 3.1), permettant ainsi une vraie interopérabilité.

En juin 2002, Sun Microsystems a publié 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). Elles rajoutent quelques éléments sur l'installation et l'effacement des applications sur la carte

1. `http://www.javacardforum.org`

et quelques nouveaux algorithmes cryptographiques.

En octobre 2003, la version 2.2.1 est publiée. Elle corrige les APIs de la version précédente et apporte quelques clarifications.

En mars 2006, la version 2.2.2 est publiée. Elle n'apporte pas de changement majeur mis à part quelques nouveaux algorithmes cryptographiques et une clarification des APIs.

C'est en 2008, qu'arrive la version 3.0 qui apporte une révolution dans les spécifications, en introduisant deux types de spécification Java Card :

- Java Card 3 *Classic 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* qui sont des applications web nécessitant un serveur web embarqué dans la carte. Elle nécessite donc des cartes avec plus de ressources (cartes modernes haut de gamme).

2.3 Avantages de Java Card

La plateforme Java Card offre aux développeurs d'applications pour cartes à puce les avantages suivants [Zhi00] :

1. *Facilité de programmation.* L'utilisation de Java comme langage évolué au lieu des langages assembleurs rend le développement des applications pour cartes à puce plus aisé et à la portée d'un plus grand nombre de développeurs. Cette facilité est due à :
   - la programmation orientée objet offrant une meilleure modularité et réutilisabilité du code ;
   - la possibilité d'utiliser les environnements de développement existants pour Java ;
   - une plateforme ouverte qui définit des APIs et un environnement d'exécution standardisé ;
   - à une plateforme qui encapsule la complexité sous-jacente et les détails du système des cartes à puce ;
2. *Indépendance du matériel.* Tout comme Java, la plateforme Java Card assure l'indépendance du code par rapport au matériel sur lequel il s'exécute. Cette portabilité offerte permet d'écrire du code qui fonctionnera sur n'importe quel microprocesseur de carte à puce (i.e. Écrire une fois, exécuter n'importe où).
3. *Sécurité.* C'est le facteur primordial pour les cartes à puce. Une partie de cette sécurité est offerte par le langage Java (un langage fortement typé, plusieurs niveaux de contrôle d'accès aux méthodes et aux variables, pas de construction de pointeurs, etc). Alors que l'autre partie est assurée par la plateforme Java Card elle même en implémentant plusieurs mécanismes de sécurité tel que le pare-feu qui isole les applications hébergées dans la carte pour empêcher tout comportement hostile de leur part. Dans la suite du présent document, on reviendra plus en détails sur ces mécanismes de sécurité (voir section 2.6).

4. *Possibilité d'avoir plusieurs applications dans la même carte.* La plateforme Java Card est 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 applications est régie par des règles strictes contrôlées par le pare-feu.
5. *Compatibilité avec les standards existants.* Java Card a été conçue autour de la norme ISO 7816. Donc elle supporte les applications qui sont compatibles avec cette norme. En outre, ceci assure une compatibilité avec tous les CAD existants.

Malgré tous ces avantages, la plateforme Java Card n'est pas exempte de défauts qui ont ouvert la porte aux attaquants (voir section 2.7).

2.4 Architecture de la plateforme

Vu les ressources limitées des cartes à puce, l'un des plus grands défis de la technologie Java Card a été de s'adapter à ces contraintes pour construire une carte Java tout en conservant suffisamment d'espace pour charger des applications. La solution adoptée a été de supporter seulement un sous-ensemble du langage Java, adapté au développement d'applications pour cartes à puce. Actuellement, avec les grands progrès technologiques qu'a connu l'industrie des cartes à puce, de nouvelles cartes haut de gamme (microprocesseur 32 bits avec architecture RISC, beaucoup plus de mémoire et des interfaces de communication multiples pouvant fonctionner en USB 2.0) commencent à voir le jour. De ce fait, de nouvelles applications plus sophistiquées, tel qu'un serveur web, peuvent désormais être hébergées dans de telles cartes.

Le nouveau défi pour Java Card est de trouver une solution qui prend en charge à la fois les cartes à faibles ressources (représentant la majorité des cartes actuellement disponibles) et les cartes de nouvelle génération. C'est l'avènement de la version 3.0 de Java Card qui a apporté cette solution en introduisant 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 développées pour les éditions précédentes (*i.e.* antérieurs à l'édition 3.0).

2.4.1 Java Card 3 édition « *Classic* »

Elle est basée sur l'évolution de la version 2.2.2 de la plateforme Java Card . Elle cible les cartes à faibles ressources qui supportent uniquement les applets comme modèle d'application. L'édition « *Classic* »adopte la même architecture (figure 2.1) de la plateforme Java Card que les versions précédentes (architecture utilisée depuis la version 2.1). Les changements apportés par cette version portent essentiellement sur le support des derniers algorithmes cryptographiques (standards de sécurité) ainsi que les standards régissant les communications sans contact.

Cette architecture est définie par les trois spécifications suivantes :

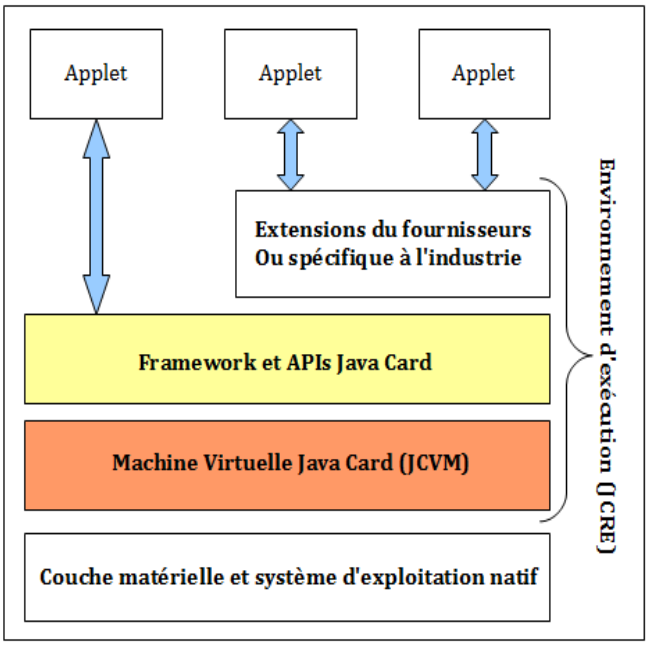

Fig 2.1 – Architecture de Java Card 3 Classic Edition

- *Java Card 3.0.1 Virtual Machine Specification*[Mic09c]. Définit la machine virtuelle nécessaire pour les applications sur cartes à puce ;
- *Java Card 3.0.1 Runtime Environment Specification*[Mic09b]. Décrit précisément le comportement de l'exécution de la Java Card ;
- *la Java Card 3.0.1 API Specification*[Mic09a]. Décrit l'ensemble des paquetages et des classes Java nécessaires à la programmation des cartes à puce mais aussi quelques extensions optionnelles.

La machine virtuelle Java Card

La machine virtuelle Java Card a une architecture (figure 2.2) semblable à celle d'une machine virtuelle Java (JVM pour *Java Virtual Machine*). Cependant, les ressources très restreintes des cartes à puce, par rapport à celles d'un ordinateur personnel, ont conduit à séparer la JCVM en deux parties :

- Partie hors carte (*Off-Card*) : tourne sur une station de travail et comprend un vérifieur de byte code et un convertisseur effectuant des transformations sur ce code en vue de l'optimiser avant de le charger dans la carte ;
- Partie sur carte (*On-Card*) : c'est la partie embarquée et elle comprend l'interpréteur de byte code qui se charge de l'exécution du code chargé.

Ensemble ces deux parties, elles assurent toutes les fonctions d'une machine virtuelle. Cependant cette architecture en deux parties est à l'origine de la problématique qui sera présentée dans la section 3.3.

Les fichiers *.class* sont produits par le compilateur Java à partir du code source. Ensuite, le convertisseur pré-traite tous les fichiers *.class* contenus dans un paquetage, représentant l'unité de conversion du convertisseur, et les convertit en un fichier *.cap* (Converted APplet). Le fichier CAP est

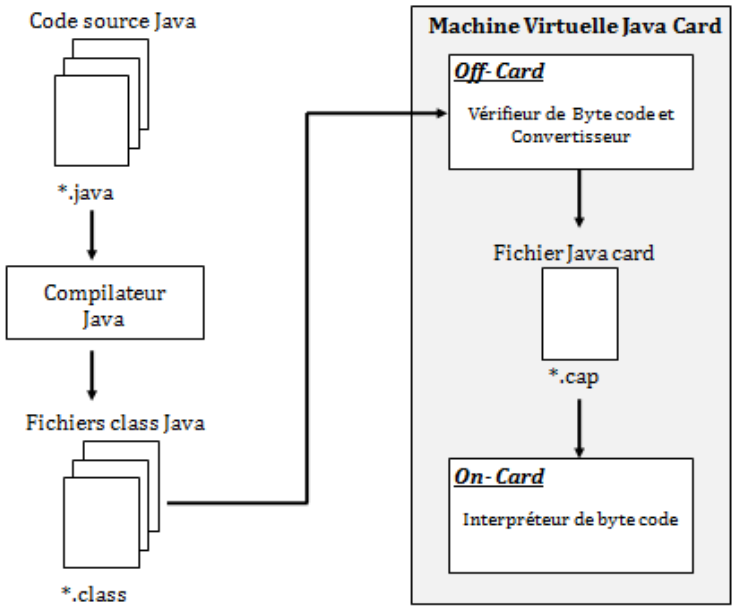

Fig 2.2 – La machine virtuelle Java Card (JCVM)

ensuite chargé sur une Java Card et exécuté par l'interpréteur.

En effet, ce n'est pas le fichier *.class* qui est chargé sur la carte, mais un fichier *.cap*. Etant donné que la carte n'est utilisée que comme un interpréteur de code, il s'agit de faciliter l'installation et prévoir un format plus simple à exécuter pour une plate-forme disposant de peu de ressources. De ce constat est né le format de fichier CAP. En effet, le fichier Class n'est pas directement exécutable et doit subir une phase importante d'édition de liens. Au contraire, le format du fichier CAP est prévu pour une exécution immédiate et repose sur une phase d'édition de liens simplifiée. Pour obtenir le fichier CAP, un outil placé hors de la carte (*i.e.* le convertisseur) transforme le fichier Class en fichier CAP une fois la vérification du byte code effectuée [Lan06].

L'Environnement d'exécution Java Card

Appelé aussi JCRE (pour *Java Card Runtime Environment*), il assure le rôle du système d'exploitation en gérant les ressources de la carte, la communication avec le réseau, l'exécution des applets (voir section 2.5) et leur sécurité. Le JCRE sépare les applets de la propriété de la technologie des cartes à puce des vendeurs. Les applets sont ainsi plus faciles à écrire et sont indépendantes de l'architecture privée des cartes à puce.

Les APIs Java Card

Elles consistent en un ensemble de classes optimisées pour la programmation d'applications pour cartes à puce conformément à la norme ISO 7816. Plusieurs classes Java ne sont pas nécessaires et donc non disponibles dans la plateforme Java Card. Les classes dans ces APIs incluent des classes adaptées à la plateforme Java Card fournissant un support pour le langage et des services de cryptographie. De

plus elles contiennent des classes spécialement conçues pour supporter la norme ISO 7816. Pour une liste complète et détaillée des APIs disponibles voir [Mic09e]. Entre autres :

- *java.lang* : fournit les fondements pour le support du langage Java. Il supporte les classes *Object, Throwable* et *Exception* ;
- *javacard.framework* : apporte les classes et les interfaces essentielles pour programmer des applet Java Card. Il définit les classes : *Applet*, *APDU*, *JCSystem*, *OwnerPIN*, *AID*, *Util* ;
- *javacard.security* : fournit une architecture aux fonctions cryptographiques supportées par la plateforme Java Card ;
- *javacardx.crypto* : est un paquetage d'extension. Il définit une classe *Cipher* qui supporte les fonctions de chiffrement et de déchiffrement pour les différents algorithmes implémentés sur la carte ;
- *java.rmi* : définit l'interface *Remote* qui identifie les interfaces dont les méthodes peuvent être appelées par des applications clientes ;

2.4.2 Java Card 3 édition « *Connected* »

Elle est conçue pour les cartes de nouvelle génération disposant de plus de ressources, supportant ainsi un sous-ensemble plus large et plus riche du langage Java. Cette édition de la plateforme introduit dans son architecture (figure 2.3) une nouvelle machine virtuelle (pas de division en deux parties) ainsi qu'un nouvel environnement d'exécution qui supporte trois modèles d'application [Mic09e], qui sont :

- *Les applets classiques.* Ce modèle est hérité des précédentes versions de Java Card et donc dispose des mêmes caractéristiques. Autrement dit, ces applications sont développées en se basant uniquement sur les fonctionnalités et librairies offertes par la version « *Classic* » et versions antérieures. Le protocole APDU est utilisé pour assurer la communication entre ces applications et la carte.
- *Les applets étendues.* Ce modèle d'application est aussi basé sur celui des applets, mais avec des améliorations apportées par la version « *Connected* » telles que la gestion du multithreading. Le protocole APDU est utilisé pour assurer la communication entre ces applications et la carte.
- *Les applications Web.* Ce modèle d'application est basé sur la construction des « *servlets* ». Une servlet est une application Java qui permet de créer dynamiquement des données au sein d'un serveur Web. Il hérite des nouvelles APIs introduites par cette édition de la plateforme (*i.e.* l'édition « *Connected* »). Le protocole HTTP ou HTTPS pour les transferts sécurisés sont utilisés pour effectuer les communications avec la carte.

Cette architecture est définie par les spécifications suivantes :

- *Java Card Runtime Environment Specification, Connected Edition*[Mic09e] ;
- *Java Card Application Programming Interface, Connected Edition*[Mic09d] ;
- *Java Card Virtual Machine Specification, Connected Edition*[Mic09g] ;
- *Java Card Servlet Specification, Connected Edition*[Mic09f].

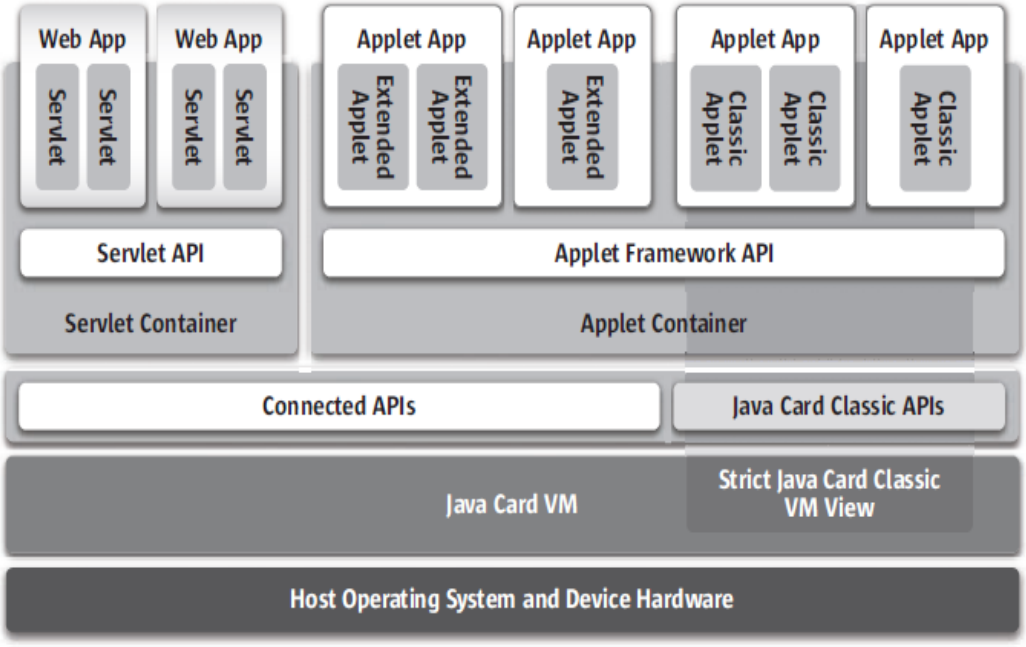

Fig 2.3 – Architecture de Java Card 3 Connected Edition [Mic08]

2.4.3 Java Card et le sous-ensemble du langage Java

Afin d'adapter la technologie Java Card aux ressources limitées des cartes à puce, seules certaines caractéristiques bien choisies du langage Java sont supportées. L'édition « *Connected* »est conçue pour les cartes de nouvelle génération, dont les contraintes matérielles sont moins fortes que celles pour les cartes classiques. De ce fait, chacune des deux éditions « *Classic* »et « *Connected* »supporte un sous-ensemble Java spécifique.

L'édition « *Classic* »et Java

- ***Les éléments non supportés du langage Java***
  - *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 ;
  - *Les types énumérés* : pas de types énumérés, ni de mot clef enum ;
  - 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. Par exemple, dans le paquetage *Java.lang* les classes supportées sont seulement *Object* et *Throwable* ;
  - *Les fichiers Class* : Le format des applications chargées est le *fichier CAP (Converted APplet)* qui est un fichier *Class* dont toutes les éditions de liens sont déjà faites afin d'accélérer l'exécution. (on reviendra plus en détails sur le fichier CAP dans la section 3.1)
- ***Les éléments supportés du langage Java***
  - *Paquetages.* La plate-forme Java Card suit les règles standards des paquetages de la plate-forme Java. Les classes des APIs Java Card sont écrites comme des fichiers sources Java qui incluent les désignations des paquetages. Les mécanismes de paquetages sont utilisés pour identifier et contrôler les accès aux classes, aux champs statiques et aux méthodes statiques.
  - *Création dynamique d'objet.* Les programmes de la plate-forme Java Card supportent la création dynamique d'objet, les instances de classes et les tableaux. Comme la machine virtuelle Java Card ne fournit pas nécessairement de ramasse-miettes. Tous les objets créés peuvent continuer à exister et à consommer des ressources même après qu'ils soient devenus inaccessibles.
  - *Méthodes virtuelles.* L'invocation de méthodes virtuelles sur des objets dans un programme écrit pour la plate-forme Java Card est exactement comme dans un programme écrit pour la plate-forme Java. L'héritage est supporté, incluant l'utilisation du mot clé *super*.
  - *Interfaces.* Les classes Java Card peuvent définir ou implémenter des interfaces comme dans le langage Java. L'invocation de méthodes sur les types d'interfaces fonctionne comme attendu.
  - *Exceptions.* Les programmes Java Card peuvent définir, lancer et attraper des exceptions, comme les programmes Java. La classe *Throwable* et ses sous-classes importantes sont supportées. Quelques sous-classes de *Exception* et *Error* sont omises.
  - les types simples : *boolean*, *short* et *byte*,
  - les classes *Object* et *Throwable*,
  - Le type *int* et le mécanisme de suppression d'objets sont supportés en option.

L'édition « *Connected* »et Java

- ***Les éléments non supportés du langage Java***
  - Pas de prise en charge des types *float* et *doubles* ;
  - Pas de *classloader* définit par l'utilisateur car les classloaders de la plateforme ne peuvent pas être modifiés. Cette fonctionnalité a été supprimée pour des raisons de sécurité ;
  - Pas de finalisation des instances de classes ;
  - *La gestion d'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.

- ***Les éléments supportés du langage Java***
  - Les types de données existants en Java sont supportés sauf les types *double* et *float* ;
  - Le *multi-threading* ;
  - Le support des fichiers *Class* comme format de chargement (au lieu des fichiers CAP) avec l'édition des liens qui se fait sur la carte ;
  - La destruction automatique des objets non utilisés ou *Automatic Garbage Collection*.

2.5 Les applet Java Card

Les applets sont donc les applications qui s'exécutent dans la carte en interagissant avec le JCRE. Elles sont identifiées par un identifiant unique **AID** (*Application IDentifier*). De même chaque paquetage est identifié par un AID unique. La construction d'un AID est définie par la norme ISO 7816-5. Il est représenté par un tableau dont la taille varie entre 5 et 16 octets (tableau 2.1). Il est constitué par la concaténation du RID (*Ressource IDentifier*) de taille fixe de 5 octets et du PIX (*Proprietary Identifier eXtension*) de taille variant entre 0 et 11 octets. C'est l'ISO qui procède à l'attribution des RIDs aux entreprises afin que chacune ait son propre RID et ensuite chaque entreprise contrôle la gestion des PIXs. Chaque applet et chaque paquetage possède un AID unique. Le RID identifiant le fournisseur de l'applet, les applets définies dans un paquetage partagent le même RID que celui-ci.

Application IDentifier (AID)	
RID (5 octets)	PIX (0-11 octets)

Tab 2.1 – Structure de l'AID

Cycle de vie d'une applet

Plusieurs applets peuvent résider dans la carte mais une seule peut être active à la fois. Afin d'interagir avec le JCRE, chaque applet doit implémenter les méthodes suivantes :

- **install/uninstall** : pour installer ou supprimer une applet de la carte ;
- **register** : pour enregistrer l'applet auprès du JCRE ;
- **select/deselect** : pour activer ou désactiver l'applet ;
- **process** : pour traiter une commande APDU et envoyer une réponse APDU.

Lorsque l'applet est chargée dans la carte elle est inaccessible tant qu'elle n'est pas installée. La vie de l'applet (figure 2.4) commence lorsqu'une instance est créée par la méthode *install()* et enregistrée au sein du JCRE grâce à la méthode *register()*. Une fois ceci fait, l'applet reste inactive jusqu'à ce qu'elle soit explicitement sélectionnée via une commande SELECT APDU envoyée au JCRE. Ce dernier consulte sa table interne pour trouver l'applet dont l'AID correspond à celui spécifié dans la commande. Une fois l'applet sélectionnée par la méthode *select()*, le JCRE fait suivre toutes les commandes APDU à la méthode *process()* où chaque commande APDU est interprétée pour exécuter la tâche qu'elle spécifie. Pour chaque commande APDU, l'applet répond en envoyant une réponse APDU informant du résultat du traitement de la commande. Ce dialogue commande-réponse continue

jusqu'à ce qu'une nouvelle applet soit sélectionnée (mais ceci après la dé-selection de l'applet courante, via la méthode *deselect()*, qui devient inactive) ou bien que la carte soit retirée du lecteur.

Le cycle de vie de l'applet se termine lorsque le JCRE reçoit une demande d'effacement de l'instance de l'applet et/ou du paquetage auquel elle appartient (*delete()*). En revanche s'il n'y a pas une telle demande, le cycle de vie de l'applet s'étend par défaut jusqu'à la fin du cycle de vie de la carte.

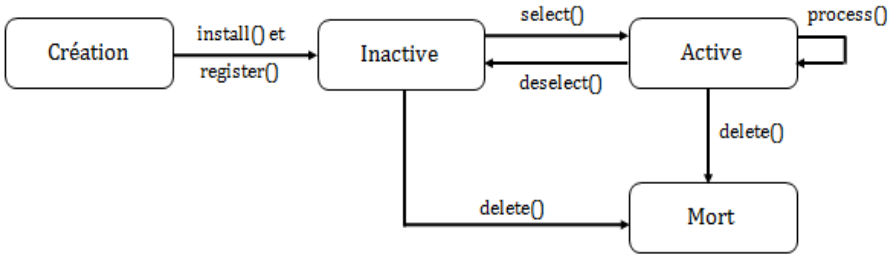

Fig 2.4 – Cycle de vie d'une applet Java Card

2.6 Sécurité de Java Card

La sécurité offerte par la plateforme Java Card est la combinaison de la sécurité intégrée au langage Java et des mécanismes intégrés à la plateforme elle même.

2.6.1 Sécurité du langage Java

Java Card supporte un sous-ensemble du langage Java, approprié au développement des applications pour cartes à puce. Ce qui implique qu'elle hérite les mécanismes de sécurité offerts par ce sous-ensemble du langage, à savoir :

- Java est un langage fortement typé ce qui permet d'éviter les conversions de types non autorisées ;
- L'opération de cast suit des règles strictes. Un cast implicite d'un sous-type vers un super-type et un cast explicite obligatoire d'un type vers un sous-type.
- Java n'utilise pas d'arithmétique sur les pointeurs, il n'a pas un moyen de forger des pointeurs ;
- Le niveau d'accès de toutes les classes, méthodes et champs est strictement contrôlé (Public, Private, Protected)

2.6.2 Sécurité de la plateforme

Elle est assurée par un ensemble de mécanismes, chacun d'entre eux assure une partie de la sécurité tout en étant complémentaire avec les autres mécanismes pour offrir une sécurité maximale à la carte. Les principaux mécanismes sécuritaires intégrés aux cartes Java Card 3.0 *Classic Edition* et versions antérieures sont :

Le vérifieur de byte code

Ce composant crucial peut être vu comme étant le processus offensif de la sécurité de Java Card , du fait qu'il intervient avant le chargement des applets sur la carte. Il a pour objectif de s'assurer que le code à charger peut être exécuté sans risques par la machine virtuelle et ne peut pas outrepasser les mécanismes de sécurité de haut niveau. La vérification consiste à effectuer une analyse statique du code mobile à charger. Cette analyse assure que le fichier contenant l'applet (fichier CAP) soit un fichier bien formé, qu'il n'y a pas de débordement de pile, que le flot d'exécution reste confiné sur du byte code valide, que chaque argument d'une instruction soit d'un type correct et que les appels de méthodes soient effectués conformément à leurs attributs de visibilité (*public, protected, private*)[Lan06]. Vu les ressources limitées des cartes à puce le processus de vérification de byte code, coûteux en termes de temps d'exécution et d'espace mémoire, se fait dans la partie hors carte pour toutes les versions de Java Card antérieures à la version 3.0 *Classic Edition* y compris celle-ci.

Comme le vérifieur de byte code constitue le cœur du présent travail, le chapitre 3 est consacré pour exposer les détails relatifs à ce composant.

Le pare-feu

Il a pour mission d'isoler les applets et les différents objets créés au sein d'espaces protégés appelés *contextes* (figure 2.5). Un contexte est associé avec un paquetage de telle sorte que toutes les applets d'un même paquetage partagent le même contexte. Le JCRE possède son propre contexte avec des privilèges spéciaux (Les applets appartenant à ce contexte peuvent accéder aux objets de n'importe quel autre contexte de la carte). Ainsi, sur la plateforme Java Card chaque objet créé appartient soit

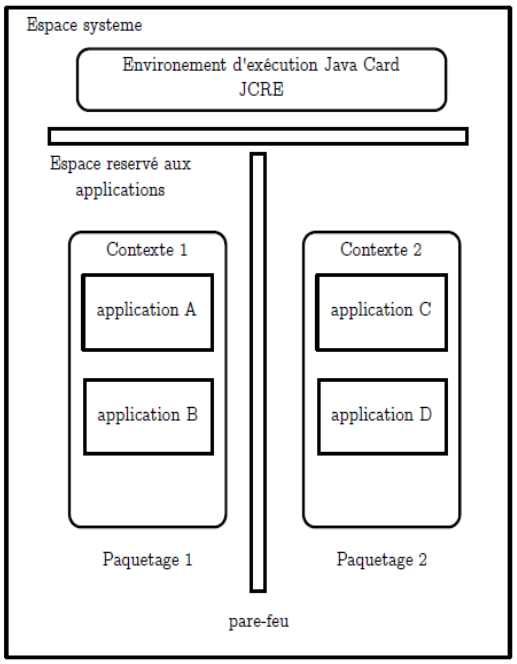

Fig 2.5 – Le pare-feu Java Card

au contexte d'une applet soit à celui du JCRE, le propriétaire de l'objet créé étant l'applet du contexte

courant. De plus, un objet est accessible seulement depuis le contexte de son propriétaire ce qui évite tout accès non autorisé à cet objet. Dans le cas où les applets ont besoin de partager des données ou d'accéder à des services du JCRE, des mécanismes sécurisés de partage accessibles via les APIs spéciales (*javacard.framework.shareable*) sont utilisés.

Atomicité et mécanisme de transaction

Ces deux notions ont été introduites dans l'environnement Java Card afin de garantir l'intégrité des données lors de la mise à jour des objets dans la mémoire persistante de la carte. L'interpréteur Java Card doit assurer que la mise à jour des champs des objets persistants soit atomique (*i.e* totalement exécutée ou pas du tout) afin de palier à tout incident (telle qu'une perte d'alimentation) pouvant remettre en cause l'intégrité des données modifiées. L'atomicité est garantie dans Java Card à l'aide d'un mécanisme de transaction qui s'assure que *toutes ou aucune* des opérations à l'intérieur d'un bloc soient terminées. Le déclenchement, la terminaison ou l'annulation des transactions se fait via des méthodes des APIs (respectivement *beginTransaction, commitTransaction* et *abortTransaction* de la classe *JCSystem*). Le mécanisme de *rollback* des données est utilisé pour restaurer les anciennes données en cas d'échec ou en cas d'annulation de la transaction.

Les cartes Java Card 3.0 *Connected Edition* disposent de mécanismes de sécurité additionnels [Mic08], adaptés à son architecture évoluée, par rapport aux autres versions Java Card. Entre autres :

- La présence obligatoire d'un vérifieur de byte code embarqué (mais il s'agit d'un vérifieur de fichiers *Class* et non pas de fichiers *CAP*) ;
- Un mécanisme d'isolation de code ;
- Un mécanisme de contrôle d'accès aux services de la plateforme ainsi qu'à ses services partagés qui gère les autorisations en utilisant un fichier définissant les permissions (fichier *policy*) rajouté lors du chargement de l'application ;
- Les annotations de sécurité qui sont utilisées pour protéger le contenu des classes ou des méthodes sensibles ;
- Les communications à travers le protocole HTTP (HyperText Transmission Protocol) sont sécurisées à l'aide de l'algorithme cryptographique TLS (*Transport Layer Security*) ;
- Une politique de sécurité renforcée par des règles à base de rôles et de permissions .

2.7 Les attaques contre cartes à puce

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 [Sér10]. Le but d'un attaquant est de pouvoir accéder aux informations et aux secrets contenus dans la carte (code PIN, clés secrètes, etc) ou tout simplement de nuire aux applications embarquées afin de tester le niveau sécuritaire de la carte [NSIcL09].

La sécurité d'une carte à puce peut être contournée de plusieurs manières : soit en prenant le matériel en défaut, soit en prenant l'applicatif ou le système en défaut. Ainsi on peut classer les

attaques connues contre cartes à puce selon l'arborescence suivante (figure 2.6).

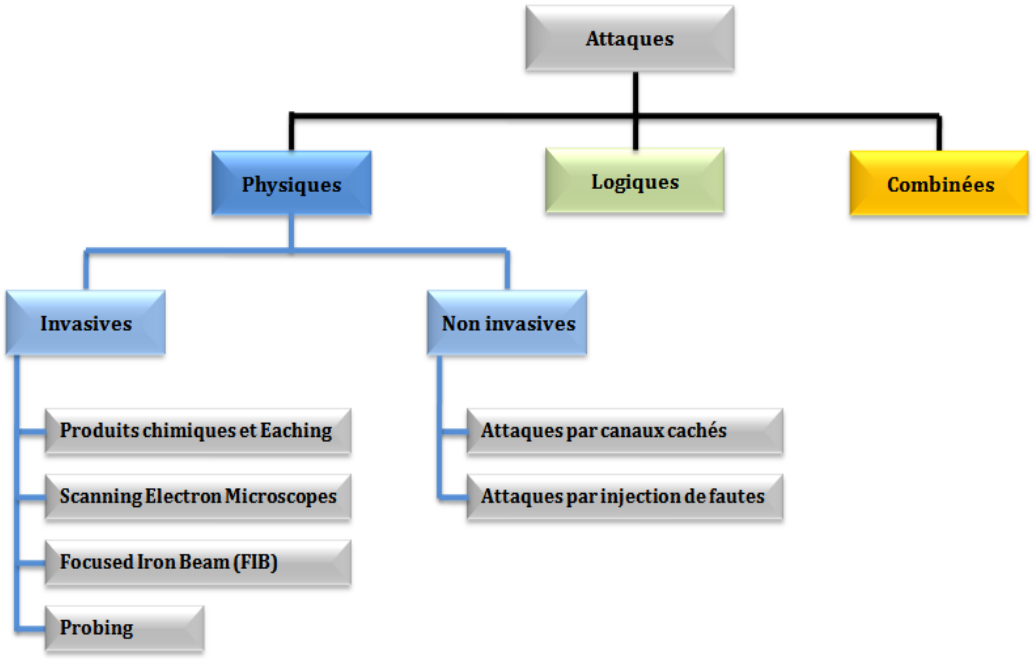

Fig 2.6 – Classification des attaques contre cartes à puce

2.7.1 Attaques physiques ou matérielles

Ces attaques touchent la partie matérielle de la carte à puce. Elles consistent en une analyse et/ou modification des circuits électroniques de la carte, souvent contenant les implémentations des algorithmes cryptographiques, afin d'obtenir des informations sensibles de cette dernière. Ces attaques peuvent être : *invasives* ou *non invasives*.

Attaques invasives

Ces attaques consistent à un accès physique aux composants internes de la carte afin de réaliser des observations directes ou encore des modifications de la structure des circuits. Il s'agit d'attaques extrêmement performantes, qui présentent cependant l'inconvénient majeur d'être destructive : une fois l'attaque effectuée, la carte devient inutilisable. Le but étant de récupérer un ensemble d'informations de la carte en se basant sur une cartographie des circuits. Pour arriver à cela, il faut une étape d'isolation physique ou chimique des circuits de la carte (d'où l'aspect *invasif*). Plusieurs méthodes pour réaliser de telles attaques sont citées dans [Wit02] :

- *Produits chimiques et gravure à eau forte (Etching).* La technique du *Etching* permet de décapsuler les circuits et isoler les différentes couches. Ce qui permet d'obtenir tous les blocs fonctionnels d'un circuit afin qu'il soit accessible pour analyse. Alors que les produits chimiques permettent de dissoudre la couche de résine qui fixe les circuits.
- *Scanning Electron Microscopes (SEM).* Ces microscopes sont utilisés pour effectuer l'analyse optique et le *reverse engineering* afin de reconstruire des circuits complets ou le code source

du système d'exploitation à partir des bits de la ROM. Les SEMs permettent aussi d'observer des circuits durant une exécution (ceci est possible si le circuit a été soigneusement préparé et peut toujours effectuer ses fonctions). L'analyse de ce circuit révèle les sections de code qui sont actives et même les valeurs des cellules de la mémoire peuvent être déterminées.

- *Le Probing.* Cette technique permet de positionner des micro-sondes, une sorte d'aiguilles d'écoute, arbitrairement sur les fils d'un circuit isolé mais qui réalise encore ses fonctions. Ceci permet de créer de nouveaux canaux vers l'extérieur de la carte. Si le bus de données peut être localisé, les sondes permettront de capter tous les échanges de données entre le CPU et les mémoires. Avec une analyse poussée, ceci permet d'obtenir le code du programme en exécution ainsi que les clés incluses dans les programmes. Il y a aussi une possibilité de modifier les micro-instructions et donc détourner l'exécution du CPU.
- *FIB (Focused Ion Beam).* Cette technique permet de modifier les circuits de la carte en rajoutant ou éliminant des pistes conductrices à l'aide de rayons d'ions. Ceci a pour conséquences : la reconnexion de circuits séparés, l'envoi de signaux internes cachés vers l'extérieur, l'élargissement des pistes fines et fragiles pour déposer des sondes, etc.

Attaques non invasives

En opposition avec les attaques invasives, les attaques non invasives ne nécessitent pas de détruire la carte à puce cible. Deux types d'attaques peuvent être distinguées à ce niveau : les attaques basées sur l'observation de l'environnement de la carte (*attaques par canaux cachés*) et celles basées sur la perturbation de cette dernière (*attaques par injection de fautes*).

1. **Attaques par canaux cachés**

   Ces attaques visent à analyser une grande quantité de données observées afin de filtrer les informations essentielles. Les phénomènes physiques utilisés pour observer le comportement d'un circuit électronique peuvent être : la consommation du courant, les radiations électromagnétiques, le temps d'exécution du CPU.

   - *Attaques par analyse de courant.* Elles sont basées sur le fait que la consommation électrique d'un module embarqué, à un instant précis, est fonction de l'instruction exécutée, des opérandes (adresses ou valeurs) manipulés ainsi que de l'état précédent du module (en particulier de la valeur des diverses cellules mémoires et des bus). En observant cette consommation, un attaquant peut donc retrouver de l'information sur les opérations effectuées ainsi que sur les valeurs des données manipulées par ces opérations [Gir07]. Les attaques par analyse de courant sont divisées en deux catégories :
     - Les attaques par analyse élémentaire de consommation appelées aussi **SPA** (*Simple Power Analysis*) [MDS99], [Man02], [Osw02]. Dont le but est d'essayer d'obtenir directement de l'information sur la clé secrète à partir d'une seule mesure de consommation.
     - Les attaques par analyse différentielle de consommation appelées aussi **DPA** (*Differential Power Analysis*) [JJK99], [DBLW02]. Elles permettent d'obtenir de l'information en utilisant des méthodes statistiques permettant de distinguer la faible corrélation qui existe entre la

valeur des données manipulées et les mesures de consommation électrique relevées.

- *Attaques par analyse du rayonnement électromagnétique.* Appelées aussi **EMA** (*ElectroMagnetic Analysis*) [SQ01], [MOG01]. Ces attaques sont très similaires à celles par analyse de courant. En effet, le courant utilisé par la puce crée un champ électromagnétique qui dépend lui aussi de l'activité de la puce. Ce champ peut être mesuré grâce à une sonde spécifique reliée à un oscilloscope et l'analyse de ce signal s'effectue ensuite de manière similaire à l'analyse de la consommation électrique. Cependant, une telle analyse permet d'isoler l'activité d'une toute petite partie de la puce. En effet, suivant la position de la sonde à la surface du composant, l'attaquant va pouvoir analyser le rayonnement électromagnétique de telle ou telle partie du composant. Par conséquent, le bruit induit par les autres parties de la puce lors d'une analyse de consommation électrique peut être grandement diminué voire même supprimé.
- *Attaques par analyse du temps d'exécution* [Koc96], [Mui01]. Elles sont basées sur le temps d'exécution des instructions dans le but de déduire des informations secrètes telles que les clés cryptographiques manipulées par un programme donné.

2. **Attaques par injection de fautes**

   Appelées aussi *attaques par perturbation* [GT04], [WNBE+04], [WNBE+04], [AG03], [Ver06] car elles visent à perturber le fonctionnement normal des circuits électroniques (modification du comportement) en introduisant des modifications physiques dans l'environnement. Le but étant d'introduire des fautes durant l'exécution d'un programme afin d'obtenir des informations sensibles comme les clés cryptographiques ou d'effectuer des traitements normalement sécurisés par la réalisation d'un évitement du test d'un code PIN. Il existe une multitude de façons pour réaliser une attaque par injection de faute, parmi lesquelles :

   - *Attaque électrique* [ABF+03] qui consiste à faire varier brusquement la tension d'entrée de l'alimentation de la puce afin de perturber l'exécution de certaines opérations.
   - *Attaque par variation de fréquence de l'horloge* [Ta10] qui consiste à faire varier la vitesse de l'horloge hors des limites autorisées par la carte afin d'induire des fautes au niveau du microprocesseur.
   - *Attaque optique ou par laser* [SA03], [AS02] dont l'idée est d'utiliser l'énergie d'une émission lumineuse (par exemple : flash d'un appareil à photos ou laser), concentrée sur la puce, pour perturber le silicium du composant. Ainsi, il est possible d'apporter suffisamment d'énergie à une cellule mémoire pour lui faire changer son contenu.
   - *Attaque par perturbation électromagnétique* [QS02], [SH07] qui consiste à émettre une forte pulsation magnétique près d'une cellule mémoire. Le champ magnétique crée des courants locaux à la surface du composant, pouvant ainsi générer une modification du contenu des cellules de la mémoire.

2.7.2 Attaques logiques

Avec l'apparition des cartes *ouvertes*, permettant de charger des applications dans la carte après émission de celle-ci, les *attaques logiques* se sont de plus en plus répandues. Elles utilisent des failles

pour contourner les protections mises en place [NSIcL09]. Ceci dans le but d'obtenir des données confidentielles ou effectuer des modifications non autorisées aux données de la carte. Les attaques logiques ne sont pas destructives (*i.e.* la partie hardware de la carte n'est pas affectée) et sont facilement reproductibles. Elles nécessitent un minimum d'équipement (une carte, un lecteur et un PC). Par contre leur succès nécessite une grande compétence, persistance et imagination des attaquants afin de trouver des *portes dérobées* (backdoors).

Comme une Java Card peut héberger des applets issues de divers fournisseurs, il devient possible d'avoir deux applets en provenance de deux fournisseurs concurrents qui co-existent dans la même carte. En outre, les fournisseurs ne suivent pas nécessairement les même standards de sécurité lors du développement de leurs applications. Par exemple, les conditions sécuritaires imposées pour une application financière sont largement plus strictes que celles d'une application de divertissement.

D'après [Wit03] les applets peuvent avoir certains comportements indésirables (classés par ordre de gravité) :

- Comportement inoffensif mais embarrassent ;
- Crash de la carte (de façon temporaire ou permanente) ;
- Déclenchement d'un comportement externe non autorisé (à l'aide de moyens externes) ;
- Révélation de la confidentialité de l'utilisateur ;
- Attaque d'autres applets de la carte ou la plateforme elle même.

C'est la dernière catégorie (*applet malicieuse*) qui présente le plus de risques pour Java Card. Les mesures de sécurité mises en œuvre par cette dernière contribuent toutes à une défense contre ce type de menaces. En considérant ce comportement indésirable (le dernier cas), trois types de menaces peuvent être identifiés :

- *Cas 1* : Une applet vérifiée qui abuse les caractéristiques de Java Card pour représenter un comportement indésirable et inoffensif.
- *Cas 2* : Une applet vérifiée qui exploite des bugs d'implémentation dans la plateforme ou les points faibles dans la spécification de Java Card pour attaquer d'autre applets ou la plateforme elle même.
- *Cas 3* : Une applet mal formée qui compromet la plateforme en arrivant à se charger dans la carte sans qu'elle soit vérifiée et procède à attaquer d'autres applets ou la plateforme.

Pour chacun de ces cas, l'auteur dans [Wit03] a donné les principes généraux pour mener des attaques logiques. Entre autre, il a proposé deux approches intéressantes : une approche qui exploite un bug d'implémentation du « *pare-feu* »et une autre approche utilisant une *confusion de type* dans une applet mal formée (cette approche a été reprise par O.Vertanen [Ver06]). La *confusion de type* consiste à utiliser deux références de types différents (et incompatibles) pour accéder à la même zone physique de la mémoire.

Par la suite, E.Poll et W.Mostowski [MP08] se sont basés sur cette approche pour proposer plusieurs méthodes d'obtention d'un code mal typé (manipulation du fichier CAP, utilisation du mécanisme de transactions, etc) ainsi que les scénarios détaillés des attaques exploitant cette confusion de type. L'objectif principal est de transformer un objet d'un type donné pour lequel un nombre limité

d'opérations est possible en un objet pour lequel d'autres opérations deviennent éligibles. Deux idées ont été exposées :

1. **Accès à un tableau de *byte* comme étant un tableau de *short***. Cette confusion de type permet de lire les données au-delà des limites du tableau de byte. Plus exactement, ça va engendrer le lecture de deux fois la taille du tableau de byte de données et qui sortent potentiellement du domaine de l'applet. La figure 2.7 résume ce cas.

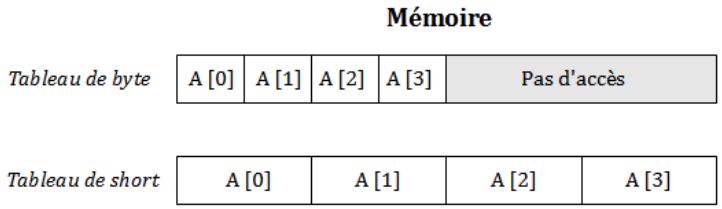

Fig 2.7 – Confusion de type entre tableaux

2. **Accès à un *objet* comme étant un *tableau***. Si les attributs du descripteur d'objet sont correctement renseignés et situés au mêmes emplacements mémoire alors il devient possible de modifier la taille du tableau en tant qu'attribut légitime d'un objet. À partir de là, il est possible de l'utiliser en tant que tableau ayant une taille égale à la mémoire de la carte et il est possible de lire/écrire partout en mémoire à partir de l'adresse à laquelle est stocké l'objet. Plusieurs façons d'exploiter cette méthode sont proposées :
   - Traiter une *référence* vers un objet en tant que *short* ;
   - Echanger la référence de deux objets même s'ils sont de types incompatibles ;
   - Manipuler des AID et rendre une applet référencée par le système invalide ;
   - Fabrication des références pour lire/écrire une partie de la mémoire.

Hyppönen dans [Hyp03] a proposé une autre approche qui exploite une faiblesse du *pare-feu*. Elle se base essentiellement sur les comportements des instructions statiques (*getstatic* et *putstatic*) et celui du composant *Reference-Location* du fichier CAP (voir section 3.1). L'instruction *getstatic* possède deux opérandes qui sont utilisées pour construire un index afin d'accéder à un champ statique dans une classe. Dans le cas normal, lors du chargement d'une applet, l'éditeur de liens se charge de résoudre toutes les références présentes dans le code et les remplace par des adresses mémoire.

L'idée de l'attaque est d'éliminer l'information, présente dans le composant *ReferenceLocation*, qui demande la résolution d'une référence (l'information représente l'opérande de *getstatic*). Ainsi, il devient possible de mettre n'importe quelle adresse mémoire à lire comme opérande de *getstatic* (par modification du fichier CAP). Ce qui permet d'avoir un accès direct à une adresse mémoire en contournant toutes les vérifications : une mauvaise vérification du fichier CAP (faiblesse du vérifieur de byte code), pas de vérification de contexte au runtime (faiblesse du pare-feu). Cependant, l'auteur n'a pas présenter des résultats expérimentaux ni une implémentation de l'attaque.

Les auteurs J.Iguchi-Cartigny et J.L.Lanet dans [ICL09] ont présenté une approche étendant celle de Hyppönen dans le but de lire et écrire n'importe où dans la mémoire de la carte (pas seulement à

une adresse mémoire). Pour cela, ils ont développé un code auto-modifiable (le cheval de Troie) qui vise la recherche et le remplacement de motifs afin de remplacer des parties de code, dans la carte, qui n'appartiennent pas au contexte de sécurité de l'applet développée en contournant les mécanismes de sécurité. Cette attaque a été menée sur plusieurs cartes Java Card. Ces dernières ont présenté des réactions différentes : résistance complète à l'aide des contremesures implémentées, exécution partielle du code en contournant certaines contremesures, aucune résistance face à l'attaque.

En se basant sur les travaux présentés dans [MP08], J.Hogenboom et W.Mostowski ont présenté dans [HM09] une nouvelle attaque qui utilise une confusion de type en exploitant un bug dans l'implémentation du mécanisme de transaction. Mais contrairement à [ICL09], cette attaque ne nécessite aucune manipulation du fichier à charger dans la carte (*i.e.* le fichier CAP) d'où sa simplicité de réalisation. Cette attaque a conduit à une lecture/écriture de la totalité du contenu de la mémoire, offrant ainsi la possibilité de manipuler des données sensibles d'autres applets ou même propres à la carte.

2.7.3 Attaques combinées

C'est la nouvelle tendance des attaques contre cartes à puce. L'idée est de combiner les attaques physiques et logiques afin de contourner le mécanisme de vérification du byte code qui a tant rendu les précédentes attaques logiques difficiles à réaliser avec succès. Ainsi il est devenu possible d'exploiter un ensemble plus large des vulnérabilités identifiées, chose qui alourdit l'impact sur la sécurité des cartes à puce.

E.Vetillard et A.Ferrari ont présenté dans [VF10] un travail théorique portant sur le développement d'attaques combinant des attaques logiques à d'autres physiques. Les auteurs ont exposé un scénario, testé sur des cartes classiques Java Card 2.2, relativement simple par rapport à d'autres cas possibles. L'idée est de forger une référence, récupérée à l'aide d'une technique de *reverse* de la mémoire, dans une application valide. Par la suite, réaliser une injection de l'instruction NOP (*i.e.* changer la valeur d'une cellule mémoire en 00) afin d'éviter un test d'accès aux références (*i.e.* contourner le *pare-feu*). Cette attaque a permis de récupérer les clés secrètes, qui sont sensées être confidentielles, possédées par une autre application.

G.Barbu *et al.* ont introduit dans [BTG10] une nouvelle approche combinant une injection de fautes avec une attaque logique. Deux cas d'étude, réalisés sur des cartes Java Card 3.0 *Connected Edition*, ont été présentés. Le premier cas montre comment introduire un code mal-formé bien qu'un vérifieur de byte code embarqué soit présent dans la carte. Alors que le second exemple expose comment transformer une méthode quelconque présente dans la carte en un code malicieux.

Et dernièrement, G.Bouffard *et al.* ont proposés dans [BICL11] une nouvelle approche combinant aussi les attaques logiques et physiques. Une modification du flux d'exécution à travers un rayon laser, permet de contourner le vérifieur de byte code embarqué et ainsi avoir le contrôle de la mémoire (dump de l'EEPROM).

2.8 Contremesures

Pour se protéger contre les attaques contre cartes à puce, de nombreuses méthodes de protection, aussi appelées contremesures, ont été publiées. Les protections peuvent être matérielles, dans ce cas elles sont déjà intégrées à la puce 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 la puce à des niveaux plus ou moins bas du système (système d'exploitation, machines virtuelles ou applications) [Sér10].

Cependant, ces méthodes de protection ont des contraintes très importantes car la carte à puce est un environnement où la taille des mémoires disponibles et les performances du processeur sont limitées. Les contremesures ont donc peu de flexibilité quant à leurs mises en œuvre alors que l'attaquant a beaucoup plus de souplesse quant au temps et à la puissance de calcul disponibles pour mener son attaque. L'objectif des contremesures implémentées dans un environnement carte à puce n'est pas d'obtenir une sécurité absolue mais plutôt de s'assurer que les moyens à mettre en œuvre pour réussir l'attaque soient jugés suffisamment importants en termes de temps, de coût et d'expertise requise suivant le niveau de sécurité désiré.

CHAPITRE 3

Le Vérifieur de Byte Code

Sommaire

La vérification de byte code est un processus essentiel dans le dispositif de sécurité de Java Card (et plus largement Java). Elle repose sur une analyse statique approfondie du code à charger dans la carte. Son but est de s'assurer du comportement correct du programme lors de son exécution.

Dans ce chapitre on va présenter le vérifieur de byte code plus en détails afin de mieux cerner la problématique qui tourne autour de ce composant de sécurité. Mais auparavant, on va présenter le fichier CAP, représentant le format binaire de compatibilité pour Java Card et le fichier d'entrée du vérifieur de byte code.

3.1 Le fichier CAP

Le fichier CAP (Converted APplet) est avant tout un fichier binaire (voir figure 3.1, un exemple d'un fichier CAP ouvert avec un éditeur hexadécimal). Il est produit par le convertisseur dans la partie hors carte de la machine virtuelle Java Card après conversion des fichiers *.class* contenus dans un

package. Chaque fichier CAP contient toutes les classes et interfaces définies dans un package Java. C'est ce fichier qui sera chargé dans la carte, au lieu du ficher *.class*, car il représente un format plus simple à exécuter pour une plateforme disposant de peu de ressources.

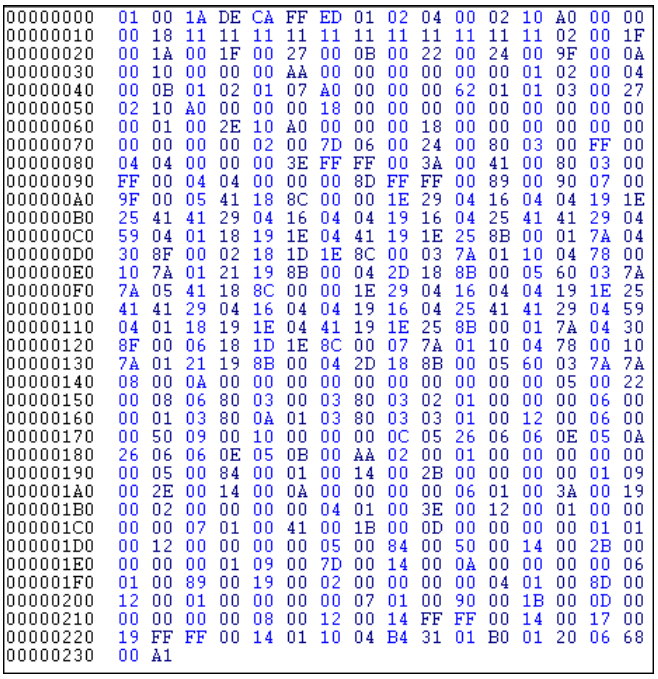

Fig 3.1 – Exemple d'un fichier CAP ouvert avec un éditeur hexadécimal

Le fichier CAP est constitué d'un ensemble de composants, chacun d'entre eux décrit un ensemble d'éléments du package en question ou un aspect du fichier CAP. Dans son ensemble, ce dernier est constitué de 12 composants dont 3 sont optionnels mais il y a une possibilité de définir de nouveaux composants (*Custom Components*). Ces derniers doivent aussi être conformes à la spécification générale d'un composant (son format) et compris pas la JCVM cible.

3.1.1 Format général

Le format général d'un composant du fichier CAP (tous les composants doivent être conformes à ce format) est le suivant (figure 3.2) :

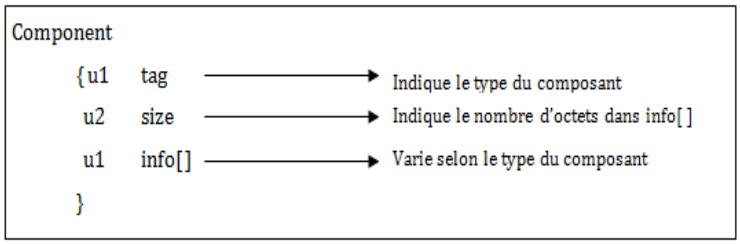

Fig 3.2 – Format général d'un composant du fichier CAP

D'après [Mic09c], u1 u2 u4 sont les types des données présentes dans le fichier CAP tels que :

– u1 : représente un octet non signé ;

– u2 : représente deux octets non signés ;
– u4 : représente quatre octets non signés.

3.1.2 Les composants du fichier CAP

Le tableau ci-dessous (tableau 3.1) résume la description des 12 composants contenus dans un fichier CAP tel que présenté dans [Mic09c].

Tag	Composant	Contenu
1	Header	– Regroupe les informations générales sur le fichier CAP et le package qui est défini
2	Directory	– Liste la taille de chaque composant défini dans le fichier CAP. – Contient aussi des entrées vers les nouveaux composants ajoutés (*custom components*)
3	Applet (optionnel)	– Contient une entrée pour chaque applet contenue dans le fichier CAP. – Si aucune applet n'est définie dans le package, ce composant ne sera pas présent dans le fichier CAP.
4	Import	– Liste l'ensemble des packages importés par les classes du package défini (ce dernier ne figure pas dans la liste)
5	ConstantPool	– Contient une entrée pour chaque : classe, méthode et champ référencé par les éléments du composant *Method* dans le fichier CAP.
6	Class	– Décrit chacune des classes et interfaces définies dans le package. – Les classes représentées dans le composant *Class* référencient d'autres entrées dans le même composant sous forme de : Superclass, Superinterface, références des interfaces implementées. – Les classes représentées dans ce composant contiennent aussi des références vers des méthodes virtuelles définies dans le composant *Method* du fichier CAP.

7	Method	– Décrit : chaque méthode déclarée dans le package, les méthodes abstraites définies par les classes, les handlers d'exception associés à chaque méthode.
8	StaticField	– Contient toutes les informations requises pour créer et initialiser une image de tous les champs statiques définis dans le package.
9	ReferenceLocation	– Représente la liste des décalages dans le composant *Method* vers des éléments qui ont des points d'entrée dans le composant *Constant Pool*.
10	Export (optionnel)	– Liste tous les éléments statiques qui peuvent être importés par des classes dans d'autres packages.
11	Descriptor	– Donne les informations nécessaires pour analyser et vérifier tous les éléments du fichier CAP.
12	Debug (optionnel)	– Contient toutes les méta-données nécessaires pour débuguer un package sur une JCVM appropriée (ce composant n'est pas chargé dans la carte).

Tab 3.1 – Description des composants du fichier CAP

3.1.3 Les liens d'interdépendance

En analysant la structure de chaque composant du fichier CAP telle que définie dans la spécification Java Card [Mic09c], il a été constaté que les composants se référencient entre eux (i.e. il y a de l'information partagée). On a résumé l'ensemble des liens d'interdépendance identifiés dans la figure 3.3.

3.2 Le vérifieur de byte code

Comme présenté précédemment dans la section 2.6.2, la vérification est l'un des principaux organes de sécurité de Java Card. Le rôle du vérifieur est de protéger la machine virtuelle contre tout débordement de l'application qui s'exécute. En particulier, il s'agit d'empêcher une application d'accéder au code ou aux données d'une autre application ainsi que de la machine virtuelle.

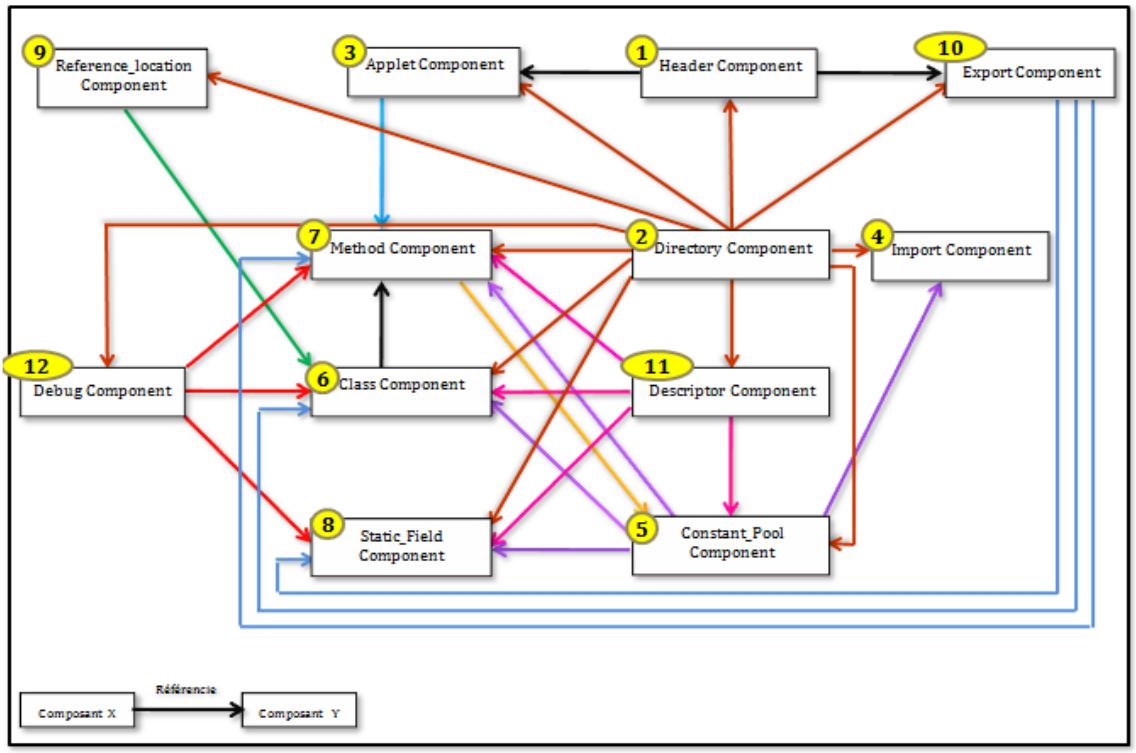

Fig 3.3 – Liens d'interdépendance entre les composants du fichier CAP

Cette vérification est réalisée en deux phases séparées mais complémentaires : une vérification de structure suivie d'une vérification de type.

3.2.1 Le vérifieur de structure

Cette partie réalise la première phase de la vérification du byte code. Son rôle est de s'assurer que le fichier à charger (i.e. le fichier CAP) dans la carte est un fichier bien formé afin qu'il ne soit pas mal interprété par le vérifieur de type ou par la machine virtuelle. Cette vérification consiste en une analyse syntaxique du flux de données en se basant sur la spécification du fichier CAP qui est donnée dans le chapitre 6 de [Mic09c]. En effet la vérification de la structure se base sur la définition de chaque composant. Mais comme les composants du fichier CAP ne sont pas indépendants les uns des autres (d'après ce qui a été présenté dans la section 3.1), on distingue deux types de la vérification structurelle :

- Les *tests internes* qui consistent à vérifier la cohérence interne de chaque composant du fichier CAP. Le but est de s'assurer que les données qui sont sensées représenter un composant ont les bonnes propriétés de ce composant. Ces dernières sont extraites de la description du fichier CAP fournie par la spécification.
- Les *tests externes* qui s'assurent de la cohérence entre les différents composants du fichier CAP (i.e. validité des liens d'interdépendance). En effet, les composants se référencient entre eux (figure 3.3) et du fait il y a de l'information partagée dont il faut s'assurer de sa cohérence. A titre d'exemple, chaque composant du fichier CAP contient un champ *"size"* qui indique sa taille.

De plus, le composant *Directory* contient un tableau *"component-sizes"* contenant les tailles des différents composants du fichier CAP. Donc pour chaque composant, il faudra vérifier que les deux tailles indiquées soient égales.

3.2.2 Le vérifieur de type

Cette partie assure la deuxième phase de la vérification du byte code. Elle consiste en une analyse sémantique du code. Elle vise à faire respecter les règles de typage définies par Java Card. Cette vérification est effectuée méthode par méthode et doit être faite pour chaque méthode du package, *i.e.* pour chaque méthode contenue dans le composant *Method* du fichier CAP à vérifier. Autrement dit, ce processus consiste à réaliser une interprétation abstraite du byte code de chaque méthode. Cette interprétation utilise des algorithmes d'inférence de type et les concepts introduits dans un article de G.Kildall [Kil73]. La partie vérification du typage s'assure qu'aucune conversion de type interdite d'après les règles de typage de Java Card n'est effectuée par le programme. Par exemple, un entier ne peut pas être converti en référence sur un objet. De même, les arguments passés en paramètre à une méthode doivent être de types compatibles avec ceux attendus par la méthode. Cette partie de la vérification est la plus complexe et la plus coûteuse à la fois en temps et en mémoire. En effet, cela nécessite de calculer le type de chaque variable et de chaque élément dans la pile pour chaque instruction et chaque chemin d'exécution possible.[BCR03]

3.3 Vérifieur de byte code et code mobile

La Java Card apporte à la carte à puce l'interopérabilité du langage Java. Ainsi, les applications développées pour Java Card peuvent s'exécuter sur toute carte implémentant la machine virtuelle Java Card. Le code généré est un code dit *mobile*, du fait qu'il peut transiter sur un réseau avant d'être chargé sur une carte à puce pour y être exécuté. En effet, Java Card prévoit la possibilité de charger du code après son déploiement. Cependant, le schéma de déploiement n'est pas exempt de problèmes.

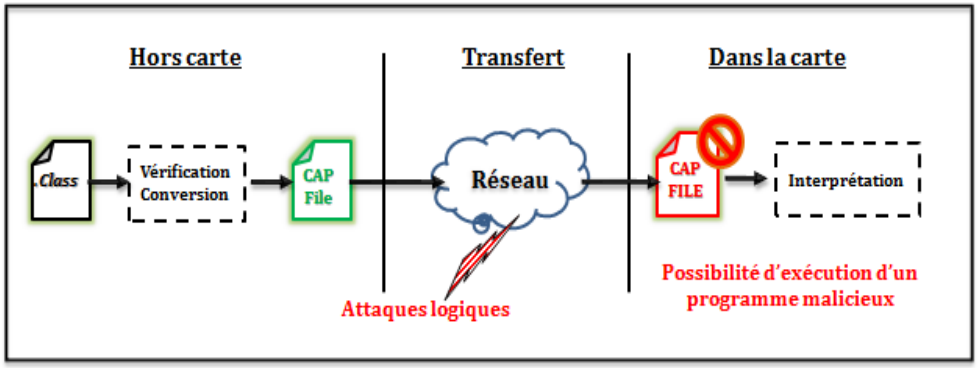

Fig 3.4 – Problématique liée au code mobile

Le problème majeur identifié concerne la validité et l'innocuité du code chargé pour la machine virtuelle. En effet, dans le schéma de chargement de Java Card (figure 3.4), le fichier CAP est produit et

vérifié dans la partie hors carte de la machine virtuelle Java Card (dans tout ce qui suit, on sous-entend par Java Card la version 3.0 « *Classic* » et versions antérieurs). Mais par la suite, ce fichier CAP peut transiter sur le réseau Internet pour arriver au terminal effectuant le chargement dans la carte. Donc, rien n'assure que ce fichier n'a pas été modifié lors de son transfert et ne menace pas l'intégrité de la machine virtuelle (voir les attaques logiques déjà présentées dans la section 2.7.2).

Dans le but d'augmenter la confiance accordée à un fichier CAP à charger après délivrance d'une carte à puce, plusieurs solutions ont été proposées dans la littérature (section 3.4) .

3.4 Travaux de recherche autour du vérifieur de byte code

La solution préventive au problème posé est que : si la carte ne peut pas charger librement du code alors, le code ne peut pas menacer l'intégrité de la carte. Ceci revient à interdire le chargement de code après le déploiement des cartes, hors environnement sécurisé. Ainsi, si l'utilisateur final de la carte veut charger de nouvelles applications, il doit se rendre dans un lieu où la sécurité est garantie (les applications sont vérifiées et certifiées).

Le chargement des applications dans un environnement sécurisé reste la solution la plus sûre et d'ailleurs elle est largement utilisée industriellement. Cependant, le problème avec cette solution est le niveau de fonctionnalité limité. En effet, le succès de Java Card vient de sa capacité à charger et exécuter des applets : la multi-application. Pour cela, il a fallut trouver des solutions répondant à la problématique.

Signature Numérique

Cette solution consiste à signer le fichier CAP à charger dans la carte à sa sortie du vérifieur puis à vérifier cette signature sur la carte. Cette solution est efficace (temps de mise en œuvre et aspect sécuritaire) et peu coûteuse en temps puisque la vérification sur la carte est faite par le processeur cryptographique. En revanche, son principal problème est un déploiement centralisé des applications puisqu'il requiert un tiers de confiance. En effet, rôle de ce dernier est d'assurer au client (porteur final de la carte) que l'application proposée par le fournisseur d'applications ne menace pas l'intégrité et le fonctionnement même de la carte. Pour cela, ce tiers de confiance teste et vérifie l'application. Après cela, il produit un certificat cryptographique identifiant l'application. Ce certificat est envoyé avec le code de l'application sur la carte du client. La carte, et plus exactement GlobalPlatforme (voir section 7.1.4), décrypte le certificat et s'assure qu'il correspond bien à l'application grâce aux protocoles cryptographiques. Si c'est bien le cas, l'application est installée sur la carte et peut s'exécuter à la demande du client. Tout cela diminue la flexibilité du système et du fait alourdit le déploiement des applications.

Machine Virtuelle Défensive

Elle consiste à construire dans la carte une machine virtuelle défensive [Coh97] qui réalisera les opérations de vérification avant l'exécution de chaque byte code. Cette solution a le grand avantage

d'être autonome et donc de pouvoir accepter le chargement d'applications malicieuses sans risque puisqu'elle les bloquera lors de leur exécution. Cependant, bien qu'elle soit très sûre, elle n'est pas très efficace à cause de la surcharge en temps et en mémoire que représente la vérification de chaque byte code.

Proof-Carrying Code

La technique du « *Proof-Carrying Code* »a été proposée par G.C.Necula et P.Lee [NL97] puis adaptée pour Java Card par E.Rose et K.Rose [RR98]. Elle consiste à générer à l'extérieur de la carte une preuve pour le code à charger qui est ensuite vérifiée dans la carte par un vérifieur spécialisé. L'avantage de cette méthode est que la preuve à vérifier sur la carte est plus simple que le travail qu'aurait dû faire un vérifieur embarqué. Cependant, cette approche souffre de quelques problèmes. D'une part, la taille des applications chargées augmente puisqu'elles incluent la preuve et leur déploiement devient complexe car l'application doit être traitée dans un générateur de preuves forcément centralisé. D'autre part, des applications valides peuvent être rejetées si la preuve n'est pas fournie.

Transformations du code

Cette solution a été proposée par X.Leroy [Ler01][Ler02]. Elle consiste à normaliser le code hors carte pour que chaque variable manipulée par la machine virtuelle soit d'un seul type (par exemple un registre de variable locale ne pourra pas dans le même appel de méthode contenir un short puis une référence à un objet). Puis, un vérifieur est embarqué sur la carte pour assurer l'application de cette propriété. Un problème de cette méthode est qu'elle nécessite que les applications passent par un outil de transformation du code propriétaire fourni par *Trusted Logic* sinon elles seront rejetées. Un autre problème plus important de cette approche est l'augmentation du nombre de variables et une diminution de leur ré-utilisabilité pour un périphérique où la mémoire est très contrainte. Donc, la carte pourra avoir besoin de plus de ressource mémoire pour exécuter un programme et elle pourrait refuser des codes optimisés.

Vérifieur de byte code Embarqué

Chacune des solutions précédentes montre certaines faiblesses qui diminuent de son efficacité. De ce fait, une solution qui paraît idéale serait un vérifieur de byte code embarqué autonome. En effet, un tel vérifieur est décentralisé et théoriquement il ne refusera pas une applet valide. Cette solution a longtemps été considérée comme impossible mais elle a pu être réalisée.

L.Casset *et al* [Cas02] [CBR02] [BCR03] ont utilisé la méthode B dans le cadre du développement sûr d'un vérifieur de byte code à embarquer dans une Java Card. Leur objectif était de formaliser le vérifieur de bytecode pour le langage Java Card complet (hormis les instructions jsr et ret qui sont traitées séparément), et de montrer qu'une implémentation générée à partir de cette formalisation, par raffinements successifs, respecte les contraintes de la carte à puce.

D.Deville et G.Grimaud [DG02] ont proposé une approche orientée système cherchant à maximiser

l'usage qui peut être fait des ressources matérielles propres aux cartes à puce ce qui a permis d'embarquer un vérifieur complet dans cette dernière. Cette approche utilise un système performant de caches de données entre les mémoires volatile et persistante de la carte et exploitent un codage des informations de types non stressant [DGR01] pour la mémoire persistante.

De tels travaux remettent en cause l'assertion de base sur l'impossibilité de réaliser la vérification de byte code sur la carte. De plus, la dernière spécification de Java Card (Java Card 3.0 « *Connected Edition* ») impose l'utilisation à l'intérieur de la machine virtuelle Java Card et donc de la carte, le vérifieur de byte code. Dès lors une grande partie des attaques logiques ne peuvent plus fonctionner. Ceci entraîne une augmentation de la sécurité des cartes à puce. Cependant, comme on a exposé précédemment dans la section 2.7.2 certaines attaques restent encore possibles bien qu'un vérifieur de byte code soit embarqué.

3.5 Problématique autour du vérifieur de byte code embarqué

Dans ce nouveau schéma de déploiement (figure 3.5) on n'a plus besoin du tiers de confiance : la carte assure seule sa propre sécurité (sa plateforme et ses applications). Une fois l'application choisie, la carte charge le code et effectue en même temps la vérification. Si le code ne menace pas l'intégrité de la machine virtuelle, alors le processus de chargement peut se poursuivre et l'application peut être installée. Dans le cas contraire, l'application est rejetée.

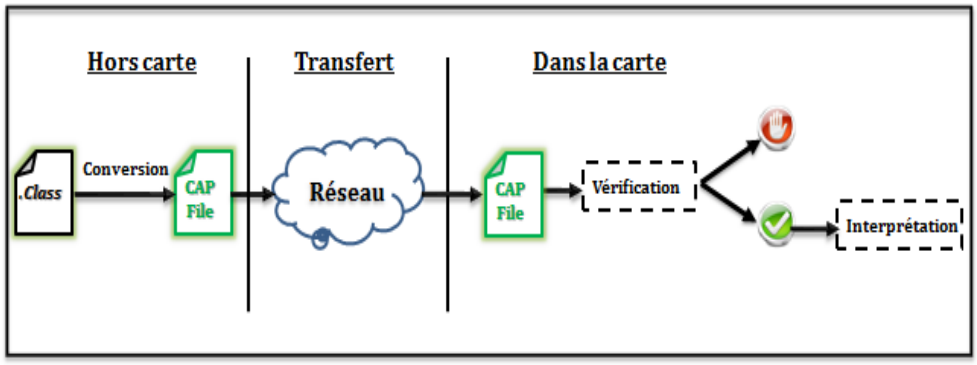

Fig 3.5 – Nouveau schéma de déploiement avec vérifieur de byte code embarqué

Il est clair que le vérifieur joue un rôle crucial dans cette nouvelle architecture. C'est lui qui autorise ou refuse l'installation d'une nouvelle application. Son implémentation doit être absolument irréprochable. Alors la question qui se pose est la suivante :

« ***Peut-on faire confiance au vérifieur embarqué ou doit-on le vérifier ?*** ».

Mais comme la sécurité d'une carte à puce est l'élément crucial pour instaurer une certaine confiance auprès de son utilisateur final, la réponse à cette question est qu'on doit impérativement procéder à une vérification de ce vérifieur embarqué en vue de préserver un niveau élevé de sécurité et du fait réduire au maximum les risques pouvant avoir lieu.

3.6 Positionnement et objectif de notre travail

Aujourd'hui certains prototypes de cartes commencent à arriver sur le marché avec ce composant de vérification embarqué. Néanmoins, vu la complexité du processus de vérification, toute faute d'implémentation du vérifieur et/ou l'implémentation d'un ensemble restreint de fonctionnalités peuvent constituer une faille de sécurité qui éventuellement aboutira à une attaque. De ce fait, notre objectif à travers le présent travail est d'étudier le niveau de sécurité de ce composant en se basant sur des suites de tests pour le caractériser et faire ressortir ses faiblesses qui peuvent être exploitées pour mener des attaques.

Tester la sécurité du vérifieur de byte code c'est analyser sa vulnérabilité. Pratiquement on doit trouver un moyen (c'est l'approche proposée dans le chapitre 6) pour déterminer un ensemble de données d'entrée invalides qui seront envoyées au vérifieur. Et par la suite il faudra analyser son comportement :

- S'il refuse une entrée X alors on déduit qu'il réagit normalement pour ce cas précis ;
- S'il accepte l'entrée X alors on peut dire qu'il y a un problème : une vulnérabilité est détectée.

Dans le cas du vérifieur de byte code pour une machine virtuelle Java Card on sous-entend par donnée d'entrée invalide un fichier CAP invalide. En effet, ce dernier représente le format d'entrée de ce composant de sécurité. L'objectif de notre travail peut être schématisé comme suit (figure 3.6) :

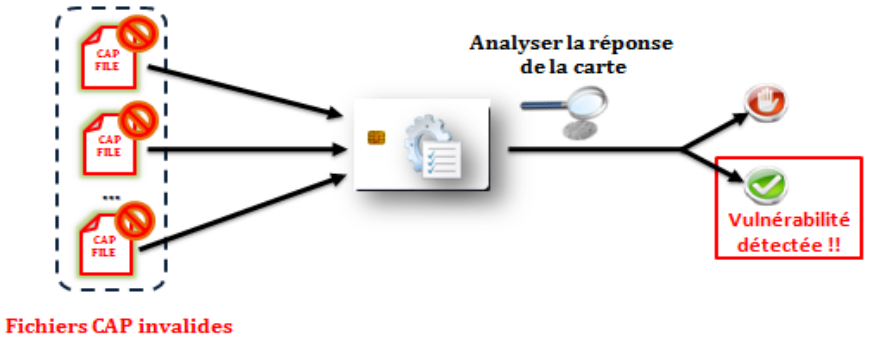

Fig 3.6 – Objectif de notre travail

Dans le présent travail on va s'intéresser uniquement au vérifieur de structure : partie en amont de la vérification du byte code. En effet, les deux parties du vérifieur de byte code sont relativement différentes :

- Se sont deux étapes successives de la vérification.
- Elles nécessitent deux stratégies différentes de vérification : la base pour le vérifieur de structure est les composants du fichier CAP qui sont traités séparément puis comme un ensemble. Par contre le vérifieur de type est constitué d'un traitement linéaire d'un ensemble de byte codes.
- Le vérifieur de type nécessite une mémorisation de l'état précédent du système pour chaque instruction à vérifier, ce qui n'est pas le cas pour le vérifieur de structure.

De nombreuses attaques existantes sont basées sur le format du fichier CAP (voir section 2.7.2). Celles-ci exploitent le fait que ce format est pensé pour faciliter le travail de l'éditeur de liens. La

vérification structurelle peut apparaître simple et dénuée d'intérêt. Cependant, c'est par la structure que la plupart des fautes ou fuites d'informations peuvent intervenir. Il s'agit de modifier quelques valeurs dans le fichier CAP et de réajuster en conséquence les décalages pour avoir l'information désirée et créer ainsi une brèche de sécurité. Donc s'il y a un problème dans la vérification de structure, la vérification de type n'a pas d'intérêt à être réalisée.

De ce fait, notre travail traite la vérification du vérifieur de structure. Et en parallèle, des travaux en cours présentés dans [SFL11a] et [SFL11b], se concentrent sur le vérifieur de type en adoptant une philosophie relativement différente de la notre.

3.7 Conclusion

Le but à travers ce chapitre était de mieux cerner notre problématique en rassemblant les détails relatifs au composant de sécurité en question : le vérifieur de byte code. En effet, on a défini la problématique liée à l'embarquement de ce composant et on a focalisé notre travail sur la première partie de ce dernier, à savoir le vérifieur de structure, en se basant sur des tests de vulnérabilité. Donc, jusqu'ici on a déterminé le « Quoi ? » de notre travail. Le « Comment ? » fera l'objet du chapitre suivant qui portera sur les approches de génération de tests. Et ceci dans l'optique de nous positionner par rapport à ce qui existe dans la littérature.

CHAPITRE 4

État de l'Art sur la Génération des Tests

Sommaire

Le test logiciel est un domaine très vaste qui a trouvé sa place parmi les techniques de vérification. Notre but à travers ce chapitre est de situer notre travail par rapport aux approches de test existantes. Pour cela on a essayé de survoler ce qui est présent dans la littérature, mais de façon non exhaustive vu la grande diversité des approches de test et des classifications possibles. Donc, partant des méthodes générales de test logiciel on est arrivé au test à partir de modèle et particulièrement son application dans le domaine des cartes à puce. Pour terminer avec un positionnement de notre travail, étape primordiale pour ce qu'on va proposer comme approche répondant à notre problématique.

4.1 Techniques de vérification

La vérification du logiciel consiste à révéler les fautes de conception et d'implémentation. Pour des raisons de coût et d'efficacité, il est important de les révéler rapidement, dès leur apparition si possible.

La vérification doit être intégrée tout au long du processus de développement.

Les approches actuelles de vérification ne permettent pas de garantir l'élimination de toutes les fautes résiduelles. Une bonne méthode de vérification fait alors appel à un ensemble de techniques qu'il faut choisir de façon judicieuse, en tenant compte du niveau de criticité du système. Les techniques de vérification peuvent être classées (figure 4.1) selon qu'elles impliquent ou non l'exécution du système. La vérification sans exécution est dite *statique*, par opposition à la vérification *dynamique* qui est basée sur l'exécution du système.

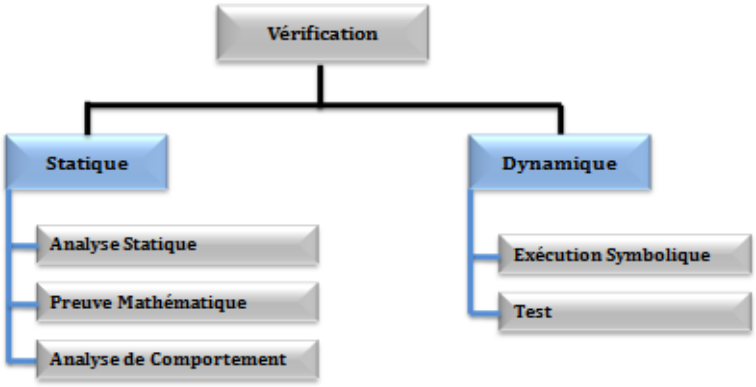

Fig 4.1 – Classification des techniques de vérification

- *L'analyse statique* correspond aux techniques de "revues" ou "inspections" qui sont applicables à n'importe quel niveau de vérification. Le principe consiste en une analyse détaillée du document produit (spécification, code source, etc.) effectuée par une équipe différente de celle ayant réalisé le document. L'analyse statique peut être manuelle, ou automatique.
- *La preuve mathématique* consiste à démontrer qu'un programme ou une implémentation est correct par rapport à sa spécification, à l'aide de concepts mathématiques. Généralement, la preuve mathématique n'est utilisée que pour des systèmes critiques de petite ou moyenne taille.
- *L'analyse de comportement* est basée sur des modèles comportementaux du système, déduits de la spécification ou de codes sources. Ces modèles doivent permettre de vérifier des propriétés de cohérence, complétude, etc. Les plus utilisés sont les systèmes de transitions étiquetées, les machines à états finis et les réseaux de Petri. Ces modèles sont bien adaptés à une décomposition hiérarchique dans le cas des systèmes complexes.
- *L'exécution symbolique* consiste à exécuter un système en lui soumettant des valeurs symboliques en entrée : par exemple, si une entrée "X" est un nombre entier positif, on peut lui affecter une valeur symbolique "a" qui représente l'ensemble des valeurs entières supérieures à 100. Une exécution symbolique consiste alors à propager les symboles sous formes de formules au fur et à mesure de l'exécution des instructions, et fournit comme résultat les expressions symboliques obtenues pour les sorties.
- *Le test* est la méthode de vérification utilisée à travers le présent travail, elle sera détaillée dans la section suivante.

4.2 Test logiciel

La taxonomie relative à l'activité de test étant relativement importante. On donne ici un panorama global (mais pas de façon exhaustive) des approches de test présentées dans la littérature et qui permettent de situer notre travail.

D'après G.Myers [Mye79], « *tester c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts* ».

Plus précisément, la norme [IEE90] définit le test comme étant « *l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus* ».

Pour satisfaire cet objectif de test, plusieurs approches complémentaires sont disponibles. Il existe également diverses classifications de ces approches.

J. Tretmans [Tre04] propose une classification tridimensionnelle (figure 4.2) des approches de test suivant trois axes :

- le niveau de test ;
- les propriétés ou caractéristiques testées du système ;
- le support de conception des tests.

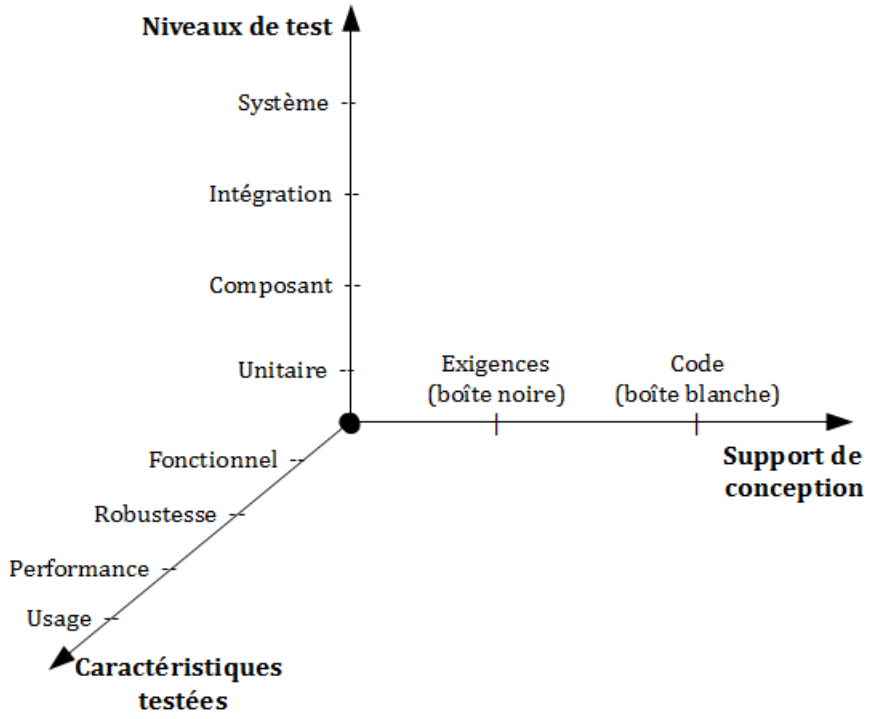

Fig 4.2 – Classification tridimensionnelle des approches de test [Tre04]

4.2.1 Niveaux de test

Cet axe caractérise le niveau de test considéré, de la plus petite unité testable jusqu'à la globalité du système. Pour cela, les niveaux de tests suivants sont identifiés :

- *Le test unitaire/composant* : « *Testing of individual software units or groups of related units* »[IEE90]. Ce test vise à révéler le plus grand nombre de fautes possibles de chaque composant, procédure ou module testable d'un programme.
- *Le test d'intégration* : « *Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them* »[IEE90]. Il se positionne au niveau des interfaces entre les unités testées au niveau précédent.
- *Le test système* : « *Testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements* »[IEE90]. Il a pour but de valider l'adéquation entre le système global sous test et ses spécifications initiales.

4.2.2 Caractéristiques testées

Ce deuxième axe catégorise les approches de test selon le caractère testé du système. Ainsi on trouve :

- *Le test fonctionnel* : garantit le bon fonctionnement du système, par rapport à ses spécifications fonctionnelles ;
- *Le test de robustesse* : teste le système dans des conditions inhabituelles, non prévues par la spécification fonctionnelle, ou des conditions extrêmes d'usage. Son objectif est de tester l'ensemble des comportements auxquelles le système doit faire face (dysfonctionnement matériel ou logiciel, actions non prévues, etc) ;
- *Le test de performance* : a pour objectif de fournir un ensemble de mesures (temps de réponse dans la plupart des cas) obtenues suite à une sollicitation excessive du système. Le test de performance est ainsi à l'origine de l'optimisation du système.
- *test d'usage* : considère le couple utilisateur/système lors de son utilisation. Par la mesure de caractères comme l'efficacité, l'ergonomie ou la correction d'un programme, il a pour objectif de révéler les points d'améliorations envisageables du produit.

4.2.3 Supports de conception des tests

Cet axe distingue les deux grands types de test traditionnels, le test en boîte-blanche (ou white-box testing) et le test en boîte-noire (ou black-box testing), en fonction du support utilisé pour leur conception.

- *Test en boîte-noire*

  Ces tests correspondent à des tests fonctionnels vérifiant que le comportement du système est conforme aux fonctionnalités exprimées dans la spécification fonctionnelle du système. Dans cette approche, le système est traité comme une boîte noire accessible à travers des points de contrôle et d'observation. L'objectif de ces tests est la vérification de la conformité des fonctionnalités de

l'implémentation par rapport à une spécification de référence. Les détails internes des systèmes ne sont pas inspectés. Ces tests ne s'intéressent qu'au comportement externe du système à tester. Plus concrètement, les tests en boîte-noire fournissent un ensemble de valeurs en entrée du système et vérifient que les valeurs ou comportements de retour obtenu(e)s correspondent à ceux (celles) attendu(e)s.

– *Test en boîte-blanche*

Ces tests sont conçus à partir de la structure interne (données, code, algorithmes) du système sous test. Autrement dit, plutôt que d'être fondé sur les fonctions du système, le test structurel est basé sur le détail du système. Les tests structurels sont effectués sur des produits dont la structure interne est accessible et sont dérivés en examinant l'implémentation du système. Ils permettent de vérifier si les aspects intérieurs de l'implémentation ne contiennent pas d'erreurs de logique et si toutes les instructions de l'implémentation sont exécutables. Ces tests ne sont pas une alternative aux tests en boîte-noire. Ils constituent plutôt une approche complémentaire qui découvre des classes différentes d'erreurs.

Parmi toutes les solutions de test existantes, on s'intéresse plus particulièrement, dans la section suivante, aux solutions à base de modèles (MBT pour Model-Based Testing). Celles-ci s'appuient sur un modèle du système afin d'en produire des cas de test. Dans la taxonomie tridimensionnelle donnée plus haut (figure 4.2), une approche MBT peut se positionner comme indiqué dans la figure 4.3 [UL06].

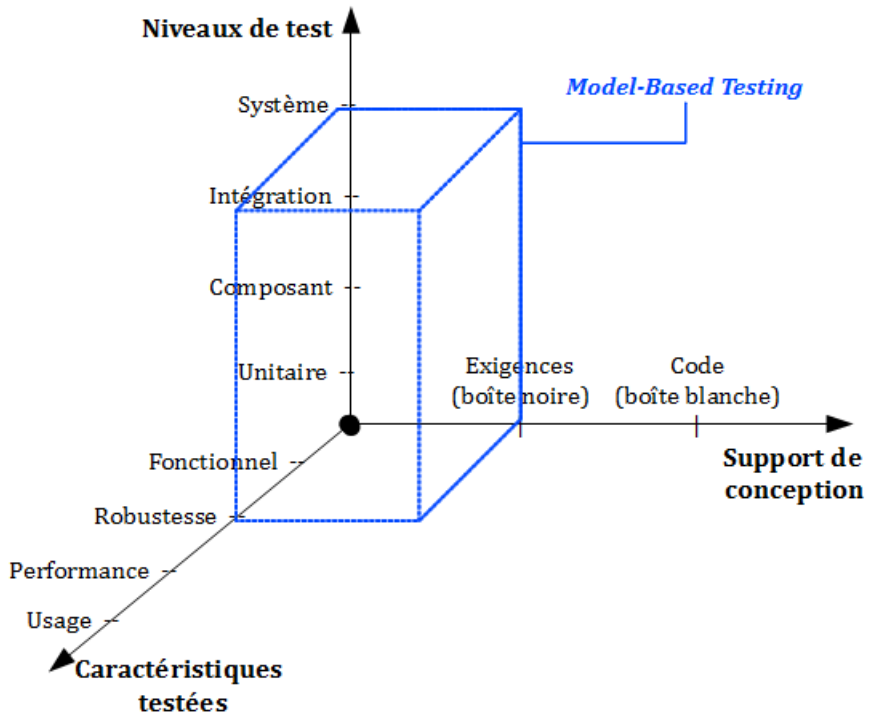

Fig 4.3 – Positionnement du MBT dans la classification tridimensionnelle des approches de test [UL06]

4.3 Test basé sur les modèles (MBT)

Le test à base de modèle repose sur un modèle du comportement attendu du système sous test (SUT : System Under Test) et éventuellement de son environnement. Ce modèle de test utilise un niveau d'abstraction et se concentre sur tout ou une partie des comportements en fonction de l'objectif de test.

Un processus MBT se décompose généralement en cinq phases qui sont :

- La construction d'un modèle abstrait du système à tester ;
- La dérivation de cas de test abstraits depuis le modèle abstrait ;
- La concrétisation des cas de test abstraits en tests exécutables (*i.e.* des cas de test concrets) sur le système sous test ;
- L'exécution des cas de test concrets sur le système sous test ;
- L'analyse des résultats ainsi obtenus.

4.3.1 Taxonomie des approches MBT

Les solutions MBT existantes sont nombreuses et se distinguent notamment par le type de modèle, la méthode de génération des cas de test, ou encore le type d'exécution de ces tests. M. Utting, A. Pretschner et B. Legeard [UPL06] ont donné une taxonomie (figure 4.4) des multiples déclinaisons du Model-Based Testing.

Modélisation

Il s'agit de concevoir un modèle qui servira à produire les cas de test à exécuter sur le système sous test. Ce modèle représente les comportements du système à un certain niveau d'abstraction. Le modèle de test est la référence, issue des spécifications du système, permettant l'établissement des verdicts de test. Pour cela, il contient une représentation des points de contrôle et d'observation du système sous test, qui permettent de comparer les valeurs réelles avec les valeurs attendues. En outre, sa structure est exploitée lors de la génération des cas de test pour garantir une certaine couverture du modèle et donc de la spécification fonctionnelle dont il est issu.

1. ***Sujet du modèle***

   Dans la pratique, un modèle de test est une abstraction du système sous test et/ou de son environnement (un *modèle d'environnement* qui contient uniquement les comportements de l'environnement ; un *modèle du système* contenant une représentation des comportements internes du système). Le modèle contient alors un sous-ensemble des comportements du système réagissant à certains stimuli de son environnement. L'abstraction choisie permet de définir le périmètre de ce sous-ensemble.

2. ***Niveau de redondance du modèle***

   Le Model-Based Testing peut être appliqué pour deux objectifs distincts :

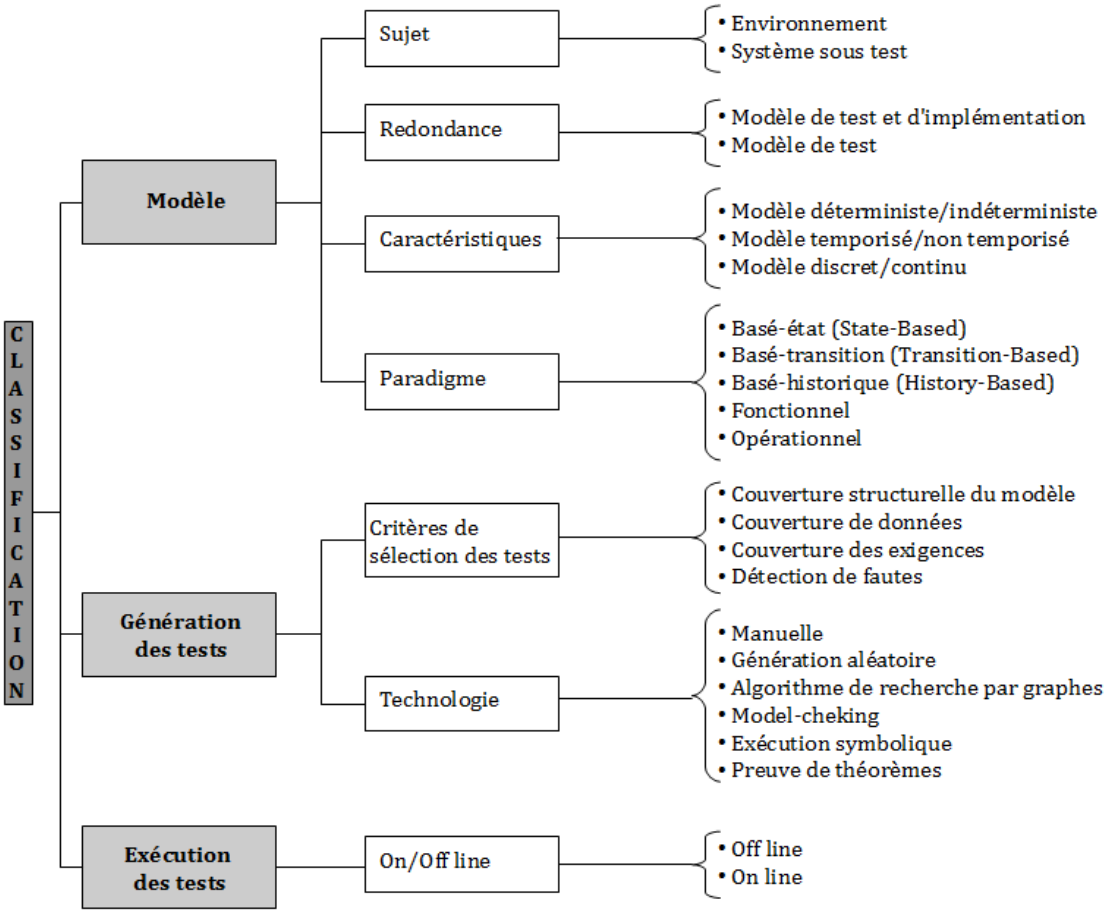

Fig 4.4 – Taxonomie des approches du MBT

- *Cas1* : Un unique modèle est utilisé à la fois pour le test et pour l'implémentation. Dans ce cas, l'abstraction du modèle est nécessairement faible puisque le niveau de détails est important. Les cas de test en résultant permettent une couverture structurelle du code (tests unitaires).
- *Cas2* : Le modèle est dédié au test. Le niveau d'abstraction est alors libre et permet la conception de modèles spécifiques au besoin de test. Ce modèle est conçu à partir de tout ou partie des spécifications fonctionnelles du système. Cette approche utilisant un modèle de test dédié est la plus courante.

3. ***caractéristiques du modèle***
   - *L'indéterminisme de comportement* [PPW+05]. C'est une caractéristique généralement liée aux systèmes concurrents. Dans ce cas, les cas de test issus de ce modèle ne sont pas des séquences de stimulus, mais des arbres ou graphes de stimulus, permettant de satisfaire cet indéterminisme.
   - *Caractéristique temporelle forte*. Les systèmes temps-réel se différencient des autres systèmes par la prise en compte de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat [BK05].
   - *Caractère dynamique* discret ou continu (ou une combinaison des deux dans le cas de modèles hybrides) [BK05] .

4. ***Le paradigme de modélisation***

   Différentes notations peuvent être utilisées lors de la modélisation du système sous test, suivant les caractéristiques de celui-ci et qui sont évoquées plus haut.

   – *Les notations basées sur l'historique* (History-Based) modélisent un système en décrivant les comportements autorisés du système dans le temps. Le système est notamment modélisé à partir d'expressions logiques temporelles sur les états passés, présents et/ou futurs du système. Dans de telles spécifications, le temps est linéaire ou non, la structure temporelle est discrète ou continue. Les propriétés temporelles s'expriment à partir d'instants et/ou d'intervalles.

   – *Les notations basées sur les états* (State-Based) permettent de modéliser le système comme un ensemble de variables, qui représentent l'état du système à un instant donné, et un ensemble d'opérations qui modifient l'état de ces variables. Chaque opération est définie par une pré-condition conditionnant l'application de l'opération par des contraintes sur les paramètres d'entrée du système, et une post-condition caractérisant l'état du système après application de cette opération par des contraintes sur les paramètres de sortie du système. Parmi les langages de spécification basés sur les états, citons B, JML ou encore OCL.

   – *Les notations basées sur les transitions* (Transition-Based) se concentrent davantage sur les transitions entre les différents états du système. Chaque transition donne, à partir d'un état d'entrée et d'un événement déclencheur, un état de sortie du système. Elles incluent : les machine à états finis (FSM [Gil62]), les machines à états-transitions (UML [RJB99], Statemate [HLN+90], Simulink Stateflow [RF00]), les systèmes de transitions étiquetées (LTS [Tre96]) et les automates d'entrées/sorties [LT88].

   – *Les notations fonctionnelles* décrivent un système comme un ensemble de fonctions mathématiques. Les fonctions peuvent être du premier ordre (spécifications algébriques) ou d'ordre supérieur (logique HOL)[GM93].

   – *Les notations opérationnelles* qui représentent le système comme un ensemble de processus exécutables. Elles sont généralement utilisées pour modéliser des systèmes distribués ou des protocoles de communication.

   – *Les notations stochastiques* décrivent le système par un modèle probabiliste (par exemple des chaînes de Markov) des valeurs d'entrées et des évènements, elles sont surtout utilisées pour modéliser l'environnement du système sous test.

Génération des tests

1. ***Le critère de sélection des tests***

   Un critère de sélection permet de générer un ensemble de cas de test partageant des propriétés communes (cet ensemble est appelé *suite de tests*). Ceci permet de mesurer la qualité d'une suite de tests, par son taux de couverture du modèle de test. Parmi les critères de sélection applicables sur un modèle de test, on trouve les deux grandes familles suivantes :

   – *Critères de couverture structurelle du modèle* dont objectif est de garantir la couverture de la structure du modèle.

- Les critères orientés *"flux de contrôle"* (Control-Flow-Oriented Coverage Criteria) sont directement issus des critères de couverture de code. La couverture porte sur les instructions, les décisions ou les chemins exprimés par le graphe de flots de contrôle issu du modèle. On a ainsi les critères *Condition Coverage, Decision/Condition Coverage, Full Predicate Coverage*, *Modified Condition/Decision Coverage* et *Multiple Condition Coverage.*
- Les critères de couverture orientés *"flux de données"* (Data-Flow-Oriented Coverage Criteria) se basent sur l'utilisation et la définition des variables du modèle. Parmi ces critères, *all-definitions* assure la couverture d'au moins une paire définition-utilisation pour chaque variable. Le critère *all-uses* assure la couverture de toutes ces paires, i.e. de toutes les utilisations de toutes les définitions. Le critère *all-def-use-paths* garantit la couverture de toutes les paires définition-utilisation (d ; u) par tous les chemins de d à u.
- Les critères de couverture basés sur les *"transitions"* (Transition-Based Coverage Criteria) sont spécifiques aux diagrammes états-transitions (FSM, LTS, machines à états UML, etc). Les éléments de base de ces types de couverture sont les états, les configurations et les transitions. Cette famille se compose également de nombreux critères de couverture de chemins (*All-loop-free-paths coverage, All-one-looppaths coverage, All-round-trips coverage, All-paths coverage*, etc).

– *Critères de couverture des données* dont l'objectif est de garantir une couverture des valeurs de chaque variable du modèle.
  - Le critère *One-value* garantit que chaque variable du modèle obtient une valeur de son domaine.
  - Le critère *All-values* garantit que chaque variable du modèle obtient toutes les valeurs de son domaine. En pratique, ce critère est inutilisable sur une variable de domaine infini ou très grand.
  - Le *test des valeurs aux bornes* a pour objectif de choisir des valeurs aux bornes de leur domaine [KLPU04].
  - Les critères de *test Pairwise* se basent sur des combinaisons de valeurs entre deux variables [BS04].

2. ***La technologie utilisée pour la génération des tests***

L'intérêt du Model-based Testing est sa capacité à produire une série de cas de test de manière automatique. Cette génération peut s'effectuer suivant différentes techniques :

– *Les techniques de test statistique* sont nombreuses dans la littérature. On peut citer la sélection d'entrées selon un profil aléatoire, l'utilisation du partitionnement, la modélisation d'un profil d'usage par chaîne de Markov ou la dérivation automatique de tests statistiques à partir de spécifications formelles.
– *Les algorithmes de recherches à partir de graphes* permettent de révéler des traces couvrant les arcs et/ou les nœuds d'un graphe.
– *Le model-checking* est une technique qui permet de vérifier la satisfaisabilité d'une propriété (les critères de test sont exprimés par des propriétés temporelles) sur un modèle donné. La

capacité des model-checkers à générer des contre-exemples est utilisée pour générer des traces de test de ces propriétés. Ainsi, les traces générées sont instanciées pour créer des séquences de test complètes satisfaisant les critères [ABM98], [BCJM04], [dHR04].

- *L'exécution symbolique de modèles* consiste à animer un modèle de test exécutable non pas avec des valeurs simples mais plutôt à partir d'ensembles de valeurs. Le système est alors représenté par un ensemble de contraintes sur ses variables d'état. Des solveurs de contraintes sont alors utilisés pour générer des traces valides respectant ces contraintes. Chaque trace contrainte est ensuite valuée, selon le domaine contraint de chaque variable, pour constituer des cas de test.
- *L'utilisation de la logique de programmation par contraintes.* L'idée est de représenter le système sous test comme étant un système de contraintes (le modèle). La résolution de ce système de contraintes permet ensuite de vérifier l'existence d'un chemin exécutable par objectif de test et générer finalement un jeu de données satisfaisant ce chemin. Dans [GBR98], Gotlieb et al. proposent une génération automatique de données de test dédiées au test structurel. Le programme sous test est transformé en un système de contraintes, appelé Kset, en utilisant une forme SSA (Static Single Assignment). Pretschner et al. utilisent cette logique pour le test de systèmes réactifs [LP00] [PL01]. Un solveur de contraintes ensembliste est utilisé dans [CLP04] pour calculer des préambules de test à partir d'une machine B.
- *La preuve déductive de théorèmes* permet de vérifier la satisfaisabilité de formules logiques. Le système sous test est modélisé par un ensemble d'expressions logiques (ou prédicats) spécifiant les comportements du système. La génération de tests se fait alors à partir d'un partitionnement en classes d'équivalence de l'ensemble des prédicats valides. Une classe d'équivalence représentant un comportement modélisé du système. Classiquement, le partitionnement est réalisé par une transformation des expressions logiques en forme normale disjonctive. Dans [HNS97], Helke et al. utilisent l'assistant de preuve de théorème Isabelle/HOL pour générer des cas de test à partir de spécifications Z.

L'exécution des tests

Cette dimension concerne l'intervalle de temps entre la génération des cas de test et leur exécution. On distingue ainsi :

- *Le test on-line.* Exploite directement les données réelles du système suite à sa stimulation. Le générateur de test est alors fortement couplé au système sous test. Le test se construit incrémentalement en fonction des réactions du système.
- *Le test off-line.* Distingue la phase de génération de la phase d'exécution. Ainsi, les tests peuvent être générés puis exécutés ultérieurement par des outils de gestion et d'exécution des tests. Les tests peuvent être générés et exécutés sur des machines différentes, dans des environnements différents.

4.3.2 Description de quelques outils MBT

Cette section expose un panorama non exhaustif [UPL06] de quelques solutions de génération de cas de test à partir de modèles, couvrant des domaines d'application variés. Ainsi qu'une classification (tableau 4.1) de ces solutions par rapport aux dimensions présentées dans la section 4.3.1.

- ***TorX*** [TB02] est une solution Model-based Testing académique de génération de cas de test à partir d'un modèle comportemental du système sous test. TorX s'appuie sur des systèmes de transitions étiquetées (LTS) et génère des cas de test de conformité entre le système (i.e. son implémentation) et le modèle de test. Cet outil gère le non-déterminisme par une approche on-line du test. Dans cette même famille d'outils se basant sur des spécifications LTS, on peut également citer, TGV [JJ05], STG [CJRZ02] et Autolink [SKGH97]. Ces outils de tests sont particulièrement adaptés pour la validation des systèmes de communication et des protocoles.
- ***LTG*** (LEIRIOS Test Generator) [BLPT04] est une solution, commercialisée par la société LEIRIOS, de génération de tests fonctionnels à partir de spécifications UML ou B. Le modèle spécifie les comportements du système sous test et constitue le support de l'oracle de test. Le pilotage de la génération de tests est permis grâce à différents critères de couverture des décisions et des données. Les cas de test sont générés à partir d'une exécution symbolique du modèle et de différents algorithmes de recherche. Les cas de test abstraits ainsi générés peuvent être transformés en scripts de test, exécutables sur le système sous test. Parmi les autres solutions basées sur la couverture structurelle du modèle : T-VEC Tester for Simulink and StateFlow [BB96], QTronic et Conformiq Test Generator de la société Conformiq. Ces solutions commerciales de test sont adaptées aux domaines d'applications des systèmes réactifs, du logiciel embarqué, de la carte à puces, ou de la transaction électronique.
- ***JUMBL*** (J Usage Model Builder Library) [Pro03] est une application académique de test statistique utilisant des chaînes de Markov. Les données de test sont générées via l'exploitation d'un modèle d'usage de type automate à états fini, dont les transitions sont étiquetées par des probabilités. Ainsi, les cas de test avec les plus grandes probabilités sont générés en premier. ALL4TEC propose une solution commerciale, MaTeLo [DZ03], basée également sur le test statistique d'usage.
- ***AETG*** (Automatic Efficient Test Generator) [CDFP97] est un outil de génération des données de test. Il utilise des algorithmes pair-wise pour garantir que toutes les combinaisons de valeurs de paires de variables sont testées. Des combinaisons supérieures à deux sont également disponibles. L'oracle de chaque test doit être fourni manuellement. Cette approche est applicable pour le test de configurations (par exemple les options de configuration de certains produits).

DIMENSIONS	TorX	LTG	JUMBL	AETG
Sujet du modèle	Modèle comportemental du système	Modèle comportemental du système	Modèle de l'environnement (modèle d'usage)	Modèle de l'environnement (Modèle des données d'entrée)
Redondance du modèle	Modèle dédié au test	Modèle dédié au test	Modèle dédié au test	Modèle dédié à la génération des données d'entrée
Caractéristiques du modèle	Non déterministe, discret, contraintes temporelles faibles	Non déterministe, discret, contraintes temporelles faibles, fini	Discret, contraintes temporelles faibles	Discret, contraintes temporelles faibles
Paradigmes de modélisation	LTS (Labelled Transition System)	Etat/transition (UML), pré/-post conditions (B)	Langage descriptif pour les chaînes de Markov (TML)	Modélisation des domaines des variables
Critères de sélection de tests	Objectifs de test	Couverture structurelle du modèle	Critères statistiques basés sur les probabilités des transitions	Couverture des données
Technologie	Exploration de l'espace d'états par des techniques on-the-fly	Exécution symbolique, algorithmes de recherche	Algorithmes de recherche statistique, modèle de Markov	Algorithmes de recherche
Exécution	on-line / off-line	off-line	off-line	off-line
Domaines d'application	Systèmes de communication, protocoles	Systèmes réactifs, logiciels embarqués, cartes à puce		Test de configuration

Tab 4.1 – Classification de quelques solutions MBT

4.4 MBT dans le domaine des cartes à puce

Ces dernières années de nombreux cas d'étude et applications industrielles basés sur le test à partir de modèles ont été conduits avec succès dans le domaine des cartes à puce. Les techniques de test basées sur des modèles ont pu être appliquées dans les différents niveaux logiciels des cartes à puce : allant des applications (modélisation des APDUs) à la validation de certaines parties de la JCVM et du JCRE.

H.Martin dans [MDB00] et [Mar01] propose une méthodologie de génération de suites de test d'applications embarquées dans des Java Card (*i.e.* les applets) basée sur la description des fonctionnalités attendues par l'utilisateur. Cette méthodologie permet de générer des suites de test concrètes à partir d'objectifs de test, en conformité avec une spécification UML du système.

D.Clarke *et al.* présentent dans [CJRZ01] des techniques de génération de cas de test symboliques basées sur un modèle du système et sur des objectifs de test. Ainsi que les résultats obtenus avec ces techniques sur les applications e-purse du standard CEPS (Commun Electronic Purse Specification).

Dans [PPS+03], J.Philipps *et al.* proposent une approche de génération automatique de tests à partir d'un modèle du système. L'approche a été appliquée sur le module WIM (Wireless Application Protocol Idendity Module) des téléphones cellulaires. Ce module se charge de la sécurité de la couche transport, des vérifications des codes du porteur de la carte (PIN et PUK) et des opérations cryptographiques (signatures numériques, cryptage, décryptage, etc.).

F.Bouquet et B.Legeard dans [BL03] ont traité le cas du mécanisme de transaction Java Card. Ceci en utilisant l'environnement BZ-Testing Tools [ABC+02] qui est dédié pour l'animation et la génération de tests aux limites à partir d'un modèle formel B.

Dans [BLLP04], E.Bernard *et al.* ont présenté les résultats de la génération de cas de test pour une partie du standard GSM 11-11 [Ins99]. En se basant sur les travaux portant sur B-Testing-Tools [LP01] et [LPU02], l'approche proposée repose sur une notation B et la programmation logique par contraintes (Constraint Logic Programming).

Dans [UPL06], deux cas d'étude utilisant l'outil propriétaire LTG (fourni par LEIRIOS Technologies et qui est dédié à la génération automatique de tests à partir de modèles formels B) ont été présentés. Le premier cas d'étude est le projet PIV II (*Personal Identity Verification*) de Gemplus. PIV II est une application pour cartes à puce qui se charge du contrôle d'accès physique aux bâtiments (et même pour des espaces plus réduits). Le second cas d'étude (repris dans [JMT08] et [JMTB09]) concerne une autre application pour cartes : IAS (*Identification, Authentification, and electronic Signature*). Comme son nom l'indique c'est une application qui se charge de tout ce qui est protocoles cryptographiques, signatures électroniques, etc.

4.5 Notre positionnement

Afin de situer notre travail par rapport à ce qui a été présenté dans ce tour d'horizon dans les méthodes de test, il faudra mettre en avant deux choses : notre objectif à travers ce travail et de quoi

on dispose pour le faire. Donc on rappelle qu'on cherche à vérifier le vérifieur de structure à travers une analyse de vulnérabilité. De ce fait, notre but consiste à trouver des entrées invalides pour les envoyer au vérifieur et analyser sa réaction. Sachant qu'on ne dispose que de la spécification [Mic09c] informelle fournie par Oracle.

Par conséquent, dans la classification de J. Tretmans (section 4.2) on a :

- Niveau de test : Système (le système en question est le vérifieur structurel) ;
- Support de conception : La spécification, donc il s'agit d'un test en boîte noire car on ne dispose pas du code ;
- Caractéristiques testées : On va tester la sécurité du vérifieur en se basant sur des données erronées dans le but de détecter des vulnérabilités (très proche d'un test de robustesse).

D'après ces critères, il s'agit d'un test à partir de modèle. Et d'après la classification de M.Utting (section 4.3.1), on trouve :

- Sujet du modèle : Système sous test et l'accent est mis sur les données à soumettre au vérifieur ;
- Redondance : Modèle pour test seulement (notre but n'est pas l'implémentation) ;
- Caractéristiques : Modèle déterministe, non temporisé et discret ;
- Paradigme : Formalisme ensembliste (voir section 5.3.2) ;
- Critères de sélection : Couverture des données (*i.e.* couverture des valeurs de chaque variable du modèle) ;
- Technologie utilisée pour la génération des tests : Résolution de contraintes ;
- Exécution des tests : Off-line.

4.6 Conclusion

Après ce panorama, non exhaustif bien sûr, de méthodes de test et particulièrement celles basées sur des modèles, on a pu positionner notre travail par rapport à ce qui a été présenté. Il s'avère qu'on va procéder par une stratégie de test à partir de modèle. Mais pour entamer notre approche il reste une autre brique de base à déterminer : le formalisme à adopter pour construire le modèle. Et ça fera l'objet du prochain chapitre.

CHAPITRE 5

Méthodes Formelles : La Méthode B

Sommaire

Les méthodes formelles constituent un domaine de recherche très large articulé autour de techniques et d'outils très variés. Fondées sur des bases mathématiques, elles s'appliquent à la modélisation d'un système (ou d'une partie d'un système) et au raisonnement sur ce dernier [Duf03].

Ces deux raisons ont orienté notre choix de modélisation vers les méthodes formelles. Dans ce chapitre on va d'abord s'intéresser à l'utilisation de telles méthodes dans le domaine du test. Et ceci dans le but de choisir un formalisme adéquat pour notre cas précis tout en justifiant ce choix.

5.1 La modélisation formelle de systèmes

Les méthodes de développement rigoureuses se basent sur l'élaboration de modèles mathématiques pour vérifier ou assurer les propriétés du système considéré. Le modèle est une abstraction mathématique du monde réel obtenue après suppression de certains détails d'implémentation ou après le choix de certaines hypothèses et de caractéristiques essentielles du système. Certaines méthodes donnent la

possibilité de raffiner ce modèle abstrait dans un modèle concret. En informatique, ce dernier se trouve très proche d'une implémentation. Il s'agit alors de synthétiser ce modèle concret dans un langage de programmation et obtenir ainsi un code exécutable. Ce programme obtenu représente l'implémentation formelle du système spécifié, en préservant les propriétés incluses au niveau le plus abstrait [Lan06].

La modélisation mathématique a de nombreux avantages dont les principaux sont [Cas02] :

- Une précision dans la spécification formelle supérieure à celle fournie par une description informelle en langage naturel qui est souvent ambiguë. C'est la force de la modélisation : lever les ambiguïtés ;
- Des fondements permettant d'énoncer des propriétés et d'étudier le comportement du système dans son ensemble.

La modélisation peut prendre diverses formes, en fonction de la nature des mathématiques retenues pour exprimer et formaliser le système. Il existe différentes techniques de modélisation. Certaines reposent sur des machines d'état abstraites, d'autres sur des automates et enfin les modèles fonctionnels qui eux sont basés sur la notion de fonction dans le sens mathématique du terme [Lan06].

5.2 Test à partir de modèles formels

L'utilisation des méthodes formelles pour le test de logiciels est probablement ce qu'il y a de plus sûr en matière de techniques de vérification. Ceci s'explique par les fondements mathématiques sur lesquels se basent ces méthodes, ce qui permet de développer un raisonnement plus rigoureux et de ce fait plus fiable. On peut requérir aux méthodes formelles pour spécifier les propriétés importantes du système testé, mais aussi pour vérifier ces propriétés sur l'implémentation finale [Bes10].

Les possibilités offertes par les techniques de génération automatique de suites de test sont intrinsèquement dépendantes des langages de modélisation auxquels elles s'appliquent [Mar01]. Ainsi, si l'objectif d'un logiciel est de réaliser du traitement de données, il faudra que le langage de spécification permette de construire un modèle de ces données. Si le logiciel implémente des comportements, il faudra que le langage de spécification permette de construire un modèle de ces comportements. Si le logiciel permet la création de nouveaux types de données, il faut que le langage de spécification permette de construire un modèle de ces nouveaux types et de leurs utilisations. Un certain nombre de langages formels permettent de construire des abstractions pour ces données, ces comportements et ces types. Ils peuvent être classés en deux catégories : algébriques ou ensemblistes.

5.2.1 Utilisation de modèles algébriques

Une spécification algébrique décrit un ensemble d'équations ou d'axiomes que doit vérifier toute mise en œuvre d'application. On peut ainsi choisir de ne préciser que les propriétés essentielles du système. Une spécification algébrique repose sur la définition de types abstraits de données, auxquels sont associés des opérateurs de création, de modification et d'observation. La mise en œuvre n'est pas précisée dans le but de séparer le plus nettement possible la spécification de l'implémentation. Une spécification algébrique peut souvent être rendue exécutable en interprétant les axiomes comme des règles de ré-écriture orientées. Plusieurs travaux [Ber91], [Gau95], [AAB+05], [Lon07], [Boi07] ont porté

sur la génération de tests à partir de spécifications algébriques. L'outil LOFT (*LOgic for Functions and Testing*) [BGM91], [Mar91], [Mar95] est l'un des outils permettant de générer automatiquement des suites de test à partir de telles spécifications. Il a été développé pour automatiser une méthode de sélection de tests par dépliage des axiomes. L'idée est d'utiliser les principes de la programmation logique pour implémenter une méthode de sélection de tests à partir de spécifications algébriques.

5.2.2 Utilisation de modèles ensemblistes

Les langages de spécification ensemblistes ont pour objectif de définir un modèle qui caractérise l'ensemble des implémentations possibles du système [VA98]. Ils reposent sur la théorie des ensembles et sur la logique des prédicats. Ils permettent d'exprimer des contraintes dans les modèles, et ainsi d'exprimer des conditions d'exécution des opérations. Les propriétés du système ne sont pas exprimées de façon explicite mais peuvent être déduites du modèle (et des propriétés des structures mathématiques le constituant) à l'aide d'un raisonnement formel. Parmi les langages formels permettant d'écrire des spécifications ensemblistes on peut citer VDM [Jon90], Z[Wor92] et B [Abr96].

Plusieurs travaux ont été menés sur la génération de tests à partir de modèles ensemblistes. Les travaux réalisés à BULL [DF93] portent sur l'automatisation de la génération de tests élémentaires à partir d'une spécification formelle exprimée dans le langage VDM. Les outils qui ont été développés lors de ces travaux utilisent des techniques de résolution de contraintes pour générer des tests. Cependant, une étude décrite dans [VA98] montre que cette technique n'est pas applicable pour de grands systèmes, à cause de la mise sous forme normale disjonctive des expressions des contraintes qui aura tendance à générer un très grand nombre de cas de tests pas nécessairement pertinents.

Les travaux développés à DST (Deutsche System-Tecknik GumbH) [HP94] s'inspirent largement de la méthode proposée par BULL, ont été appliquées à des spécifications écrites en Z.

L.Van Aertryck propose, dans [VA98], une technique de génération de cas de tests, nommée CASTING (Computer Assisted Software TestING), à partir de modèles spécifiés avec un sous-ensemble du langage B. Elle se base sur une utilisation du modèle ensembliste et des contraintes qui y sont exprimées, ainsi que sur une stratégie fournie par le testeur. La technique qu'il propose a été appliquée à d'autres langages que le langage B (UML-CASTING [VAJ03]).

D'autres travaux sur l'identification d'objectifs de test pour des modèles B ont été également présentés par S.Behnia dans [Beh00].

5.3 La méthode B

5.3.1 Présentation

La méthode B est une méthode formelle destinée au développement de logiciels. Cette méthode englobe le processus complet de développement depuis la spécification jusqu'à l'implémentation, et permet de prouver que l'implémentation du logiciel est conforme à sa spécification. La preuve fait partie intégrante du processus, et chaque étape, de la spécification à l'implémentation doit être prouvée.

La méthode B est née dans les années 80. Le livre de référence de la méthode est le *B Book* [Abr96] écrit par Jean-Raymond Abrial, le fondateur de la méthode et qui avait déjà participé à la conception de la méthode Z (d'où les similitudes entre la méthode B et la méthode Z). La méthode B a bénéficié du soutien de l'industrie ferroviaire française qui a permis le passage à l'échelle de son utilisation. L'exemple industriel le plus abouti de l'utilisation de la méthode B concerne le pilote automatique embarqué (PAE) de METEOR de la ligne 14 du métro parisien [BBFM99].

Lors du développement de la méthode, deux principaux outils ont été mis en œuvre : l'Atelier B [1] (développé par CLEARSY) et le B-Toolkit [2] (développé par B-Core). Ces outils permettent de développer formellement un logiciel, de sa spécification abstraite à sa forme concrète qui sera utilisée pour générer le code correspondant, le tout en assistant le développeur dans les phases de preuves qui doivent être menées pour établir la sûreté du développement.

5.3.2 Pourquoi choisir la méthode B

Notre choix d'un langage pour la modélisation du vérifieur de structure a porté sur le langage B. Et ce choix n'est pas dû au hasard.

En effet, pour réaliser notre objectif visant le test du vérifieur de structure on avait indiqué, dans la section 4.5, qu'on allait effectuer du test basé sur un modèle de ce dernier. La modélisation concerne à la fois les données et les contraintes qui les régissent. Et d'après la spécification, qui servira comme base pour la construction du modèle, les données sont de nature ensembliste (des valeurs appartenant à des ensembles finis) et les contraintes peuvent être vues comme étant des prédicats de la logique du premier ordre. Donc, ceci correspond bien à ce qui peut être offert par un modèle ensembliste tel que présenté dans la section 5.2.2. De ce fait le premier choix a porté sur l'utilisation d'un modèle ensembliste.

Mais arrivé à ce niveau là, il fallait faire un deuxième choix, celui du langage ensembliste à utiliser. Comme on l'a déjà présenté, il existe une multitude de langages faisant partie de cette catégorie. D'une part, d'après [Mar01], la méthode B semble plus particulièrement adaptée pour traiter les aspects données en plus des comportements et même permet de mettre l'accent sur les données. Et c'est exactement ce qu'on cherche : s'intéresser aux données plutôt qu'aux comportements. Car notre objectif principal est d'arriver à générer des données d'entrée invalides à partir d'un modèle. D'une autre part, on peut très bien voir que la méthode B est largement utilisée dans le milieu industriel est bénéficie de retours d'expérience dans le domaine de la carte à puce, point qu'on a déjà présenté dans la section 4.4 (travaux relatifs aux tests à partir de modèles).

Et plus particulièrement, on s'est inspirés des travaux de [Cas02] pour la construction de notre modèle. Mais à la différence de ces travaux, qui utilisent le modèle B du vérifieur dans le but de son implémentation, notre travail utilise le modèle B dans un but de test seulement. Par conséquent, le niveau d'abstraction dans les deux modèles n'est pas le même.

Pour toutes ces raisons, le choix d'un langage ensembliste pour la modélisation du vérifieur de

1. `http://www.atelierb.eu`
2. `http://www.b-core.com`

structure a porté sur la méthode B (méthode car avant d'être un langage c'est une méthode de développement mais dans notre cas on va plutôt profiter de la puissance offerte par le langage du moment qu'on va faire du test et non pas de l'implémentation).

5.3.3 Fondements théoriques de la méthode B

La notion de base de la méthode B est la machine abstraite. Celle-ci encapsule des données et des opérations sur ces données. Une machine abstraite peut être vue comme un ensemble de données et de services permettant d'accéder à ces données. Une spécification B peut être constituée de plusieurs machines abstraites. Les principaux éléments composant une machine abstraite sont :

- Les variables représentent les données encapsulées par la machine. Celles-ci ne peuvent être modifiées que par l'intermédiaire des opérations définies dans la machine ;
- L'invariant consiste en une série de prédicats définissant le type des variables, ainsi que les différentes propriétés devant être garanties par la machine ;
- Les opérations définissent les services proposés par la machine. Il peut leur être associé des pré-conditions, définissant les conditions devant être remplies pour pouvoir appeler l'opération.

Différents concepts sont associés à la spécification des éléments précédents :

- La théorie des ensembles. La théorie des ensembles est principalement utilisée pour définir les données manipulées par une machine ;
- La logique des prédicats. Celle-ci est utilisée dans l'invariant pour décrire les propriétés devant être conservées par la machine, ainsi que pour préciser les pré-conditions nécessaires à l'appel d'une opération ;
- Les substitutions généralisées. C'est le formalisme utilisé pour la définition des opérations fournies par la machine.

Les machines abstraites peuvent être raffinées afin d'ajouter, à chaque niveau de raffinement, de nouveaux détails permettant d'expliciter l'algorithme choisi et la manière d'opérer. Ceci est assuré par le mécanisme de raffinement qui consiste à reformuler, par étapes successives, les variables et les opérations de la machine abstraite, de manière à aboutir finalement à un module qui constitue un programme informatique. Les étapes intermédiaires de reformulation se nomment les raffinements et le dernier niveau de raffinement s'appelle l'implantation (ou implémentation) [Cle06]. A chaque niveau de raffinement, des obligations de preuve représentent ce qu'il faut prouver pour garantir que le comportement d'une nouvelle opération ne contredit pas l'opération qu'elle raffine. Ainsi, après la dernière itération, et par transitivité, le code d'une implantation sera effectivement conforme aux spécifications de la machine abstraite correspondante.

5.3.4 Le langage B

Dans le chapitre suivant traitant notre approche, nous allons illustrer les différentes étapes de cette dernière à l'aide d'exemples issus de la modélisation B. Donc, pour faciliter la lecture de ces exemples,

nous présentons dans cette section une description générale du langage B, complétée par une annexe listant quelques notations utilisées dans ce langage (Annexe A). Pour plus de détails sur chaque notion présentée, le manuel de référence [Cle09] peut être consulté. Le langage B se compose des parties suivantes :

- Le langage des prédicats ;
- Le langage des expressions ;
- Le langage des substitutions généralisées
- Le langage des composants (machines abstraites, raffinements, implantations)

Les prédicats

Les prédicats utilisés en B font partie de la logique du premier ordre. Ce sont des formules qui permettent d'exprimer des propriétés sur des données. Par exemple, ils expriment les propriétés invariantes des variables, les pré-conditions sous lesquelles une opération peut être appelée (ces conditions portent particulièrement sur les paramètres d'entrée de l'opération). Les prédicats du langage B sont :

1. Les propositions simples (conjonction, négation, disjonction, implication, équivalence) ;
2. Les prédicats quantifiés (universellement, existentiellement) ;
3. Les relations entre expressions : égalité, appartenance, inclusion, comparaison arithmétique.

Les expressions

Une expression est une formule qui désigne une donnée. Toute donnée possède un type qui est l'ensemble de toutes ses valeurs possibles. Les catégories d'expressions sont :

1. Les expressions de base (désignation d'une donnée),
2. Les expressions booléennes,
3. Les expressions arithmétiques,
4. Les couples,
5. Les ensembles (ensemble vide, ensemble des entiers relatifs, des booléens...),
6. Les constructions d'ensembles (ensemble des sous-ensembles, union, intersection...),
7. Les relations (identité, inverse, projection, composition, itération, domaine...),
8. Les fonctions (injections, surjections, bijections...),
9. Les constructions de fonctions (fonctions constantes, lambda expressions...),
10. Les suites (suites, permutations, concaténation, insertion, restriction...).

Les substitutions

Afin de garantir que l'appel d'une opération préserve les données et leurs propriétés, il est nécessaire de formuler la transformation effectuée par cette opération. La notion de substitution généralisée est

utilisée pour construire les prédicats à prouver. Les substitutions généralisées (tableau 5.1) décrivent le comportement des opérations, au niveau des machines abstraites, des raffinements et des implantations. Les substitutions de spécifications peuvent être non-déterministes et non-exécutables, alors que les substitutions d'implantations correspondent à des instructions d'un langage de programmation impératif classique.

Opérateur ou mot réservé	Nom de la production grammaticale
BEGIN	Substitution bloc
skip	Substitution identité
:=	Substitution devient égal
:( )	Substitution devient tel que
:∈	Substitution devient élément de
PRE	Substitution pré-condition
ASSERT	Substitution assertion
CHOICE	Substitution choix borné
IF	Substitution conditionnelle
SELECT	Substitution sélection
CASE	Substitution cas
ANY	Substitution choix non borné
LET	Substitution définition locale
VAR	Substitution variable locale
;	Substitution séquence
WHILE	Substitution tant que
→	Substitution appel d'opération
‖	Substitution simultanée

Tab 5.1 – Les substitutions généralisées [Cle09]

Les composants

Dans la terminologie B, le terme composant est utilisé de façon générique pour représenter une machine abstraite, un raffinement ou une implantation. Les composants sont décrits par des clauses. Les principales clauses sont résumées dans le tableau 5.2.

Clause	Description
CONSTRAINTS	définition du type et des propriétés des paramètres scalaires formels
SEES	liste des instances de machines vues
INCLUDES	liste des instances de machines incluses
PROMOTES	liste des opérations promues des instances de machines incluses
EXTENDS	liste des instances de machines étendues
USES	liste des instances de machines utilisées
SETS	liste des ensembles abstraits et définition des ensemble énumérés
CONCRETE-CONSTANTS	liste des constantes concrètes
ABSTRACT-CONSTANTS	liste des constantes abstraites
PROPERTIES	définition du type et des propriétés des constantes de la machine
CONCRETE-VARIABLES	liste des variables concrètes
ABSTRACT-VARIABLES	liste des variables abstraites
INVARIANT	définition du type et des propriétés des variables
ASSERTIONS	définition de propriétés déductibles de l'invariant
INITIALISATION	initialisation des variables
OPERATIONS	liste et définition des opérations propres

Tab 5.2 – Les clauses d'un composant B [Cle09]

5.4 Conclusion

Arrivé à ce niveau là dans notre présentation des domaines d'étude liés à notre travail, on peut dire qu'on dispose de tous les ingrédients nécessaires pour entamer la présentation de l'approche qu'on a proposé afin d'apporter une solution à la problématique qui a été définie. Ainsi, notre approche d'analyse de vulnérabilité consistera en une approche de test à partir d'un modèle formel exprimé en langage B. La suite du présent document est consacrée pour la présentation en détails des différentes étapes de cette approche ainsi que leur agencement dans l'optique de répondre à notre objectif principal : tester la sécurité du vérifieur de structure et détecter ses vulnérabilités.

Deuxième partie

Contribution

CHAPITRE 6

Approche Proposée pour l'Analyse de Vulnérabilité

Sommaire

6.1 Pourquoi une nouvelle approche ?

En élaborant l'état de l'art sur les méthodes de génération de test, et spécialement les travaux portant sur les approche basées sur des modèles, on a constaté qu'une très grande partie de ces travaux traite l'aspect *conformité* des implémentations par rapport aux spécifications. Cependant, dans notre cas, on s'intéresse plutôt à l'aspect *sécurité*. Autrement dit, est-ce qu'une implémentation donnée ne comporte pas des failles de sécurité (*i.e.* des vulnérabilités) qui font de telle sorte qu'un comportement devant être rejeté d'après la spécification soit accepté par cette implémentation.

D'après nos recherches dans la littérature, aucun travail similaire n'a pu être trouvé, *i.e.* portant sur les tests de sécurité et répondant à nos attentes. Néanmoins, ce qui est sûr est que le cas du vérifieur de byte code n'a pas été traité auparavant dans le but d'effectuer des études de sécurité.

Ce constat nous a amené à élaborer une nouvelle approche de test [BHLS11] [HLM11] répondant à

cet objectif de sécurité : l'analyse de vulnérabilité. Et comme on l'a évoqué précédemment (section 4.6), notre approche correspond à une approche de test en boite noire car elle est basée uniquement sur la spécification. Ce qui signifie qu'elle est générique du moment qu'elle n'est pas spécifique à une implémentation donnée. De ce fait, cette approche peut être très bien utilisée pour analyser la vulnérabilité de toute implémentation de vérifieur embarqué pour une plateforme Java Card (version 3.0 *Classic Edition* et versions antérieures *i.e.* les versions 2.x).

6.2 Processus général

Notre approche d'analyse de vulnérabilité du vérifieur de structure s'articule autour des cinq étapes ci-dessous et qu'on va présenter en détails dans les sections qui vont suivre :

1. Partant de la spécification de la machine virtuelle [Mic09c], nous avons construit un modèle abstrait du vérifieur de structure (un modèle B regroupant des données et les contraintes correspondantes) ;
2. Ce modèle est dérivé en appliquant successivement la négation des contraintes du modèle pour obtenir des modèles avec fautes (chacun d'entre eux comporte exactement une seule faute) ;
3. L'instanciation des modèles avec fautes à l'aide d'un solveur de contraintes donne les cas de test abstraits ;
4. Chacun des cas de test obtenus précédemment est transformé en un fichier CAP invalide constituant ainsi un cas de test concret pouvant être chargé directement dans la carte ;
5. L'analyse des réponses de la carte, au chargement des fichiers CAP invalides obtenus, et détection d'éventuelles vulnérabilités constitue la dernière étape de l'approche.

Le processus général de l'approche proposée est schématisé dans la figure 6.1.

6.3 Les étapes de l'approche proposée

6.3.1 Construction du modèle abstrait du vérifieur de structure

Le vérifieur de structure s'assure que les données reçues par la carte (*i.e.* celles contenues dans un fichier CAP) soient dans un format correct *i.e.* celui défini par Oracle [Mic09c]. Donc, il peut être vu comme une collection de tests qui vont être exécutés les un après les autres pour accepter, dans le cas favorable, les données. Dans la section 3.2.1 on a déjà évoqué le fait que le vérifieur de structure opère sur la base des composants : il effectue les tests internes (composant par composant) suivis de tests externes (pour les interdépendances) qui reposent sur le succès des premiers. De ce fait, son architecture est modulaire et donc plus facilement découpable.

Ainsi la modélisation du vérifieur de structure repose sur les trois éléments suivants :

- La modélisation de chaque composant du fichier CAP ;
- La modélisation des tests internes ;
- La modélisation des tests externes.

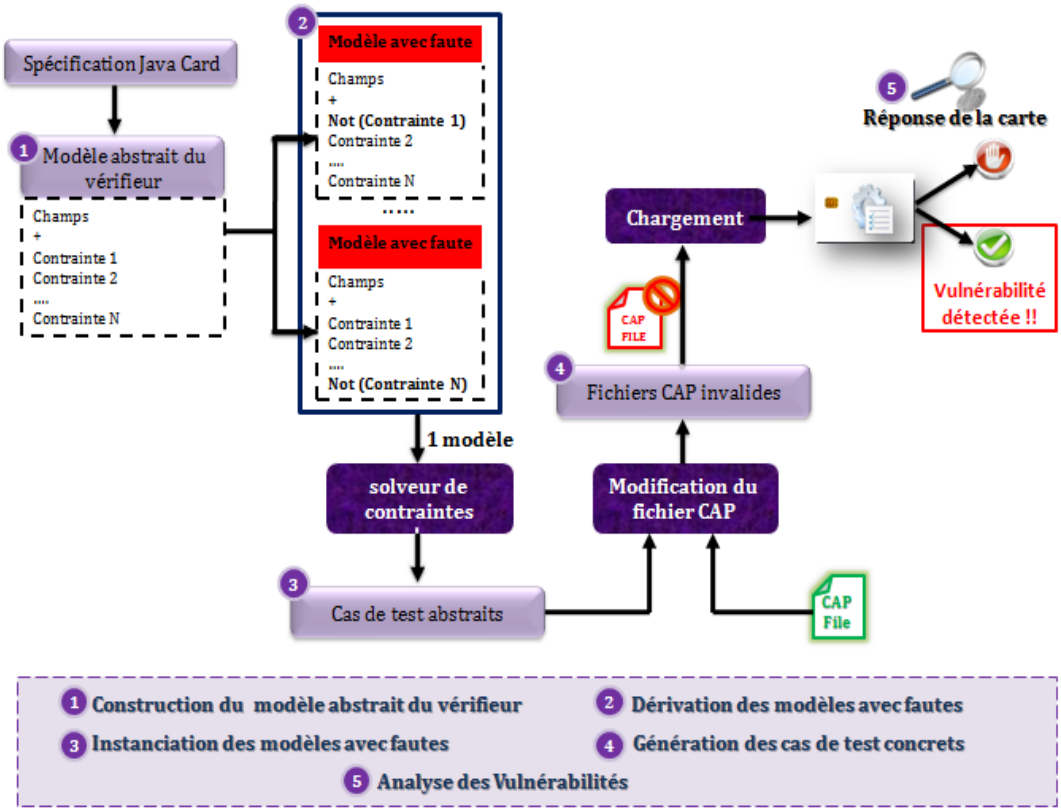

Fig 6.1 – Proposition de l'approche d'analyse de vulnérabilité du vérifieur de structure

L'obtention de ces 3 éléments clés pour la première étape de notre approche a nécessité un travail approfondi d'analyse et de synthèse des spécifications d'Oracle [Mic09c].

Travail préliminaire pour la modélisation

Le point de départ de notre modélisation est la spécification fournie par Oracle. Et plus exactement dans le chapitre 6 de [Mic09c], Oracle spécifie informellement le format du fichier CAP. Cela concerne la définition de tous les composants du fichier CAP. Pour chacun d'entre eux on trouve une description de sa structure en utilisant une notation proche du langage C suivie d'une description de chaque élément (ou champ, terme qu'on utilisera souvent dans la suite du chapitre) de cette structure en langage naturel. Un exemple d'une telle spécification est donné en annexe (annexe B). Il s'agit du composant Applet.

À partir d'une telle spécification, on doit extraire toute information nécessaire pour la vérification de structure. Ainsi, pour chaque composant on a classé les informations extraites en trois groupes :

1. Les éléments (ou champs) constituant le composant et leurs propriétés de typage ;
2. Les tests internes ;
3. Les tests externes.

Si on reprend l'exemple du composant Applet (annexe B), on peut extraire les informations suivantes :

– Les champs constituant le composant :

Champ	Type
tag	U1 : 1 octet non signé
size	U2 : 2 octets non signés
count	U1 : 1 octet non signé
applet [count]	Un tableau de structures qui comprend *count* éléments. Chaque structure comprend les champs suivants : – *aid-length* : 1 octet – *aid[ ]* : un tableau d'octet dont la taille est *aid-length* – *install-method-offset* : 2 octets

- Les tests internes :
  - Test1 : vérifier que $tag = 3$.
  - Test2 : vérifier que *size* correspond à la taille du composant (*tag* et *size* étant exclus du calcul).
  - Test3 : vérifier que *count* est supérieur à 0.
  - Test4 : pour chaque applet du tableau *applet[ ]* vérifier que la valeur *aid-length* est comprise entre 5 et 16 (valeurs incluses).
  - Test5 : vérifier que toutes les applets du tableau *applet[ ]* possèdent le même RID (*i.e.* les 5 premiers octets sont les mêmes).
- Les tests externes :
  - Test6 : vérifier que les RIDs des applets du tableau *applet[ ]* sont égaux au RID du package défini par le fichier CAP (information dans le composant Header).
  - Test7 : vérifier que tous les *install-method-offset* sont des décalages valides dans le composant *Method*.

Une fois ce travail d'analyse terminé, on peut entamer la modélisation de notre vérifieur de structure. Comme première étape on va traiter la modélisation des composants incluant les tests internes et par la suite on inclura les tests externes pour chaque composant.

Modélisation d'un composant

Avec une vision modulaire, chaque composant du fichier CAP est modélisé par une machine abstraite B qui inclue sa structure (champs et leurs types) ainsi que les tests internes correspondant. Tous les composants sont modélisés de la même façon. Ainsi, nous avons proposé le format général du modèle d'un composant X comme présenté dans la figure 6.2.

Dans la clause **USES**, on a fait référence à une machine abstraite *types* (figure 6.3) qui regroupe la définition des types de données utilisés à savoir : t-u1, t-u2, t-u4. En effet, ces types (définis dans la spécification) sont utilisés dans tous les composants donc par toutes les machines abstraites correspondantes. Afin d'éviter la redondance on a factorisé la définition des types dans la machine *types* à qui on fait appel à chaque fois qu'on en a besoin.

```
MACHINE
        cpn_composantX
USES
        types
CONCRETE_CONSTANTS
        /* Tous les champs définissant le composantX */
        ComposantX_champ1,
        ComposantX_champ2,
        ...
PROPERTIES
        /* Les propriétés de typage des champs */
        /* t_type est t_u1 ou t_u2 ou t_u4 */
        ComposantX_champ1 : t_type &
        ComposantX_champ2 : t_type &
        ...

        /* Les tests internes */
        L'expression de tous les tests identifiés
END
```

Fig 6.2 – Format général du modèle d'un composant X

```
MACHINE
    types
CONCRETE_CONSTANTS
    t_u1,
    t_u2,
    t_u4
PROPERTIES
    t_u1 = 0 .. 255 &
    t_u2 = 0 .. 65535 &
    t_u4 = 0 .. MAXINT
END
```

Fig 6.3 – Machine abstraite « *types* »

La clause **CONCRETE-CONSTANTS** contient une liste de tous les champs figurant dans la structure d'un composant. Dans un modèle B, on peut avoir des constantes abstraites ou concrètes. La différence entre le deux, d'après [Cle09], est que les premières sont des données de valeurs constantes qui seront raffinées dans le raffinement des composants alors que les secondes sont des données dont la valeur sera conservée au cours du raffinement. Et comme on ne va pas procéder au raffinement de nos modèles B (car notre but n'est pas leur implémentation) on a besoin de données constantes tout court, *i.e.* directement implémentables (représentant le contenu d'un fichier CAP). D'où le choix de la clause **CONCRETE-CONSTANTE**. La clause **PROPERRTIES** permet de typer les constantes (concrètes et abstraites) définies dans un composant et d'exprimer des propriétés sur ces constantes [Cle09]. Ça correspond exactement à notre but : typer les champs représentant un composant (exprimés sous forme de constantes) et définir les tests internes qui représentent des contraintes régissant les champs. Ces contraintes sont exprimées sous forme de prédicats (formules de la logique du premier ordre).

Modéliser les tests externes

Les tests externes impliquent des informations, *i.e.* les valeurs des champs, partagées par plusieurs composants du fichier CAP. Ainsi, leur modélisation consiste en la construction des interdépendances entre les composants. Ces derniers étant déjà modélisés dans l'étape précédente par des machines abstraites B, il devient plus facile de modéliser les tests externes. Pour le faire, il y a deux possibilités, pour chaque composant du fichier CAP :

- Choix 1 : modéliser ses tests externes par une machine abstraite B qui fait référence aux composants interdépendants avec le composant en question (ce dernier inclus aussi).
- Choix 2 : modéliser ses tests externes dans la même machine abstraite le définissant. Ceci en injectant les contraintes relatives à ces tests dans la clause **PROPERTIES**, à la suite des tests internes, et en rajoutant les composants référencés dans la clause **USES**.

Dans notre cas le premier choix ne fait qu'augmenter le nombre de machines abstraites constituant le modèle global du vérifieur. Et par conséquence, augmenter la difficulté du travail qu'aura à faire le solveur de contrainte dans l'étape 3 de notre approche lors de l'instanciation des modèles. Comme il n'y a pas un besoin de séparation des tests internes et externes pour un composant du fichier CAP on a opté pour le deuxième choix. Ainsi, pour chaque composant du fichier CAP on va utiliser une seule machine abstraite B pour le modéliser. Et cette machine regroupe les champs, leurs types, les tests internes et les tests externes relatifs au composant modélisé. Ainsi, pour l'exemple du composant Applet, on obtient la machine abstraite B le modélisant et qui est présentée dans la figure 6.4.

```
MACHINE
    applet_cpn
USES
    types,
    header_cpn
CONCRETE_CONSTANTS
    app_tag,
    app_size,
    app_count,
    app_aid_length,
    app_aid,
    app_install_method_offset
PROPERTIES
    app_tag  : t_u1 &
    app_size : t_u2 &
    app_count   : t_u1 &
    app_aid_length : t_u1 +-> t_u1 &
    app_aid : t_u1 +-> (t_u1 +-> t_u1) &
    app_install_method_offset : t_u1 +-> t_u2 &
    app_aid_length : 0..app_count-1 --> t_u1 &
    app_install_method_offset : 0..app_count-1 --> t_u2 &
    /*Check that the AID of each applet is different of the other applet's AID */
    app_aid        : 0..app_count-1 >-> (t_u1 +-> t_u1) &

    /*------------------------- internal tests ------------------------*/
    /*Test1---check the tag */
    app_tag = 3 &

    /*Test2---check the size*/
    app_size = 1 + ( SIGMA index.(index: 0 .. app_count-1|(3+ app_aid_length(index)))) &

    /*Test3---Check that Count >0*/
    app_count > 0 &

    /*Test4---Check that AID_length represents the length of the AID array*/
    !aid.(aid : 0..app_count-1
          => dom(app_aid(aid)) = 0..app_aid_length(aid)-1) &

    /*Test5---AID_length between 5 and 16 inclusive*/
    ! aid . ( aid: 0 .. app_count-1
          => app_aid_length (aid) : 5..16 ) &

    /*Test6---check that all the AIDs have the same RID= the first 5 bytes */
    !(aid1, aid2).((aid1 : 0..app_count-1 & aid2 : 0..app_count-1)
          => 0..4 <| app_aid(aid1) = 0..4 <| app_aid(aid2))&

    /*------------------------ external tests ------------------------*/
    /*Test7---check that applet's RID = package's RID for all applets */
    !aid.(aid : 0..app_count-1
          => 0 .. 4<| app_aid(aid) = 0 .. 4<| hdr_pkg_aid)
END
```

Fig 6.4 – Machine abstraite B représentant le composant Applet

Arrivé à ce niveau dans la modélisation, on dispose de modèles séparés des différents composants du fichier CAP (ils se référencient mais ils sont représentés par des machines distinctes). Cependant, notre but était d'avoir un modèle global du vérifieur de structure. De ce fait, il nous fallait un moyen pour lier tous les modèles des composants construits jusqu'ici. La solution est de définir une nouvelle machine abstraite B (figure 6.5) faisant référence, à travers une clause **USES**, à toutes les machines abstraites B représentant les composants du fichier CAP ainsi que la machine abstraite *types* définissant les types des données. Cette nouvelle machine (nommée *global*) joue le rôle d'une machine globale qui

sera transmise au solveur de contrainte à l'étape d'instanciation de modèles (Etape 3 section 6.3.3).

```
MACHINE
    global
USES
    types,
    header_cpn,
    directory_cpn,
    applet_cpn,
    import_cpn,
    constant_pool_cpn,
    class_cpn,
    method_cpn,
    static_field_cpn,
    reference_location_cpn,
    export_cpn,
    descriptor_cpn
END
```

Fig 6.5 – La machine abstraite « *global* »

Ainsi, on peut schématiser notre modèle du vérifieur de structure comme indiqué dans la figure 6.6.

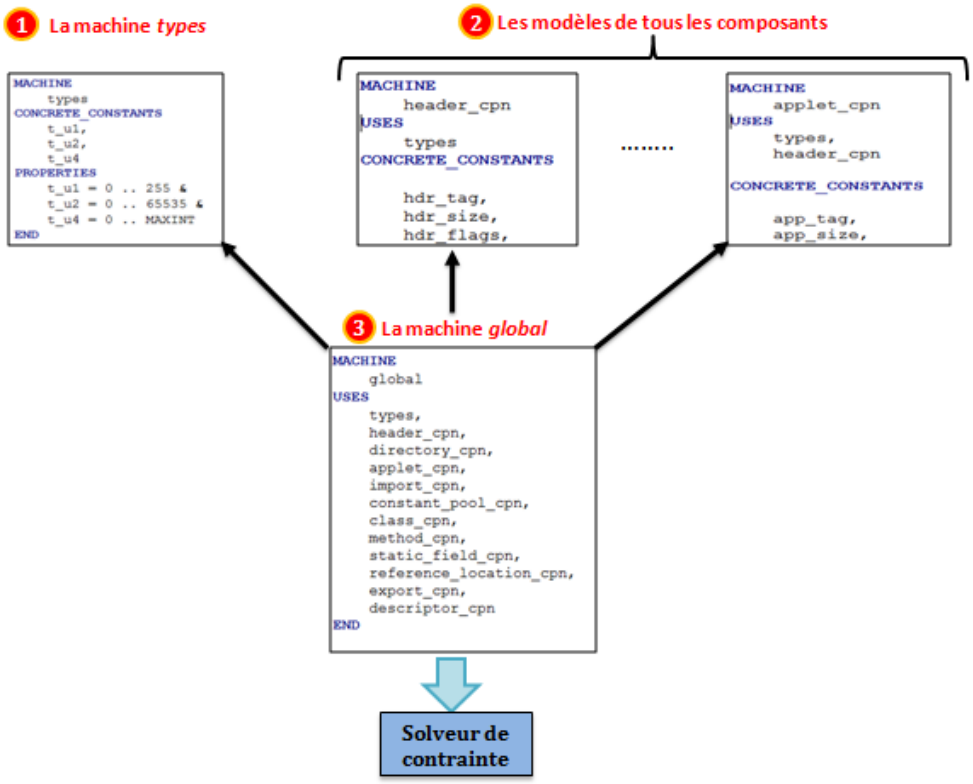

Fig 6.6 – Schéma global du modèle du vérifieur de structure

6.3.2 Dérivation des modèles avec fautes

Rappelons notre objectif principal qui consiste à générer un ensemble d'entrées invalides pour le vérifieur de structure sans avoir à générer l'ensemble de tous les cas possibles (valides et invalides). Autrement dit, on cherche à générer puis exécuter des données d'entrée intentionnellement corrompues

pour perturber le fonctionnement du vérifieur et analyser par la suite sa réaction. Ainsi, il faudra introduire une certaine intelligence pour avoir un ensemble pertinent (vis-à-vis de notre objectif) de cas de test réduisant ainsi l'effort nécessaire à la réalisation de la compagne de test.

L'idée est de construire des modèles qui contredisent la spécification, on les a appelé : *modèles avec fautes*. On a pu obtenir de tels modèles en se basant sur le modèle abstrait du vérifieur structurel construit à l'étape 1 de notre approche (section 6.3.1). En effet, ce modèle respecte la spécification fournie par Oracle [Mic09c] qui a été la base de son élaboration. Afin d'avoir les modèles avec fautes on a procédé à la négation successive des contraintes, relatives aux tests internes et externes, du modèle abstrait précédemment obtenu : seulement une contrainte niée à la fois afin d'avoir un modèle avec exactement une seule faute.

Ceci a l'avantage de permettre une injection de faute (au sens de données erronées) mieux ciblée. Autrement dit, dans le cas où un fichier invalide X, soumis au vérifieur structurel, soit accepté et donc il y a détection d'une vulnérabilité on peut très bien déterminer avec précision la source de cette faille en remontant au modèle avec faute qui a servi pour l'élaboration du fichier invalide en question. Et donc, trouver la contrainte niée causant ceci, *i.e.* le test qui n'a pas été correctement effectué par le vérifieur structurel.

Pour résumer, pour chaque modèle d'un composant du fichier CAP (*i.e.* au niveau de chaque machine abstraite B représentant un composant) on dérivera autant de modèles avec fautes que le nombre de tests internes et externes le constituant (bien évidemment les contraintes de typage sont exclues de la négation car elles n'ont aucun lien avec les tests structuraux).

On reprend l'exemple du composant Applet et son modèle abstrait pour voir ce que nous donne sa dérivation en modèles avec fautes. Par la suite, ceci peut être étendu à tous les autres composants (*i.e.* les modèles correspondants pour avoir une dérivation complète des modèles avec fautes pour le vérifieur structurel). Le résultat obtenu pour le composant Applet peut être schématisé dans la figure 6.7.

Dans ce schéma on fait abstraction aux autres modèles de composants pour des raisons de simplification seulement. Mais en réalité, quand on parle d'un modèle avec faute on sous-entend un modèle du système complet où il y a toutes les contraintes des composants avec seulement une contrainte niée. Si on regarde de plus près le premier modèle avec faute (un cas très simple pour mieux expliquer le principe) de la figure 6.7. La contrainte niée indique que : le champ *tag* doit être différent de la valeur 3. Ce qui est faux au regard de la spécification et ce sont exactement ces cas là qui nous intéressent. En répétant cette opération de négation à toutes les contraintes du modèle (relatives aux tests internes et externes seulement) une à une, nous n'aurons que des modèles qui permettront, dans la suite de l'approche, d'avoir des entrées (*i.e.* des fichiers CAP) invalides, ce qui correspond à notre objectif.

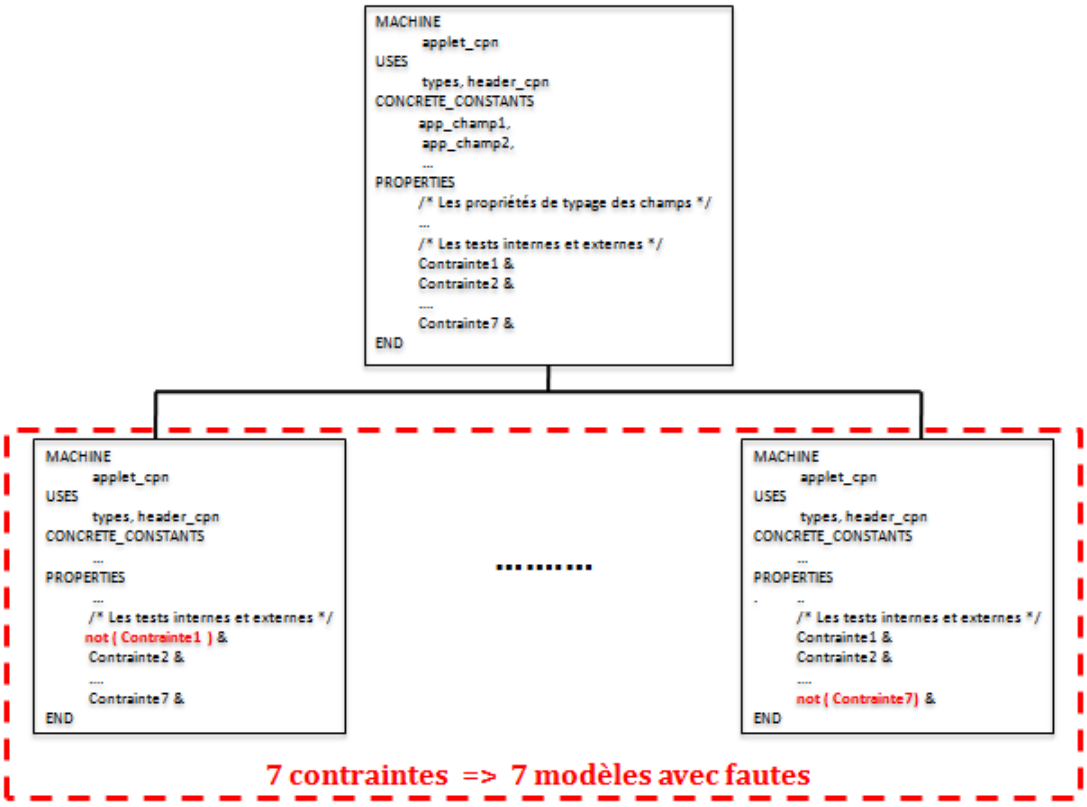

Fig 6.7 – Dérivation des modèles avec fautes pour le composant Applet

6.3.3 Extraction des cas de test abstraits

Une fois les modèles avec fautes obtenus, il nous faudra un moyen pour trouver les bonnes valeurs de tous les champs définis telles que ces valeurs respectent toutes les contraintes y compris celle qui a été niée pour introduire l'erreur.

D'après le format du fichier CAP (surtout les interdépendances des champs, figure 3.3), en plus de notre étude préliminaire de la spécification [Mic09c], on peut constater qu'on est face à un modèle contenant assez de champs et de contraintes les liant (largement supérieur à une centaine) que ça devient vraiment impossible d'en trouver les bonnes combinaisons de valeurs satisfaisant l'ensemble des contraintes. La solution est de faire recours à un solveur de contraintes. En effet, un tel outil nous a permis d'instancier toutes les variables (*i.e.* les champs) de chaque modèle avec faute en respectant toutes les contraintes définies dans ce dernier y compris celle qui a été modifiée pour introduire l'erreur. Ainsi, pour chaque modèle avec faute le solveur de contraintes peut trouver plusieurs instanciations possibles. Chacune de ces dernières constitue ce qu'on appelle : un cas de test abstrait.

6.3.4 Génération des cas de test concrets : les fichiers CAP invalides

Les tests abstraits obtenus à l'étape précédente (*i.e.* les instanciations des modèles avec fautes), tels qu'ils sont présentés ne peuvent pas être directement chargés dans une carte à puce. Ceci est dû

à leur format qui ne correspond pas exactement à celui attendu par le vérifieur de byte code. Il est donc nécessaire de procéder à une transformation des tests abstraits issus de la spécification vers des tests concrets qui seront soumis au vérifieur de byte code. Ces tests concrets consistent en des fichiers d'entrée compréhensibles par ce dernier : des fichiers CAP. Et plus précisément des fichiers CAP invalides car ils sont issus à l'origine des modèles avec fautes. Pour obtenir un fichier CAP invalide on a besoin de deux éléments : une instanciation d'un modèle avec faute et un fichier CAP valide. L'idée est simple (figure 6.8) : il faudra utiliser les valeurs présentes dans l'instanciation du modèle avec faute pour modifier les champs correspondants dans le fichier CAP valide. Il s'agit d'une modification d'un fichier CAP valide en un autre invalide mais pouvant être transmis directement à la carte.

Ainsi, on aura autant de fichiers invalides que d'instanciations de tous les modèles avec fautes. Et chacun de ces fichiers contient exactement une seule erreur.

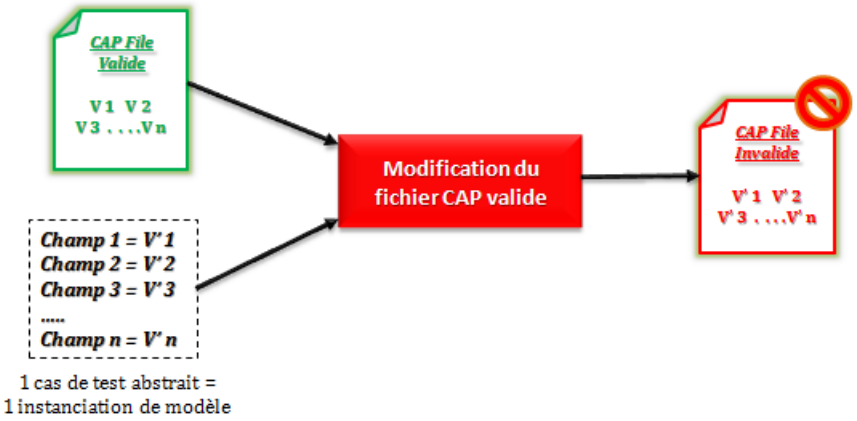

Fig 6.8 – Des tests abstraits aux tests concrets

6.3.5 Analyse de vulnérabilité

Arrivé à cette dernière étape, on ne dispose que de fichiers CAP invalides et qui sont prêts à être envoyés à la carte. Ce qui correspond bien à l'objectif fixé au début de notre travail. Ainsi, il ne reste plus qu'à envoyer successivement les différents fichiers CAP obtenus à la carte et analyser sa réponse. Cette dernière correspond en réalité à la réaction du vérifieur de byte code face au fichier transmis. Dans notre cas, ceci constitue l'oracle. En effet, le rejet d'un fichier implique que le vérifieur a accompli son rôle pour ce cas précis. Par contre, l'acceptation d'un fichier revient à dire qu'une vulnérabilité est détectée (*i.e.* une faiblesse au niveau du vérifieur de structure). Et comme on a fait de telle sorte à introduire exactement une seule erreur pour chaque fichier CAP invalide, on peut très bien remonter vers le modèle avec faute qui a servi comme base pour la construction de ce fichier et déterminer avec précision la contrainte concernée par la vulnérabilité détectée.

L'analyse des réponses de la carte revient à l'interprétation des codes renvoyés par cette dernière. En effet, on a évoqué précédemment (section 1.6) que dans le protocole APDU on a deux types de messages : les APDUs de commande et les APDUs de réponse. Ces derniers sont envoyés par la carte vers l'extérieur suite à l'exécution d'une commande. Cette réponse est constituée des deux octets SW1

et SW2 (et éventuellement de données) qui servent à indiquer l'état de la carte après cette exécution. Cependant, le code formé par ces deux octets est élaboré avec une certaine logique qui facilité son décodage afin de savoir si les commandes reçus par la carte se déroulent correctement ou, dans le cas contraire, quelles sont les causes des erreurs ou anomalies rencontrées. D'après la figure 6.9, on voit que les codes sont séparés en deux grandes familles concernant les commandes arrivées à leur terme et celles qui ont été interrompues.

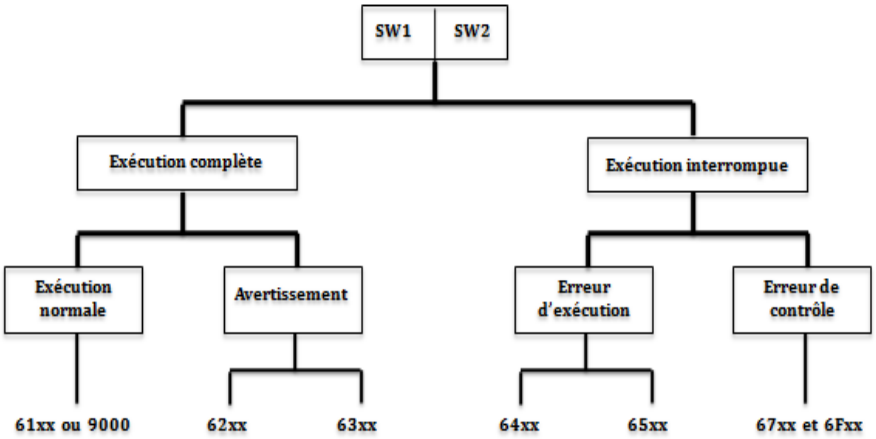

Fig 6.9 – Arborescence des valeurs des octets d'état SW1 et SW2 [Tav07]

La norme ISO 7816-4 définit un certain nombre de ces codes dont quelques uns sont regroupés dans le tableau 6.1.

Code	**Type**	**Signification**
61xx	EN	xx octets de données sont disponibles
6281	AV	Les données fournies peuvent être fausses
6283	AV	le fichier sélectionné est bloqué de manière irréversible
6581	EE	Erreur mémoire
6700	EC	Longueur incorrecte
6800	EC	Fonction non supportée par cette classe
6900	EC	Commande non autorisée
6981	EC	Commande incompatible de la structure du fichier
6982	EC	Conditions de sécurité non satisfaites
6983	EC	Authentification bloquée

6985	EC	Conditions d'utilisation non autorisées
6A00	EC	Paramètre P1 ou P2 incorrects
6A81	EC	Fonction non supportée
6A82	EC	Fichier introuvable
6A88	EC	Données référencées par la commande introuvables
6D00	EC	Commande non supportée
6F00	EC	Commande interrompue. Diagnostic plus précis impossible
9000	EN	Commande exécutée avec succès
Légende : EN : Exécution Normale ; AV : Avertissement ; EC : Erreur de Contrôle ; EE : Erreur d'Exécution		

Tab 6.1 – Principaux codes d'état définis par la norme ISO 7816-4 [Tav07]

Après cette présentation des codes relatifs aux réponses possibles d'une carte à puce, on peut conclure que dans notre cas où on n'envoie à la carte que des fichiers CAP invalides, une réponse de 90 00 atteste sur la présence d'une vulnérabilité au niveau du vérifieur de structure. Alors que tout autre code indique que le fichier a été rejeté par le vérifieur.

6.4 Optimisation de l'approche : les modèles pré-remplis

Pour arriver à de meilleurs résultats lors de l'application de notre approche, on a pensé à une optimisation de cette dernière. On a pu constater que la négation d'une contrainte correspond à un nombre relativement restreint de variables qui seront impliquées sans que les autres variables ne prennent effet. L'ensemble restreint de variables en question correspond aux variables impliquées directement dans la contrainte niée (*i.e.* dans sa définition) en plus de celles qui le sont par transitivité (on sous-entend l'interdépendance entre les champs d'un même composant ou ceux des différents composants).

Dans cette idée, on peut construire nos cas de test concrets en s'appuyant sur des fichiers valides et n'apporter de modifications qu'aux variables concernées par la négation d'une contrainte. Pour ce faire, on a introduit un modèle pré-rempli.

Pratiquement, il correspond à une nouvelle machine abstraite B (nommée : *valid-cap*) qu'on a défini et qui regroupe tous les champs de tous les composants d'un fichier CAP. C'est une machine spéciale qui constitue une sorte de projection d'un fichier CAP valide (celui qu'on utilise dans l'étape de génération des cas de test concrets) dans un modèle. Autrement dit, pour chaque champ défini on récupère sa valeur à partir du fichier CAP valide afin de la lui associer au niveau du modèle pré-rempli, formant ainsi les contraintes qui sont toutes de la forme suivante : *Champ-x = valeur*. Donc, au final on aura une machine abstraite avec une clause **USES** dans laquelle on fait appel à toutes les machines représentant les composants (on n'aura pas à déclarer une autre fois tous les champs d'un fichier CAP).

En outre, on a une clause **PROPERTIES** regroupant toutes les contraintes (*Champ-x = valeur*). Un exemple d'un tel modèle est donné dans la figure 6.10.

```
MACHINE
    valid_cap
USES
    header_cpn,
    directory_cpn,
    applet_cpn,
    ...
PROPERTIES

    /*================= Header_cpn =========
    hdr_tag =1 &
    hdr_size= 26 &
    hdr_magic= DECAFFED &
    hdr_minor_version =1 &
    hdr_major_version =2 &
    hdr_flags=4 &
    hdr_pkg_minor = 0 &
    hdr_pkg_major = 2 &
    ...
    /*================= Directory_cpn ======
    dir_tag = 2 &
    dir_size = 31 &
    dir_component_sizes(0) = 26 &
    dir_component_sizes(1) = 31 &
    dir_component_sizes(2) = 39 &
    ...
END
```

Fig 6.10 – Exemple d'un modèle pré-rempli

Pour prendre en considération ce modèle pré-rempli, on doit adapter légèrement la machine globale qui sera transmise au solveur de contraintes. Dans sa clause **USES** on rajoute l'appel au modèle pré-rempli. Ainsi, on aura l'architecture globale du modèle du vérifieur qui est représentée dans la figure 6.11.

Au niveau de l'étape de dérivation des modèles avec fautes (section 6.3.2), on aura une adaptation de plus à faire. Elle consiste à masquer au niveau du modèle pré-rempli les champs qu'on a identifié et qui sont impliqués dans la négation d'une contrainte. Autrement dit, ce modèle contiendra toutes les valeurs des champs d'un fichier CAP sauf ceux liés à la contrainte niée. Ainsi, le travail du solveur de contraintes sera largement simplifié car au lieu de résoudre le système de contraintes pour trouver toutes les valeurs, il n'aura qu'à trouver les valeurs des champs masqués dans le modèle pré-rempli. Ce qui constitue vraiment un réduction du temps de calcul (résultats section 7.3).

En plus de cette optimisation à cette étape de l'approche, on peut constater une autre amélioration au niveau de l'étape de construction des cas de test concrets (*i.e.* les fichiers CAP invalides). En effet, au lieu de récupérer toutes les valeurs et modifier les champs correspondant dans le fichier CAP valide, on ne récupère que les valeurs des champs impliqués dans la contrainte niée pour modifier leurs correspondants dans ce fichier. Et ceci, toujours dans le but d'avoir un fichier CAP invalide. Ce qui permet aussi de réduire considérablement le temps nécessaire pour la modification d'un fichier CAP.

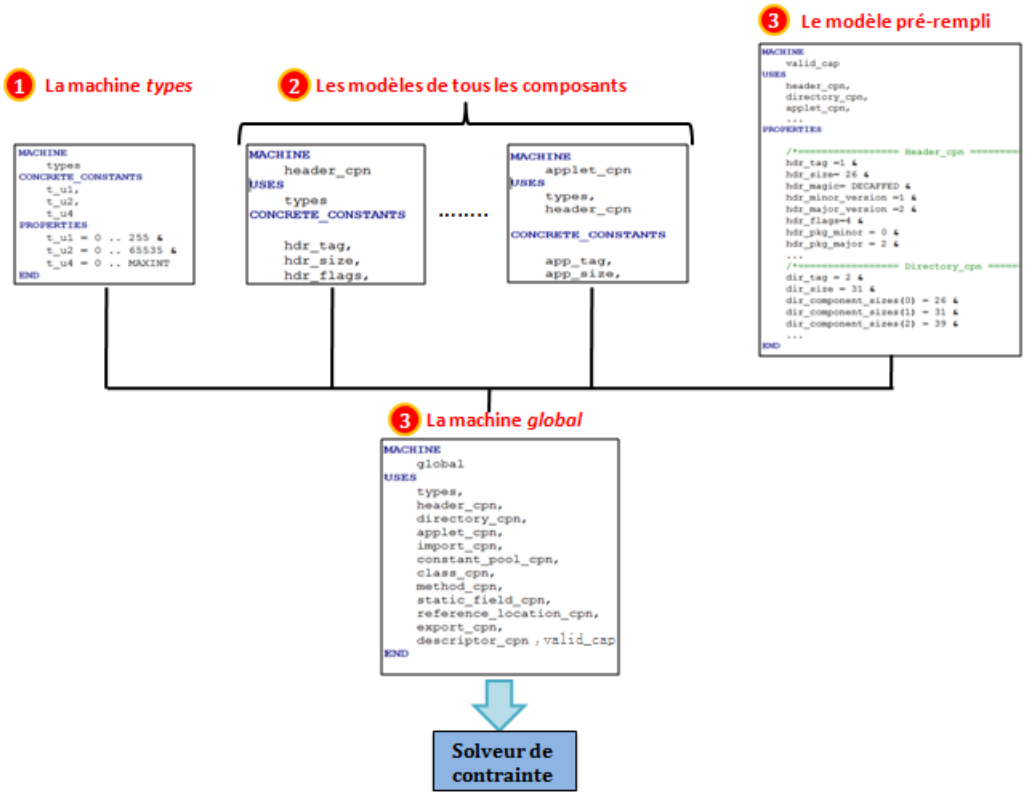

Fig 6.11 – Architecture globale du modèle du vérifieur après optimisation

6.5 Conclusion

Ce chapitre a permis d'apporter une réponse à la problématique posée au début de ce travail (section 3.5) : « comment vérifier le vérifieur de structure ? ». Cette réponse se résume en une approche d'analyse de vulnérabilité à travers des données de tests de sécurité générées à partir d'un modèle formel B (construit à partir de la spécification d'Oracle [Mic09c]). Le contenu de chacune des 5 étapes de cette approche a été présenté en détail dans ce chapitre. De plus, une idée d'optimisation de l'approche à été présentée. Elle se base sur l'utilisation des modèles pré-remplis pour la génération des données de test. A présent, il ne reste plus qu'à passer à la pratique : la mise en œuvre de l'approche et l'évaluation des résultats expérimentaux.

CHAPITRE 7

Implémentation et Evaluation des Résultats

Sommaire

Ce chapitre porte sur l'implémentation de l'approche proposée, qui a été présentée dans le chapitre 6, ainsi que l'évaluation des résultats expérimentaux obtenus sur une vraie carte à puce disposant d'un vérifieur de byte code embarqué. Tout d'abord, on va présenter les différents outils, dans l'ordre de leur utilisation, qui ont permis la mise en œuvre de notre approche. Ensuite, on exposera nos résultats expérimentaux qui seront une sorte de validation de cette dernière.

7.1 Les choix techniques d'implémentation

7.1.1 Construction du modèle abstrait : Atelier B

L'outil de formalisation et de modélisation de notre vérifieur de structure est l'Atelier B [1]. Cet outil permet une utilisation opérationnelle de la méthode B. Il offre, au sein d'un environnement cohérent, de nombreuses fonctionnalités permettant de gérer des projets en langage B. Ces fonctionnalités (tableau 7.1) se regroupent en quatre catégories [Cle06] :

1. `http://www.atelierb.eu`

- Les tâches automatisables lors du développement d'un projet : les vérifications syntaxiques des composants, la génération automatique des obligations de preuve, la traduction automatique des implantations B vers les langages C ou Ada ;
- Une aide à la preuve, pour démontrer les obligations de preuve, grâce à des outils de preuve adaptés ;
- Une aide au développement : gestion automatique des dépendances entre composants B, bibliothèques réutilisables, génération de documentation et génération automatique de métriques ;
- Des outils de confort pour l'utilisateur : représentation graphique de projets, navigateur hypertexte pour se diriger dans un projet, affichage de l'état et des statistiques d'un projet, génération automatique du dictionnaire des termes d'un projet, archivage de projets.

Fonctionnalité	Description
Les analyseurs syntaxiques	– un éditeur de modèle a été intégré à l'Atelier B. Celui-ci intègre un analyseur syntaxique, la complétion automatique ainsi que des fonctionnalités de navigation au travers du modèle – le Type Checker effectue d'abord la vérification grammaticale d'un composant B, et ensuite un certain nombre de vérifications contextuelles dont le contrôle de type et le contrôle de portée des identificateurs. – le B0 Checker effectue des vérifications spécifiques au langage B0 utilisées dans les implantations (une sous-partie du langage B) pour assurer que celles-ci peuvent être traduites. – le vérificateur de projets effectue des vérifications sur l'ensemble des composants d'un projet pour contrôler son architecture (les liens entre les composants). – Les modèles B peuvent être sauvegardés aux formats pdf, rtf et LaTeX
Le raffinement automatique	L'outil de raffinement automatique (BART) permet de générer raffinement et implémentation à partir d'une base de règle de raffinement extensible par l'utilisateur.
Les traducteurs automatiques	Facilitant la génération de code sur n'importe quel système cible, les implantations sont traduites automatiquement en langage de programmation classique. Les programmes obtenus peuvent alors être compilés et assemblés sur la machine cible pour produire le logiciel exécutable.

Les outils de preuve	– la génération automatique des obligations de preuve à partir des composants en langage B – la preuve en mode automatique : sans intervention de l'utilisateur, la plupart des obligations de preuve sont démontrées – la preuve en mode interactif utilisée lorsque le mode automatique a échoué – la visualisation des obligations de preuve, leur origine et leur état – le prouveur de prédicats : démonstration de prédicats, démonstration des règles ajoutées par l'utilisateur (triviales, prouvées, non prouvées)
La représentation graphique des projets	Représenter graphiquement les composants d'un projet et leurs liens, en les positionnant automatiquement au sein du graphe.
La gestion de projets	– utilisation de l'Atelier B par plusieurs utilisateurs en réseau – archiver et restaurer un projet complet – architecturer un projet en plusieurs sous-projets ou bibliothèques. – visualiser de manière synthétique l'état d'un projet, en fournissant pour chaque composant, son état, le nombre d'obligations de preuve et le pourcentage prouvé, – générer automatiquement les dépendances entre composants d'un projet – intégration du Paralléliseur : une application permettant de lancer sur des machines distantes partageant le même système de fichiers des tâches de preuves sur un projet B

Tab 7.1: Principales fonctionnalités de l'atelier B

7.1.2 Le solveur de contraintes : ProB

Un problème de satisfaction de contraintes (CSP pour *Constraint Satisfaction Problem*) est représenté par un triplet (V, D, C) où V est un ensemble de variables prenant leur valeur dans un domaine précisé pour chaque variable (leur ensemble est D) et sur lesquelles portent des contraintes C. Une solution pour CSP est une affectation complète des variables (*i.e.* l'attribution d'une valeur unique à

chaque variable) telle que toutes les contraintes soient vérifiées. La résolution d'un CSP met en œuvre de manière entrelacée des étapes de filtrage (réduction sophistiquée des domaines des variables pour éliminer les portions de l'espace de recherche ne pouvant faire partie d'une solution) et des étapes d'énumération (prise de décision pour se rapprocher d'une solution). Un solveur de contrainte est un système programmable qui réalise le contrôle de cet entrelacement en faisant appel à divers composants de filtrage (les contraintes elles-mêmes) et de contrôle de l'énumération [Jus03].

Dans notre cas, le choix du langage B comme langage de modélisation nous a amené à choisir un solveur de contraintes ensembliste qui soit compatible : ProB [2] [LB03b], [LB03a]. C'est un outil développé en programmation logique avec contraintes permettant d'exprimer des modèles formels via la méthode B. ProB permet une animation complètement automatique des modèles B. Il autorise entre autres les opérations non-déterministes, les instructions *ANY*, les opérations avec arguments complexes, les ensembles, les séquences, les fonctions, les lambda expressions, les constantes et les propriétés. Contrairement à d'autres outils similaires, l'utilisateur n'a pas à deviner les bonnes valeurs pour les arguments d'opérations ou les variables de choix. Ceci est géré par co-routinage et résolution de contraintes dans les domaines finis. Enfin, ProB incorpore un vérifieur temporel et un vérifieur à base de contraintes, pouvant tous les deux détecter des erreurs dans les spécifications B.

ProB a été développé en SICStus Prolog, avec une interface graphique implémentée en Tcl/Tk. L'animateur permet de faire une animation *step-by-step* d'une machine B. Comme le montre la figure 7.1, ProB fournit à l'utilisateur :

- une description de l'état courant de la machine (State Properties), c'est là qu'on va récupérer les valeurs des champs de notre modèle ;
- l'historique qui a permis d'atteindre cet état courant (History) ;
- et une liste des opérations permises (Enabled Oprations). Dans notre cas, elle constitue la liste des cas de test abstraits *i.e.* les instanciations d'un modèle avec faute.

Si on reprend un exemple d'un modèle avec faute (contrainte : not(app-tag = 3)), et on le fait passer à ProB. Il résout le système de contraintes pour rendre de façon automatique un ensemble d'instanciations possibles (Enabled operations). En choisissant une instanciation de la liste on obtient un état courant de la machine énumérant l'ensemble des champs du modèle avec leurs valeurs (figure 7.2). De plus, ProB peut fournir un fichier texte en sortie, facilement exploitable, sauvegardant un état courant (*i.e.* un cas de test abstrait).

2. `http://www.stups.uni-duesseldorf.de/ProB/index.php5/The_ProB_Animator_and_Model_Checker`

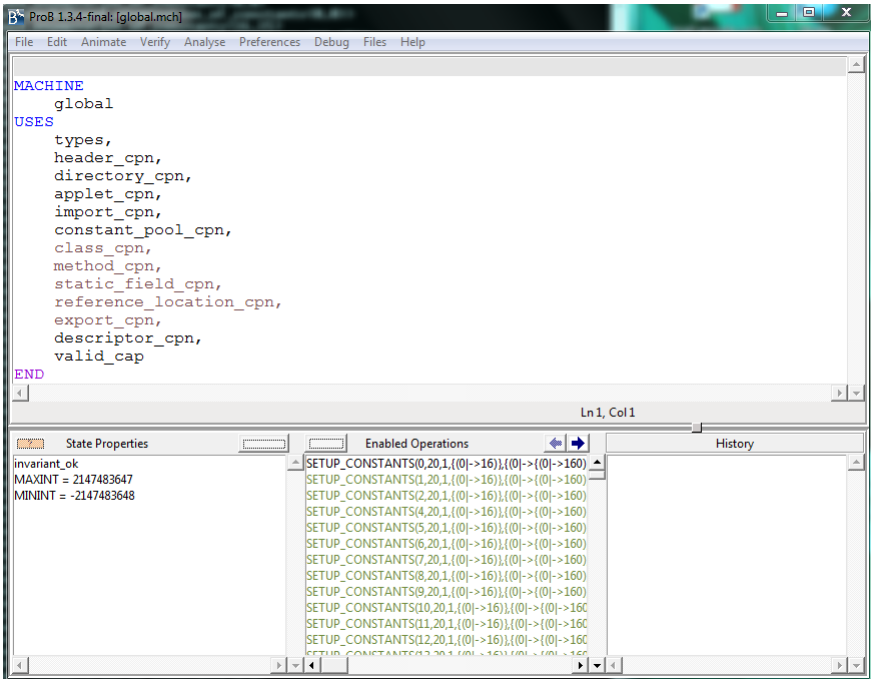

Fig 7.1 – Animation d'un modèle B dans ProB

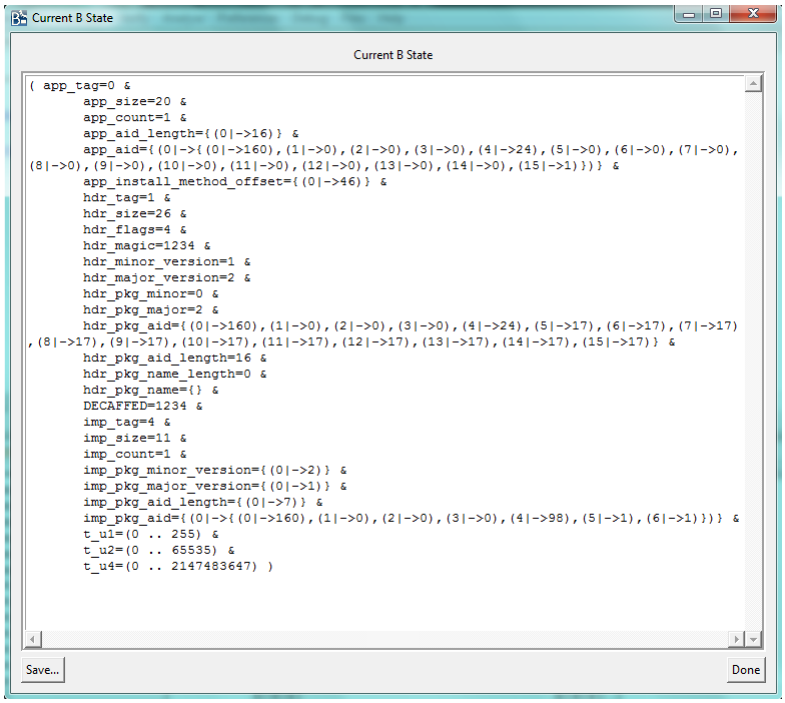

Fig 7.2 – Exemple d'une instanciation d'un modèle avec faute

7.1.3 Modification du fichier CAP : Cap File Manipulator

La modification d'un fichier CAP s'avère une opération complexe, fastidieuse et surtout très délicate. Ceci s'explique par le fait qu'un fichier CAP est un fichier binaire dont la modification nécessite une localisation avec précision du composant puis de l'élément en question. Et une fois la modification

établie, il faut arranger tous les décalages et interdépendances pour maintenir la cohérence du fichier. D'où la nécessité d'avoir un outil qui permet d'offrir une représentation plus adéquate du fichier CAP, et surtout une modification plus facile avec un maintien automatique de la cohérence de ce dernier.

Le *Cap File Manipulator* est une librairie Java qui offre la possibilité de la lecture et la modification des fichiers CAP (compatible avec la Java Card 3.x Classic Edition et versions antérieures). Elle est disponible [3] sous la forme d'un plug-in pour Eclipse (figure 7.3). Cette librairie permet de [NSIcL09] :

- Charger un fichier CAP et le parser ;
- Afficher les données du fichier composant par composant et sous un format intelligible ;
- Modifier les éléments d'un fichier CAP et vérifier sa validité en maintenant sa cohérence ;
- Et de sauvegarder les fichiers CAP éventuellement altérés.

Dans notre cas, cet outil nous a été d'un grand intérêt pour l'obtention de nos fichiers CAP invalides. En effet, on s'est basé sur un fichier CAP valide qu'il fallait à chaque fois ouvrir avec le *Cap File Manipulator* pour apporter les modifications nécessaires (les valeurs retournées par le solveur de contraintes) dans le but d'obtenir un fichier CAP invalide.

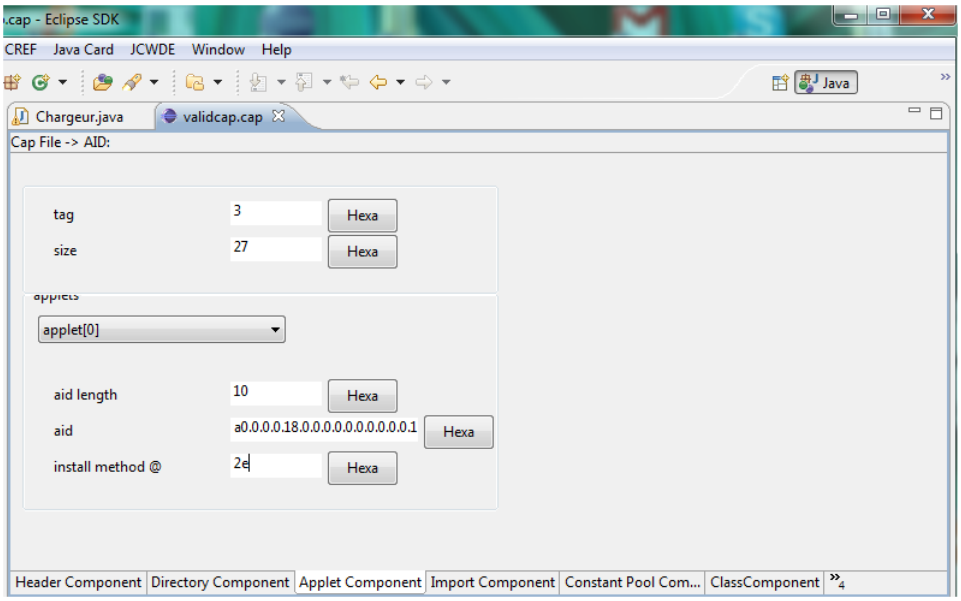

Fig 7.3 – Exemple d'un fichier CAP ouvert avec le *Cap File Manipulator*

7.1.4 Chargement des fichiers CAP : OPAL

GlobalPlatform

L'utilisation des cartes multi-applicatives (dont Java Card en fait partie) à grande échelle est régie par les spécifications GlobalPlatform [4]. Elles fournissent une architecture globale de gestion de la sécurité et des cartes. Le but de ces spécifications est d'apporter un standard pour gérer les cartes de façon indépendante du matériel, des vendeurs et des applications. Les spécifications GlobalPlatfom portent sur les aspects relatifs :

3. `http://secinfo.msi.unilim.fr/software/cap-file-manipulator/`
4. `http://www.globalplatform.org/specificationscard.asp`

1. aux cartes, incluant principalement :
   - Card Specification v2.1.1
   - Guidelines for Developing Java Card applications on GlobalPlatform Cards v1.0
   - Card Security Requirements Specifications v1.0
2. aux terminaux, incluant :
   - GlobalPlatform Device API v2.0
   - Device Tools Supporting Documents
   - GlobalPlatform Guidelines for Using GPD 2.0 /STIP 2.1 API v1.0
3. aux systèmes, incluant :
   - GlobalPlatform Messaging Specification v1.0
   - Key Management Requirements Systems Functional Requirements Specification
   - GP Load and Personalization Interface v1.0
   - GP Guide to Common Personalization v1.0

Dans la suite, nous présenterons brièvement l'architecture GlobalPlatform embarquée sur la carte uniquement, *i.e.* la partie relative au document *GlobalPlatform Card Specification v2.1.1*. Elle spécifie principalement :

- Les conditions d'interopérabilité et d'indépendance relativement à la cible pour implémenter GlobalPlatform ;
- Les communications hors carte avec le terminal et les communications sur la carte pour la gestion des applications. Dans notre cas, c'est ce point là qui nous intéresse : comment communiquer avec la carte pour procéder au chargement de nos fichiers CAP.

L'architecture GlobalPlatform

Cette architecture comprend un certain nombre de composants permettant de s'abstraire du matériel du vendeur grâce à des interfaces pour les applications (API standardisée) et pour le système de gestion hors carte (APDUs standardisées). Principalement, on trouve les composants suivants :

1. Le gestionnaire de la carte (*Card Manager*) : Il joue le rôle d'administrateur central de la carte. C'est le représentant de l'émetteur de la carte sur cette dernière au travers du domaine de sécurité de l'émetteur (ISD pour *Issuer Security Domain*). Ce dernier a la possibilité de charger, d'installer et d'effacer les applications appartenant à l'emetteur mais aussi à celles d'autres fournisseurs d'applications.
2. Les domaines de sécurité (*Security Domains*) : Sont les représentants des fournisseurs d'applications sur la carte. Ils proposent des services de sécurité comme la manipulation des clés, le chiffrement, le déchiffrement, la génération et la vérification des signatures. Chaque domaine de sécurité implémente un protocole de canal sécurisé (SCP pour *Secure Channel Protocol*) qui permet de sécuriser la communication entre l'émetteur de carte, le fournisseur d'applications, ou l'autorité de contrôle et son domaine de sécurité sur la carte. Chacun de ces acteurs possède ses propres clès et qui sont complètement isolées de celles des autres domaines de sécurité.

3. Les APIs de GlobalPlatform : Elles fournissent des services aux applications (vérification du détenteur de la carte, personnalisation, services sécuritaires). Elles offrent aussi aux applications des services de gestion du contenu de la carte (blocage de la carte ou mise à jour des états du cycle de vie de l'application).

GlobalPlatform propose plusieurs mécanismes de sécurité, la plupart sont d'ordre cryptographique. Ils spécifient des méthodes pour : sécuriser les communications, s'assurer que les applications chargées sont officiellement signées et vérifier l'identité du porteur de la carte.

Pour sécuriser les échanges entre la carte et une entité extérieure, GlobalPlatform spécifie les mécanismes :

- D'authentification mutuelle : C'est une phase durant laquelle une carte et une unité extérieure vérifient l'identité de leur correspondant en s'assurant qu'ils partagent les mêmes secrets. L'établissement de cette authentification se fait à travers plusieurs étapes (des échanges entre les deux parties).
- Canal sécurisé : Après une authentification mutuelle sécurisée, GlobalPlatform propose plusieurs niveaux de canal sécurisé :
  - Niveau authentification ;
  - Niveau authentification et intégrité ;
  - Niveau authentification, intégrité et confidentialité.

Donc, la communication avec une carte à puce (et spécialement une Java Card dans notre cas) nécessite l'utilisation d'une librairie spécifique qui doit implémenter des fonctionnalités de haut niveau conformément aux spécifications GlobalPlatform.

OPAL

OPAL [5] [BBICL11] est une librairie Java open source qui implémente les spécifications GlobalPlatform définissant entre autres les protocoles d'authentification, de cryptographie et de transfert pour les cartes à puce. Ce qui constitue un support complet pour la gestion du cycle de vie des applets Java Card.

Ainsi, pour parvenir au chargement de nos fichiers CAP, il fallait préparer un script de chargement en se basant sur la librairie OPAL (répondant aux spécifications GlobalPlatform). Pour cela, ce programme (donné en Annexe C) implémente les étapes suivantes :

- Obtention d'un canal de communication avec la carte (Card Channel)
- Obtention de l'ATR et chargement de la configuration de la carte (ISD, SCPmode, Protocole de transmission, les clés de chargement) ;
- Sélection du domaine de sécurité ;
- Authentification mutuelle dans le domaine de sécurité (Initialize Update et External Authenticate) ;

5. `http://secinfo.msi.unilim.fr/opal/`

- Installation de l'applet, *i.e.* le chargement du fichier CAP : choix du fichier CAP, load et install for install and make selectable ;
- Suppression de l'applet puis du package (le but est de remettre la mémoire de la carte dans son état avant le chargement du fichier CAP)

Dans notre cas, pour chaque fichier CAP invalide qu'on veut soumettre à la carte on fait exécuter ce programme. Toutes les réponses de la carte sont sauvegardées dans un fichier Log qu'on a prévu à cet effet. Après chaque exécution, il suffit de repérer la réponse de la carte (APDU de réponse) et l'analyser pour conclure de la présence d'une vulnérabilité ou non.

7.2 Plateforme de test

1. Les expérimentations se sont déroulées sur une machine disposant d'un processeur Intel Core 2 Duo (1.83 GHz, 2 Mb L2 cache) et d'une mémoire RAM DDR2 de 2GB. Sous le système d'exploitation Windows7 (32 bits).
2. Les tests ont été exécutés sur une carte à puce Java Card 2.x disposant d'un vérifieur de byte code embarqué (on n'a pas pu obtenir d'autres prototypes de telles cartes car elles ne sont pas très disponibles à l'heure actuelle).
3. Pour les outils qu'on a présenté précédemment, on a utilisé les versions suivantes :
   - Atelier B 4.0
   - ProB 1.3.4
   - Eclipse 3.5.2
   - Cap File Manipulator 0.0.2T
   - OPAL 0.1

7.3 Évaluation des résultats expérimentaux

Arrivé à ce niveau, on peut dire que toute la méthodologie adoptée par notre approche est bien définie. Ce qui reste alors c'est la validation de l'approche à travers des résultats expérimentaux. Pour mieux justifier l'apport de notre approche on a établi une comparaison entre elle et un plan manuel de test dont on dispose. On rappelle qu'actuellement il n'y a aucune approche existante qui soit dédiée à l'analyse de vulnérabilités d'un vérifieur de byte code. De ce fait, la comparaison qui est faite est la seule qui peut avoir lieu. Pour se faire, on a étudié deux métriques : le nombre de contraintes et le nombre de cas de test générés.

Tout d'abord, partant de la spécification de Oracle, on a construit un modèle partiel du vérifieur et qui a servi de base pour l'application de toutes les autres étapes de l'approche. Le modèle considéré porte sur les quatre composants suivants : Header, Directory, Applet et Import.

Nombre de contraintes

Les contraintes considérées sont celles relatives aux tests internes et aux tests externes seulement (*i.e.* les contraintes de typage sont exclues du calcul). Les résultats obtenus sont résumés dans le tableau 7.2.

Composants	Nombre de contraintes						Taux de réduction
	Approche proposée			Plan de test manuel			
	Internes	Externes	Total	Internes	Externes	Total	
Header	8	/	8	9	7	16	50%
Directory	9	17	26	15	25	40	35%
Applet	6	1	7	9	10	19	63.2%
Import	5	1	6	7	9	16	62.5%

Tab 7.2 – Résultats expérimentaux sur le nombre des contraintes

Pour chacun des quatre composants on a recensé les contraintes correspondantes (internes et externes). On peut constater que dans tous les cas on obtient moins de contraintes dans les modèles des composants avec notre approche que ce qui existe dans le plan de test manuel. Après l'analyse de ce dernier, on peut dire que ceci est dû aux redondances qui existent lors de la définition des contraintes. Chose qui est évitée avec notre approche.

Cette réduction du nombre de contraintes est représentée par un taux de réduction qu'on a calculé pour chaque composant (dernière colonne du tableau 7.2). Donc pour les quatre composants considérés on a déjà un taux moyen de réduction égal à 52.7% dans notre approche par rapport au plan de test manuel.

Nombre de cas de test

On sous-entend par cas de test les fichiers CAP invalides qui seront soumis à la carte. Il faut préciser une chose importante. Dans notre cas, le solveur de contraintes retourne un grand nombre de cas de test abstraits suite à l'instanciation d'un modèle avec faute. Donc ça serait fastidieux et peut-être sans grand intérêt le fait de les transformer tous en cas de test concrets. La solution a été donc de faire recours au test aux limites (qui est adoptée même dans le plan de test manuel). Cette stratégie de sélection des valeurs des données de test part du principe qu'un logiciel a plus de chance de dévoiler ses erreurs lorsqu'on utilise des valeurs limites du domaine de définition d'une donnée. Donc si une valeur amène à détecter une erreur, toutes les valeurs de ce même intervalle auraient mené à cette erreur. Si au contraire cette valeur ne permet pas de détecter une erreur, aucune valeur de l'intervalle considéré ne le permet. Un test aux limites correspond à tester les valeurs extrêmes de l'intervalle considéré. Par exemple, si on prend la contrainte *not (app-tag = 3)* qui implique que app-tag $:\in [0, 2] \cup [4, 255]$.

D'après le principe du test aux limites on va construire des cas de test pour les valeurs 0, 2, 4, 255 de la variable app-tag. Les résultats obtenus sont résumés dans le tableau ci-dessous (tableau 7.3).

	Nombre de cas de test		Taux de réduction
Composants	Approche proposée	Plan de test manuel (incomplet)	
Header	14	18	22.3%
Applet	9	16	43.8%
Import	9	13	30.8%

Tab 7.3 – Résultats expérimentaux sur le nombre des cas de test

Pour cette métrique on n'a pas considéré le composant Directory car on n'a pas encore un nombre final des cas de test. Ceci est dû au fait que ce composant a des liens d'interdépendance avec tous les autres composants du fichier CAP. Donc une fois qu'on aura tous les modèles de ces derniers on pourra calculer avec précision ce nombre.

Pour les trois autres composants considérés, les cas de test générés permettent d'obtenir une couverture totale des contraintes du modèle construit. Par contre, avec le plan de test manuel cette couverture est partielle. Il y a certains cas qui n'ont pas été considérés (c'est pour cela qu'on a indiqué « incomplet »dans la colonne correspondante du tableau 7.3). Cependant malgré cette incomplétude dans le plan de test manuel notre approche offre des résultats nettement meilleurs avec une couverture totale des contraintes.

Comme on l'a déjà fait pour la première métrique, on a calculé la taux de réduction du nombre de cas de test qu'on a pu obtenir (dernière colonne du tableau 7.3) et qui est en moyenne égal à 32.3%.

Bien que la stratégie de test aux limites soit efficace et simple à appliquer, elle n'est pas suffisante pour garantir une couverture complète des cas possibles. Mais elle constitue un compromis entre couverture et nombre de cas de test (que généralement on vise à réduire).

Temps d'exécution du solveur de contraintes

Cette métrique ne rentre pas dans la comparaison de notre approche par rapport au plan de test manuel. Mais plutôt, elle concerne l'optimisation de l'approche qu'on a présenté dans la section 7.3. Plus précisément, on veut illustrer l'apport de cette optimisation en termes de temps d'exécution avec et sans des modèles pré-remplis. Le tableau 7.4 résume les temps d'exécution de ProB dans les deux cas. On a paramétré ProB pour un nombre maximal d'instanciations égal à 256 cas.

		Runtime de ProB	
Composant	Contrainte niée	Sans modèle pré-rempli	Avec modèle pré-rempli
Applet	C1	2.215 s	2.012 s
	C2	2.183 s	2.215 s
	C3	2.043 s	1.919 s
	C4	timeout (> 10 min)	2.184 s
	C5	timeout (> 10 min)	3.603 s
	C6	timeout (> 10 min)	2.683 s
	C7	timeout (> 10 min)	2.402 s

Tab 7.4 – Résultats du temps d'exécution

On peut constater que pour les modèles avec fautes où la contrainte niée est relativement simple (pas de quantificateurs), les temps d'exécution du solveur de contraintes sont assez proches. Par contre, dès que les contraintes deviennent complexes (comme c'est le cas pour C4, C5, C6 du composant applet, voir figure 6.3.1) on peut conclure que l'utilisation des modèles pré-remplis optimise vraiment le temps d'exécution nécessaire à ProB pour la résolution des contraintes. En effet, sans modèles pré-remplis ce temps est de l'ordre des minutes (on a fixé le délai à 10 minutes) alors que leur utilisation le réduit à quelques secondes seulement.

Résultat très important : piste d'une faille !

Une fois tous nos fichiers CAP invalides obtenus, on a entamé la dernière étape de notre approche : le chargement des fichiers moyennant le script qu'on a développé (Annexe C) et l'analyse des réponses de la carte. Les réponses qu'on a pu obtenir variaient entre un SW **6985** à l'étape *load* ou à l'étape *install for load*. Ce qui est normal pour une tentative de chargement d'un fichier CAP invalide dans la carte disposant du vérifieur de byte code embarqué.

Mais ce qui est important dans tout cela est le fait que la carte ne renvoyait que le code *load :6985*. Mais au moment où on à tenter de charger le fichier contenant une faute relative au composant applet, la carte a commencé à renvoyer le code *install for load :6985*. Et encore, elle a commencé à se comporter d'une façon " anormale " : des réponses différentes pour le chargement d'un même fichier, l'acceptation et le refus aléatoires de fichiers valides.

La faute qui a provoqué cette déstabilisation de la carte sous test concerne l'AID d'une applet. Et plus exactement, c'est le choix d'un AID d'applet qui soit différent au niveau du RID (5 premiers octets) du package contenant cette applet. La valeur du RID choisi est égale à 0.0.0.0.0. Alors que le RID du package est égal à A0.00.00.00.18. Certes, cette valeur spéciale n'a pas conduit à l'acceptation d'un fichier CAP invalide. En revanche, elle a déclenché un comportement assez étrange qui nous fait penser au fait qu'il est fort possible qu'il y ait quelque chose qui soit caché derrière cette valeur. Cette

piste de recherche est actuellement en cours d'exploration dans le but de découvrir une nouvelle faille de sécurité dans l'implémentation du vérifieur de structure qu'on a pu tester.

7.4 Conclusion

Ce chapitre a permis de présenter d'abord les choix techniques qui nous ont permis la mise en œuvre de notre approche d'analyse de vulnérabilité. Par la suite on a procédé à l'application de l'approche, sur une carte à puce disposant d'un vérifieur de byte code embarqué, cible de notre travail. Ceci dans le but d'avoir des résultats expérimentaux qui ont permis de mettre en avant l'apport de notre approche par rapport à un plan de test manuel à travers une comparaison basée sur deux métriques : le nombre de contraintes et le nombre de cas de test. Dans les deux cas, on a pu obtenir des résultats nettement meilleurs. De plus, on a montré l'apport de l'utilisation des modèles pré-remplis (l'optimisation proposée) à travers des résultats pratiques concernant le temps d'exécution du solveur de contraintes utilisé.

Conclusion et Perspectives

Le vérifieur de byte code embarqué est la première ligne de défense pour Java Card. En effet, c'est lui qui décide d'accepter ou de refuser le chargement d'une nouvelle application dans la carte à puce. De ce fait, toute faille découverte dans son implémentation peut remettre en cause toute la sécurité de la plateforme. Donc, une étude de la sécurité du vérifieur de byte code s'avère nécessaire car elle permettra de faire ressortir ses faiblesses en vue d'apporter les corrections adéquates pour maintenir le niveau de sécurité envisagé.

Le but de notre travail était de faire cette étude de sécurité à travers une analyse de vulnérabilité. Pour ce faire, on a proposé une nouvelle approche de génération de test de sécurité basée sur un modèle formel, en langage B, du vérifieur de structure et qui a été construit à partir de la spécification fournie par Oracle [Mic09c]. Ce modèle a été dérivé en appliquant successivement la négation des contraintes pour obtenir des modèles avec fautes. L'instanciation de ces modèles à l'aide d'un solveur de contraintes donne lieu aux cas de test abstraits. Par la suite, chacun de ces derniers a été transformé en un cas de test concret par une modification d'un fichier CAP valide. Chacun des cas de test concrets obtenus constitue un fichier CAP invalide à charger dans la carte moyennant une librairie destinée à cet effet. Enfin, il suffit d'analyser chaque réponse de la carte pour conclure sur la présence d'une vulnérabilité ou non.

Suite à l'implémentation de cette proposition d'approche, en combinant plusieurs outils existants, elle a été appliquée sur une vraie carte à puce Java Card disposant d'un vérifieur de byte code embarqué. Ceci dans le but d'obtenir des résultats expérimentaux qui ont servi à la fois à établir une comparaison avec un plan de test manuel et à faire ressortir l'apport de l'utilisation des modèles pré-remplis. Bien que nous ayons appliqué notre approche sur un fragment du modèle abstrait du vérifieur de structure, mais on peut déjà voir que nos résultats préliminaires sont encourageants :

– *Utilisation d'un modèle formel.* Ceci dans le but de modéliser le vérifieur de structure. Ce qui nous a permis de profiter de la force des fondements mathématiques des méthodes formelles.

- *Introduction des contraintes.* Nous constatons que la non prise en compte des contraintes revient à la génération de toutes les valeurs possibles pouvant être envoyées à la carte. Mais le problème avec une telle approche est le nombre de cas de test à générer. En outre, nous obtenons des fichiers acceptables par le vérifieur et d'autres qui ne le sont pas. Donc cette suite de test ne permet pas de tester uniquement les problèmes de sécurité. Ce qui est intéressant avec les contraintes est le fait qu'elles nous permettent de réduire considérablement le nombre de cas de test à générer car elles limitent les valeurs que peuvent prendre les constantes du modèle. Cependant, il reste encore beaucoup de cas de test. L'utilisation de la technique de test aux limites, *i.e.* tester les éléments extrêmes des ensembles,a permis de réduire encore ce nombre.
- *La négation des contraintes.* Le fait d'avoir une contrainte qui est fausse dans le modèle nous a permis de ne générer que des fichiers CAP invalides au regard de la spécification, ce qui correspond à notre objectif. Par conséquent l'oracle n'aura plus qu'à attendre que le vérifieur de structure indique que le fichier chargé soit valide pour conclure sur la présence d'une vulnérabilité. De plus, le fait d'avoir une seule faute à la fois permet une injection de faute mieux ciblée.
- *Utilisation des modèles pré-remplis.* Elle a permis de simplifier le travail du solveur de contraintes. Ceci se justifie par les résultats expérimentaux qu'on a obtenu pour le temps d'exécution du solveur avec et sans utilisation de ces modèles. En effet, ce temps a été largement optimisé, on est passé de quelques minutes à quelques secondes.
- *Comparaison avec le plan de test manuel.* Il a été démontré à travers les résultats obtenus que le taux moyen de réduction du nombre de contraintes est égal à 52.7 %. De plus, le taux moyen de réduction du nombre de cas de test est égal à 32.3 %.

Notre travail a permis d'apporter une approche originale pour l'étude de la sécurité du vérifieur de structure, problématique qui n'a pas été traitée auparavant. Du moment qu'on s'est basé uniquement sur la spécification de ce composant, l'approche est générique et peut être utilisée pour analyser la vulnérabilité de toute implémentation du vérifieur qui soit conforme à cette spécification (plateforme Java Card 3.0 *Classic Edition* et versions antérieures).

Les perspectives d'amélioration et de développement de ce travail sont multiples. En premier lieu, il s'agit de compléter le modèle abstrait du vérifieur de structure pour prendre en charge tous les composants du fichier CAP et avoir ainsi une couverture totale des fautes qui peuvent exister. Et par conséquent, pouvoir détecter toutes les vulnérabilités relatives aux différentes implémentations sous test.

Une amélioration possible de l'approche serait l'automatisation des différentes étapes. A priori, on voit que ceci est possible car le format de sortie de chaque étape peut être exploité dans l'étape suivante directement ou après un simple traitement (par exemple, utiliser un parser pour récupérer les valeurs des champs contenues dans le fichier texte correspondant à une instanciation d'un modèle avec faute et qui peut être obtenu dans ProB). Une telle automatisation permettrait de gagner encore du temps lors de la génération des jeux de tests de sécurité.

Troisième partie

Annexes

ANNEXE A

Quelques Notations du Langage B

Notation	Description	Notation	Description
!	quantificateur universel (quelque soit)	#	quantificateur existentiel (il existe)
%	lambda expression	&	conjonction (ET logique)
(	parenthèse ouvrante	)	parenthèse fermante
*	multiplication ou produit cartésien	**	puissance
+	addition	+->	fonction partielle
+-> >	surjection partielle	-	soustraction
- ->	fonction totale	- -> >	surjection totale
->	insertion en tête d'une suite	.	renommage ou séparateur de données utilisé avec certains opérateurs
..	intervalle	/	division entière
/ :	non-appartenance	/< :	non-inclusion
/< < :	non-inclusion stricte	/=	inégalité
:	appartenance	<	strictement inférieur ou délimiteur de fichier de définitions
<+	surcharge d'une relation	<->	ensemble des relations
<-	insertion en fin de suite	<- -	paramètres de sortie d'opération
< :	inclusion	< < :	inclusion stricte
< <|	soustraction sur le domaine	<=	inférieur ou égal

<=>	équivalence	<|	restriction sur le domaine
=	égalité	= =	définition
=>	implique	>	strictement supérieur ou délimiteur de fichier de définitions
>+>	injection partielle	>->	injection totale
>-> >	bijection totale	><	produit direct de relations
>=	supérieur ou égal		
END	terminateur des clauses ou des substitutions	FALSE	constante booléenne littérale "faux"
FIN	ensemble des sous-ensembles finis		
INT	ensemble des entiers relatifs concrets	INTEGER	ensemble des entiers relatifs
MAXINT	plus grand entier implémentable	MININT	plus petit entier implémentable
NAT	ensemble des entiers naturels concrets	NAT1	ensemble des entiers naturels non nuls concrets
NATURAL	ensemble des entiers naturels	NATURAL1	ensemble des entiers naturels non nuls
PI	produit quantifié d'entiers	POW	ensemble des sous-ensembles
POW1	ensemble des sous-ensembles non vides	SIGMA	somme quantifié
STRING	ensemble des chaînes de caractères	TRUE	constante booléenne littérale "vrai"
bool	conversion d'un prédicat en booléen	card	cardinal
dom	domaine d'une fonction	id	fonction identité
max	maximum d'un ensemble d'entiers	min	minimum d'un ensemble d'entiers
mod	modulo	not	négation (NON logique)
or	disjonction (OU logique)	pred	prédécesseur d'un entier
ran	codomaine d'une relation	succ	successeur
{	début d'ensemble	}	fin d'ensemble
|	barre verticale	|->	maplet
|>	restriction sur le codomaine	|> >	soustraction sur le codomaine

ANNEXE B

Spécification du composant Applet

Le contenu de cette annexe est directement extrait de la spécification d'Oracle [Mic09c], chapitre 6 portant sur le fichier CAP.

Applet Component

The Applet Component contains an entry for each of the applets defined in this package. Applets are defined by implementing a non-abstract subclass, direct or indirect, of the javacard.framework.Applet class. If no applets are defined, this component must not be present in this CAP file. The Applet Component is described by the following variable-length structure :

```
applet_component {
     u1 tag
     u2 size
     u1 count
       { u1 AID_length
         u1 AID[AID_length]
         u2 install_method_offset
       } applets[count]
}
```

The items in the applet-component structure are as follows :

– *tag*

The tag item has the value COMPONENT-Applet (3).

– *size*

The size item indicates the number of bytes in the applet-component structure, excluding the tag and size items. The value of the size item must be greater than zero.

– *count*

The count item indicates the number of applets defined in this package. The value of the count item must be greater than zero.

– *applets[]*

The applets item represents a table of variable-length structures each describing an applet defined in this package. The items in each entry of the applets table are defined as follows :

– *AID-length*

The AID-length item represents the number of bytes in the AID item. Valid values are between 5 and 16, inclusive.

– *AID[]*

The AID item represents the Java Card platform name of the applet. Each applet is assigned an AID conforming to the ISO 7816-5 standard (Section 4.2, "AID-based Naming" on page 4-3). The RID (first 5 bytes) of all of the applet AIDs must have the same value. In addition, the RID of each applet AIDs must have the same value as the RID of the package defined in this CAP file.

– *install-method-offset*

The value of the install-method-offset item must be a 16-bit offset into the info item of the Method Component (Section 6.9, "Method Component" on page 6-36). The item at that offset must be a method-info structure that represents the static install(byte[],short,byte) method of the applet. The install(byte[],short,byte) method must be defined in a class that extends the javacard.framework.applet class, directly or indirectly. The install(byte[],short,byte) method is called to initialize the applet.

ANNEXE C

Script de chargement des CAP

```
import fr.xlim.ssd.opal.library.SecLevel;
import fr.xlim.ssd.opal.library.GetStatusFileType;
import fr.xlim.ssd.opal.library.GetStatusResponseMode;
import fr.xlim.ssd.opal.library.SecurityDomain;
import fr.xlim.ssd.opal.library.commands.CommandsImplementationNotFound;
import fr.xlim.ssd.opal.library.params.CardConfig;
import fr.xlim.ssd.opal.library.params.CardConfigFactory;
import fr.xlim.ssd.opal.library.params.CardConfigNotFoundException;
import fr.xlim.ssd.opal.library.utilities.Conversion;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import javax.smartcardio.*;
import java.io.*;
import java.util.List;
import java.util.logging.Level;

public class Chargeur {
    //the logger
    private final static Logger logger = LoggerFactory.getLogger(Chargeur.class);
    //timeout to detect card presence
    private final static int TIMEOUT_CARD_PRESENT = 1000;
    private final static byte[] PACKAGE_ID = {(byte) 0x.., (byte) 0x.., ... };
    private final static byte[] APPLET_ID = {(byte) 0x.., (byte) 0x.., ... };
    //channel to the card
    private static CardChannel channel;
    private static CardConfig getCardChannel(int cardTerminalIndex, String transmissionProtocol) {
        TerminalFactory factory = TerminalFactory.getDefault();
        List<CardTerminal> terminals;
        try {
```

```
        terminals = factory.terminals().list(State.ALL);
    } catch (CardException ex) {
        logger.error("Cannot get list of terminals", ex);
        return null;
    }
    if (terminals.isEmpty()) {
        logger.error("No card terminal found");
        return null;
    } else {
        for (int i = 0; i < terminals.size(); i++) {
            logger.info("Card terminal found: " + i + " - " + terminals.get(i));
        }
    }
    if (terminals.size() == 1) {
        cardTerminalIndex = 0;
    } else if (terminals.size() < cardTerminalIndex) {
        logger.error("Card terminal index not available: " + cardTerminalIndex);
        return null;
    }
    logger.info("Card terminal selected: " + terminals.get(cardTerminalIndex));
    CardTerminal terminal = terminals.get(cardTerminalIndex);
    logger.info("Wait for card (during " + TIMEOUT_CARD_PRESENT + "ms)");
    boolean cardFound;
    try {
        cardFound = terminal.waitForCardPresent(TIMEOUT_CARD_PRESENT);
    } catch (CardException ex) {
        logger.error("Cannot detect card presence", ex);
        return null;
    }
    if (!cardFound) {
        logger.error("Card not found");
        return null;
    }
    logger.info("Connect to card with transmission protocol " + transmissionProtocol);
    Card card;
    try {
        card = terminal.connect(transmissionProtocol);
    } catch (CardException ex) {
        logger.error("Cannot connect to card", ex);
        return null;
    }
    logger.info("Card description: " + card);
```

```
        channel = card.getBasicChannel();
        ATR atr = card.getATR();
        logger.info("Card ATR: " + Conversion.arrayToHex(atr.getBytes()));
        try {
                return CardConfigFactory.getCardConfig("my_card_name");
        } catch (CardConfigNotFoundException ex) {
            logger.error(ex.getMessage());
        }
        return null;
    }
    //======================================================
    public static void main(String[ ] args) throws CardException, CardConfigNotFoundException,
CommandsImplementationNotFound,ClassNotFoundException,IOException {
        channel = null;
        SecLevel secLevel = SecLevel.NO_SECURITY_LEVEL;

        // get the card config and card channel, detection of t=0 or t=1 is automatic
        CardConfig cardConfig = getCardChannel(1, "*");
        if (channel == null) {
            logger.error("Cannot access to the card");
            System.exit(-1);
        }

        // select the security domain
        logger.info("Selecting Security Domain");
        SecurityDomain securityDomain = new SecurityDomain(cardConfig.getImplementation(),channel,
 cardConfig.getIssuerSecurityDomainAID());
        securityDomain.setOffCardKeys(cardConfig.getSCKeys());
        try {
            securityDomain.select();
        } catch (Exception ex) {
            java.util.logging.Logger.getLogger(Chargeur.class.getName()).log(Level.SEVERE, null, ex);
        }

        // Initialize Update
        logger.info(" Initialize Update");
        securityDomain.initializeUpdate(cardConfig.getDefaultInitUpdateP1(),cardConfig.
getDefaultInitUpdateP2(),cardConfig.getScpMode());

        // External Authenticate
        logger.info("External Authenticate");
        securityDomain.externalAuthenticate(secLevel);
```

```
        // Install Applet
        logger.info(" Installing Applet");
        logger.info("* Install For Load");
        securityDomain.installForLoad(PACKAGE_ID, null, null);
        File file = new File(cap_file_path);
        InputStream is = new FileInputStream(file);
        byte [] convertedBuffer = new byte[(int)file.length()];
        is.read(convertedBuffer);
        logger.info("* Loading file");
        securityDomain.load(convertedBuffer);
        logger.info("* Install for install");
        securityDomain.installForInstallAndMakeSelectable(
                PACKAGE_ID,
                APPLET_ID,
                APPLET_ID,
                Conversion.hexToArray("00"), null);

        // Deleting Applet
        logger.info("Deleting applet");
        securityDomain.deleteOnCardObj(APPLET_ID, false);

        // Deleting package
        logger.info("Deleting package");
        securityDomain.deleteOnCardObj(PACKAGE_ID, false);
    }
}
```

Bibliographie

[AAB+05] M. Aiguier, A. Arnould, C. Boin, Le Gall.P. et B. Marre : Testing from algebraic specifications : Test data set selection by unfolding axioms. *In Formal Approaches to Testing of Software (FATES'05)*, volume 3997, pages 203–217. LNCS, 2005.

[ABC+02] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet et M. Utting : BZ-TT : A tool-set for test generation from Z and B using contraint logic programming. *In Formal Approaches to Testing of Software, FATES 2002 workshop of CONCUR'02*, pages 105–120, 2002.

[ABF+03] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter et J.P. Seifert : Fault attacks on RSA with CRT : Concrete results and practical countermeasures. *Cryptographic Hardware and Embedded Systems-CHES 2002*, pages 81–95, 2003.

[ABM98] P.E. Ammann, P.E. Black et W. Majurski : Using Model Checking to Generate Tests from Specifications. *In ICFEM '98 : Proceedings of the Second IEEE International Conference on Formal Engineering Methods*, Washington, USA, 1998. IEEE Computer Society.

[Abr96] J.R. Abrial : *The B Book - Assigning Programs to Meanings.* Cambridge University Press, 1996.

[AG03] A. Appel et S. Govindavajhala : Using Memory Errors to Attack a Virtual Machine. *In IEEE Symposium on Security and Privacy*, 2003.

[AS02] R. Anderson et S. Skorobogatov : Optical Fault Induction Attacks. *In Workshop on Cryptographic Hardware and Embedded Systmes (CHES 2002)*, USA, 2002.

[BB96] M.R. Blackburn et R.D. Busser : T-VEC : A tool for developing critical systems. *In Compass'96 : Eleventh Annual Conference on Computer Assurance*, Gaithersburg, Maryland, 1996.

[BBFM99] P. Behm, P. Benoit, A. Faivre et J.M Meynadier : METEOR : A successful application of B in a large project. *In Proceedings of FM'99 : World Congress on Formal Methods*, pages 369–387, 1999.

[BBICL11] A. Bkakria, G. Bouffard, J. Iguchy-Cartigny et J.L. Lanet : Opal : an open-source global platform java library which includes the remote application management over http. *In e-Smart 2011*, Nice (France), September 2011.

[BCJM04] D. Beyer, A.J. Chlipala, R. Jhala et R. Majumdar : Generating Tests from Counterexamples. *In ICSE'04 : Proceedings of the 26th International Conference on Software Engineering*, pages 326–335, Washington, USA, 2004. IEEE Computer Society.

[BCR03] L. Burdy, L. Casset et A. Requet : Développement formel d'un vérifieur embarqué de byte-code java. *Techniques et sciences informatiques*, 22(1):33–60, 2003.

[Beh00] S. Behnia : *Test de modèles formels en B : cadre théorique et critères de couvertures.* Thèse de doctorat, Institut National Polytechnique de Toulouse, 2000.

[Ber91] G. Bernot : Testing against formal specifications : a theoretical view. *In* LNCS, éditeur : *Theory and Practice of Software Developement (TAPSOFT'91)*, volume 494, pages 99–119, 1991.

[Bes10] F. Bessayah : *Une Approche Complémentaire de Test de Robustesse Basée sur l'Injection de Fautes et le Test Passif.* Thèse de doctorat, Télécom & Management SudParis, 2010.

[BGM91] G. Bernot, M.C. Gaudel et B. Marre : Software testing based on formal specifications : a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.

[BHLS11] G. Bouffard, S. Hamadouche, J.L. Lanet et A. Savary : Vulnerability Analysis of a Smart Card Security Component through Test Suites Generation. *In The 6th IEEE/ACM International Workshop on Automation of Software Test AST'11 (ICSE 2011)*, Waikiki, Honolulu, Hawaï, 2011. (Article accepté mais participation annulée).

[BICL11] G. Bouffard, J. Iguchi-Cartigny et J.L. Lanet : Combined Software and Hardware Attacks on the Java Card Control Flow. *In CARDIS'11*, Leuven (Belgique), Septembre 2011.

[BK05] K. Berkenkötter et R. Kirner : Real-Time and Hybrid Systems Testing. *In Model-Based Testing of Reactive Systems*, volume 3472, pages 355–387. LNCS, Springer-Verlag, 2005.

[BL03] F. Bouquet et B. Legeard : Reification of executable test scripts in formal specification-based test generation : the Java Card transaction mechanism case study. *In* LNCS, éditeur : *Proceedings of the International Conference on Formal Methods Europe (FME'03)*, volume 2805, pages 778–795, Pisa, Italie, Septembre 2003. Springer Verlag.

[BLLP04] E. Bernard, B. Legeard, X. Luck et F. Peureux : Generation of test sequences from formal specifications : GSM 11.11 standard case-study. *Journal of Software Practice and Experience*, 34(10):915–948, 2004.

[BLPT04] F. Bouquet, B. Legeard, F. Peureux et E. Torreborre : Mastering Test Generation from Smart Card Software Formal Models. *In Procs. of the Int. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04)*, volume 3362 de *LNCS*, pages 70–85, Marseille, France, 2004. Springer.

[Boi07] C. Boin : *Une méthode de sélection de tests à partir de spécifications algébriques.* Thèse de Doctorat, Université d'Evry-Val d'Essonne, 2007.

[BS04] J. Bach et P. Shroeder : Pairwise Testing - A Best Practice That Isn't. *In Proceedings of 22nd Pacific Northwest Software Quality Conference*, pages 180–196, 2004.

[BTG10] G. Barbu, H. Thiebeauld et V. Guerin : Attacks on java card 3.0 combining fault and logical attacks. *In IFIP International Federation For Information Processing*, pages 148–163, 2010.

[Cas02] L. Casset : *Construction Correcte de Logiciels pour Carte à Puce : Développement formel d'un vérifieur embarqué de byte code Java Card à l'aide de la méthode B.* Thèse de doctorat, Université de la Méditerranée, 2002.

[CBR02] L. Casset, L. Burdy et A. Requet : Formal Development of an Embedded Verifier for JavaCard ByteCode. *In Proceedings of DSN'02*. IEEE Computer Society, 2002.

[CDFP97] D.M. Cohen, S.R. Dalal, M.L. Fredman et G.C. Patton : The AETG System : An Approach to Testing Based on Combinatiorial Design. *Software Engineering*, 23(7):437–444, 1997.

[CJRZ01] D. Clarke, T. Jéron, V. Rusu et E. Zinovieva : Automated test and oracle generation for smart card applications. *In Proceedings of the International Conference on Research in Smart Cards (e-Smart'01)*, volume 2140 de *LNCS*, pages 58–70, Cannes, France, Septembre 2001. Springer Verlag.

[CJRZ02] D. Clarke, T. Jéron, V. Rusu et E. Zinovieva : STG : A Symbolic Test Generation Tool. *In TACAS*, pages 470–475, 2002.

[Cle06] ClearSy : Méthode B, 2006.

[Cle09] ClearSy : *Manuel de Référence du Langage B.* France, Février 2009. Version 1.8.7.

[CLP04] S. Colin, B. Legeard et F. Peureux : Preamble Computation in Automated Test Case Generation using Constraint Logic Programming. *In Journal of Software Testing, Verification and Reliability*, volume 14, pages 213–235, 2004.

[Coh97] R. M. Cohen : Defensive Java Virtual Machine. Rapport technique Version 0.5 alpha, 1997.

[DBLW02] B. Den Boer, K. Lemke et G. Wicke : A DPA Attack against the Modular Reduction within a CRT Implementation of RSA. *In Cryptographic Hardware and Embedded Systems (CHES'2002)*, pages 228–243. Springer, 2002.

[DF93] J. Dick et A. Faivre : Automating the generation and sequencing of test cases from model-based specifications. *In FME'93 : Industrial-Strength Formal Methods*, pages 268–284. Springer, 1993.

[DG02] D. Deville et G. Grimaud : Building an "impossible" verifier on a Java Card. *In 2nd USENIX Workshop on Industrial Experiences with Systems Software*, Boston, USA, 2002.

[DGR01] D. Deville, G. Grimaud et A. Requet : Efficient representation of code verifier structures. 2001. A compléter.

[dHR04] L. deMoura, G. Hamon et J. Rushby : Generating Efficient Test Sets with a Model Checker. *In 2nd International Conference on Software Engineering and Formal Methods*, pages 261–270, Beijing, China, 2004. IEEE Computer Society.

[Duf03] G Dufay : *Vérification formelle de la plate-forme Java Card.* Thèse de doctorat, Université de Nice-Sophia Antipolis, 2003.

[DZ03] W. Dulz et F. Zhen : MaTeLo - Statistical Usage Testing by Annotated Sequence Diagrams, Markov Chains and TTCN-3. *In QSIC'03 : Proceedings of the Third International Conference on Quality Software*, pages 336–342. IEEE Computer Society, 2003.

[Eur10] EuroSmart : Courrier de la Monétique, Décembre 2010. `http://www.eurosmart.com/images/doc/Eurosmart-in-the-press/2010/courrier_monetique_dec2010.pdf`.

[Gau95] M.C. Gaudel : Testing can be formal. *In Theory and Practice of Software Developement (TAPSOFT'95)*, volume 915 de *LNCS*, pages 82–96, 1995.

[GBR98] A. Gotlieb, B. Botella et M. Rueher : Automatic Test Data Generation using Constraint Solving Techniques. *In Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, volume 2, pages 53–62. ACM SIGSOFT, 1998.

[Gil62] A. Gill : *Introduction to the theory of finite-state machines.* 1962.

[Gir07] C. Giraud : *Attaques de cryptosystèmes embarqués et contremesures associées.* Thèse de doctorat, Université de Versailles Saint-Quentin, 2007.

[GM93] M.J.C. Gordon et T.F. Melham : *Introduction to HOL : a theorem proving environment for higher order logic.* 1993.

[GT04] C. Giraud et H. Thiebeauld : A survey on fault attacks. *In CARDIS 2004*, 2004.

[HLM11] S. Hamadouche, J.L. Lanet et M. Mezghiche : Méthode d'Analyse de Vulnérabilité Appliquée à un Composant de Sécurité d'une Carte à Puce. *In Rencontres sur la Recherche en Informatique (R2I)*, Tizi-Ouzou, Algérie, Juin 2011.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring et M.B. Trakhtenbrot : STATEMATE : A Working Environment for the Development of Complex Reactive Systems. *Software Engineering*, 16(4):403–414, 1990.

[HM09] J. Hogenboom et W. Mostowski : Full Memory Read Attack on a Java Card. *In 4th Benelux Workshop on Information and System Security*, 2009.

[HNS97] S. Helke, T. Neustupny et T. Santen : Automating Test Case Generation from Z Specifications with Isabelle. *In ZUM*, pages 52–71, 1997.

[HP94] H.M Höercher et J. Peleska : The role of formal specification in software test. *In FME'94 Industrial Benefit of Formal Methods*, 1994.

[Hyp03] K. Hyppönen : Use of cryptographic codes for byte code verification in smart card environment. Mémoire de master, Université de Kuopio, 2003.

[ICL09] J. Iguchi-Cartigny et J.L. Lanet : Developing a Trojan applets in a smart card. *Journal in Computer Virology*, pages 1–9, 2009.

[IEE90] IEEE : *Standard glossary of software engineering terminologie*, 1990. IEEE Standard 610.12-1990.

[Ins99] European Telecommunications Standards Institute : *GSM 11-11 V7.2.0 Technical Specifications*, 1999.

[JJ05] C. Jard et T. Jéron : TGV : theory, principles and algorithms : A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *In Int. J. Softw. Tools Technol. Transf.*, volume 7, pages 297–315, 2005.

[JJK99] B. Jun, J. Jaffe et P. Kocher : Differential Power Analysis. *In Cryptology Conference on Advances in Cryptology*, pages 388–397. Springer, 1999.

[JMT08] J. Julliand, P.A. Masson et R. Tissot : Generating Tests from B Specifications and Test Purposes. *In ABZ 2008*, pages 139–152. Springer-Verlag, 2008.

[JMTB09] J. Julliand, P.A. Masson, R. Tissot et P.C. Bué : Generating tests from B specifications and dynamic selection criteria. *Formal Aspects of Computing*, 23:3–19, 2009.

[Jon90] C.B. Jones : *Systematic Software Development Using VDM*. Deuxième édition, 1990.

[Jus03] N. Jussien : Programmation par contraintes pour les technologies logicielles. Actes du colloque GEMSTIC, 2003.

[Kil73] G.A. Kildall : A unified approach to global program optimization. *In Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM Press, 1973.

[KLPU04] N. Kosmatov, B. Legeard, F. Peureux et M. Utting : Boundary Coverage Criteria for Test Generation from Formal Models. *In ISSRE*, pages 139–150, 2004.

[Koc96] P. Kocher : Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and other systems. *In International Cryptology Conference on Advances in Cryptology*. Springer-Verlag, 1996.

[Lan06] J.L. Lanet : Support de cours : Carte à puce, 2006.

[LB03a] M. Leuschel et M. Butler : ProB : A Model Checker for B. *In FME 2003 : FORMAL METHODS, LNCS 2805*, pages 855–874. Springer-Verlag, 2003.

[LB03b] M. Leuschel et M. Butler : The ProB animator and model checker for B – a tool description , 2003.

[Ler01] X. Leroy : On-Card Bytecode Verification for Java Card. *In International Conference on Research in Smart Cards, E-Smart 2001*, pages 150–164. Springer-Verlag, 2001.

[Ler02] X. Leroy : Bytecode verification on Java smart cards. *Software-Practice & Experience*, pages 319–340, 2002.

[Lon07] D. Longuet : *Test à partir de spécifications axiomatiques.* Thèse de Doctorat, Université d'Evry-Val d'Essonne, 2007.

[LP00] H. Lötzbeyer et A. Pretschner : Testing Concurrent Reactive Systems with Constraint Logic Programming. *In Proceedings of 2nd workshop on Rule-Based Constraint Reasoning and Programming*, Singapore, 2000.

[LP01] B. Legeard et F. Peureux : Generation of functional test sequences from B formal specifications - Presentation and industrial case-study. *In Proceedings of the 16th International Conference on Automated Software Engineering (ASE'01)*, pages 377–381, San Diego, USA, November 2001. IEEE Computer Society Press.

[LPU02] B. Legeard, F. Peureux et M. Utting : A comparison of the BTT and TTF testgeneration methods. *In Proceedings of the International Conference on Formal Specification and Development in Z and B (ZB'02)*, volume 2272 de *LNCS*, pages 309–329, Grenoble, France, January 2002. Springer Verlag.

[LT88] N. Lynch et M. Tuttle : An Introduction to Input/Output Automata. Rapport technique MIT/LCS/TM-373, MIT Laboratory for Computer Science, 1988.

[Man02] S. Mangard : A Simple Power-Analysis (SPA) Attack on Implementations of the AES Key Expansion. *In Information Security and Cryptology (ICISC'2002)*, pages 343–358. Springer, 2002.

[Mar91] B. Marre : Toward automatic test data set selection using algebraic specifications and logic programming. *In International Conference on Logic Programming (ICLP'91)*, pages 202–219, 1991.

[Mar95] B. Marre : Loft : a tool for assisting selection of test data sets from algebraic specifications. *In* LNCS, éditeur : *Theory and Practice of Software Developement (TAPSOFT'95)*, volume 915, pages 799–800, 1995.

[Mar01] H. Martin : *Une méthodologie de génération automatique des suites de tests pour applets Java Card.* Thèse de doctorat, Université de Lille 1, 2001.

[MDB00] H. Martin et L. Du Bousquet : Automatic test generation for Java Card applets. *In Proceedings of the Java Card Workshop*, Cannes, France, Septembre 2000.

[MDS99] T. Messerges, E. Dabbish et R. Sloan : Power Analysis Attacks of Modular Exponentiation in Smartcard. *In Cryptographic Hardware and Embedded Systems (CHES'99)*, pages 144–157. Springer, 1999.

[Mic08] Sun Microsystems : The Java Card™3 Platforme, Août 2008. White paper.

[Mic09a] Sun Microsystems : *Java Card™Platform, Version 3.0.1 Classic Edition : Application Programming Interface.* Sun Microsystems, 2009.

[Mic09b] Sun Microsystems : *Java Card™Platform, Version 3.0.1 Classic Edition : Runtime Environment Specification.* Sun Microsystems, 2009.

[Mic09c] Sun Microsystems : *Java Card™Platform, Version 3.0.1 Classic Edition : Virtual Machine Specification*. Sun Microsystems, 2009.

[Mic09d] Sun Microsystems : *Java Card™Platform, Version 3.0.1 Connected Edition : Application Programming Interface*. Sun Microsystems, 2009.

[Mic09e] Sun Microsystems : *Java Card™Platform, Version 3.0.1 Connected Edition : Runtime Environment Specification*. Sun Microsystems, 2009.

[Mic09f] Sun Microsystems : *Java Card™Platform, Version 3.0.1 Connected Edition : Servlet Specification*. Sun Microsystems, 2009.

[Mic09g] Sun Microsystems : *Java Card™Platform, Version 3.0.1 Connected Edition : Virtual Machine Specification*. Sun Microsystems, 2009.

[MOG01] C. Mourtel, F. Olivier et K. Gandol : ElectroMagnetic Analysis : Concrete Results. *In CHES'2001*, 2001.

[MP08] W. Mostowski et E. Poll : Malicious code on Java Card smartcards : Attacks and countermeasures. *Smart Card Research and Advanced Applications*, pages 1–16, 2008.

[Mui01] J.A. Muir : Techniques of Side Channel Cryptanalysis. Mémoire de master, Université de Waterloo, Ontario, Canada, 2001.

[Mye79] G.J. Myers : *The Art of Software Testing*. New York, USA, 1979.

[NL97] G.C. Necula et P. Lee : Proof-Carrying Code. *In Conference Record of POPL'97 : The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, Janvier 1997.

[NSIcL09] A.C. Noubissi, A.A. Séré, J. Iguchi-cartigny et J.L. Lanet : Cartes à puce : Attaques et contremesures. *In MajecSTIC 2009*, 2009.

[Osw02] E. Oswald : Enhancing Simple Power-Analysis Attacks on Elliptic Curve Cryptosystems. *In Cryptographic Hardware and Embedded Systems (CHES'2002)*, pages 82–97. Springer, 2002.

[PL01] A. Pretschner et H. Lötzbeyer : Model Based Testing with Constraint Logic Programming : First Results and Challenges. *In Proceedings of 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing and Verification (WAPATV)*, pages 1–9, Toronto, 2001.

[PPS+03] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel et K. Scholl : Model-based test case generation for smart cards. *In Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03)*, volume 80, Trondheim, Norway, Juin 2003. Elsevier.

[PPW+05] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, Zölch et T. R. Stauner : One evaluation of modelbased testing and its automation. *In Proc. ICSE'05*, pages 392–401, 2005.

[Pro03] S.J. Prowell : JUMBL : A Tool for Model-Based Statistical Testing. *In Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03)*, 2003.

[QS02] J.-J. Quisquater et D. Samyde : Eddy Current for Magnetic Analysis with Active Sensor. *In E-Smart 2002*, 2002.

[RF00] M. Rivoire et J. Ferrier : *Matlab simulink stateflow*. 2000.

[RJB99] J. Rumbaugh, I. Jacobson et G. Booch : *The Unified Modeling Language Reference Manual*. 1999.

[RR98] E. Rose et K. Rose : Lightweight bytecode verification. *In Workshop on Fundamental Underpinnings of Java, OOPSLA'98 Workshop*, Vancouver, Canada, Octobre 1998.

[SA03] S.P. Skorobogatov et R.J. Anderson : Optical fault induction attacks. pages 2–12, 2003.

[SFL11a] A. Savary, M. Frapier et J.-L. Lanet : Automatic generation of vulnerability test suite for the Java Card verifier. *In E-smart 11*, Sophia Antipolis (France), Septembre 2011.

[SFL11b] A. Savary, M. Frapier et J.L. Lanet : Génération de tests de vulnérabilité pour vérifieur du typage de Java Card. *In Sar-SSI 2011*, La Rochelle (France), Mai 2011.

[SH07] J.M. Schmidt et M. Hutter : Optical and EM Fault-Attacks on CRT-based RSA : Concrete results. *In Proceedings of the Austrochip*, pages 61–67. Citeseer, 2007.

[SKGH97] M. Schmitt, B. Koch, J. Grabowski et D. Hogrefe : Autolink : A tool for the automatic and semi-automatic test generation. *In Proceedings of the 7th GI/ITG Technical Meeting on Formal Description Techniques for Distributed Systems*, Berlin, 1997.

[SQ01] D. Samyde et J.J. Quisquater : ElectroMagnetic Analysis (EMA) : Measures and Countermeasures for Smart Cards. *In E-smart*, 2001.

[Sér10] A.A. Séré : *Tissage de contremesures pour machines virtuelles embarquées*. Thèse de doctorat, Université de Limoge, 2010.

[Ta10] A. Tria et al. : When Clocks Fail : On Critical Paths and Clock Faults. *In Smart Card Research and Advanced Application*, pages 182–193, 2010.

[Tav07] C. Tavernier : *Les Cartes à Puce : Théorie et mise en œuvre*. Dunod, Paris, 2e édition, 2007.

[TB02] J. Tretmans et E. Brinksma : Automated model based testing. *In Progress 2002 - 3rd Workshop on Embedded Systems*, pages 246–255, 2002.

[Tre96] J. Tretmans : Conformance testing with labelled transition systems : implementation relations and test generation. *Comput.Netw.*, 29(1):49–79, 1996.

[Tre04] J. Tretmans : Model-based testing : Property checking for real. *In International Workshop for Construction and Analysis of Safe Secure, and Interoperable Smart Devices*, 2004.

[UL06] M. Utting et B. Legeard : *Practical Model-Based Testing : A Tools Approach*. Morgan Kaufmann Publishers Inc, San Francisco, USA, 2006.

[UPL06] M. Utting, A. Pretschner et B. Legeard : A Taxonomy of Model-Based Testing. Rapport technique, Université de Waikato, 2006.

[VA98] L. Van Aertryck : *Une méthode est un outil pour l'aide à la génération de jeux de tests de logiciels*. Thèse de doctorat, Université de Rennes 1, 1998.

[VAJ03] L. Van Aertryck et T. Jensen : UML-CASTING : Test synthesis from UML models using constraint. *In AFADL'2003 (Approches Formelles dans l'Assistance au Développement de Logiciel)*, Janvier 2003.

[Ver06] O. Vertanen : Java Type Confusion and Fault Attacks. *Fault Diagnosis and Tolerance in Cryptography*, pages 237–251, 2006.

[VF10] E. Vetillard et A. Ferrari : Combined Attacks and Countermeasures. *Smart Card Research and Advanced Application*, pages 133–147, 2010.

[Wit02] M. Witteman : Advances in smartcard security. *Information Security Bulletin*, (July):11–22, 2002.

[Wit03] M. Witteman : Java card security. *Information Security Bulletin*, 8(October):291–298, 2003.

[WNBE+04] C. Whelan, D. Naccache, H. Bar-El, H. Choukri et M. Tunstall : The Sorcerer's Apprentice Guide to Fault Attacks. *In Workshop on Fault Detection and Tolerance in Cryptography*, Italy, 2004.

[Wor92] J.B. Wordsworth : *Software Development with Z*. Addison Wesley, 1992.

[Zhi00] C. Zhiqun : *Java Card technologie for smart cards : Architecture and Programmer's Guide*. Addison-Wesley, 2000.