

N° d'ordre : 2930

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET
D'INFORMATIQUE

Par **Damien SAUVERON**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

**Étude et réalisation d'un environnement d'expérimentation et de
modélisation pour la technologie Java CardTM.
Application à la sécurité.**

Soutenue le : 3 décembre 2004

Après avis des rapporteurs :

Isabelle Attali	Directrice de Recherche INRIA
Jean-Pierre Borel	Professeur des Universités
Pierre Dusart	Maître de Conférences

Devant la commission d'examen composée de :

Isabelle Attali	Directrice de Recherche INRIA	Rapporteur
Jean-Pierre Borel	Professeur des Universités	Rapporteur
Richard Castanet	Professeur des Universités	Président
Serge Chaumette	Professeur des Universités	Directeur de thèse
Pierre Dusart	Maître de Conférences	Rapporteur
Alain Griffault	Maître de Conférences	Invité
Jean-Pierre Lacoustille	Ingénieur	Examineur

Remerciements

Je tiens à exprimer toute ma reconnaissance aux membres du jury pour avoir accepté d'assister à la présentation de mes travaux. En particulier, je prie les rapporteurs d'accepter ma profonde gratitude. Ce mémoire rassemble le fruit de trois années de réflexion et je les remercie pour leurs commentaires pertinents et constructifs.

Mes remerciements sont aussi à l'attention Serge Chaumette, mon directeur de thèse. L'autonomie qu'il a su me laisser et les responsabilités qu'il m'a confié ont rendu mon travail des plus agréables. Je souhaite également citer ici les membres de l'équipe Systèmes et Objets Distribués que j'ai cotoyé : Pascal Grange, Iban Hatchondo, Achraf Karray et Pierre Vignéras.

Je suis aussi très reconnaissant envers l'ANRT et SERMA Technologies pour m'avoir permis de mener à bien cette thèse. Je remercie également tous mes collègues du CESTI de SERMA Technologies pour la bonne ambiance dans laquelle nous avons travaillé durant trois ans.

Enfin, mes remerciements vont vers ma famille sans qui je n'en serai pas là. Je pense en particulier à ma grand-mère, Mamie Margot, qui m'a offert mon premier ordinateur, à mon amie, Anaïs, qui m'a soutenu lors de cette dernière année difficile et à ma mère que j'aime très fort. C'est à elle que je dédie ce mémoire de thèse car plus que tout le monde, elle m'a permis de poursuivre les études qui me plaisaient.

Table des matières

Remerciements	i
Introduction	9
0.1 Précision du domaine	9
0.1.1 Contexte : le développement sécurisé sur cartes à puce	9
0.1.2 La problématique : les technologies multi-applicatives des cartes à puce	9
0.2 Cadre de travail	11
0.3 Un environnement libre pour Java Card	11
0.3.1 État de l’art	12
0.3.2 Les JCatools	12
0.3.3 Notre contribution : la <i>pré-persistence</i>	12
0.3.4 Perspectives	13
0.4 Les problèmes de sécurité des cartes multi-applicatives	13
0.4.1 État de l’art	13
0.4.2 Notre contribution : la mise en évidence de nouveaux problèmes . . .	13
0.4.3 Perspectives	14
0.5 Le projet Java Card Grid	14
0.5.1 État de l’art	14
0.5.2 Notre contribution : la preuve de concept	15
0.5.3 Perspectives	15
0.6 Organisation de la thèse	15
I Présentation des standards	17
1 La carte à puce	19
1.1 Qu’est ce qu’une carte à puce ?	19

1.2	Historique	19
1.3	Les différents types de cartes à puce	20
1.3.1	La carte à mémoire	20
1.3.2	La carte à microprocesseur	20
1.4	Les différents types de mémoire	22
1.4.1	La mémoire programmée en usine	22
1.4.2	La mémoire volatile	22
1.4.3	La mémoire persistante	23
1.4.4	Résumé	24
1.5	Les communications	24
1.5.1	La carte à contact	24
1.5.2	La carte sans contact	25
1.5.3	La carte dual-interface	27
1.5.4	La pile de communication	27
1.6	La fabrication et le cycle de vie de la carte	30
1.6.1	Les acteurs	30
1.6.2	La fabrication	30
1.6.3	Le cycle de vie de la carte	32
1.7	Les sécurités	32
1.7.1	Au niveau physique	33
1.7.2	Au niveau matériel	34
1.7.3	Au niveau logiciel	35
1.7.4	Au niveau de l'environnement	36
1.8	Les applications de la carte à puce	36
1.9	Les cartes multi-applicatives	37
1.9.1	MULTOS	38
1.9.2	Windows for Smart Cards de Microsoft	40
1.9.3	The ZeitControl BasicCard	42
1.9.4	Smartcard.NET	42
2	La technologie Java Card	45
2.1	Présentation	45
2.2	Historique	45
2.3	Les avantages de la technologie Java Card	47
2.4	L'architecture	48

2.4.1	Le langage, un sous-ensemble du langage Java	49
2.4.2	La machine virtuelle : JCVM	52
2.4.3	L'environnement d'exécution : JCRE	57
2.4.4	Les APIs	59
2.4.5	Les <i>applets</i>	62
2.4.6	Les objets Java Card	65
2.5	Les spécificités de Java Card	65
2.5.1	Cycles de vie	65
2.5.2	La nature des objets Java Card	67
2.5.3	Atomicité et transaction	69
2.5.4	Gestion de la mémoire	71
2.6	Les sécurités	72
2.6.1	Le vérifieur	72
2.6.2	Le pare-feu	74
2.7	Les inconvénients de la technologie Java Card	78
2.7.1	Les problèmes de sémantique	78
2.7.2	Les problèmes de la politique de développement	81
2.7.3	Les problèmes d'intégration	81
2.7.4	Les problèmes de sécurité des applications	82
2.8	Les différentes recherches sur Java Card	82
3	GlobalPlatform	87
3.1	Quelques problématiques de la multi-application	87
3.2	Présentation	88
3.3	L'architecture GlobalPlatform	89
3.3.1	L'environnement d'exécution (<i>RTE</i>)	90
3.3.2	Le gestionnaire de la carte (<i>Card Manager</i>)	90
3.3.3	Les domaines de sécurité (<i>Security Domains</i>)	92
3.3.4	Les APIs GlobalPlatform	92
3.4	Les mécanismes de sécurité	93
3.4.1	La sécurité des communications	93
3.4.2	La signature des applications	96
3.4.3	La gestion de la vérification du porteur de carte (CVM)	96
3.5	Le verrou psychologique versus le verrou technologique	97
4	PC/SC : communication entre le PC et la carte	99

4.1	Présentation	99
4.2	L'architecture	100
4.3	Les différentes implantations	102
4.3.1	Microsoft PC/SC	102
4.3.2	<code>pcsc-lite</code>	104
4.4	Quelques applications	105
4.4.1	Les wrappers	105
4.4.2	JPC/SC vs OCFPCSC	106
4.4.3	MuscleCard	107
5	La certification	111
5.1	Le schéma français	113
5.2	L'évaluation selon les Critères Communs	115
5.2.1	Les Critères Communs	115
5.2.2	Les concepts	118
5.2.3	L'évaluation	120
5.3	Conclusion	125
II	Un environnement libre pour Java Card	127
6	Contexte	129
6.1	Les problèmes d'évaluation apportés par Java Card	129
6.2	Le projet <i>Sécurité Java Card</i>	131
6.2.1	Présentation	131
6.2.2	Déroulement	132
6.2.3	Les choix et quelques résultats	132
6.3	Les outils existants au début du projet	133
6.4	Les outils existants aujourd'hui	134
7	Les outils de la suite JCatools	137
7.1	JCat Emulator	137
7.1.1	Fonctionnalités	138
7.1.2	Utilisations possibles	138
7.1.3	Limitations	139
7.1.4	Aperçu de JCat Emulator	139
7.1.5	Travaux en cours et améliorations futures	148

7.2	JCat View	149
7.3	JCat Converter	149
7.4	Les autres JCatools	150
7.5	Les problèmes de licence	150
7.6	L'architecture des JCatools	151
7.6.1	L'extensibilité	153
7.6.2	La sécurité	155
8	Résultats	159
8.1	Bilan de la suite JCatools	159
8.2	Interprétations des spécifications	160
8.2.1	Le problème de la définition du tas	160
8.2.2	Le problème du contenu du tas	161
8.2.3	Le problème de la localisation du tas	163
8.2.4	Pré-persistence et organisation du tas	163
8.2.5	Travaux connexes	167
8.2.6	Conclusion et perspectives	168
III	Les problèmes de sécurité des cartes multi-applicatives ouvertes	169
9	Les attaques classiques	171
9.1	Les attaques non-invasives	171
9.1.1	Les attaques hors conditions opérationnelles	171
9.1.2	Les attaques par rayonnement	172
9.1.3	Les attaques par injection de fautes	172
9.1.4	Les attaques par canaux cachés	173
9.1.5	Les attaques logicielles	182
9.1.6	Les attaques par erreur	183
9.2	Les attaques invasives	183
9.2.1	Le <i>microprobing</i>	183
9.2.2	La modification de circuit	183
9.2.3	La rétro-conception matérielle	186
10	Les attaques internes	187
10.1	L'identification de service	187
10.2	La collecte d'information	187

10.3	Les attaques contre la machine virtuelle, le pare-feu, les APIs	188
11	De nouvelles classes d'attaques	191
11.1	Les méthodes d'aide à l'identification physique	191
11.1.1	La méthode à base de <i>glitches</i>	191
11.1.2	Méthode basée sur la répétition de motifs	196
11.1.3	La méthode hybride utilisant les <i>glitches</i> et les motifs	197
11.2	Les applications des méthodes d'aide à l'identification physique	197
11.2.1	Les attaques par couplage	198
11.2.2	Les attaques physiques	198
11.3	Expérimentations	198
11.4	Travaux connexes	199
11.5	Conclusion et perspectives	200
11.6	Méthodologie pour modifier un fichier CAP	200
IV	Le projet Java Card Grid	203
12	Contexte	205
12.1	Objectifs	205
12.2	Les différentes grilles	206
12.2.1	Modèle client-serveur	206
12.2.2	Modèle d'égal à égal	208
12.3	Problèmes de sécurité sur les grilles	209
12.3.1	Sécurité du système vis à vis des applications et des infrastructures de calcul	209
12.3.2	Confidentialité du code de l'application	209
12.3.3	Sécurité de l'exécution de l'application	210
12.3.4	Sécurité des communications	211
12.4	Exemple d'un système de sécurité existant	211
13	Le projet Java Card Grid	213
13.1	Présentation	213
13.2	Infrastructure	214
13.2.1	Infrastructure matérielle	214
13.2.2	Infrastructure logicielle	215
13.3	Les défis	217

13.3.1	L'appel de méthodes à distance	217
13.3.2	La proactivité des cartes	218
13.3.3	Le canal sécurisé	222
13.4	Les applications de démonstration	227
13.4.1	La fractale Mandelbrot	228
13.4.2	FBI – Air France	235
13.5	Bilan	241
13.5.1	Défis résolus	241
13.5.2	Défis partiellement résolus	242
13.5.3	Défis à résoudre	242
13.6	Travaux connexes	243
13.7	Partenaires	244
14	Perspectives	245
14.1	Une plate-forme de confiance pour les TMCs	245
14.1.1	Les besoins explicites	246
14.1.2	Les besoins implicites	247
14.2	Les micro-services	247
14.2.1	Les micro-services dans le cadre des besoins explicites	247
14.2.2	Les micro-services dans le cadre des besoins implicites	248
14.2.3	D'autres micro-services	249
	Conclusion	251
	Glossaire	253
	Bibliographie	255

Table des figures

1	Un microprocesseur de carte à puce (source : Anonyme).	21
2	Le micromodule.	25
3	La carte à contact (source : Gemplus).	26
4	La carte sans contact (source : Gemplus).	26
5	Pile de protocoles de communication entre le lecteur et la carte.	28
6	Exemple de puce observée en microscopie optique (source : SERMA Technologies).	31
7	Coupe d'une puce observée en microscopie électronique (source : SERMA Technologies).	32
8	Grille de protection en microscopie optique (source : SERMA Technologies).	35
9	Architecture d'une carte à puce multi-applicative.	38
10	Architecture de MULTOS.	39
11	Architecture de Windows for Smart Card.	41
12	Architecture de la Smartcard.NET.	43
13	Architecture de la Java Card.	48
14	La machine virtuelle Java Card (JCVM).	52
15	Conversion d'un paquetage.	55
16	Installateur Java Card et programme d'installation hors carte.	57
17	L'architecture du système dans la carte.	58
18	Les étapes du développement d'une <i>applet</i> Java Card.	64
19	Communication APDU.	68
20	Le processus de vérification.	73
21	Le pare-feu Java Card.	75
22	Illustration du changement de contexte.	76
23	Architecture d'une carte GlobalPlatform (source : GlobalPlatform).	90
24	Procédure d'authentification mutuelle.	94

25	Procédure d'authentification mutuelle d'une application avec l'extérieur. . .	95
26	L'architecture de PC/SC 2.0.	100
27	L'architecture CryptoAPI.	104
28	L'architecture MuscleCard.	108
29	Déroulement de l'évaluation.	112
30	Rôles des différents intervenants et échanges d'information.	113
31	Concepts de sécurité et relations.	118
32	Concepts de l'évaluation et relations.	119
33	Contenu d'une ST et d'un PP.	121
34	Modèle de développement d'une TOE.	122
35	L'assurance des Critères Communs.	123
36	Les exigences d'assurance des Critères Communs.	124
37	Les exigences d'assurance des Critères Communs pour le niveau EAL1. . .	124
38	L'échelle d'assurance des Critères Communs.	125
39	Vue globale de l'émulateur.	140
40	Les différentes fonctions de la barre d'outils.	141
41	Vue de l'exécution en mode débogage.	142
42	Vue de l'exécution après un arrachage virtuel.	143
43	Vue d'un objet.	144
44	Édition des valeurs de l'objet.	145
45	Vue d'un objet point d'entrée temporaire.	146
46	Vue d'un objet point d'entrée permanent.	146
47	Vue d'un tableau global transient de type CLEAR_ON_RESET.	147
48	Présentation des statistiques sur les <i>bytecodes</i>	147
49	Fenêtre des paramètres d'impression.	148
50	Vue de JCat View.	149
51	L'architecture de JCat Emulator.	151
52	Architecture pour l'observation et la modification de la mémoire.	152
53	Extensibilité de l'architecture.	152
54	Factory de création des différents lecteurs de composant.	154
55	Factory de création des différents types de tableaux et diagramme UML des objets.	155
56	Le problème du contenu du tas.	162

57	Le problème de l'étendue du tas.	164
58	Le problème global de l'étendue et du contenu du tas.	164
59	La solution n°1 : la solution sans la pré-persistence.	165
60	La solution n°2 : la solution avec l'espace pré-persistant en EEPROM. . .	166
61	La solution n°3 : la solution avec l'espace pré-persistant en RAM.	167
62	La solution n°4 : la solution avec l'espace pré-persistant en RAM et en EEPROM.	168
63	Une station d'étude de la consommation en courant (source : SERMA Technologies).	175
64	Schéma de câblage de l'oscilloscope afin de pouvoir observer la consommation en courant (source : SERMA Technologies).	175
65	Schéma de câblage pour pouvoir observer la consommation en courant aux bornes de la puce (source : SERMA Technologies).	176
66	La consommation de différentes instructions (source : SERMA Technologies).176	
67	La signature en courant d'un chiffrement DES (source : SERMA Technologies).	176
68	La consommation des opérations modulaires (source : SERMA Technologies).177	
69	La consommation de l'algorithme RSA présenté (source : SERMA Technologies).	177
70	Dépendance de la consommation en courant de l'instruction I (source : SERMA Technologies).	178
71	DPA effectuée avec succès (source : SERMA Technologies).	179
72	DPA ayant échoué (source : SERMA Technologies).	179
73	Dispositif de cartographie électromagnétique (source : SERMA Technologies).	180
74	Cartographie des valeurs maximales pour l'algorithme 1 (source : SERMA Technologies).	180
75	Cartographie des valeurs moyennes pour l'algorithme 1 (source : SERMA Technologies).	180
76	Cartographie des valeurs maximales pour l'algorithme 2 (source : SERMA Technologies).	181
77	Cartographie des valeurs moyennes pour l'algorithme 2 (source : SERMA Technologies).	181
78	Consommation en courant de l'instruction 1 (source : SERMA Technologies).182	
79	Consommation en courant de l'instruction 2 (source : SERMA Technologies).182	
80	Consommation en courant de l'instruction 1 et instruction 2 (source : SERMA Technologies).	182

81	Le schéma d'un FIB (source : SERMA Technologies).	184
82	Un circuit modifié au FIB et sa schématique (source : SERMA Technologies).184	
83	Un plot de micro-probing réalisé à l'aide du FIB (source : SERMA Technologies).	185
84	Un circuit sur lequel SERMA Technologies a gravé son nom au FIB (source : SERMA Technologies).	185
85	Trace d'une exécution normale vue au travers des différentes couches. . . .	192
86	Trace d'une exécution <i>glitchée</i> vue au travers des différentes couches. . . .	192
87	Trace d'une exécution de multiples motifs vue au travers des différentes couches.	196
88	Trace d'une exécution <i>glitchée</i> de multiples motifs vue au travers des différentes couches.	197
89	Une solution pour le calcul avec des Java Cards.	214
90	La grille de Java Cards.	215
91	Infrastructure logicielle.	216
92	La solution PC/SC.	216
93	Mécanisme d'invocation d'une méthode.	218
94	Modèle d'exécution d'une carte proactive.	219
95	Modèle d'exécution classique d'une Java Card.	220
96	Invocation de méthode depuis une autre carte.	221
97	Détails de l'invocation de méthode depuis une autre carte.	222
98	Procédure d'authentification mutuelle avec chiffrement du <i>Host Challenge</i> . 225	
99	Procédure d'authentification mutuelle avec l'utilisation d'un certificat. . .	226
100	Une illustration de l'algorithme utilisé dans DJFractal.	230
101	Architecture de l'application de calcul de la fractale de Mandelbrot. . . .	233
102	Capture de la fractale de Mandelbrot calculée sur les cartes.	233
103	Tests sans canal sécurisé.	234
104	Saisie des informations sur les passagers.	237
105	Gestion des passagers sur la carte.	238
106	Partage de données.	239
107	Saisie de critères de recherche.	239
108	Diffusion de la requête d'analyse aux <i>applets</i> <i>CheckPassengers</i>	240
109	Retour des résultats.	241

Liste des tableaux

1	Caractéristiques des différents types de mémoire.	24
2	Structure d'une commande APDU.	29
3	Structure d'une réponse APDU.	29
4	Les différents échanges de commande et de réponse APDU.	30
5	Caractéristiques Java supportées, optionnelles et non supportées	49
6	Format de fichier d'un composant du packaging	53
7	Identifiant d'application : AID	63
8	Réponse des JCatools aux objectifs de SERMA Technologies.	137
9	<i>Benchmark</i> sans canal sécurisé.	234

Table des listings

2.1	Différences sémantiques dues à la persistance.	79
2.2	Différences sémantiques dues au pare-feu.	84
2.3	Une méthode pour le partage d'objet inter- <i>applet</i>	85
7.1	Le <i>factory design pattern</i> utilisé pour le lecteur de fichier CAP	154
7.2	Implantation JCatools de la classe OwnerPIN	156
7.3	Code d'une application officielle.	157
8.1	Tableau transient d'objets.	166
10.1	La recherche des services cryptographiques.	188
10.2	Le code pour scanner toutes les applications et essayer de récupérer une interface shareable.	189
10.3	Le code pour balayer toutes les références.	189
11.1	Chiffrement entouré par des <i>glitches</i> de waitExtension()	193
11.2	Chiffrement entouré par des <i>glitches</i> de donnée.	193
11.3	<i>Bytecodes</i> générés pour l'invocation du chiffrement entouré par les <i>glitches</i> . .	194
11.4	Amélioration de la précision de l'encadrement.	194
11.5	La méthode des motifs.	196
11.6	Exemple de code utilisant la méthode hybride.	197
11.7	Code Java factice.	200
11.8	<i>Bytecodes</i> générés par le code factice.	201
11.9	Motif à observer entouré par des <i>glitches</i>	201
13.1	La reprise de code.	223

Introduction

0.1 Précision du domaine

0.1.1 Contexte : le développement sécurisé sur cartes à puce

Dans le domaine des cartes à puce, la spécification et le développement des systèmes d'exploitation embarqués sont toujours des phases délicates qui nécessitent la mise en commun de compétences fortes en sécurité informatique, en électronique mais également en cryptologie. En effet les travaux menés lors de la dernière décennie ont montré qu'il était possible de réussir des attaques matérielles sur les cartes à puce avec un équipement parfois rudimentaire – par exemple en exploitant les canaux cachés. Dès lors, la fabrication de circuits toujours plus sécurisés a été un préalable indispensable à la conception d'architectures matérielles et logicielles totalement sûres. Par ailleurs la sécurité n'étant pas de la seule responsabilité de la plate-forme, les applications destinées à être embarquées sur les cartes (originellement mono-applicatives fermées) devaient impérativement utiliser les services de sécurité offerts par la plate-forme sous-jacente afin de ne pas mettre en péril leurs propres biens ou ceux de la plate-forme. C'est ainsi que devant l'importance des enjeux économiques des marchés exploitant la carte à puce, les gouvernements ont mis en place des schémas de certification sécuritaire des produits des technologies de l'information (TI) afin d'établir la sécurité des produits certifiés. Dans ce cadre, des tiers indépendants sont chargés de conduire des évaluations sécuritaires de produits TI en examinant le code du produit, les documentations de développement et les éléments de preuve nécessaires afin de s'assurer que le produit du fabricant fait bien ce qu'il prétend faire et de façon efficace. Au terme de ces processus de développement et de certification, la carte à puce et l'application qu'elle embarque constituent en principe l'un des périphériques les plus sécurisés du marché.

0.1.2 La problématique : les technologies multi-applicatives des cartes à puce

Si ces étapes permettent d'assurer l'obtention d'un produit possédant un très haut degré de sécurité, c'est au prix d'un coût élevé aussi bien pour la fabrication du produit (*i.e.* puce sécurisée, OS sécurisé, application sécurisée) que pour l'évaluation et la certification de ces différents éléments. De plus, le processus de masquage est souvent très long et ne permet pas de répondre efficacement aux besoins extrêmement volatiles des marchés. Ainsi, afin de réduire ces coûts, mais aussi afin d'avoir une plus grande flexibilité et donc d'être plus

réactif aux demandes des industriels utilisant les cartes, des technologies de cartes multi-applicatives ont été proposées. Ces technologies devraient permettre une réduction des coûts à plusieurs niveaux :

- lors du développement, grâce principalement à l'utilisation de langages de programmation de haut niveau qui simplifient considérablement la mise au point des applications, mais également grâce à l'utilisation d'une plate-forme commune comme support d'exécution d'applications pouvant être de natures très diverses ;
- lors de l'évaluation et de la certification, puisqu'il suffit théoriquement d'évaluer et de certifier une seule fois la plate-forme puis au coup par coup les applications qu'elle devra embarquer, plutôt que d'évaluer l'ensemble à chaque fois qu'on veut installer une nouvelle application.

Ces technologies répondent aux besoins de flexibilité et de réactivité en fournissant le dynamisme qui manquait jusqu'alors aux cartes à puce classiques, en autorisant le chargement d'applications même après la phase de fabrication, et en permettant aux fournisseurs d'applications de faire jouer la concurrence entre les divers fabricants de cartes proposant une implantation de la même technologie. De plus, comme ces cartes permettent de gérer plusieurs applications au sein d'un même environnement sécurisé, elles permettent aussi de développer des applications collaborant de manière sécurisée sur un même support – chose auparavant assez difficile.

Néanmoins, si ces technologies multi-applicatives résolvent beaucoup de problèmes du monde de la carte à puce, elles en créent également de nouveaux et notamment dans le domaine essentiel de la sécurité. En effet, puisqu'il est possible d'embarquer plusieurs applications de différents fournisseurs, un émetteur d'application potentiellement malhonnête, ou tout simplement peu informé des recommandations sécuritaires, pourrait charger une application malicieuse ou involontairement mal programmée qui risquerait de mettre en péril la sécurité de toute la plate-forme.

Bien évidemment, afin de contrer de telles menaces, et tant que ces technologies n'ont pas été prouvées sûres à 100% à l'aide de méthodes formelles, plusieurs procédures et mécanismes peuvent être déployés. Parmi les procédures il est toujours possible de recourir à l'évaluation sécuritaire de la plate-forme et de l'application à charger afin de s'assurer de leur bon fonctionnement et de l'efficacité des fonctions de sécurité mises en œuvre. Un consortium d'industriels a également développé un standard, GlobalPlatform, pour n'autoriser le chargement d'applications sur les cartes multi-applicatives qu'aux seules personnes dûment authentifiées. Aujourd'hui, ces deux exemples illustrent bien l'état de l'art des méthodes utilisées pour assurer la sécurité d'une plate-forme multi-applicative sur carte à puce. S'il est certain que ces procédures conduisent à améliorer la sécurité globale du système, c'est au détriment de sa flexibilité. Heureusement, dans un futur proche et au vu des travaux de recherche récents sur les vérificateurs embarqués, il devrait être possible de disposer de plates-formes sécurisées et totalement ouvertes. Dans ce cadre, la plate-forme sera évaluée et certifiée une fois et une seule et les applications pourront être chargées dessus sans aucune restriction puisque la plate-forme assurera la sécurité d'une application vis-à-vis des autres. Ainsi, les industriels n'auront plus qu'à faire évaluer les applications critiques nécessitant une assurance forte quand au fait qu'elles ne dévoilent pas malgré elles leurs biens indépendamment de toute plate-forme, *i.e.* il faudra juste vérifier qu'elles sont

bien programmées en accord avec les spécifications de l'application et il ne sera plus nécessaire de prendre en compte tout l'environnement puisqu'elle sera destinée à être chargée sur une plate-forme déjà évaluée et certifiée.

0.2 Cadre de travail

Les travaux présentés sont le fruit de ma thèse sous convention CIFRE entre SERMA Technologies et le LaBRI mais également d'un projet intitulé « Sécurité Java Card », labellisé PROGSI (*i.e.* « PROGramme Société de l'Information ») par le Ministère de l'Économie, des Finances et de l'Industrie, et dont les partenaires étaient SERMA Technologies et le LaBRI. Les objectifs du côté de SERMA Technologies étaient d'améliorer son expertise Java et d'acquérir les compétences et les moyens tant en terme de logiciels que de méthodologies pour mener à bien des évaluations sécuritaires de cartes à puce multi-applicatives. Les objectifs du côté du LaBRI était de se confronter à des vrais problèmes industriels afin de contribuer par une démarche scientifique à les résoudre et donc de faire profiter la communauté de son savoir faire dans le domaine des technologies Java.

A cet effet, lors du projet « Sécurité Java Card », nous avons développé un environnement libre pour Java Card, la suite JCtools qui comprend comme outil principal un émulateur Java Card. Ce développement nous a par ailleurs permis de relever des incohérences dans les spécifications Java Card et de proposer la notion nouvelle de *pré-persistance* pour clarifier et expliquer les propriétés des différentes implantations possibles des modèles de mémoire.

D'autres part, nos travaux se sont concentrés sur l'étude des nouvelles problématiques de sécurité occasionnées par les cartes multi-applicatives. Nous avons ainsi pu mettre en évidence des vulnérabilités potentielles à prendre en compte lors du développement de tels systèmes. Nous avons également proposé quelques solutions pour compliquer d'éventuelles attaques.

Enfin, au sein de l'équipe « Systèmes et Objets Distribués » du LaBRI nous avons contribué à la proposition d'une utilisation des cartes à puce multi-applicatives pour sécuriser le calcul sur la grille : « Java Card Grid ». Ces travaux ont conduit à la réalisation et au déploiement d'une première plate-forme afin d'obtenir une preuve de concept. Dans ce cadre nous avons participé au développement de deux applications utilisant cette architecture pour valider les premières hypothèses.

0.3 Un environnement libre pour Java Card

Développé pour partie lors du projet « Sécurité Java Card » au sein de l'équipe « Systèmes et Objets Distribués » du LaBRI par Iban Hatchondo (ingénieur de recherche au LaBRI) et par moi-même, cet environnement libre a eu pour vocation d'accroître l'expertise du LaBRI et de SERMA Technologies dans le domaine de la technologie Java Card. À terme nous souhaiterions que cet environnement puisse être disponible sous licence libre afin de permettre à la communauté scientifique de greffer ses outils (*e.g.* vérifieur, etc.) et

aux industriels d'expérimenter leurs applications.

0.3.1 État de l'art

Au début du projet, SERMA Technologies souhaitait établir une liste des vulnérabilités des plates-formes Java Card et définir des batteries d'attaques pour tester ces plates-formes. Puisqu'il fallait préserver la confidentialité des informations de ses clients les travaux devaient être menés indépendamment de tout produit existant. Par ailleurs si Sun microsystems possédait bien une suite de tests complète, fournie seulement aux licenciés Java Card, mais qui aurait pu intéresser SERMA Technologies, ces derniers ne souhaitaient pas investir dans cette licence puisque leur cœur de métier n'est pas de fabriquer des cartes et que l'opération n'aurait pas été rentable. De plus, cette batterie de tests était plutôt orientée vers du test fonctionnel et non pas vers du test sécuritaire comme le souhaitait plutôt SERMA Technologies.

0.3.2 Les JCatools

Comme aucun produit ouvert n'était alors disponible et comme cela étant nécessaire pour d'autres activités de l'équipe SOD nous avons fait le choix de développer notre propre environnement d'exécution afin de maîtriser complètement la technologie. Il est intéressant de noter que trois ans après le début de ce projet c'est aujourd'hui **le seul** émulateur libre existant même si nous ne pouvons le distribuer pour le moment en raison d'une discussion en cours avec Sun microsystems relative à la licence. JCatools est donc un environnement pour l'étude de plates-formes et d'applets Java Card. Il est composé d'un ensemble d'outils qui permet en particulier de répondre aux besoins de SERMA Technologies pour mener les évaluations sécuritaires de produits Java Card. Ce développement réalisé en Java comprend trois outils principaux (*i.e.* JCat Emulator, JCat View, JCat Converter) ainsi que quelques utilitaires destinés à simplifier les opérations de modification de fichiers CAP. Bien évidemment JCat Emulator, qui est une implantation complète des spécifications Java Card 2.1.1 (*i.e.* VM, JCRE et APIs, sauf les APIs cryptographiques), est le cœur de notre *framework*. Cet outil inclut le pare-feu, les mécanismes d'atomicité et de transactions et lit le format standard des fichiers CAP, ce qui en fait un vrai émulateur et non pas un simple simulateur. Il est destiné à tester et à attaquer des implantations d'applets afin de détecter des problèmes dans leur comportement comme par exemple une mauvaise utilisation des services offerts par la plate-forme. Ce *framework* nous a permis de développer un ensemble d'applets agressives pour tester certains points précis des implantations que SERMA Technologies a et aura à évaluer.

0.3.3 Notre contribution : la *pré-persistence*

La technologie Java Card a été conçue afin d'apporter aux programmeurs d'applications embarquées sur cartes à puce tous les avantages offerts par le langage Java. Ainsi, adapter l'environnement Java aux cartes à puce consistait à répondre à la question : « Comment intégrer Java sur un périphérique de nature persistante ? ». En effet, grâce à la technologie des mémoires persistantes qu'elle embarque, la carte à puce peut stocker de façon sécurisée

des informations mais aussi les conserver une fois hors tension. Cependant, l'environnement Java étant un environnement de programmation temporaire (*i.e.* qui s'exécute dans la RAM, donc dont toutes les données sont détruites à la fin de l'exécution de la machine virtuelle), pour satisfaire aux contraintes des cartes à puce les spécifications Java Card ont choisi d'adopter les concepts de *persistance* (*i.e.* les données continuent d'exister entre deux exécutions du programme) et de *transience* (*i.e.* des données possèdent un cycle de vie qui dépend de différents facteurs : *e.g.* étendue de la méthode qui les a créées, occurrence d'un événement, etc.) issus des systèmes persistants orthogonaux (*i.e.* des systèmes où la transience et la persistance dépendent du type de la donnée).

Malheureusement des ambiguïtés subsistent dans les spécifications Java Card et nous présenterons les problèmes rencontrés lors de la réalisation de notre plate-forme JCatools. Pour cela nous introduirons la notion de *pré-persistance* afin d'expliquer les différents modèles de mémoire que nous envisageons.

0.3.4 Perspectives

Faute de temps et dans le contexte d'une thèse sous convention CIFRE nous n'avons pas poussé plus avant une formalisation des concepts de persistance, transience et pré-persistance, mais il est certain qu'à terme nous souhaitons proposer un modèle formel. D'autre part, dès que la suite JCatools aura pu être placée sous une licence libre nous espérons pouvoir poursuivre son développement afin d'implanter les fonctionnalités manquantes.

0.4 Les problèmes de sécurité des cartes multi-applicatives

Avec l'apparition des cartes multi-applicatives de nouvelles problématiques se sont posées. En effet, il est maintenant non seulement possible, comme pour la génération précédente, de mener des attaques depuis l'extérieur, mais également depuis l'intérieur de la carte à puce.

0.4.1 État de l'art

Si quelques articles traitent de possibles attaques sur les cartes utilisant ces nouvelles technologies, aucun d'eux ne se place sous l'angle que nous avons choisi pour mettre en évidence de nouveaux problèmes de sécurité. Ainsi, ils abordent principalement les problèmes liés au chargement, à la machine virtuelle, au partage d'objets, et aux flux d'informations entre les applets, alors que nous avons choisi de nous intéresser à l'impact du chargement d'applications au niveau matériel de la puce.

0.4.2 Notre contribution : la mise en évidence de nouveaux problèmes

De notre côté, et sûrement à cause de la forte expertise de SERMA Technologies dans le domaine de l'analyse de composants électroniques, nous nous sommes focalisés sur la

recherche des vulnérabilités exploitables au niveau matériel (*e.g.* les canaux cachés) et causées par la possibilité offerte à n'importe quel utilisateur de charger sa propre application. Ainsi, nous avons décrit comment l'utilisation de méthodes d'aide à l'identification physique de motifs (*i.e.* bytecode, ou plus généralement une suite d'opérations) peut permettre d'attaquer ou de faire de la rétro-conception du code d'une application officielle (*i.e.* une application émise par une entité officielle : *e.g.* une banque) déjà présente sur la carte.

0.4.3 Perspectives

Si certaines des attaques que nous proposons sont utilisées avec succès lors d'évaluations de cartes multi-applicatives, nous n'avons néanmoins pas encore réussi à constituer un dictionnaire qui permettrait de faire de la rétro-conception de code. Dans le cadre de collaborations en cours de définition avec les équipes « Arithmétique, Cryptographie, Codage » et « Sécurité de l'Information » de l'Université de Limoges nous espérons pouvoir poursuivre ces recherches.

0.5 Le projet Java Card Grid

Un enjeu actuel essentiel dans le domaine du calcul distribué, et en particulier du calcul sur la grille, est celui de la sécurité. Or dans ce domaine, peu de travaux s'intéressent à la protection des codes de calcul. La seule solution appliquée jusqu'à présent est celle de la confiance : le propriétaire d'un code fait confiance aux propriétaires des ressources de calcul qu'il exploite pour ne pas tenter de corrompre l'exécution de son code ou d'espionner celle-ci. Dans les situations où la confiance aveugle est impossible, nous proposons l'utilisation de cartes à puce multi-applicatives et en particulier de la technologie Java Card pour protéger un code de calcul.

0.5.1 État de l'art

Des frameworks tels que JiniCard, Jason ou OrbCard ont déjà été développés pour communiquer de manière transparente et sécurisée avec les services des cartes à puce. Il est de fait possible de les utiliser pour faire du calcul distribué sur des cartes. Toutefois, comme ce n'est pas leur paradigme principal, il manque à tous ces frameworks des mécanismes utiles dans le contexte des applications réparties tels que l'invocation asynchrone de méthode requise dès que l'on utilise des ressources informatiques lentes comme les cartes à puce.

Des approches plus formelles pour protéger des codes mobiles exécutés dans des environnements d'exécution classique (*i.e.* pas de confiance) ont également été développées. Par exemple, il existe une solution originale basée sur une extension des fonctions de cache utilisant les codes correcteurs d'erreurs. Seulement, bien qu'utilisant des bases solides, il semble difficile d'utiliser ces solutions dans un cas général en raison des hypothèses fortes qu'elles imposent sur les programmes à exécuter pour être applicables.

0.5.2 Notre contribution : la preuve de concept

Nous avons choisi de développer une preuve de concepts basée sur l'utilisation massive de Java Cards pour réaliser du calcul distribué sécurisé. Bien évidemment, même s'il était illusoire d'espérer obtenir de bonnes performances sur de tels périphériques nous avons développé deux applications afin de valider nos hypothèses, et nous avons ébauché ce que devrait être une architecture destinée à faire du calcul sécurisé et exploitant du matériel sécurisé. Cette architecture est basée sur la plate-forme Mandala elle aussi développée dans l'équipe « Systèmes et Objets Distribués » du LaBRI.

0.5.3 Perspectives

Dans l'avenir l'équipe SOD souhaite développer cette plate-forme afin de lui donner un plus grand dynamisme nécessaire à la validation de certains aspects sécuritaires, puis, par la suite l'étendre au cadre de la mobilité.

0.6 Organisation de la thèse

En raison de la diversité des travaux de recherche réalisés, cette thèse est constituée de quatre parties. La première partie fait un état de l'art de la carte à puce, des cartes multi-applicatives et tout spécialement Java Card, des standards GlobalPlatform et PC/SC et de la certification.

Dans la seconde partie nous décrirons les travaux réalisés dans le projet « Sécurité Java Card ». Nous présenterons les JCatools et nous introduirons la notion de pré-persistance. Dans une troisième partie nous reviendrons sur les nouvelles vulnérabilités des cartes à puce multi-applicatives. Enfin dans la dernière partie nous exposerons comment la fusion de certains travaux des divers membres de l'équipe « Systèmes et Objets Distribués » a permis de proposer une solution à la protection des code mobiles pour faire du calcul distribué.

Nous avons donc développé un tout cohérent autour des cartes à puce multi-applicatives en cherchant tout d'abord à identifier leurs vulnérabilités intrinsèques et en proposant des solutions pour s'en prémunir. Nous avons également cherché à utiliser ces nouvelles technologies de cartes pour sécuriser des systèmes plus larges.

Première partie

Présentation des standards

La première partie de cette thèse dresse un état de l'art de la carte à puce, des cartes multi-applicatives et tout spécialement de Java Card. Elle présente également les standards GlobalPlatform et PC/SC permettant respectivement de gérer la multi-application sur les cartes à puce et d'établir le dialogue entre les applications hors-carte et les cartes (*i.e.* les applications sur les cartes). Pendant toute la durée cette thèse sous convention CIFRE – trois ans –, j'étais salarié de SERMA Technologies au poste d'ingénieur de recherche et développement pour le CESTI (Centre d'Évaluation de la Sécurité des Technologies de l'Information). J'exposerai donc le contexte sous-jacent aux travaux menés dans cette thèse, *i.e.* l'évaluation sécuritaire de produits des Technologies de l'Information.

Notre contribution dans cette partie est principalement l'effort de présentation scientifique des divers domaines pointus du monde de la carte à puce.

Chapitre 1

La carte à puce

Tantôt encensée par les uns pour sa sécurité et sa facilité d'utilisation et tantôt décriée par les autres pour les piratages répétés des systèmes dont elle assure la sécurité, la carte à puce est l'objet de toutes les passions. Ce chapitre a pour vocation de la présenter de façon objective et pose les bases sur lesquelles s'appuient nos travaux. L'accent sera mis sur les multiples avantages de la carte à puce et principalement sa sécurité.

1.1 Qu'est ce qu'une carte à puce ?

Une carte à puce est une carte plastique qui intègre un circuit électronique capable de manipuler (stocker, calculer, etc) des informations de façon sécurisée. C'est en quelque sorte un ordinateur portable et résistant aux altérations. Il s'agit donc bien d'une carte *intelligente* (*smart card* comme on l'appelle dans les pays anglo-saxons) et non d'une simple carte à piste magnétique puisqu'elle embarque un circuit logique. D'ailleurs contrairement à cette dernière, la carte à puce n'a pas besoin d'accéder à une base de données distante lors d'une transaction puisqu'elle possède sa propre puissance de calcul et sa propre capacité de stockage d'informations dans un mode sécurisé. La plupart des normalisations applicables à ces cartes sont décrites dans l'ISO 7816 [108].

1.2 Historique

Voilà déjà plus de trois décennies que le premier prototype de carte à puce a été conçu et pourtant elle est toujours aussi méconnue car volontairement entourée par ses fabricants de beaucoup de secrets. Même la paternité de son invention est sujette à discussion et ainsi suivant le pays où l'on se trouve, son inventeur change. Le bref historique ci-après est celui qui semble néanmoins le plus universellement admis.

- En 1968, Jürgen Dethloff et Helmut Grötrupp deux inventeurs allemands sont les premiers à introduire un circuit intégré dans une carte plastique.
- En 1970, indépendamment Kunitaka Arimura de l'institut de technologie Arimura au Japon dépose un brevet sur la carte à puce.

- Entre 1974 et 1978, Roland Moreno dépose 47 brevets dans 11 pays. En France, il est considéré comme l’inventeur de la carte à puce.
- En 1979, la première carte est créée et assemblée à Toulouse par Motorola pour Bull CP8 avec 1 Ko de mémoire programmable et un cœur à base de microprocesseur 6805.
- En 1983 apparaissent les premières cartes téléphoniques à mémoire.
- En 1984, le G.I.E carte bancaire adopte la « carte bleue » proposée sur un prototype Bull CP8. Cela aboutit à notre carte bleue actuelle (version B0’).
- Entre 1984 et 1987, les normes internationales de l’ISO sur la carte à puce à contact voient le jour sous la référence 7816.
- En 1997, les premières cartes multi-applicatives apparaissent.

1.3 Les différents types de cartes à puce

Il existe plusieurs sortes de cartes à puce que l’on peut classer de plusieurs façons. Cette section va présenter un classement suivant les technologies utilisées en interne. Dans la section 1.5 nous présenterons un classement basé sur les possibilités de communication de ces cartes.

1.3.1 La carte à mémoire

Premiers modèles de cartes à puce, elles représentaient encore la majorité des cartes vendues dans le monde en 1999.

Comme son nom l’indique, une telle carte possède une puce mémoire. Elle peut aussi comporter une logique câblée non programmable (instruction programmée et non reprogrammable) mais pas de microprocesseur. Autrefois la taille de sa mémoire était seulement de quelques Kilo-octets alors qu’aujourd’hui certaines peuvent atteindre le Méga-octets. Les avantages de cette carte sont sa technologie simple et son faible coût de revient (1 €). Ses principaux inconvénients sont sa dépendance vis-à-vis du lecteur de carte pour le calcul – elle même en est incapable – et la possibilité d’être assez facile à dupliquer (mais toujours beaucoup moins que les cartes utilisant la piste magnétique).

Pour l’anecdote, on peut noter qu’il existe plusieurs types de carte à mémoire et par exemple en décembre 2000, la société LaserCard (anciennement Drexler Technology Corporation) [37] avait annoncé le développement de cartes à mémoire optique capable de stocker plus de 4 Mo. Sur ces cartes, la mémoire pour stocker des informations est une couche plastique à la surface de la carte, un peu à la façon des CDs. Bien entendu elles ne correspondent pas au standard international ISO7816 mais constituent sûrement une voie de développement pour les cartes de demain.

1.3.2 La carte à microprocesseur

Cette carte est l’essence même de ce que l’on nomme une *smart card*. La puce de cette carte embarque un microprocesseur, ce qui la rend autonome (*i.e.* intelligente) et donc plus

sûre. Les dimensions de la puce sont d'environ 25mm^2 pour sa surface et $200\mu\text{m}$ pour son épaisseur (*cf.* FIG. 1).

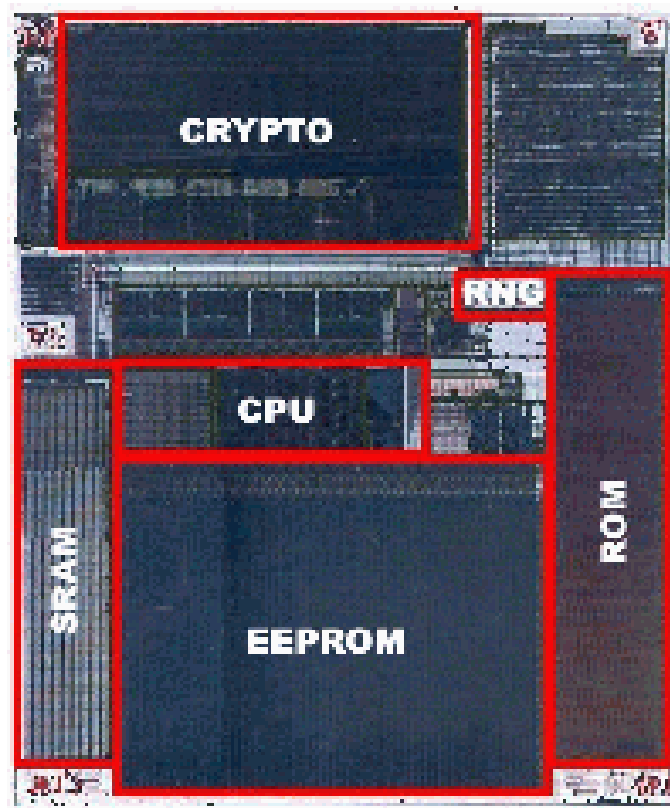


FIG. 1 – Un microprocesseur de carte à puce (source : Anonyme).

Précédemment, ces cartes étaient basées sur des processeurs de type Motorola 6805 ou l'Intel 8051, 8 bits CISC, cadencés selon une horloge externe à 5 MHz. Aujourd'hui, on dispose de processeurs plus puissants 16 ou 32 bits CISC ou RISC travaillant à des fréquences comprises entre 5 et 40 MHz. Cependant, Le standard ISO7816 obligeant à respecter une certaine plage de fréquence d'horloges pour rester compatible avec les matériels déjà en place (*e.g.* lecteurs, etc), les fabricants utilisent alors en interne des multiplicateurs pour atteindre de plus hautes fréquences permettant de faire les calculs dans la carte plus rapidement. Actuellement, il existe ainsi des cartes à microprocesseur de dernière génération RISC fonctionnant avec une horloge interne de 100 à 200 MHz en utilisant un multiplicateur $\times 20$ ou $\times 40$.

Côté mémoire, comme nous pouvons le voir sur la figure 1 les cartes en comportent trois types que nous détaillerons dans la section 1.4.

Ces puces possèdent également un coprocesseur cryptographique qui réalise les opérations de chiffrement et déchiffrement de manière câblée améliorant ainsi grandement les performances. Ce coprocesseur cryptographique est associé à un générateur de nombres aléatoires RNG (Random Number Generator).

L'avantage d'une telle carte est le microprocesseur qui fournit la sécurité, via le coprocesseur cryptographique et d'autres mécanismes, en interdisant que les données ne soient accessibles de l'extérieur. Le coût pour un produit aussi sûr est tout à fait acceptable puisque situé entre 1 € et 20 €.

Pour résumer, dans cette carte c'est la triple alliance électronique, informatique et cryptographie qui assure un très haut degré de sécurité.

1.4 Les différents types de mémoire

Une caractéristique originale des cartes à puce provient des différents types de mémoire qu'elles embarquent. Le choix des différentes mémoires présentes sur une carte est tellement important et crucial pour la sécurité qu'il nous a semblé essentiel de les présenter dans une section dédiée. De plus, une partie de nos travaux, qui seront développés ultérieurement, se basent sur les propriétés spécifiques de ces différents types de mémoire. Dans cette section nous n'évoquerons pas les aspects sécuritaires relatifs aux mémoires car ceux-ci sont décrits pour partie dans la section 1.7 et pour le reste dans nos travaux.

De façon schématique, les cartes à puce utilisent trois types de mémoire aux propriétés bien spécifiques : la mémoire figée en usine, la mémoire persistante et la mémoire volatile. Ces types de mémoires, suivant la technologie employée, occupent plus ou moins d'espace sur la puce. Or, comme nous l'avons vu précédemment la taille de la puce est limitée. Le constructeur est donc obligé de jouer sur la capacité accordée à chaque type de mémoire afin d'obtenir un produit performant et viable.

À titre d'exemple, suivant les technologies de gravure utilisées, une cellule d'EEPROM prend quatre fois plus de place qu'une cellule de ROM, et une cellule de RAM prend quatre fois plus de place qu'une cellule d'EEPROM.

1.4.1 La mémoire programmée en usine

La mémoire programmée en usine est une mémoire persistante et figée. Elle n'a donc pas besoin d'énergie pour sauvegarder l'information qu'elle contient et son contenu n'est pas modifiable. Cette mémoire est aussi appelée ROM (Read Only Memory). Elle sert à stocker le COS (Card Operating System) ainsi que des données permanentes. Elle est programmée en usine une fois pour toute. Sa taille est le plus souvent de 32 Ko mais il existe des cartes disposant de 64 Ko, voire plus.

1.4.2 La mémoire volatile

La mémoire volatile qui est utilisée comme espace de stockage temporaire pour le calcul grâce à la rapidité de ses temps d'accès. Elle possède un caractère non persistant et elle perd donc son contenu dès que la carte n'est plus sous tension. Elle peut être lue et écrite à l'infini. Il s'agit d'une mémoire RAM (Random Access Memory) équivalente à celle présente sur un ordinateur classique. Néanmoins, pour une carte, sa capacité est seulement de 1 à 4 Ko.

1.4.3 La mémoire persistante

La mémoire persistante tout comme la ROM est aussi une mémoire persistante mais dont le contenu est modifiable. Elle est souvent désignée sous le terme de NVM (*i.e.* Mémoire Non Volatile ou en anglais Non-Volatile Memory). Elle permet le stockage d'informations qui peuvent évoluer dans le temps et qui doivent être conservées même lorsque la carte n'est plus sous tension.

Encore récemment il n'existait qu'un seul type de mémoire possédant ces caractéristiques : l'EEPROM (Electrical Erasable Programmable Read Only Memory). Aujourd'hui, on trouve des produits basés sur d'autres types de mémoire aux propriétés similaires comme la Flash ou la FeRAM. Une autre alternative est aussi en cours de développement avec la MRAM. Ces différentes technologies de mémoire (*i.e.* EEPROM, Flash, FeRAM et MRAM) seront décrites ci-dessous.

L'EEPROM présente deux inconvénients :

- sa durée de vie limitée en nombre de cycles d'écriture – *i.e.* son endurance – (100000 cycles) et en temps de rétention de l'information (10 ans) ce qui nécessite de changer régulièrement la carte pour éviter des dysfonctionnements ;
- sa lenteur d'accès – *i.e.* sa latence – puisqu'elle est 1000 fois plus lente en écriture que la RAM.

La mémoire Flash a le gros avantage de ne pas occuper beaucoup de place sur la puce. En revanche, elle possède une latence en écriture très supérieure à l'EEPROM et une endurance équivalente. Toutefois, grâce à sa technologie, on a vu apparaître des produits disposant de peu ou pas de ROM et de beaucoup de mémoire Flash ce qui permet de reprogrammer entièrement la carte. Le plus souvent il s'agit de cartes prototypes pour les développeurs de systèmes d'exploitation. Avec la technologie très intégrée de cette mémoire, Sharp a présenté un produit en 2002 qui possédait 1 Méga-octets de mémoire Flash [58].

La mémoire utilisant la technologie FeRAM [208] (Ferroelectric Random Access Memory) est une mémoire basée sur les propriétés ferromagnétiques des matériaux qui la composent. La latence des opérations de lecture et d'écriture est constante. De plus, cette latence est bien plus faible que celle d'une opération de lecture de l'EEPROM. Elle possède une endurance presque illimitée et elle consomme aussi beaucoup moins d'énergie. Cette dernière caractéristique la rend parfaite pour les cartes sans contact. Un de ses défauts est une occupation sur la puce plus importante par rapport à la Flash, mais identique à celle de l'EEPROM.

La MRAM (Magnetic Random Access Memory) fonctionne grâce à la polarisation magnétique des matériaux. Cette technologie lui confère une endurance illimitée, tout comme la RAM. L'avantage de la MRAM est que sa lecture n'est pas destructive contrairement à la FeRAM.

La taille de ces mémoires varie de 16 à 64 Ko, voire 256 Ko. Les derniers développements technologiques devraient nous conduire très bientôt à des mémoires non volatiles presque aussi performantes que les mémoires volatiles (*i.e.* la RAM) présentes sur nos stations de travail.

1.4.4 Résumé

Le tableau 1 résume des caractéristiques des différents types de mémoire issues de diverses sources [114, 186, 196]

	SRAM	EEPROM	Flash	FeRAM	MRAM
Volatile	Oui	Non	Non	Non	Non
Occupation	6T	2T	1T	2T/2C	N/A
Endurance	∞	10^5	10^5	$10^{12} - \infty$	$10^{15} - \infty$
Cycle de lecture (ns)	12	200	70	30 - 200	10 - 1000
Cycle d'écriture (ns)	12	3000	10^6	30 - 200	10 - 50
Consommation lecture (mA/operation)	180	30	15	15	N/A
Consommation écriture (mA/operation)	180	30	35	15	N/A
Rétention (année)	0	10	10	10	10

TAB. 1 – Caractéristiques des différents types de mémoire.

1.5 Les communications

Nous avons vu que la carte à puce peut être assimilée à un ordinateur. Or, pour interagir avec un environnement extérieur, (*e.g.* d'autres ordinateurs) la carte a besoin de communiquer.

Plusieurs technologies de communications ont donc été développées pour réaliser ces échanges avec le monde extérieur. Toutefois les cartes, de part leur conception extrêmement compacte, sont des ordinateurs de nature passive qui doivent être alimentés depuis l'extérieur pour pouvoir fonctionner (même s'il existe maintenant des batteries assez compactes et performantes pour être embarquées). C'est ainsi que les cartes ne communiquent pas directement avec d'autres ordinateurs, mais plutôt avec un lecteur de carte à puce qui leur fournit l'alimentation nécessaire à leur fonctionnement. Par conséquent, le lecteur sert d'interface (*i.e.* d'adaptateur) entre la carte et la machine hôte avec laquelle elle doit dialoguer.

Dans un premier temps, nous présenterons une classification des cartes suivant leur mode de communication avec le lecteur. La communication de la machine hôte avec le lecteur sera quant à elle détaillée dans un chapitre ultérieur (*cf.* chapitre 4). Dans un second temps nous nous concentrerons sur les protocoles de communication utilisés. Ces protocoles sont importants puisque nous les détournons de leur utilisation normale pour réaliser les attaques présentées dans le chapitre 11 et pour rendre la carte « active » dans le chapitre 13.

1.5.1 La carte à contact

Ce type de carte communique via un microcontact relié à la puce (*microchip*) par des fils d'or. Cet assemblage se nomme micromodule (*cf.* FIG. 2).

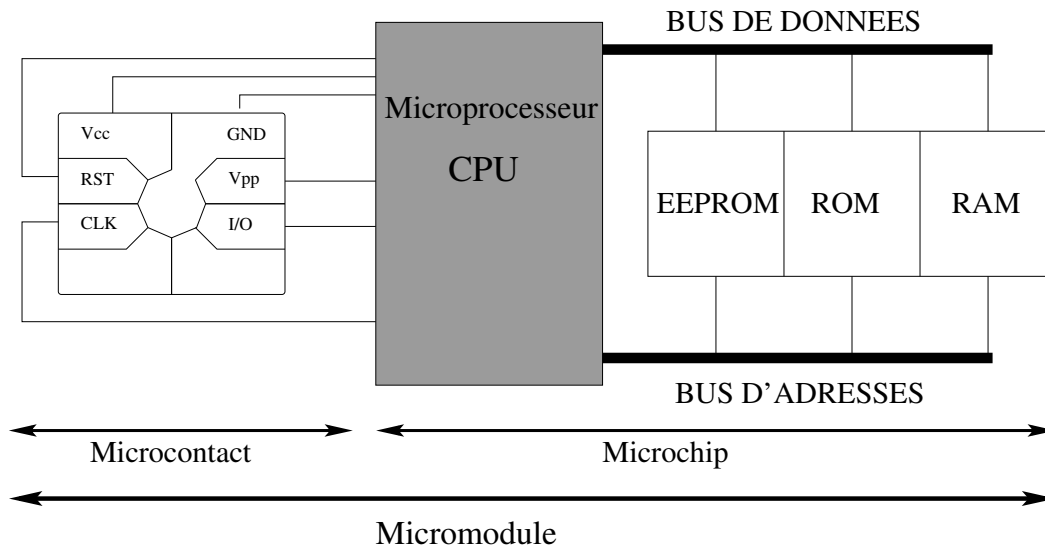


FIG. 2 – Le micromodule.

Il est placé dans le corps de carte (*i.e.* la partie plastique) (*cf.* FIG. 3) et correspond à la partie intelligente de la carte.

Ces cartes utilisent une communication série via les huit contacts définis dans le standard ISO 7816. En pratique, seulement cinq de ces huit contacts sont réellement utilisés pour réaliser la communication série. Le contact Vpp n'est quasiment plus utilisé puisqu'il servait juste à fournir une alimentation suffisante lors de la programmation de mémoires utilisant des technologies anciennes et donc consommant beaucoup d'énergie. En revanche, les deux derniers contacts qui étaient jusqu'alors réservés à une utilisation future sont aujourd'hui utilisés pour communiquer en USB (Universal Serial Bus). Il s'agit donc toujours d'une communication série, mais universelle et beaucoup plus rapide que le mode de communication classique. Les cartes utilisant cette technologie intègrent donc directement la gestion de l'USB reléguant ainsi le lecteur à un simple adaptateur entre la carte et le PC. Cela permet d'avoir des lecteurs de faible coût.

Quoiqu'il en soit, pour pouvoir fonctionner toutes ces cartes doivent être insérées dans un lecteur de cartes qui leur fournit l'énergie dont elles ont besoin. C'est d'ailleurs une source de problèmes puisque l'insertion et le retrait sont des facteurs d'usure de la carte. Un autre problème est la bonne orientation de la carte à respecter lors de son insertion dans le lecteur.

1.5.2 La carte sans contact

Les cartes sans contact communiquent via une antenne interne reliée au *microchip* (*cf.* FIG. 4).

Les problèmes rencontrés avec les cartes à contact ne se posent plus puisque ces cartes n'ont pas besoin d'être insérées dans un lecteur pour fonctionner. La puce tire son énergie soit d'un couplage capacitif (couplage intérieur à la carte, *e.g.* une batterie) soit d'un

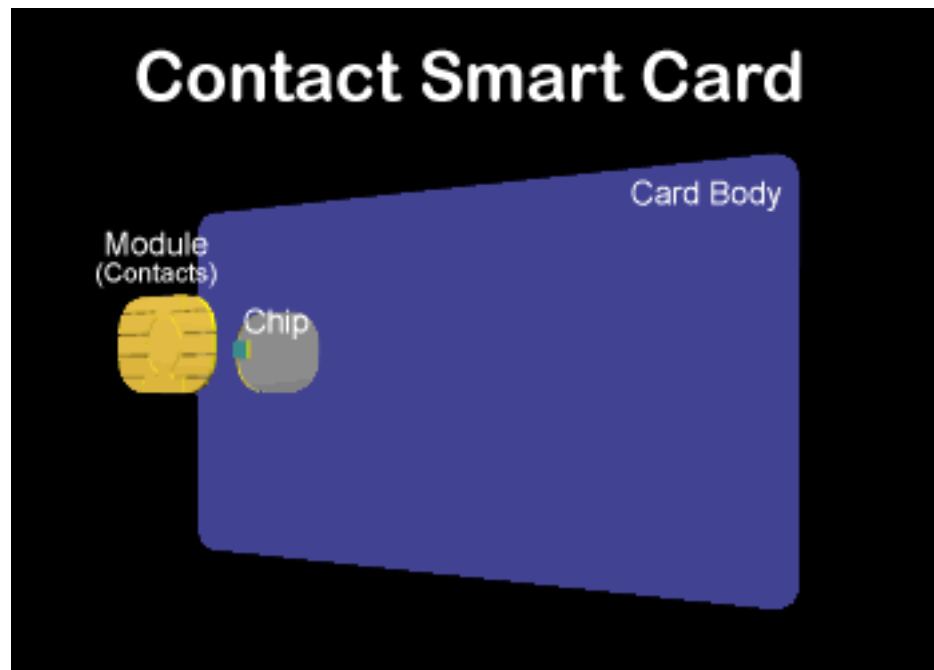


FIG. 3 – La carte à contact (source : Gemplus).

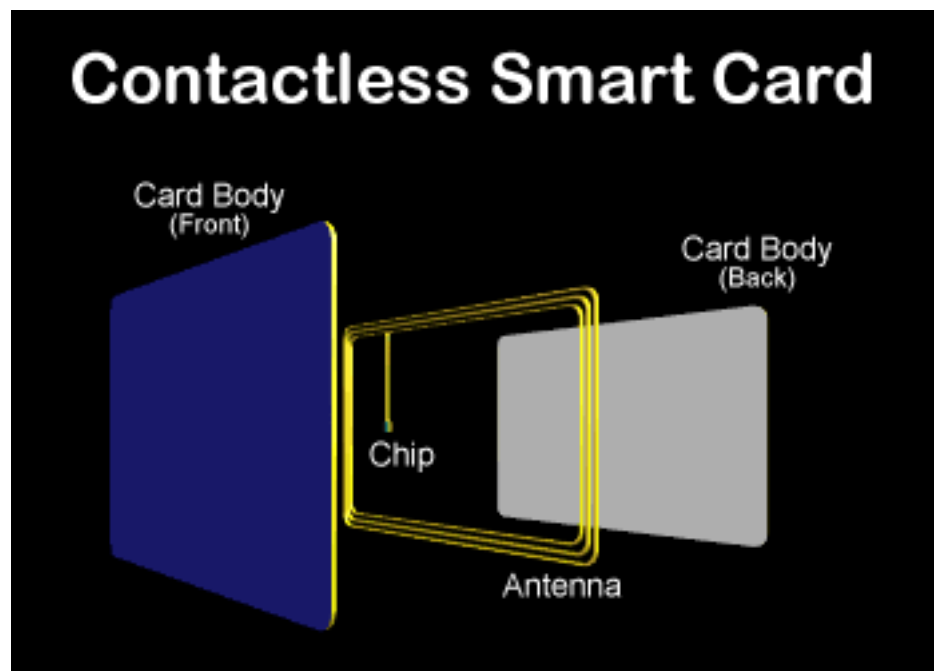


FIG. 4 – La carte sans contact (source : Gemplus).

couplage inductif (couplage distant, collecté par l'antenne). Le couplage inductif fonctionne sur le principe du transformateur où une bobine dans le lecteur induit un courant dans une autre bobine (*i.e.* l'antenne dans la carte). La fréquence de transmission utilisée est de l'ordre de quelques MHz. La puce est capable, en changeant sa résistance, de transmettre un signal qui est capté par le lecteur et interprété comme un signal de données.

Bien sûr, ces cartes posent aussi quelques problèmes. Par exemple, la distance de communication entre la borne de lecture et la carte est limitée (environ 10 cm). Un autre problème est que la carte est souvent en mouvement rapide dans l'espace d'échange et que le temps pour réaliser la transaction est de l'ordre de 200 ms ce qui limite la taille des données à échanger (souvent quelques centaines d'octets). Par ailleurs, ces cartes sont beaucoup plus chères que les cartes à contact.

Elles sont surtout utilisées pour les applications où l'insertion et le retrait dans un lecteur ne sont pas pratiques.

1.5.3 La carte dual-interface

Une carte dual-interface est simplement la combinaison d'une carte à contact et d'une carte sans contact. Elle dispose donc des deux possibilités de communication ce qui en fait la carte idéale.

1.5.4 La pile de communication

Avec les cartes à puce classiques la communication entre le lecteur et la carte se fait en mode *half-duplex*, *i.e.* à un instant donné seule la carte ou seul le terminal envoie des données sur le support physique mais pas les deux en même temps.

Ce mécanisme nécessite donc un canal de synchronisation pour savoir qui est autorisé à communiquer et donc qui est autorisé à écouter. Pour ce faire, il faudra un maître pour initier la communication et un esclave pour la recevoir. Le non-respect de ces règles pourrait entraîner deux situations désastreuses pour le dialogue :

- les deux entités envoient leurs données en même temps sur le support physique et il y aura une collision et les données seront perdues ;
- les deux entités se mettront à écouter dans l'attente d'une communication en provenance de l'autre et dans ce cas il y aura une situation de *deadlock*.

La communication entre la carte à puce et le lecteur suit donc un modèle de communication de type maître-esclave où le lecteur est le maître. C'est d'ailleurs lui qui fournit à la carte l'énergie dont elle a besoin pour fonctionner.

La communication lecteur/carte s'effectue au travers d'une pile protocolaire qu'on peut essayer d'aligner sur le modèle OSI. Bien évidemment ce modèle de communication maître-esclave ainsi que les particularités des protocoles mis en jeu n'aboutissent pas à une pile protocolaire aussi parfaite que celle décrite par OSI, mais globalement on obtiendra la pile représentée figure 5.

Dans ce document nous utiliserons indifféremment les termes CAD (*i.e.* Card Acceptance Device) et IFD (*i.e.* InterFace Device) pour désigner un lecteur de carte à puce.

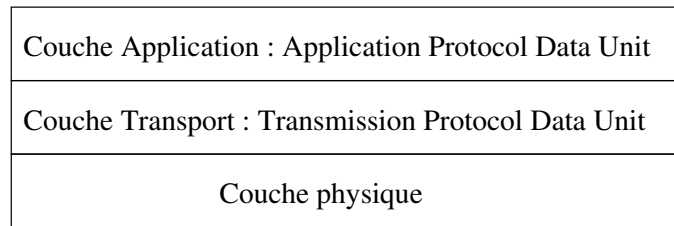


FIG. 5 – Pile de protocoles de communication entre le lecteur et la carte.

Dans une partie de nos travaux présentés à la section 13.3.2, nous détournerons ce protocole maître-esclave pour rendre la carte proactive.

La couche physique

Elle est normalisée par l'ISO 7816-3 qui décrit la fréquence d'horloge (entre 1 MHz et 5 MHz), la vitesse des communications (jusqu'à 115200 bauds), etc.

La couche transport : TPDU

C'est aussi dans l'ISO 7816-3 que sont définis les échanges de données pour les protocoles de transmission (TPDU : Transmission Protocol Data Unit) les plus couramment utilisés. Parmi eux deux sont particulièrement utilisés :

- le premier appelé T=0 est un protocole orienté octet ;
- le second appelé T=1 est un protocole orienté paquet.

Il y a aussi certaines cartes qui utilisent T=14 (réservé pour les protocoles propriétaires). On trouve aussi des cartes utilisant T=CL pour les communications sans contact.

Remarque : L'ISO 7816-3 normalise aussi :

- la sélection du type de protocole (PTS : Protocol Type Selection) si plusieurs protocoles sont supportés ;
- la réponse au reset (ATR : Answer To Reset) qui correspond aux données envoyées par la carte immédiatement après sa mise sous tension.

La couche application : APDU

Les échanges de données élémentaires pour cette couche sont les APDU (Application Protocol Data Unit). Ils sont normalisés par l'ISO 7816-4 et il en existe de deux types :

- la commande APDU (C-APDU) qui est émise par le CAD en direction de la carte ;
- la réponse APDU (R-APDU) qui, elle, transite de la carte au CAD.

Rappelons que le modèle de discussion entre le CAD et la carte est un modèle maître-esclave où le CAD est le maître et la carte est l'esclave. Ainsi, la carte est toujours en

attente d'une commande APDU et une réponse APDU aura toujours lieu en retour d'une commande APDU. En outre, les deux types de structures de communication commande et réponse sont toujours couplés.

La commande APDU est constituée par (*cf.* TAB. 2) :

- une en-tête obligatoire de quatre octets :
 - **CLA**, l'octet de classe identifie la catégorie de la commande et de la réponse APDUs,
 - **INS**, l'octet d'instruction spécifie l'instruction de la commande,
 - **P1** et **P2**, deux octets utilisés pour paramétrer l'instruction ;
- un corps optionnel de taille variable :
 - **Lc**, l'octet qui spécifie la taille du champ de données,
 - le **champ de données** qui contient les données à envoyer à la carte pour exécuter l'instruction spécifiée dans l'en-tête,
 - **Le**, l'octet qui spécifie le nombre d'octets attendus par le CAD pour le champ de données dans la réponse APDU de la carte.

En-tête obligatoire				Corps optionnel		
CLA	INS	P1	P2	Lc	Champ de données	Le

TAB. 2 – Structure d'une commande APDU.

La réponse APDU est constituée par (*cf.* TAB. 3) :

- un corps optionnel de taille variable :
 - le **champ de données** de taille **Le** déterminée dans la commande APDU correspondante ;
- une en-queue obligatoire de deux octets :
 - **SW1** et **SW2** (Status Word), deux champs d'un octet précisant l'état de la carte après l'exécution de la commande APDU. Par exemple, **0x9000** signifie que l'exécution s'est déroulée complètement et avec succès.

Corps optionnel	En-queue obligatoire	
Champ de données	SW1	SW2

TAB. 3 – Structure d'une réponse APDU.

Comme certaines parties des APDUs sont optionnelles il y a quatre cas possibles d'échanges APDU (*cf.* TAB. 4) :

- cas 1 : Aucune donnée n'est échangée entre le CAD et la carte autre que l'en-tête de la commande APDU et l'en-queue de la réponse APDU.
- cas 2 : Aucune donnée (dans le champ de données de la commande APDU) n'est envoyée à la carte. Le corps de la commande contient le champ **Le** qui spécifie le nombre d'octets que la carte doit lui fournir dans le champ de données de sa réponse APDU.

	commande APDU	réponse APDU
cas 1	en-tête	SW
cas 2	en-tête Le	données SW
cas 3	en-tête Lc données	SW
cas 4	en-tête Lc données Le	données SW

TAB. 4 – Les différents échanges de commande et de réponse APDU.

- cas 3 : Des données, de taille Lc octets, sont fournies à la carte (dans le champ de données de la commande APDU) et la carte ne renvoie dans la réponse APDU que son en-queue.
- cas 4 : Des données sont échangées dans le champ de données de la commande et de la réponse APDU.

1.6 La fabrication et le cycle de vie de la carte

Cette section a pour vocation de présenter les étapes de la vie d’une carte et les différentes acteurs qui y prennent part. En effet, ces intervenants sont tous essentiels pour la sécurité de la carte car pouvant être tour à tour commanditaire d’une évaluation (cf chapitre 5), utilisateur bien ou mal intentionné, etc.

1.6.1 Les acteurs

On peut distinguer dans le monde de la carte à puce, quatre principaux acteurs :

- le fabricant de puces (*e.g.* Fujitsu, Philips, ST Microelectronics), aussi appelés fondeurs ;
- le fabricant de cartes (*e.g.* Axalto, Gemplus, Oberthur) ;
- l’émetteur de cartes ou fournisseur de cartes (*card issuer*) (*e.g.* les banques) qui fournit la carte à l’utilisateur final ;
- le porteur de carte qui est l’utilisateur final.

1.6.2 La fabrication

Le processus de fabrication décrit ci-après est volontairement simplifié mais reste assez proche de la réalité.

Au commencement de la vie d’une carte à puce se trouve le fabricant de puces. C’est lui qui va construire la puce, choisir les technologies pour le microprocesseur, les différentes mémoires embarquées, etc. Il va aussi, lors de l’opération de masquage, figer dans la mémoire ROM un système d’exploitation minimum et certaines données afin de permettre à son produit de fonctionner.

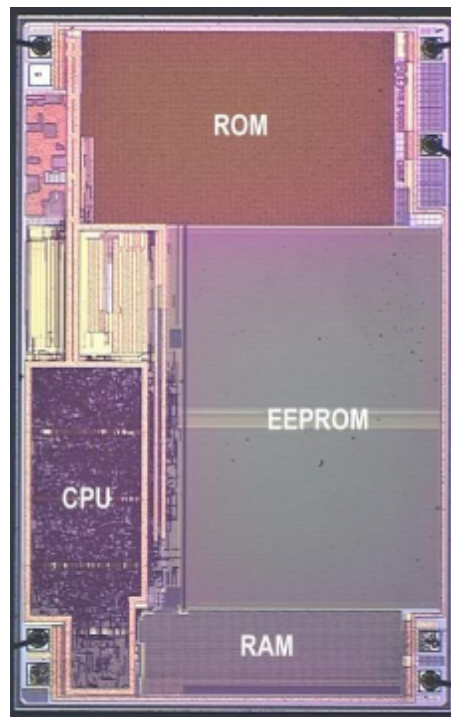


FIG. 6 – Exemple de puce observée en microscopie optique (source : SERMA Technologies).

Dans les faits, l'opération de masquage est une suite itérative d'opérations consistant à reproduire un circuit électronique par dépôts successifs de couches de métallisation puis de couches de passivation (*i.e.* un isolant) sur un substrat de silicium. La réalisation d'un circuit électronique dans un monde à deux dimensions est impossible puisqu'obligatoirement des pistes électriques se croiseraient. Aussi, les fabricants utilisent la troisième dimension pour câbler leurs circuits et emploient des plots de tungstène pour relier les différents niveaux de métal. Dès lors, si l'on réalise une coupe de la puce on a l'impression d'observer un mille-feuilles (*cf.* FIG. 7). En général, suivant les technologies utilisées, les niveaux de métal sont au nombre de quatre ou cinq.

Par la suite, le fabricant de cartes reçoit du fabricant de puces des galettes de silicone (*silicon wafer*) contenant les puces qu'il va découper, tester, puis assembler avec le microcontact pour former le micromodule (s'il désire par exemple faire une carte à contact). Parallèlement il produit le corps de carte à partir de PVC (Poly Vinyl Chloride), d'ABS (Acrylonitrile Butadiène Styrol) ou de PC (PolyCarbonate). Le matériau choisi pour le corps de carte dépendra de l'usage auquel est destinée la future carte. En effet, ces matériaux n'ont pas les mêmes caractéristiques (*e.g.* de longévité), donc pas les mêmes fonctions, ni les mêmes coûts.

L'étape suivante consiste à insérer le micromodule sur le corps de carte dans le cas d'une carte à contact – opération appelée encartage. On a maintenant une carte vierge qui va pouvoir commencer son cycle de vie.

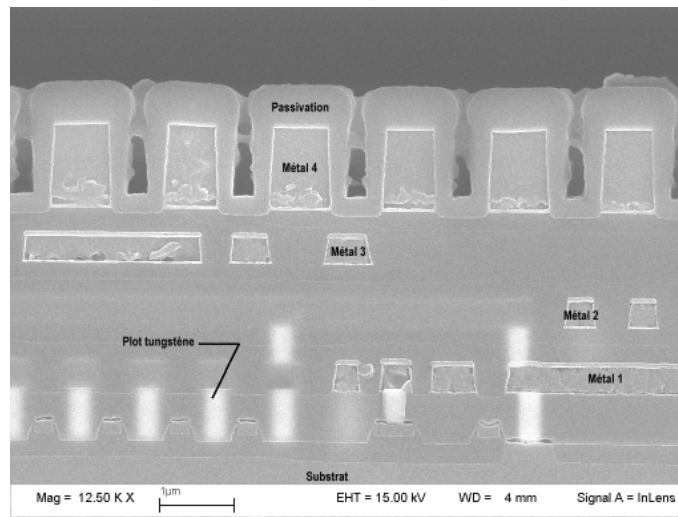


FIG. 7 – Coupe d’une puce observée en microscopie électronique (source : SERMA Technologies).

1.6.3 Le cycle de vie de la carte

Souvent le fabricant de cartes développera pour le compte de l’émetteur de cartes les applications à embarquer et les installera sur la carte.

Le cycle de vie de la carte commencera avec une phase d’initialisation de la carte qui consiste à inscrire en mémoire persistante des données communes aux applications et éventuellement à charger d’autres caractéristiques de sécurité. Cette étape, comme la suivante, est souvent réalisée par le fabricant de cartes en fonction de la demande de son client émetteur de cartes. L’étape suivante est la personnalisation et se fait à deux niveaux :

- au niveau électrique :
 - par l’inscription en mémoire persistante des données relatives à chaque porteur de carte ;
- au niveau graphique :
 - par la personnalisation du corps de carte suivant le désir de l’émetteur de carte (*e.g.* son logo),
 - par l’embossage de données relatives à chaque porteur de carte.

La carte peut alors passer du fabricant de cartes au porteur de carte par l’intermédiaire de l’émetteur de cartes. Le cycle de vie de la carte se poursuit ainsi avec son utilisation par le porteur de carte et se termine soit par invalidation logique, saturation de la mémoire, bris, perte, vol, etc.

1.7 Les sécurités

Nous allons maintenant présenter les différents niveaux de sécurité offerts par la carte à puce car nous en aurons besoin par la suite, dans la partie 3, lorsque nous exposerons les

attaques.

Pourtant mise en doute par quelques journalistes pressés ou à la recherche d'un scoop, la sécurité d'une carte à puce est une merveille de technologie basé sur les techniques les plus abouties en matière de sécurité. Toutefois, même si certains acteurs du monde de la carte (*e.g.* le secteur bancaire) par une mauvaise intégration et une mauvaise utilisation de la carte à puce dans un système global ont pu mettre à mal son crédit, elle jouit encore de la réputation d'être le périphérique le plus sécurisé au monde.

Dans ce domaine de la sécurité, les fabricants de produits des Technologies de l'Information et les attaquants se livrent une véritable guerre. Et comme dans toutes les guerres, on assiste à une escalade des protections. Si aujourd'hui son niveau de sécurité est si élevé et si reconnu c'est grâce à une multitude de caractéristiques sécuritaires que nous allons détailler ci-après.

1.7.1 Au niveau physique

Le premier niveau de sécurité d'une carte réside dans son apparence.

En effet, quel commerçant accepterait un vulgaire morceau de plastique, ou pire encore, une palette avec un connecteur ISO 7816 relié à un ordinateur pour se faire payer ? Les méthodes de fabrication de telles cartes sont décrites dans les livres de Patrick Gueulle [134, 135]. Les contrefacteurs doivent donc être capables de reproduire des cartes assez fidèles d'aspect. Or, pour rendre leur tâche un peu plus ardue, les encarteurs utilisent des technologies d'impression sophistiquées lors de la fabrication du corps de carte. Il peut y avoir différentes couches d'impression à diverses profondeurs dans le plastique du corps de carte. Souvent le corps de carte comprend aussi un hologramme et il possède un embossage des informations relatives au porteur.

Mais bien souvent, ces techniques ne suffisent pas à protéger des malfaçons les commerçants peu attentifs. C'est pourquoi la carte intègre beaucoup d'autres sécurités matérielles et logicielles que nous examinerons dans les prochaines sections.

Nous verrons dans la section 9 que certaines attaques nécessitent un accès direct au *microchip*. En conséquence, afin de gêner l'ouverture (*i.e.* l'opération d'extraction du micromodule du corps de carte) différents procédés d'encollage du micromodule ont été développés. Le matériau constituant le corps de carte peut aussi être de différentes natures selon le niveau de résistance souhaité. Ces différentes protections visent à faire échouer les attaques chimiques d'extraction du micromodule. Les constructeurs jouent aussi sur le design du microcontact pour rendre la puce plus fragile à l'extraction.

Cet ensemble de techniques de défense, bien que contournables par des spécialistes de la micro-électronique et des matériaux, est une première barrière assez difficile à franchir pour le non-initié. Le but des fabricants est de pousser l'attaquant à endommager la puce lors de l'ouverture, la rendant ainsi inutilisable.

Enfin nous concluons cette partie en rappelant que même si les normes ISO 7816 sont principalement fonctionnelles, elles assurent aussi un certain niveau de sécurité au travers de la multitude de tests de caractérisation effectués. On pensera tout particulièrement à l'ISO 7816-1 qui porte sur :

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

1.7.2 Au niveau matériel

Le second niveau de sécurité d'une carte à puce se situe au niveau du matériel. C'est lui qui est à la base de la pyramide sécuritaire que constitue une carte à puce. S'il peut être pris en défaut, alors c'est tout l'édifice qui s'écroule. Les constructeurs de puces ont donc rivalisé d'ingéniosité pour assurer un haut degré de sécurité.

Une des premières sécurités a été d'introduire un numéro de série unique pour chaque pièce. Puis, avec le développement de puces de plus en plus complexes, les systèmes d'exploitation devenaient de plus en plus conséquents et donc sujets à de plus en plus de bugs. Aussi, les constructeurs se sont mis à intégrer des mémoire de type PROM (mémoire persistante) qui permettaient de corriger les problèmes en ajoutant du code aux bonnes adresses mémoire. Nous avons déjà vu toutes les propriétés de ces mémoires dans le cadre de la section 1.4.

Comme nous l'avons évoqué auparavant certaines attaques nécessitent un accès direct à la puce. Dès lors, les fondeurs sont mis à intégrer des détecteurs sur la carte afin de déterminer son utilisation dans un environnement hostile. Par exemple, on peut trouver sur une carte des détecteurs de lumière ou d'autres rayonnements, des détecteurs de température, des détecteurs de tension et de fréquence.

Tous ces mécanismes même s'ils ne sont pas garants de l'invulnérabilité de la puce, ont pour but de rendre le travail d'un attaquant plus difficile. En effet, l'attaquant ne disposant pas des compétences et du matériel nécessaires à la désactivation de ces détecteurs devrait voir ses attaques échouer. À l'inverse, un attaquant qualifié pourra, lui, passer outre ces dispositifs de protection et avoir un accès direct à la puce.

Comme nous le verrons au chapitre 9, parmi les attaques que pourra réaliser un individu mal-intentionné on trouve une observation des émissions électromagnétiques. Afin d'endiguer ce type d'attaque, les fabricants avaient, dans un premier temps, réalisé un blindage physique de leur composant en ajoutant une grille de protection (*cf.* FIG. 8) au mille-feuilles que représente une puce. Ainsi, l'attaquant ne pouvait plus avoir un accès direct au circuit. Seulement, l'amélioration des techniques de micro-électronique permet aujourd'hui de retirer la grille. Les fabricants rendent maintenant leur bouclier actif (*i.e.* la grille est à une certaine tension) et lorsqu'il n'est plus actif la puce se verrouille. Une fois encore, des mécanismes permettant de contourner cette protection ont été inventés.

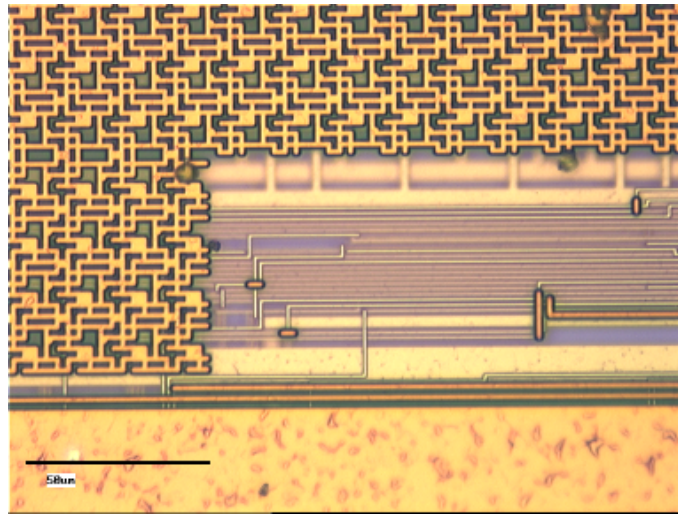


FIG. 8 – Grille de protection en microscopie optique (source : SERMA Technologies).

Parmi les autres dispositifs de protection qu'on trouve sur une carte il y a le chiffrement des données sur les bus et dans les mémoires, mais aussi l'enfouissement des bus sensibles dans les différents niveaux de métallisation, la pose de leurres (*i.e.* pistes reliées à aucune autre), la redondance de certaines fonctionnalités matérielles et la dissémination de l'information. Ce dernier point signifie que des octets situés à des adresses mémoires consécutives ne sont pas forcément situés à des positions consécutives sur le circuit.

Deux autres mécanismes sécuritaires fondamentaux sur la carte sont le co-processeur cryptographique qui accélère les calculs cryptographiques et rend leur observation beaucoup plus difficile, ainsi qu'un vrai générateur de nombres aléatoires pour tirer les aléas nécessaires dans quasiment tous les algorithmes cryptographiques.

Enfin, nous terminerons ici notre catalogue non exhaustif en citant le mécanisme de pompe de charge qui sert à lisser la consommation en courant. En effet, comme nous le verrons dans le chapitre 9, certaines attaques consistent à analyser la consommation de la puce pour en déduire des informations.

1.7.3 Au niveau logiciel

Le troisième niveau de sécurité des cartes à puce est réalisé par le logiciel et plus particulièrement par le système d'exploitation. Le système implante souvent une politique très stricte de contrôle d'accès aux données. L'accès à des données nécessite la vérification de règles qui dépendent des opérations à réaliser. Le système d'exploitation assure aussi l'intégrité des données via des mécanismes de *checksum* sur toute ou partie de la mémoire mais aussi via des mécanismes de transaction et d'atomicité des opérations. C'est encore lui qui sécurise les entrées/sorties, en les chiffrant par exemple. Il peut également implanter des mécanismes de migration du code et des données dans les mémoires afin d'empêcher un attaquant de les localiser. Sur certaines puces le système d'exploitation peut aussi changer

les fréquences de travail pour perturber les observations de l'attaquant mais il peut également assurer qu'une suite d'opérations va s'effectuer en un nombre de cycles constant indépendamment des données traitées afin que l'observateur ne puisse déterminer ces données. Nous décrirons une attaque qui peut être réalisée en l'absence d'un tel mécanisme dans la section 9.1. Pour résumer, le système d'exploitation est essentiel pour assurer la sécurité des applications qui feront appel à ses services.

La sécurité apportée par le logiciel est aussi la conséquence de l'application de techniques de programmation sécurisée. Ainsi, tous les grands développeurs d'applications ou de systèmes d'exploitations pour carte à puce disposent de guides sécuritaires de programmation, à l'état de l'art des attaques, voire même en avance, et qui décrivent les règles de bonne programmation (*e.g.* décrémentation du compteur d'essais du PIN avant de faire la comparaison, description des informations nécessaires pour restaurer les états une transaction échouée, etc.).

1.7.4 Au niveau de l'environnement

La confiance dans la carte à puce vient aussi de ce que l'environnement de sa chaîne de production (*i.e.* de fabrication, d'assemblage, de tests, etc.) est physiquement sécurisée. Ainsi, un individu malveillant ne devrait pas pouvoir s'introduire par une porte dérobée ni dans le code ni dans le matériel, mais il ne devrait pas non plus pouvoir récupérer des échantillons (*e.g.* dans les poubelles). En effet, le vol d'échantillons est très problématique puisqu'il peut par exemple permettre à l'attaquant de se familiariser avec les procédés d'ouverture de carte et donc de réussir l'opération sur une carte complète.

Mais pire encore, le voleur peut dérober directement le micromodule et avoir ainsi un accès direct à la puce. Heureusement, les personnels travaillant dans ces milieux ont souvent passé, avant d'être embauchés, une série de tests moraux mais ont aussi subi des enquêtes plus ou moins approfondies en fonction du poste occupé. De plus, les environnements sont régulièrement audités, que cela soit dans le cadre d'évaluation sécuritaire (*cf.* chapitre 5) ou dans le cadre d'audit interne, assurant ainsi un peu plus la sécurité des sites et donc de la carte à puce.

1.8 Les applications de la carte à puce

Les principaux secteurs qui utilisent actuellement la carte à puce [141] sont :

- l'industrie des télécommunications avec, par exemple, les cartes téléphoniques prépayées (comme par exemple les cartes téléphoniques classiques qui sont des cartes à mémoires avec contact) ou bien avec les cartes SIMs insérées dans les téléphones GSM ;
- l'industrie bancaire et monétaire avec les cartes de crédits (*e.g.* Europay, MasterCard, Visa) et les porte-monnaie électroniques (*e.g.* Monéo) ;
- le secteur de la santé (*e.g.* la carte Vitale) ;
- l'industrie audiovisuelle avec la télévision à péage ;

- l’industrie du transport avec les cartes sans contact pour les transports en commun, ou pour le transport routier le remplacement des chronotachygraphes par des cartes à puce ;
- l’industrie du contrôle d’accès physique des personnes aux locaux utilise de plus en plus la carte sans contact ;

Mais de nouvelles applications pour la carte à puce sont aujourd’hui à l’étude pour :

- l’authentification (*e.g.* à des sites sur l’Internet) ;
- les applications de fidélisation (*e.g.* l’accumulation de *miles* de voyage en avion gratuits à chaque transaction) ;
- les jeux comme le *pocket-gaming* (qui consiste en des terminaux de poche – comme le deviennent les téléphones ou les consoles portables – pour jouer par exemple à des jeux de casino).

Le domaine principal qui permettra à la carte à puce de s’imposer est l’Internet qui au travers des e-services exigent de plus en plus fréquemment une identification et une sécurisation des transactions. Ainsi, la carte peut être utilisée pour ces services en stockant par exemple les clefs servant à l’authentification et à la sécurisation des communications. Quelques exemples de e-services sont :

- le e-commerce ;
- la banque distante ;
- le e-courrier ;
- le télétravail.

Pour notre part, nous présenterons une application de la carte à puce dans un domaine un peu inattendu au chapitre 13 puisque nous l’utiliserons comme nœud de calcul sécurisé.

1.9 Les cartes multi-applicatives

Dans cette section nous traiterons les cartes multi-applicatives les plus connues exceptée la Java Card qui sera l’objet du prochain chapitre.

Une carte multi-applicative est une carte capable d’embarquer plusieurs applications. Ces applications ne s’exécutent pas de façon concurrente car les systèmes d’exploitation des cartes à puce possèdent une unique *thread*.

Les deux principaux standards de cartes multi-applicatives sont Java Card [93, 169, 170, 171] et MULTOS [176]. Mais, récemment, deux nouvelles technologies les ont rejoint avec la naissance de Smartcard.NET [142, 144] de Hive-Minded et la MultiApplication BasicCard ZC6.5 [207] de ZeitControl. Windows for Smart Cards de Microsoft peut également être cité même s’il semble aujourd’hui défunt. Java Card a été le premier standard mais ils partagent tous les mêmes concepts. L’architecture d’une carte implantant ces technologies (*cf.* FIG. 9) consiste en :

- un système d’exploitation embarqué qui supporte le chargement et l’exécution de plusieurs applications (chacune à leur tour). Ce système d’exploitation est en fait constitué d’un environnement d’exécution et d’une machine virtuelle (VM) qui mettent en place des caractéristiques sécuritaires (*e.g.* un pare-feu entre les applications).

- des applications qui sont interprétées par la VM.

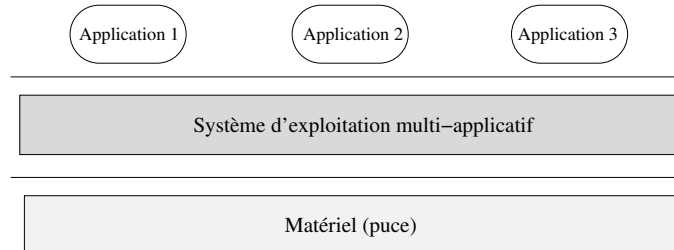


FIG. 9 – Architecture d'une carte à puce multi-applicative.

1.9.1 MULTOS

MULTOS[176] est la première carte à puce ouverte, hautement sécurisée et possédant un système d'exploitation multi-applicatif.

Un consortium industriel, nommé MAOSCO, est chargé de promouvoir MULTOS comme système d'exploitation multi-applicatif pour les cartes à puce, de gérer les spécifications de MULTOS, de fournir les licences et les certifications de services MULTOS. Les membres fondateur du consortium MAOSCO sont : American Express, Dai Nippon Printing, Mondex International Ltd, Siemens, Fujitsu, Hitachi, Motorola, MasterCard International, Keycorp.

Les éléments clés de la plate-forme MULTOS sont :

- une architecture hautement sécurisée ;
- la possibilité d'avoir plusieurs applications sur la même carte ;
- l'indépendance des applications par rapport à la plate-forme matérielle ;
- le chargement et l'effacement des applications durant la vie de la carte ;
- la compatibilité avec les standards industriels ISO 7816 et EMV [12].

Pour le développement des applications, Mondex International Ltd a mis en place :

- un langage optimisé pour les cartes à puce : MEL (Multos Executable Language) basé sur les P-codes Pascal et la machine virtuelle décrite par David A. WATT [203] ;
- des spécifications fixes pour les APIs MULTOS. En d'autres termes, les développeurs de cartes MULTOS ne sont pas libres d'ajouter leur propres extensions pour des raisons d'interopérabilité. Ainsi, si un service MULTOS optionnel est présent sur une carte, alors s'il est aussi présent sur une autre carte, il possédera exactement la même interface. Sur ce point, MULTOS diffère beaucoup de Java Card puisque Java Card permet l'ajout de composants propres au développeur de cartes. Par exemple, avant Java Card 2.2 on se retrouvait ainsi avec un manque d'interopérabilité lorsque l'on voulait utiliser les mécanismes de ramasse-miettes fournis par différents constructeurs car leurs interfaces n'étaient pas homogènes.

Les applications pour MULTOS sont en général programmées soit en C soit en Java, mais, comme il existe une multitude de compilateurs, on peut aussi les programmer en

Basic ou en Modula-2. Il faut noter qu'avec MULTOS il est même possible d'inclure du code assembleur dans une application programmée dans un langage de haut niveau. C'est d'ailleurs la seule carte multi-applicative qui le permet. En bref, il existe donc plusieurs environnements de développement pour la plate-forme MULTOS.

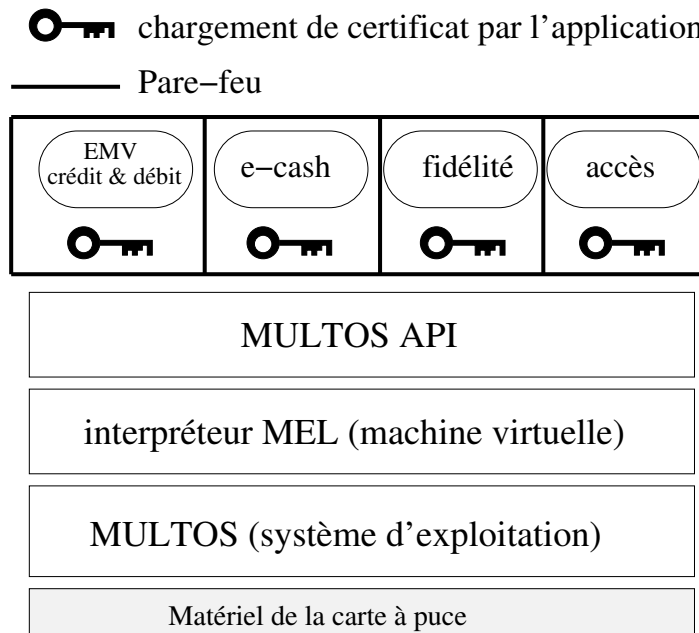


FIG. 10 – Architecture de MULTOS.

MULTOS s'exécute sur la puce de la carte. A l'installation d'une nouvelle application, la carte à puce MULTOS vérifie la validité de l'application qui a été envoyée, alloue au programme un espace mémoire protégé grâce au pare-feu. Chaque nouveau service ou application reste aussi rigoureusement séparé des autres programmes déjà sur la carte afin qu'aucun d'eux, en cas de dysfonctionnement, ne puisse interférer avec un autre programme.

Outre un accès à son propre espace mémoire, l'application peut aussi accéder à deux autres segments de données en mémoire volatile : un dynamique et un public. Le segment de données dynamique est privé et contient la pile et d'autres données de session alors que le segment de données public contient lui le buffer de communication pour gérer le trafic entre le terminal et les applications, mais aussi entre les applications. En effet, les applications peuvent communiquer entre elles et s'échanger des données au moyen des APDUs simplifiant considérablement l'écriture d'une application par rapport à Java Card. C'est un système simple et efficace qui permet la migration d'une application du terminal vers la carte sans nécessiter beaucoup de développement puisque le protocole de communication reste le même.

Virtuellement aucune structure n'est imposée dans le modèle de programmation MULTOS. Un peu dans l'esprit Unix, une application consiste simplement en une fonction `main` et elle peut ensuite définir ses propres fonctions et faire appel à des primitives systèmes.

On notera d'ailleurs que la bibliothèque C pour MULTOS inclut beaucoup de fonctions de la bibliothèque standard C (*e.g.* gestion du tas, manipulation sur les chaînes). Bien

évidemment on trouve aussi beaucoup de primitives relatives à la cryptographie.

MULTOS est pour le moment implanté sur des cartes à puce par les sociétés :

- DNP avec des puces fabriquées par Renesas (anciennement Hitachi) ;
- Keycorp avec des puces fabriquées par Infineon (anciennement Siemens) et Philips.

Il existe déjà des dizaines d'applications pour MULTOS écrites par les fabricants de cartes à puce comme Axalto, Gemplus, Orga, etc. L'implantation réalisée par Keycorp est l'implantation complète de MULTOS 4.0 basée sur des puces Infineon et Philips. Cette implantation est certifiée au niveau E6 des critères ITSEC (Information Technology Security Evaluation Criteria) faisant ainsi de ce produit une des cartes les plus sécurisées du monde.

Les caractéristiques de cette carte sont une implantation complète de MULTOS, la disponibilité de 16, 32 ou 48 Ko d'EEPROM libre pour les applications, l'intégration des standards de sécurité de MULTOS 4.

MULTOS est aujourd'hui un standard bien établi qui concurrence Java Card.

1.9.2 Windows for Smart Cards de Microsoft

La WfSC, proposée par Microsoft, est une carte multi-applicative orientée authentification compatible ISO 7816 . Elle a été développée en 1998 (par Scott B. Guthery [147]) et aujourd'hui elle n'est quasiment plus utilisée.

La philosophie de cette carte est d'adapter le niveau de compétence et le modèle de programmation de la carte à puce pour la rendre accessible aux développeurs d'applications plutôt que de les forcer à acquérir de nouvelles compétences et à apprendre de nouveaux modèles de programmation pour utiliser la carte à puce. Évidemment, il y a eu tout de même des petites adaptations à cause des ressources limitées de la carte.

Cette carte embarque un vrai système de fichiers FAT et parce que les cartes à puce sont utilisées internationalement, les noms de fichiers sont représentés en Unicode plutôt qu'en ASCII.

Elle possède aussi un contrôle d'accès très fin puisqu'elle utilise les ACLs (*i.e.* Access Control List) qui existent sur beaucoup de systèmes Unix mais aussi sous Windows NT.

La machine virtuelle embarquée est une version compacte de l'Intel 8048 et l'API proposée est une version considérablement réduite de l'API Windows (Win32).

Elle peut être utilisée selon deux modes, un mode système de fichiers tout simple avec un accès par le jeu de commandes standard ISO7816 ou un mode multi-applicatif qui est celui qui nous intéresse ici.

Un programme compilé pour WfSC consiste en deux fichiers, un fichier de *bytecodes* (.RTE) et un fichier pour les données (.DAT). Cette séparation du code et des données en plus d'être une pratique courante de bonne programmation permet de mettre facilement à jour les applications tout en conservant leurs données. Ce n'est pas aussi évident, par exemple, avec Java Card. Sur cette plate-forme le partage de données se fait avec les ACL au travers du système de fichiers. Au départ, Microsoft avait complètement intégré le développement des applications dans son outil Visual Studio. En fait, sous cet environnement un projet WfSC était simplement un autre type de projet Visual Basic ou Visual

C++. L'environnement permettait grâce à un simulateur de déboguer pas à pas à la fois l'application sur la carte mais aussi l'application sur le PC en lançant deux instances de l'environnement. Après cette étape de débogage, on pouvait déployer son application sur la carte avec l'outil en passant différentes étapes pour un chargement sécurisé. Microsoft fournissait aussi des applications simples et prêtes à être déployées notamment pour faire de l'authentification sous Windows. Depuis, Microsoft a abandonné le support de WfSC dans Visual Studio et il faut utiliser d'autres environnements de développement.

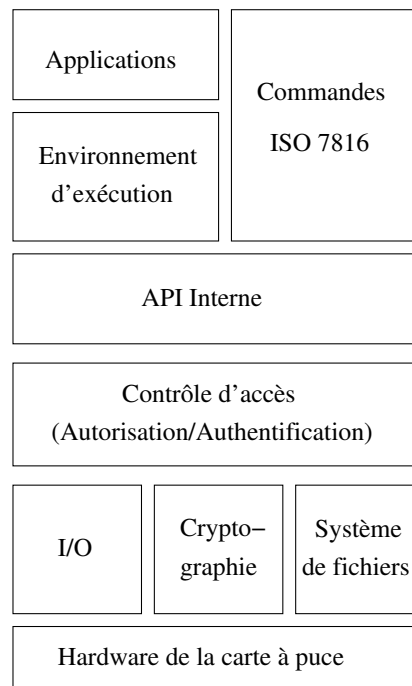


FIG. 11 – Architecture de Windows for Smart Card.

Gemplus, avec l'annonce de GemShield, a été la première société à réaliser une carte supportant Windows for Smart Card. Depuis, quelques autres sociétés telles NexSmart technology ou DAWAR technologies, ont développé des WinCards. La carte de NexSmart technology, Windows Powered Smart Card With Crypto, possède les caractéristiques suivantes :

- un système d'exploitation Windows Powered Smart Card 1.1 ;
- un processeur RISC 8 bits ;
- un coprocesseur cryptographique 1024 bits ;
- 32 Ko de mémoire FLASH ;
- 32 Ko d'EEPROM ;
- 1 Ko de RAM.

1.9.3 The ZeitControl BasicCard

Même si les BasicCard existent depuis 1996 c'est seulement en avril 2004 que ZeitControl a sorti sa première carte multi-applicative la *MultiApplication BasicCard ZC6.5* [207].

À l'instar des cartes multi-applicatives déjà présentées, les ZeitControl BasicCard possèdent une machine virtuelle, mais celle-ci se différencie des autres en étant la seule à supporter les nombres flottants.

Cette carte se programme en ZeitControl Basic qui contient la plupart des constructions habituelles du langage Basic telles que les chaînes et les fonctions pour les manipuler, les tableaux, les données définies par l'utilisateur, etc. Elle possède aussi un système de fichiers un peu similaire à celui de la FAT et propose une API Basic classique pour son utilisation. Un peu comme pour la WfSC, le partage de données se fait au travers du système de fichiers avec un contrôle d'accès par groupe d'applications et les applications ne respectant pas les règles associées aux fichiers ne peuvent accéder qu'à leurs propres données.

L'environnement de développement proposé par ZeitControl est complet et bien pensé. En effet, il possède deux fenêtres adjacentes, une pour développer la partie embarquée sur la carte et une pour développer la partie résidant du côté du terminal qui permet de déboguer facilement les applications. Il permet également de charger l'application sur la carte et une fois cette étape effectuée, on peut encore déboguer son application du côté terminal en communiquant directement avec la carte.

Par ailleurs, cet environnement est gratuit et peut être téléchargé sur leur site web. Pour les personnes souhaitant s'initier au monde des cartes à puce il a l'énorme avantage de ne nécessiter aucun matériel pour commencer à développer (*i.e.* ni lecteur, ni cartes).

En outre, l'énorme avantage de ces cartes est leur très faible coût.

1.9.4 Smartcard.NET

Cette plate-forme ne marque pas le retour de Microsoft dans le monde de la carte. En fait, c'est une pousse issue de Berkeley, Hive-Minded, qui a lancé Smartcard.NET[142, 144] avec l'idée que si quelqu'un peut écrire un programme pour Windows, il peut le faire pour une carte à puce.

Pour cela, l'architecture proposée est très proche de celle de Java Card mais avec une différence de philosophie énorme. En effet, nous verrons dans le prochain chapitre que l'API Java Card n'est pas un sous-ensemble de l'API Java mais vraiment quelque chose de différent. Leur objectif a au contraire été de fournir un vrai sous-ensemble de l'API .NET pour les cartes à puce.

Ainsi, leur machine virtuelle supporte tous les types exceptés les nombres flottants, la vérification de code, les entiers sur 64 bits et le *Remoting* (*i.e.* les RPC dans .NET). Leur API inclut tous les types et toutes les méthodes requis par le langage pour supporter .NET (y compris les chaînes). Elle gère également les entrées/sorties, la cryptographie, les synchronisations entre *threads*, les collections, etc.

Un des gros avantages de cette carte est que .NET supporte beaucoup de langages

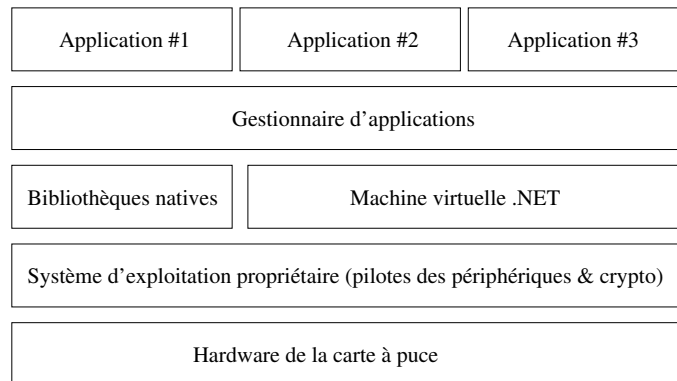


FIG. 12 – Architecture de la Smartcard.NET.

de programmation. En conséquence, on peut écrire des applications en C#, VB.NET, Jscript.NET, J#, etc.

La plate-forme Nectar (*i.e.* l'implantation de référence de SmartCard.NET) développée par Hive-Minded requiert environ 60 Ko pour la machine virtuelle et les APIs et un minimum de 128 octets de RAM mais 2 Ko sont recommandés.

L'objectif de Hive-Minded est de vendre des licences logicielles aux fabricants de cartes pour sa plate-forme et comme Nectar a été écrit en C ANSI standard, il doit être facilement portable sur plusieurs puces.

Il propose d'ailleurs une suite de plus de 11000 tests pour vérifier la bonne implantation de SmartCard.NET et des applications classiques de démonstration (*e.g.* système de fichier ISO 7816, implantation d'une carte SIM). Hive-Minded propose aussi aux développeurs d'application pour SmartCard.NET une chaîne de développement intégrée dans Visual Studio.NET rendant très simple la construction, le développement et le déploiement de leurs applications.

Smartcard.NET bien que très récent pourrait bien rejoindre très rapidement Java Card et MULTOS parmi les poids-lourds de la multi-application.

Chapitre 2

La technologie Java Card

2.1 Présentation

La technologie Java Card permet aux cartes à puce et à d'autres périphériques à mémoire limitée de faire fonctionner des applications écrites en langage Java. Une Java Card est donc une carte à puce sur laquelle il est possible de charger et d'exécuter des programmes Java, appelés *applets*. Contrairement aux cartes à puce traditionnelles, les programmes exécutés par la carte ne sont pas forcément fournis par l'émetteur de la carte.

De manière synthétique, on peut dire que la technologie Java Card, telle qu'elle est vendue par Sun microsystems, définit une plate-forme sécurisée pour cartes à puce, portable et multi-application qui intègre beaucoup des avantages du langage Java.

2.2 Historique

En Novembre 1996, un groupe d'ingénieurs du centre de production de Schlumberger à Austin au Texas cherche à simplifier la programmation des cartes à puce tout en préservant la sécurité. Le langage de programmation Java apparaît comme la solution. Schlumberger devint alors la première entreprise à acquérir une licence en proposant un *draft* de quatre pages (la spécification Java Card 1.0).

En Février 1997, Bull et Gemplus se joignent à Schlumberger pour cofonder le Java Card Forum [29]. Le but de ce consortium industriel est d'identifier et de résoudre les problèmes de la technologie Java Card en proposant des spécifications à JavaSoft (la division de Sun microsystems à qui appartient Java Card). Il a également pour objectif de promouvoir des APIs Java Card afin de permettre son adoption par l'industrie de la carte à puce.

Aujourd'hui, le Java Card Forum regroupe les fabricants de cartes, Sun et des utilisateurs. À partir de ce moment, avec un large soutien de l'industrie, Sun microsystems va acquérir la société Integrity Arts afin de développer la technologie Java Card comme plate-forme de la technologie Java pour les cartes à puce et les périphériques comportant peu de mémoire. Cette orientation va être un gros avantage pour Gemplus qui était alors spécialisé dans le développement de machine virtuelle et de systèmes d'exploitation pour

les cartes à puce.

En Novembre 1997, Sun microsystems présente les spécifications Java Card 2.0 qui consistent en un sous-ensemble du langage et de la machine virtuelle Java. Elles définissent des concepts de programmation et des APIs très différentes de celles de la version 1.0. Il n'y a encore rien sur le format des *applets* téléchargeables.

En Mars 1999 sort la version 2.1 des spécifications Java Card. Elle se compose en fait de trois spécifications :

- la *Java Card 2.1 API Specification*,
- la *Java Card 2.1 Runtime Environment Specification*,
- la *Java Card 2.1 Virtual Machine Specification*.

Dans cette nouvelle version, il y a eu quelques modifications des APIs notamment au niveau de la cryptographie et des exceptions. L'environnement d'exécution des *applets* a également été standardisé. Mais la contribution la plus significative de la version 2.1 est la définition explicite de la machine virtuelle de la Java Card (JCVM : Java Card Virtual Machine) et du format des *applets*, permettant ainsi une vraie inter-opérabilité.

En Mai 2000, de petites corrections et clarifications de la version précédente ont abouti à la version 2.1.1 des spécifications.

En Octobre 2000, ce sont près de quarante entreprises qui ont acquis la licence d'exploitation de la technologie Java Card : AOL, American Express, CP8, Centra Systems, Citicorp, Dallas Semiconductor, Datacard/Platform 7, ETRI, Fujitsu, Gemplus, Giesecke & Devrient, Goldkey, Hitachi, IBM, InCard, Inside Technologies, IPM Group, ITRI, IRIS Automation, Keycorp, Matsushita, Mitsubishi, Motorola, NCT/Advancel, NEC, NTT Corporation, Oberthur, Orga, Schlumberger, Sermepa, Setec, ST Microelectronics, Toppan Printing Co. Ltd., Toshiba, Visa International, Watch Data, Wave Systems et Xilinx.

Puis, en Juin 2002, Sun microsystems publie la version 2.2 des spécifications dont les principales nouveautés sont la prise en charge des canaux logiques et de l'appel de méthode à distance RMI [133]. Elle ajoute aussi des considérations sur la façon d'installer et d'effacer des applications sur la carte et quelques nouveaux algorithmes cryptographiques.

Enfin, en Octobre 2003, la publication de la version 2.2.1 [169, 170, 171] corrige l'API de la version précédente et apporte quelques clarifications mais aucune nouveauté.

Aujourd'hui, en Août 2004, les licenciés sont ARM, Aspects, CCL/ITRI, Fujitsu, Gemplus, Giesecke & Devrient GmbH, HiSmarTech, I'M Technologies Ltd., IBM, InCard, KEB-Technology, Logos Smart Card, Oberthur, ORGA Kartensysteme, Schlumberger, Sermepa, Setec, Sharp, Smart Card Laboratory Inc., SSP Solutions, Inc., STMicroelectronics, Toppan Printing, Trusted Logic. Ils sont donc au nombre de 23 ayant accepté de figurer sur la liste officielle de Sun microsystems. On pourra constater que parmi les acteurs présents en 2000 sur ce marché beaucoup n'y sont plus mais que quelques petits nouveaux viennent prendre la relève.

2.3 Les avantages de la technologie Java Card

Les avantages de la technologie Java Card pour les développeurs d'applications pour cartes à puce sont pour l'essentiel les mêmes que ceux que Java a pu apporter par rapport aux langages de programmation classiques.

Ainsi, Java Card a permis de faciliter le développement des applications grâce :

- à la programmation orientée objet offerte par Java (contre l'assembleur auparavant) ;
- à la possibilité d'utiliser les environnements de développement existants pour Java ;
- à une plate-forme ouverte qui définit des APIs et un environnement d'exécution standardisé ;
- à une plate-forme qui encapsule la complexité sous-jacente et les détails du système des cartes à puce.

Dès lors, les développeurs peuvent concentrer totalement leurs efforts sur les *applets* ou les bibliothèques qu'ils créent, réduisant ainsi le temps de développement.

La sécurité est également au rendez-vous grâce :

- à plusieurs niveaux de contrôle d'accès aux méthodes et aux variables (**public**, **protected**, **private**) ;
- à un langage fortement typé ;
- à l'impossibilité de construire des pointeurs qui pourraient être utilisés dans des programmes malicieux afin d'espionner le contenu de la mémoire ;
- à un pare-feu qui sépare les *applets* dans la plate-forme.

Tous ces mécanismes empêchent un programme hostile de créer des dommages sur les autres parties du système.

Évidemment, un des buts premiers de la technologie Java Card a été d'apporter l'indépendance des applications par rapport au matériel en s'inspirant des principes du langage Java où une application est exécutable sur n'importe quel système puisque son code pré-compilé en *bytecode* est exécuté par la machine virtuelle Java. Cette portabilité permet d'écrire du code qui fonctionnera sur n'importe quel microprocesseur de carte à puce (*i.e. Write Once, Run Anywhere*).

Une des plus grosses nouveautés qu'apportait Java Card était la capacité de stockage et de gestion de plusieurs applications. En effet, les Java Cards étaient parmi les premières technologies de cartes à pouvoir héberger sur une même carte des applications provenant de fournisseurs et de domaines d'utilisation totalement différents. Ainsi, une fois la carte fournie à son propriétaire, il est encore possible de télécharger des *applets*. Les applications de la Java Card peuvent donc être continuellement mises à jour sans avoir besoin de changer de carte. Par ailleurs, un mécanisme de pare-feu interdit l'accès d'une *applet* à une autre si celui-ci n'est pas explicitement permis.

Enfin, la technologie Java Card a été conçue dans un esprit de compatibilité avec les standards existants pour les cartes à puce. Ainsi, elle est basée sur le standard ISO 7816 [108] ce qui lui permet de supporter des applications compatibles ISO 7816. Par conséquent, les *applets* peuvent interagir non seulement avec toutes les cartes à puce Java mais aussi avec les CADs existants.

Si nous avons présenté ici les avantages de la technologie Java Card, nous ne manquerons pas de présenter ses inconvénients dans la dernière section de ce chapitre lorsque le lecteur sera un peu plus initié aux concepts sous-jacents.

2.4 L'architecture

Comme nous l'avons vu au chapitre précédent, les configurations mémoires des cartes à puce sont extrêmement réduites, de l'ordre de 2 Ko de RAM, 32 Ko d'EEPROM et de 32 Ko de ROM. C'est pourquoi, l'un des plus grands défis de la technologie Java Card a été de s'adapter à ces contraintes pour construire une carte Java tout en conservant suffisamment de place pour embarquer des applications.

La solution choisie a été de supporter seulement un sous-ensemble des caractéristiques du langage Java et de découper la machine virtuelle Java (JCVm) en deux parties : une partie qui s'exécute en dehors de la carte et une partie qui s'exécute sur la carte. La partie de la JCVm embarquée sur la carte comprend principalement l'interpréteur de *bytecode*. Ainsi, beaucoup de tâches ne sont plus réalisées à l'exécution mais sont déportées vers la partie de la machine virtuelle hors carte (*e.g.* le chargement de classes, la vérification de *bytecode*, la résolution de liens, l'optimisation, etc).

Afin de pallier aux problèmes de sécurité dus à l'absence de vérifieur embarqué, la technologie Java Card a ajouté à l'environnement d'exécution, le JCRE (Java Card Runtime Environment), un mécanisme sécuritaire (*i.e.* le pare-feu) qui permet une séparation claire entre le système de la carte et les applications, et entre les applications elles-même.

Ainsi, depuis la version 2.1 des spécifications, l'architecture de la Java Card suit le modèle présenté figure 13. Dans ce système, on peut considérer l'interpréteur de *bytecodes*

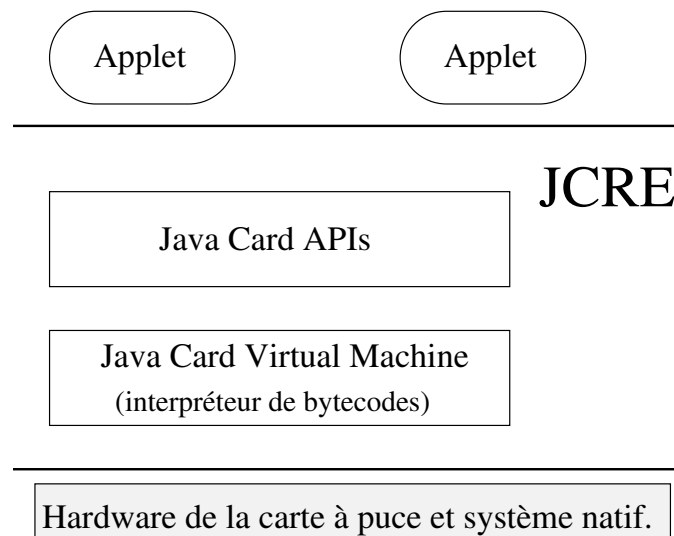


FIG. 13 – Architecture de la Java Card.

comme un processeur virtuel et le JCRE comme un système d'exploitation puisqu'il cache

la complexité sous-jacente du système des cartes. Les interfaces utilisateur de ces deux entités sont donc universelles et identiques pour toutes les cartes implantant la technologie Java Card. Les *applets* s'exécutent au-dessus de ce processeur virtuel et demandent des ressources et des services systèmes au JCRE à travers les APIs qui peuvent être vues comme des bibliothèques universelles présentes sur toutes les Java Cards.

Cette architecture de la technologie Java Card est définie par les trois spécifications suivantes :

- la *Java Card 2.2.1 Virtual Machine Specification* définit un sous-ensemble du langage de programmation Java et la définition de la machine virtuelle nécessaire pour les applications sur cartes à puce ;
- la *Java Card 2.1 Runtime Environment Specification* décrit précisément le comportement de l'exécution de la Java Card, comme la gestion de la mémoire, des *applets* et d'autres caractéristiques ;
- la *Java Card 2.1 API Specification* décrit l'ensemble des paquetages et des classes Java nécessaires à la programmation des cartes à puce mais aussi quelques extensions optionnelles.

2.4.1 Le langage, un sous-ensemble du langage Java

Afin d'adapter la technologie Java Card aux environnements à mémoire limitée, seules un certain nombre de caractéristiques bien choisies de Java ont été spécifiées. Le tableau 5 résume les caractéristiques importantes supportées, celles optionnelles et celles non supportées.

Caractéristiques supportées	Caractéristiques non supportées
▷ Types simples de donnée de petite taille : boolean, byte, short ▷ Tableaux à une dimension ▷ Paquetages, classes, interfaces et exceptions ▷ Caractéristiques orientées objet : héritage, méthodes virtuelles, surcharge et création dynamique d'objets, contrôle d'accès	▷ Types simples de donnée de grande taille : long, double, float ▷ Tableaux à plusieurs dimensions ▷ Caractères et chaînes ▷ Chargement dynamique de classe ▷ Security Manager ▷ Finalisation ▷ Threads ▷ Sérialisation d'objets ▷ Clonage d'objets
Caractéristiques optionnelles	
▷ Le mot clé int et le support des entiers sur 32 bits sont optionnels ▷ Ramasse-miettes	

TAB. 5 – Caractéristiques Java supportées, optionnelles et non supportées

Afin de bien comprendre les choix faits par les concepteurs de la technologie Java Card, nous allons détailler certains points du tableau 5.

Caractéristiques Java non supportées

- Chargement dynamique des classes.
Une implantation de la plate-forme Java Card ne doit pas charger les classes dynamiquement. Les classes sont soit incluses dans le masque à la fabrication, soit installées au travers d'un processus d'installation sur la carte après son émission. En conséquence, les programmes qui s'exécutent sur la carte doivent seulement se référer aux classes déjà existantes sur la carte. En effet, il n'y a aucune façon de télécharger des classes durant une exécution normale du code d'une application.
- **Security Manager.**
Le gestionnaire de sécurité sur la plate-forme Java Card diffère significativement de celui de la plate-forme Java. Dans la plate-forme Java, il y a une classe **Security Manager** (`java.lang.SecurityManager`) responsable des caractéristiques sécuritaires d'implantation qui fabrique des règles de décision pour autoriser des opérations. Dans la plate-forme Java Card, les règles de sécurités du langage sont implantées par la machine virtuelle.
- Finalisation.
La finalisation n'est pas nécessaire. `finalize()` ne sera pas automatiquement appelée par la JCVM, et le programme ne doit pas dépendre de ce comportement.
- Threads.
La JCVM ne peut pas supporter plusieurs *threads* de contrôle. Un programme Java Card ne peut pas utiliser la classe **Thread** ou n'importe quel mot clé du langage Java relatif aux *threads*.
- Clonage d'objet.
La plate-forme Java Card ne supporte pas le clonage d'objet. La classe **Object** de l'API Java Card n'implante pas la méthode `clone`, et ne fournit pas l'interface **Cloneable**.
- Le contrôle d'accès des paquetages Java.
Le langage Java Card supporte le contrôle d'accès aux paquetages comme défini dans le langage Java. Pourtant, il y a des cas non supportés tels que les suivants :
 - Si une classe implante une méthode avec un accès de visibilité **package**, une sous-classe ne peut pas surcharger la méthode et changer l'accès de visibilité de la méthode en **protected** ou **public**.
 - Une classe de visibilité **public** ne peut pas contenir un champ **public** ou **protected** de type référence à une classe de visibilité **package**.
 - Une classe de visibilité **public** ne peut pas contenir une méthode **public** ou **protected** avec un type de retour de type référence à une classe de visibilité **package**.
 - Une méthode **public** ou **protected** d'une classe **public** ne peut pas contenir un paramètre formel de type référence à une classe de visibilité **package**.
 - Une classe de visibilité **package** qui peut être héritée par une classe **public** ne peut pas définir de champs ou de méthodes **public** et **protected**.

- Une interface définie avec un accès de visibilité **package** qui est implémenté par une classe de visibilité **public** ne peut définir aucun champ.
- Une interface définie avec un accès de visibilité **package** ne peut pas être héritée par une interface avec un accès de visibilité **public**.

Le but de ces restrictions est de renforcer les règles de contrôle d'accès définies par le programmeur d'*applets* et d'éviter qu'une erreur d'inattention de sa part (*e.g.* l'attribution d'une mauvaise visibilité sur une méthode lors d'un héritage, etc.) n'aille à l'encontre des contrôles précédemment définis.

Caractéristiques Java supportées

- Paquetages.
La plate-forme Java Card suit les règles standards des paquetages de la plate-forme Java. Les classes des APIs Java Card sont écrites comme des fichiers sources Java qui incluent les désignations des paquetages. Les mécanismes de paquetages sont utilisés pour identifier et contrôler les accès aux classes, aux champs statiques et aux méthodes statiques. Exceptée la remarque faite ci-dessus sur le contrôle d'accès, les paquetages sur la plate-forme Java Card sont utilisés exactement comme sur la plate-forme Java.
- Création dynamique d'objet.
Les programmes de la plate-forme Java Card supportent la création dynamique d'objet, les instances de classes et les tableaux. Cette opération est réalisée comme d'habitude par l'opérateur **new**. Les objets sont alloués en dehors du tas. Comme souligné dans la remarque plus haut, la JCVm ne fournit pas nécessairement de ramasse-miettes. Tous les objets créés par la machine virtuelle peuvent donc, en l'absence d'un ramasse-miettes continuer à exister et à consommer des ressources même après qu'ils soient devenus inaccessibles.
- Méthodes virtuelles.
Comme les objets Java Card sont des objets du langage de programmation Java, l'invocation de méthodes virtuelles sur des objets dans un programme écrit pour la plate-forme Java Card est exactement comme dans un programme écrit pour la plate-forme Java. L'héritage est supporté, incluant l'utilisation du mot clé **super**.
- Interfaces.
Les classes Java Card peuvent définir ou implanter des interfaces comme dans le langage de programmation Java. L'invocation de méthodes sur les types d'interfaces fonctionnent comme attendu. La vérification de type et l'opérateur **instanceof** fonctionne aussi correctement avec les interfaces.
- Exceptions.
Les programmes Java Card peuvent définir, lancer et attraper des exceptions, comme les programmes Java. La classe **Throwable** et ses sous-classes importantes sont supportées. Quelques sous-classes de **Exception** et **Error** sont naturellement omises, ce sont celles dont les exceptions ne peuvent apparaître dans la plate-forme Java Card.

Caractéristiques Java optionnelles

- Ramasse-miettes.

La version 2.2.1 de la technologie Java Card offre un mécanisme optionnel de suppression des objets. Ainsi, les applications conçues pour fonctionner sur ces implantations peuvent utiliser ce nouveau service via la méthode appropriée de l'API. En revanche, ce service ne devrait être utilisé que pour la mise à jour de gros objets (*i.e.* avec parcimonie) tels que des certificats et des clés. Par ailleurs, cette mise à jour est atomique pour garder la contrainte de *pointer-safety*. Mais de façon générale, les développeurs d'applications doivent gérer eux-mêmes l'allocation des objets et ils doivent considérer que l'espace mémoire occupé par des objets inaccessibles ne sera pas nécessairement récupéré puisque seules quelques Java Card avancées fournissent le mécanisme de ramasse-miettes.

2.4.2 La machine virtuelle : JCVM

La machine virtuelle Java Card (JCVM) a une architecture comparable à celle de la machine virtuelle Java (JVM) ; la différence est que la JCVM est découpée en deux parties (*cf.* FIG. 14) :

- une partie embarquée sur la carte qui inclue l'interpréteur de *bytecodes* ;
- une partie hors-carte qui tourne sur une station de travail et qui comprend le convertisseur.

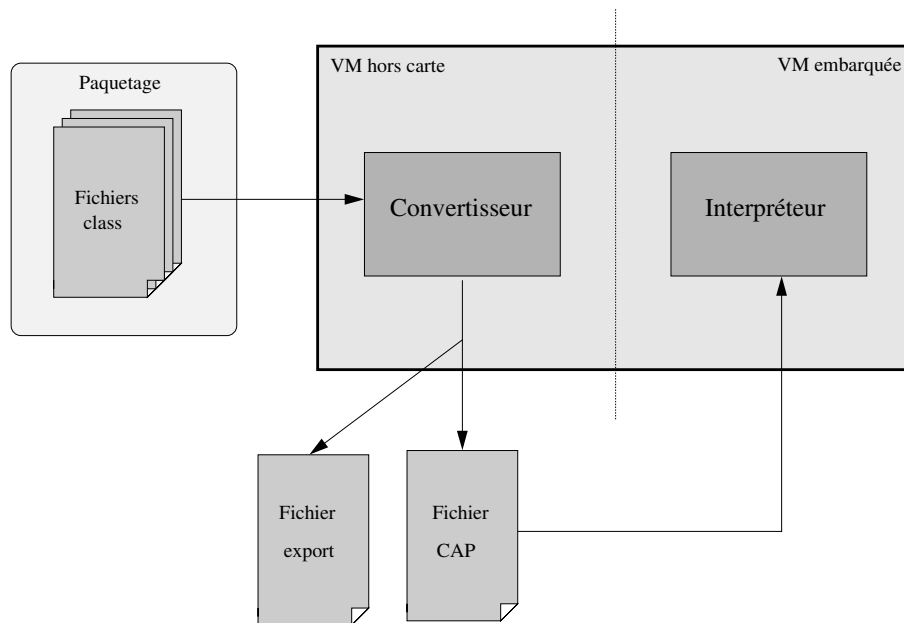


FIG. 14 – La machine virtuelle Java Card (JCVM).

Ensemble, ces deux parties implantent toutes les fonctions d'une machine virtuelle. Le convertisseur charge et traite les fichiers `class` qui composent le paquetage Java à

convertir et produit en sortie un fichier **CAP** (Converted APplet) qui est une représentation binaire exécutable des classes. En plus de la création du fichier **CAP**, le convertisseur génère également un fichier **export** représentant les APIs publiques du paquetage converti.

Par ailleurs, toute caractéristique non supportée du langage utilisée dans une *applet* est détectée par le convertisseur et conduit à un échec du processus de conversion. Si la conversion réussit, le fichier **CAP** peut alors être chargé sur la carte à puce afin d'y être exécuté par l'interpréteur.

À cause du découpage de la JCVM, la plate-forme est distribuée entre la carte à puce et la machine de développement, et ce, dans le temps et dans l'espace.

Fichier CAP et fichier export

La technologie Java Card introduit deux nouveaux formats de fichiers binaires qui apportent l'indépendance de la plate-forme de développement, de la distribution et de l'exécution de logiciels Java Card. Ce sont :

- le fichier **CAP** ;
- le fichier **export**.

Un fichier **CAP** contient une représentation binaire exécutable des classes d'un paquetage Java. Ce fichier **CAP** est une archive utilisant le format de fichier JAR [26] pour stocker un ensemble de composants. Chacun de ces composants est stocké comme un fichier individuel dont le nom a pour extension **.cap**. Le format du fichier de chacun de ces composants suit le modèle suivant :

Étiquette	Taille	Données
-----------	--------	---------

TAB. 6 – Format de fichier d'un composant du paquetage

Chaque composant décrit un aspect du contenu d'un fichier **CAP**. Aujourd'hui avec les spécifications 2.2.1, il y a douze composants officiels dont trois sont optionnels. Mais, il existe aussi la possibilité de créer de nouveaux composants pour des besoins spécifiques des vendeurs. Voici la liste des composants officiels avec leur étiquette et une brève description :

- header (1) : contient le « nombre magique » et les numéros de versions de l'implantation de la machine virtuelle Java Card pouvant supporter le paquetage ;
- directory (2) : contient la liste des autres composants avec leur taille respective ;
- applet (3) (optionnel) : contient la liste des *applets* avec leur AID et une référence (*i.e.* un *offset*) sur la méthode **install()** (*i.e.* la première méthode appelée par l'interpréteur pour installer l'*applet*) ;
- import (4) : contient la liste des paquetages référencés dans ce paquetage ;
- constant pool (5) : contient la table de références aux classes, aux méthodes et aux champs, respectivement, dans les composants de classes, de méthodes et de champs ;
- class (6) : contient la table des classes et interfaces ;
- method (7) : contient le code des méthodes des classes ;

- static field (8) : contient les valeurs initiales de tous les champs statiques du paquetage ;
- reference location (9) : contient la liste des références relatives dans le code des méthodes à transformer en références absolues au chargement, en vue de procéder à des vérifications dans le code ;
- export (10) (optionnel) : contient la liste des éléments de ce paquetage que les autres paquetages peuvent référencer directement (méthodes et champs statiques **public** ou **protected** dans les classes publiques) ;
- descriptor (11) : contient des informations de typage (*i.e.* signatures des méthodes) si ce paquetage est accessible par d'autres paquetages ;
- debug (12) (optionnel) : contient toutes les méta-données nécessaires pour déboguer un paquetage dans une JCVm adaptée. Il n'est pas requis pour exécuter des programmes hors d'un environnement de débogage.

Dans la technologie Java, le fichier **class** est la pièce centrale de l'architecture Java. Il définit le standard pour la compatibilité binaire de la plate-forme Java. Dans la technologie Java Card, à cause des caractéristiques distribuées de l'architecture système, c'est le fichier **CAP** qui est le format standard de fichier pour la compatibilité binaire. Si un fichier **CAP** ne contient pas d'*applet* (*i.e.* aucun fichier **class** n'hérite de **javacard.framework.Applet**) alors le composant applet est absent et il s'agit d'une bibliothèque.

Les fichiers **export** ne sont pas chargés dans la carte et ne sont pas directement utilisés par l'interpréteur. Ils sont produits et utilisés par le convertisseur dans des buts de vérifications et d'édition de liens. Ainsi, les fichiers peuvent être vus comme les fichiers entêtes dans la programmation en langage C. Il sert au développeur, client de bibliothèques, pour la phase d'édition entre les bibliothèques utilisées et les *applets*/bibliothèques qu'il développe. Un fichier **export** contient donc les informations publiques sur les APIs pour un paquetage de classes complet. Il définit l'étendue et le nom des classes et l'étendue et la signature des méthodes et des champs des classes. En revanche, un fichier **export** ne contient pas d'information relative à l'implantation (*i.e.* pas de *bytecode*). Ce fichier peut donc être librement distribué par un développeur d'*applet* aux utilisateurs potentiels de l'*applet* ou de la bibliothèque sans révéler les détails de l'implantation interne.

Convertisseur Java Card

Contrairement à la machine virtuelle Java, qui travaille sur une classe à la fois, l'unité de conversion Java Card est le paquetage. Les fichiers **class** sont produits par le compilateur Java depuis le code source. Le convertisseur examine ensuite les fichiers **class** qui composent le paquetage Java et convertit ce paquetage en un fichier **CAP**.

Durant la conversion, le convertisseur réalise des tâches qu'une machine virtuelle Java classique (*i.e.* sur une station de travail) doit réaliser au chargement :

- vérifier que les images des classes Java sont bien formées ;
- contrôler des violations du langage Java Card ;
- réaliser des initialisations de variables **static** ;

- résoudre les références symboliques aux classes, méthodes et champs dans la forme la plus compacte qui peut être traitée efficacement sur la carte ;
- optimiser le *bytecode* en tirant avantage des informations obtenues au chargement des classes et à l'édition de liens ;
- allouer l'espace et créer les structures de données de la machine virtuelle pour représenter les classes.

Grâce aux tâches réalisées par le convertisseur, la partie de la machine virtuelle embarquée sur la carte peut être plus petite et plus performante, car déchargée d'une partie de son travail.

En fait le convertisseur ne prend pas seulement des fichiers **class** à convertir mais aussi un ou plusieurs fichiers **export**. En plus de la production du fichier **CAP**, le convertisseur génère également un fichier **export** pour le paquetage converti. La figure 15 montre comment un paquetage est converti. Le convertisseur charge toutes les classes d'un paquetage Java. Si le paquetage importe des classes provenant d'autres paquetages, le convertisseur charge aussi les fichiers **export** de ces paquetages. En effet, comme nous l'avons vu les fichiers **export** contiennent des informations sur les paquetages chargés sur la carte qui vont servir au convertisseur pour réaliser son travail. La sortie produite par le convertisseur pour le paquetage converti est un fichier **CAP** à charger et un fichier **export** pour les futurs paquetages clients.

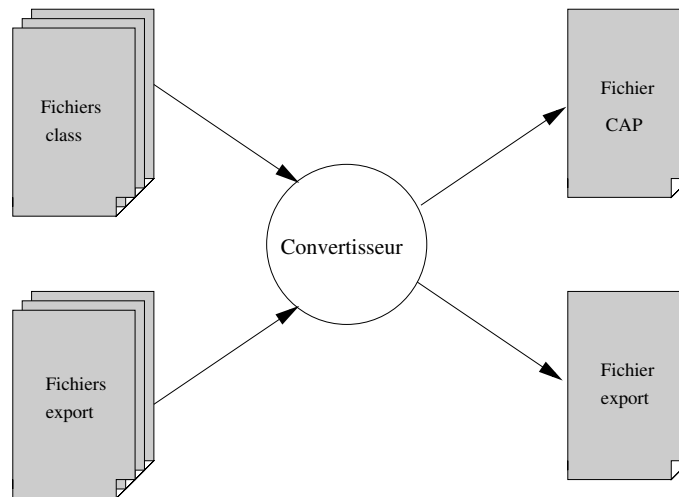


FIG. 15 – Conversion d'un paquetage.

Interpréteur Java Card

L'interpréteur Java Card fournit le support d'exécution du modèle du langage Java qui autorise une indépendance du code de l'*applet* par rapport au matériel. L'interpréteur est donc en quelque sorte le processeur virtuel « universel » pour la Java Card. Il réalise les tâches suivantes :

- il exécute les instructions du *bytecode* et donc les *applets* ;

- il contrôle les allocations de mémoire et les créations d’objets ;
- il joue un rôle crucial pour assurer la sécurité lors de l’exécution.

On notera que l’interpréteur peut exécuter 186 *bytecodes* différents dont deux réservés qui ont un comportement dépendant de l’implantation. Ces derniers sont inutilisables dans une application mais peuvent être employés en interne par le développeur de Java Cards pour ses propres besoins. Nous utiliserons d’ailleurs l’un d’eux dans la section 7.6.1 pour implanter les méthodes natives dans notre émulateur.

Par la suite, il nous arrivera d’assimiler la JCVM à l’interpréteur.

Installateur Java Card et programme d’installation hors carte

L’interpréteur Java Card ne charge pas le fichier **CAP** lui-même. Il exécute seulement le code du fichier **CAP**. Dans la technologie Java Card, le mécanisme de téléchargement et d’installation du fichier **CAP** est contenu dans une unité appelée l’installateur.

L’installateur Java Card réside sur la carte et il coopère avec un programme d’installation situé hors de la carte. Ce programme transmet à l’installateur qui s’exécute sur la carte, via le terminal (*i.e.* CAD), l’exécutable binaire contenu dans un fichier **CAP**. L’installateur écrit le fichier binaire dans la mémoire de la carte à puce, le lie avec les autres classes déjà présentes sur la carte, puis crée et initialise les structures de données qui sont utilisées par le *Java Card Runtime Environment* (JCRE). Ce processus est illustré figure 16.

La séparation entre l’interpréteur et l’installateur de fichier **CAP** permet à l’interpréteur de rester compact et offre une grande flexibilité dans l’implantation de l’installateur. Par exemple, ce dernier peut être écrit comme une *applet* Java Card ou non.

Les spécifications Java Card précisent que l’installateur est un composant optionnel mais s’il est absent une Java Card ne pourra pas accepter le chargement des applications après son émission : toutes les *applets* doivent alors être écrites dans la mémoire de la carte à puce lors de sa fabrication.

Par ailleurs, la description de l’installateur dans les spécifications est assez brève et ce n’est que depuis les versions 2.2 des spécifications que des remarques sur la possibilité d’effacer des *applets* ont fait leur apparition. En effet, les spécifications ne détaillent que peu la gestion des *applets* sur la carte (*i.e.* « Comment les charger ? », « Comment les effacer ? », « Quel est leur cycle de vie ? », etc.) et le standard GlobalPlatform que l’on présentera au chapitre 3 permettra de résoudre ces problèmes. Toutes les parties concernant l’installateur (*i.e.* le chargement, l’installation et l’effacement) sont donc plus des recommandations minimales qu’il faudrait suivre, plutôt que des spécifications visant à standardiser ce composant. Il semble que Sun microsystems n’a pas voulu imposer d’interface standard risquant d’aller à l’encontre du formidable travail fait par GlobalPlatform. Ainsi, on ne trouve, par exemple, aucune standardisation des APDUs pour le chargement, l’installation ou l’effacement dans les spécifications du JCRE.

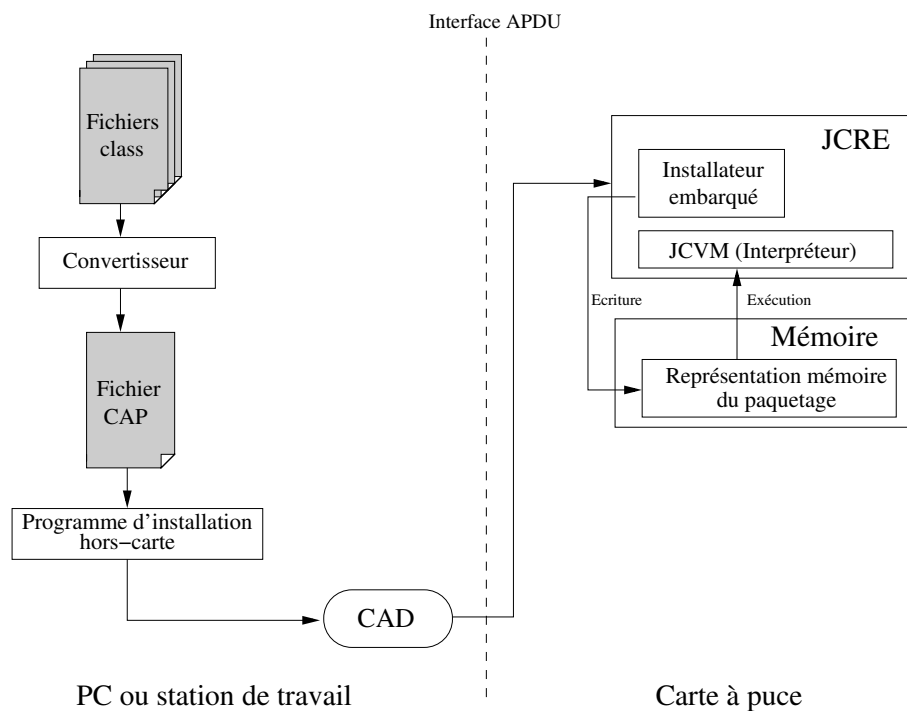


FIG. 16 – Installateur Java Card et programme d'installation hors carte.

2.4.3 L'environnement d'exécution : JCRE

L'environnement d'exécution de la Java Card (JCRE) représente l'ensemble des composants du système Java Card présents à l'intérieur de la carte à puce. Il est responsable de la gestion des ressources de la carte, de la communication réseau, des *applets*, du système de la carte et de la sécurité des *applets*.

Le JCRE est une sorte de système d'exploitation « universel » pour la Java Card. Il apporte l'indépendance des *applets* par rapport aux technologies propriétaires des vendeurs de cartes à puce en leur fournissant un système et des APIs standards. Il en résulte que les *applets* sont plus faciles à écrire et sont portables sur diverses architectures de cartes à puce.

Comme on peut le voir figure 17, le JCRE se situe au-dessus du matériel de la carte à puce et du système natif. Il englobe :

- les méthodes natives ;
- la JCVM (l'interpréteur de *bytecodes*) ;
- les classes systèmes du JCRE ;
- les APIs Java Card ;
- des extensions spécifiques à l'industrie ;
- l'installateur.

La couche basse du JCRE comprend les méthodes natives et la machine virtuelle Java Card. Les méthodes natives procurent un accès aux services bas niveau de la puce pour

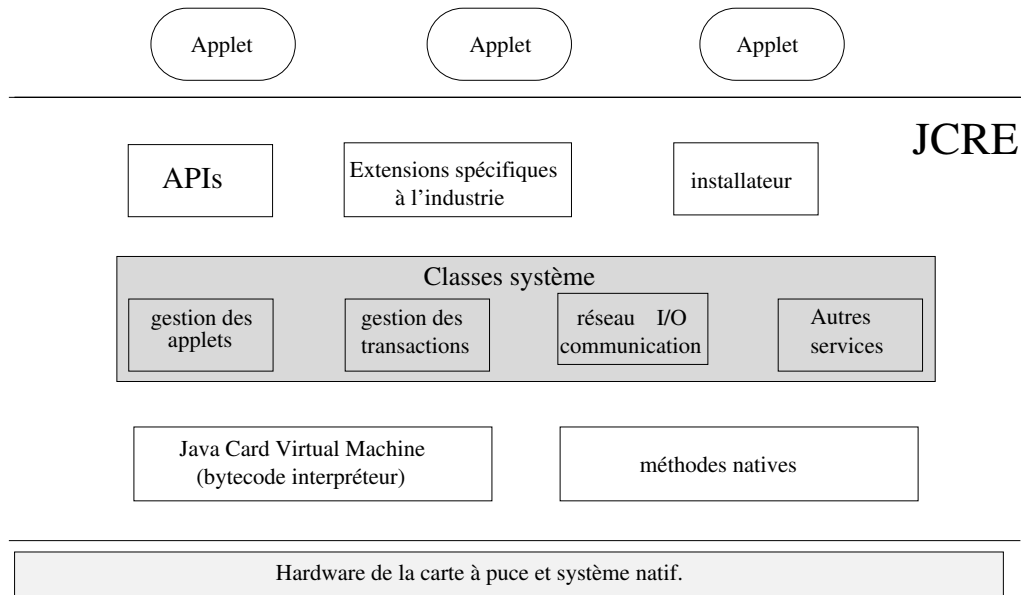


FIG. 17 – L'architecture du système dans la carte.

la JCVM et pour la couche supérieure que forment les classes systèmes du JCRE. Ces méthodes natives sont responsables du traitement des protocoles de communication, de la gestion de la mémoire, du support de la cryptographie, etc.

Les classes systèmes sont la partie essentielle du JCRE. Elles sont analogues au noyau d'un système d'exploitation et elles ont en charge :

- la gestion des transactions ;
- la gestion des communications entre les applications hôtes (applications qui tournent sur la machine reliée au terminal) et les *applets* Java Card ;
- le contrôle de la création, sélection et désélection des *applets*.

Pour réaliser ces tâches, les classes systèmes se basent généralement sur le système natif de la carte via les méthodes natives présentées ci-dessus.

Les classes des APIs Java Card sont compactes et optimisées pour le développement d'*applets* pour les cartes à puce. Les développeurs peuvent donc concentrer leurs efforts sur les détails de leurs *applets* plutôt que sur les détails de l'infrastructure du système des cartes à puce. Les *applets* accèdent aux services du JCRE à travers les APIs Java Card.

Les extensions spécifiques à l'industrie sont des bibliothèques chargées de fournir des services supplémentaires ou de redéfinir la sécurité et le modèle du système. Par exemple, GlobalPlatform ajoute des services au JCRE pour la gestion des applications issues de partenaires différents.

Comme nous l'avons déjà vu l'installateur est un composant optionnel du JCRE et c'est lui qui permet le chargement des *applets* sur la carte après leur émission.

Les *applets* Java Card sont les applications sur la plate-forme Java Card. Elles sont évidemment écrites dans le sous-ensemble du langage Java déjà décrit et sont contrôlées et

gérées par le JCRE. En effet, elles sont téléchargeables via le processus d'installation mais elles interagissent aussi de façon très étroite avec les différents services du JCRE.

Les caractéristiques du JCRE

En plus du support du modèle d'exécution du langage Java, le JCRE supporte quatre autres caractéristiques d'exécution :

- Les objets persistants et transients.
Par défaut les objets Java Card sont persistants et sont donc créés dans la mémoire persistante. L'espace et les données de tels objets existent à travers les sessions CAD. Pour des raisons de sécurité et de performance, les *applets* peuvent également créer des objets dont les données seront placées en mémoire volatile. De tels objets sont appelés transients. Les objets transients contiennent des données temporaires qui ne persisteront pas à travers les sessions CAD.
- L'atomicité et les transactions.
La JCVM assure que chaque opération d'écriture dans un champ d'un objet ou d'une classe est atomique. Donc, en cas de problème lors de l'écriture, le champ mis à jour prend soit la nouvelle valeur soit il est restauré à la valeur précédente. Le JCRE propose également des mécanismes de transactions via les APIs. Ainsi, une *applet* peut réaliser plusieurs écritures durant une transaction ce qui permet de s'assurer que soit tout est mis à jour dans une transaction complète, soit rien n'est réalisé si une erreur apparaît au milieu de la transaction.
- Le pare-feu entre les *applets* et les mécanismes de partage.
Le pare-feu isole chaque *applet* des autres *applets* à l'intérieur de son propre espace appelé « contexte ». Ainsi, l'existence et les opérations d'une *applet* n'ont pas d'effet sur les autres *applets* présentes sur la carte. Ce pare-feu est renforcé par la JCVM puisque c'est elle qui exécute les *bytecodes*. Par ailleurs, dans les situations où les *applets* ont besoin de partager des données ou d'accéder à des services du JCRE, la machine virtuelle permet l'utilisation de mécanismes sécurisés de partage accessibles via les APIs.
- Le service d'invocation de méthodes à distance.
Depuis Java Card 2.2, les spécifications ont introduit un sous-ensemble de RMI appelé JCRMI (Java Card Remote Method Invocation). Il fournit à une application cliente fonctionnant côté CAD, un mécanisme pour appeler une méthode sur un objet distant présent sur la carte. Ce service a pour objectif de faciliter le développement d'applications clientes en utilisant le protocole RMI plutôt que les APDUs. Cela permet de cacher la complexité sous-jacente du passage d'argument et de la pile protocolaire entre l'application cliente hors-carte et l'application serveur sur la carte. Un paquetage embarqué sur la Java Card assure la gestion de la couche transport pour JCRMI.

2.4.4 Les APIs

Les APIs Java Card sont un ensemble de classes optimisées pour la programmation des cartes à puce en accord avec le modèle ISO 7816. Beaucoup de classes de la plate-forme Java

(*e.g.* celles relatives au réseau, à l’affichage, etc) ne sont pas nécessaires et donc pas disponibles sur la plate-forme Java Card. Jusqu’à la version 2.1.1 des spécifications Java Card les APIs ne comprenaient qu’un noyau de trois paquetages (`java.lang`, `javacard.framework` et `javacard.security`) et un paquetage d’extension (`javacardx.crypto`). Mais depuis la version 2.2 trois autres paquetages ont été ajoutés pour supporter JCRMI : `java.io`, `java.rmi` et `javacard.framework.service`.

Le paquetage `java.lang`

Le paquetage `java.lang` est, sur la plate-forme Java Card, un sous-ensemble strict de son équivalent sur la plate-forme Java. Ce paquetage supporte les classes `Object`, `Throwable` et `Exception`. C’est lui qui fournit les fondements pour le support du langage Java. La classe `Object` est définie comme la classe racine de la hiérarchie des classes sur la plate-forme Java Card. Les classes d’exceptions fournies permettent également d’assurer une sémantique cohérente lorsqu’une erreur apparaît à cause d’une violation du langage de programmation Java.

Par exemple, la machine virtuelle Java et celle Java Card lanceront toutes les deux `NullPointerException` quand on voudra accéder à une référence `null`.

Le paquetage `javacard.framework`

Ce paquetage apporte les classes et les interfaces essentielles pour programmer des *applets* Java Card. La classe la plus importante qu’il définit est la classe de base `Applet` qui fournit la structure sous-tendant le modèle d’exécution de l’*applet* en interaction avec le JCRE pour toute la durée de vie de l’*applet*.

Il fournit également la classe `APDU` nécessaire pour gérer les communications des *applets* avec l’extérieur. Depuis Java Card 2.2, c’est aussi dans ce paquetage que se trouve l’interface `MultiSelectable` qui indique si l’*applet* accepte les canaux logiques et la multi-sélection. Puisque la classe `System` du paquetage `java.lang` de la plate-forme Java n’est naturellement pas supportée, la plate-forme Java Card propose dans le paquetage `javacard.framework` la classe `JCSys` afin d’accéder aux services du JCRE.

Cette classe permet notamment de gérer les mécanismes de transaction et de partage d’objets au travers du pare-feu en association avec l’interface `Shareable` mais aussi de créer des objets transients et d’appeler le mécanisme de ramasse-miettes si celui-ci est présent.

Enfin, ce paquetage fournit d’autres classes comme la classe `OwnerPIN` qui propose une implantation de référence du mécanisme de PIN. Les deux autres classes restantes sont `AID` pour faire des opérations sur les AIDs et `Util` qui propose des méthodes souvent utilisées dans la programmation en Java Card.

Le paquetage `javacard.security`

Le paquetage `javacard.security` fournit une architecture aux fonctions cryptographiques supportées sur la plate-forme Java Card. Sa structure est basée sur le paquetage `java.security` de la plate-forme Java. Il propose une classe `KeyBuilder` pour fabriquer des clés cryptographiques de différents types et des interfaces pour les manipuler. Il met principalement à disposition du programmeur les classes abstraites `RandomData`, `Signature`, `MessageDigest`, et `Checksum` pour générer respectivement des nombres aléatoires, des signatures, des empreintes et des sommes de contrôle. Il définit également l'exception `CryptoException` pour la gestion des différentes erreurs.

Le paquetage `javacardx.crypto`

Le paquetage `javacardx.crypto` est un paquetage d'extension. Il contient les classes cryptographiques et les interfaces qui sont sujettes à une demande pour l'exportation aux États-Unis.

Il définit une classe abstraite de base `Cipher` qui supporte les fonctions de chiffrement et de déchiffrement pour les différents algorithmes implantés sur la carte. En effet, dans le domaine de la cryptographie, Java Card n'impose pas aux vendeurs d'implanter tous les algorithmes cryptographiques, ni même un ensemble minimal. On peut ainsi se retrouver avec des Java Cards aux possibilités bien différentes dans les domaines cryptographiques.

Le paquetage `java.io`

Ce paquetage est un sous-ensemble strict de celui de la plate-forme Java. Il ne définit qu'une exception `IOException` pour indiquer un problème de communication. Celle-ci n'est présente que pour les besoins de JCRMI qui en hérite au travers de la classe `RemoteException` du paquetage `java.rmi`.

Le paquetage `java.rmi`

Ce paquetage définit l'interface `Remote` qui identifie les interfaces dont les méthodes peuvent être appelées par des applications clientes du CAD. Il définit également l'exception `RemoteException` qui pourra être lancée pour indiquer qu'une exception s'est produite lors de l'exécution d'un appel de méthode à distance.

Le paquetage `javacard.framework.service`

Ce paquetage fournit un ensemble de classes et d'interfaces qui permettent à une *applet* Java Card d'être conçue comme le regroupement de plusieurs services. Il fournit aussi une classe d'agrégation, appelée `Dispatcher` qui propose des méthodes pour ajouter et retirer des services de sa base interne, et pour router les commandes APDUs vers les services enregistrés.

Le paquetage contient également l'interface **Service** que doivent implanter tous les services. Cette interface propose toutes les méthodes nécessaires pour traiter les commandes APDUs.

Il existe aussi deux sous-interfaces de **Services**, **RemoteService** et **SecurityService** pour l'implantation de services aux fonctionnalités plus spécifiques. **RemoteService** est utilisé pour définir des services qui permettent à des processus distants d'accéder aux services présents sur la Java Card alors que **SecurityService** est utilisé pour définir des services qui fournissent des méthodes pour connaître le niveau de sécurité actuelle (*e.g.* intégrité des données entrantes, confidentialité, etc.).

La classe **BasicService** fournit une implantation par défaut des méthodes définies dans l'interface **Service** et par conséquent, la fonctionnalité de base d'un service. Tous les services seront donc des sous-classes de **BasicService**.

Par ailleurs, afin de permettre à un programme Java fonctionnant côté poste client d'appeler des méthodes sur les objets distants d'une *applet* Java Card, le paquetage fournit les classes **CardRemoteObject** et **RMIService**. Ces classes contiennent donc les fonctionnalités minimales exigées pour permettre l'invocation de méthodes à distance pour la plate-forme Java Card (JCRMI).

2.4.5 Les *applets*

Tout d'abord, il ne faut pas confondre les *applets* Java destinées à être exécutées dans un navigateur web et les *applets* Java Card qui, elles, sont destinées à être exécutées au sein du JCRE. Si les programmes Java Card portent aussi le nom d'*applet* c'est parce qu'ils sont chargeables après la fabrication de la carte et n'ont pas besoin de se trouver dans la ROM. Cette possibilité de charger les *applets* confère à la plate-forme Java Card le dynamisme qui manquait jusqu'alors aux systèmes embarqués sur carte à puce.

Sur cette plate-forme, les *applets* Java Card doivent suivre un ensemble de règles, ce qui se traduit en terme de programmation par faire hériter toute *applet* Java Card de la classe `javacard.framework.Applet`. Cette classe définit le modèle d'exécution standard pour la plate-forme Java Card en fournissant les méthodes qu'appellera le JCRE pour interagir avec l'*applet*. Une *applet* exécutable est donc une instance de la classe `javacard.framework.Applet`.

Le JCRE étant un environnement multi-applicatif, il peut supporter plusieurs *applets* sur la même carte.

Convention de nommage des paquetages et des *applets* Java Card

Alors que sur la plate-forme Java un paquetage ou un programme est identifié de manière unique par une chaîne Unicode et un schéma de nommage basé sur les noms de domaine internet, sur la plate-forme Java Card un paquetage ou un programme (*i.e.* une *applet*) est identifié par un AID.

La construction d'un AID (*cf.* TAB. 7) est définie par l'ISO 7816-5. Il est représenté par un tableau dont la taille varie entre 5 et 16 octets. Il est constitué par la concaténation

du RID (Ressource IDentifier de taille fixe de 5 octets) et du PIX (Proprietary Identifier eXtension de taille variant entre zéro et onze octets). C'est l'ISO qui procède à l'attribution

RID (5 octets)	PIX (0-11 octets)
----------------	-------------------

TAB. 7 – Identifiant d'application : AID

aux entreprises des RIDs afin que chacune ait son propre RID et ensuite chaque entreprise contrôle la gestion des PIXs.

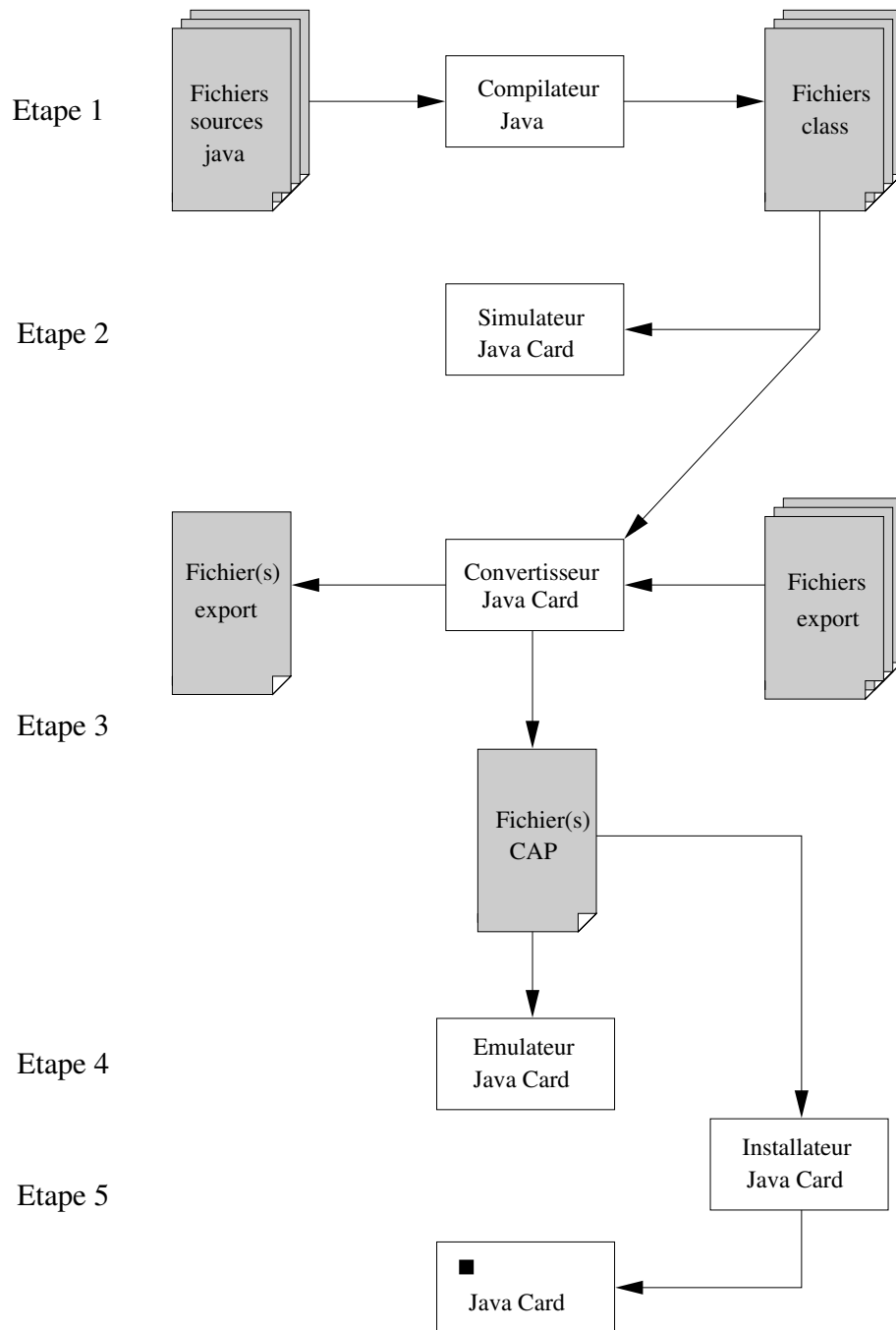
Chaque *applet* et chaque paquetage possède un AID unique. Le RID identifiant le fournisseur de l'*applet*, les *applets* définies dans un paquetage partagent le même RID que celui-ci. L'AID d'un paquetage et celui par défaut de chaque *applet* de ce paquetage sont spécifiés dans le fichier **CAP**. On parle d'AID par défaut pour une *applet* car il est possible de modifier, à l'installation, l'AID qui était le sien dans le fichier **CAP** à condition de conserver le même RID. Les AIDs sont fournis au convertisseur lorsque le fichier **CAP** est généré.

On notera que les exigences définies dans les spécifications Java Card concernant le RID ne sont pas reprises par GlobalPlatform (*cf.* chapitre 3). Or, suivant les choix d'implantation, cela peut engendrer des problèmes comme l'attaque connue sous le nom de *AID Impersonation* que nous décrivons au chapitre 10.

Le cycle de développement d'une *applet* Java Card

Le cycle de développement d'une *applet* peut se décomposer en cinq étapes (*cf.* FIG. 18) :

- La première étape consiste comme pour tout programme Java à compiler les sources afin d'obtenir les fichiers **class**.
- La seconde étape se limite à tester et à déboguer le comportement global de l'*applet* sur un simulateur tournant sur une station de travail. Dans ces environnements de simulation utilisant la machine virtuelle Java de la station, on ne peut pas tester le comportement de l'*applet* vis-à-vis du pare-feu, des objets transients et persistants et d'autres caractéristiques du JCRE.
- La troisième étape est la conversion des paquetages Java afin d'obtenir les fichiers **CAPs** (*cf.* section 2.4.2).
- La quatrième étape permet de tester réellement les *applets* Java Card sur un émulateur. Un émulateur Java Card émule le JCRE sur une station de travail et comporte donc une JCVM. Par conséquent, le comportement de l'*applet* sera le même sur l'émulateur que sur la carte après son chargement. La plupart des simulateurs et des émulateurs vont de paire avec un débogueur permettant ainsi au développeur de mettre des points d'arrêts, de suivre les changements d'états de son *applet*, etc. Nous présenterons d'ailleurs dans le chapitre 7 l'émulateur, nommé JCat Emulator, que nous avons développé.
- Enfin, lorsque l'*applet* est finalisée, l'étape suivante consiste à la charger puis à l'installer (*cf.* section 2.4.2) sur un périphérique supportant la technologie Java Card. On pourrait également déboguer l'*applet* sur la carte, mais c'est une procédure peu commode et extrêmement risquée.

FIG. 18 – Les étapes du développement d'une *applet* Java Card.

2.4.6 Les objets Java Card

Pour représenter, stocker, et manipuler des données le JCRE et les *applets* créent des objets. Sur la plate-forme Java Card ces objets obéissent aux règles de programmation du langage Java :

- tous les objets sur la plate-forme Java Card sont des instances, créées dans le tas, de classes ou de tableaux, qui ont la même classe racine `java.lang.Object` ;
- les champs d'un nouvel objet ou les éléments d'un nouveau tableau sont initialisés à leurs valeurs par défaut (0, `null` ou `false`) sauf si ils sont initialisés à une autre valeur par le constructeur.

La technologie Java Card autorise les objets *persistants* et les objets *transients*. Pourtant, les concepts d'objets persistants et transients et les mécanismes associés sur la plate-forme Java Card ne sont pas les mêmes que ceux de la plate-forme Java standard ainsi que nous le verrons plus loin (*cf.* section 2.5.2).

2.5 Les spécificités de Java Card

Cette section détaille quelques aspects qui font la spécificité de la technologie Java Card par rapport à la technologie Java standard.

2.5.1 Cycles de vie

Sur les périphériques à mémoire persistante, le cycle de vie des différentes entités dépend de leur localisation mémoire. Ainsi, de par l'architecture de la plate-forme Java Card, la ROM contient le système d'exploitation et la machine virtuelle. La RAM quant à elle renferme les différentes structures de données nécessaires à la machine virtuelle Java Card (JCVM) pour fonctionner (*e.g.* la pile d'appels). L'EEPROM est utilisée pour stocker les applications et les objets persistants.

Le cycle de vie de la JCVM

Dans un PC ou une station de travail, la JVM se comporte comme un processeur virtuel et les données et les objets dont elle a besoin sont créés dans la RAM. Ainsi, quand l'alimentation est coupée ou lorsque la machine virtuelle est arrêtée, les applications Java et leurs objets sont automatiquement détruits.

Sur la plate-forme Java Card, les informations de la machine virtuelle sont préservées au travers des sessions de communication avec le lecteur grâce à la mémoire persistante. Donc, contrairement à la machine virtuelle Java classique utilisée sur une station de travail qui voit son cycle de vie limité à une exécution, la JCVM possède un cycle de vie identique à celui de la carte.

Ainsi, lorsque l'alimentation est coupée, la machine virtuelle est seulement suspendue et à la prochaine remise sous tension, le JCRE relancera l'exécution de la machine virtuelle en chargeant les données depuis la mémoire persistante. Il faut noter que le JCRE ne redémarre

pas la JCVM exactement au point où la tension a été coupée mais qu'elle s'exécute depuis le début de la boucle principale de *boot* après que le JCRE ait remis la carte dans un état cohérent.

Le cycle de vie du JCRE

Sur une machine classique le JRE (Java Runtime Environnement) se comporte comme un système d'exploitation virtuel et son cycle de vie est le même que celui de la JVM, *i.e.* réduit à une exécution.

En revanche, dans la carte à puce Java, le cycle de vie du JCRE est le même que le cycle de vie de la carte. L'initialisation du JCRE est réalisée seulement une fois dans la vie de la carte, lors de sa première mise en service. Durant cette phase, le JCRE initialise la machine virtuelle et crée les objets nécessaires pour fournir des services et gérer les *applets*. Grâce à la mémoire persistante les états du JCRE et des objets créés sur la carte sont préservés même lors des pertes d'alimentation.

À chaque remise sous tension, le JCRE relance la JCVM. Ce redémarrage du JCRE diffère de la phase d'initialisation car il préserve les *applets* et les objets créés sur la carte. Lors du redémarrage, si une transaction n'a pas été terminée, le JCRE réalise tous les nettoyages nécessaires pour remettre la carte dans un état cohérent.

Le cycle de vie des *applets*

Les applications Java Card sont programmées en langage Java Card et doivent toutes hériter de la classe **Applet**. Elles peuvent être chargées tout au long de la vie de la carte sous la forme d'un paquetage au format **CAP**. Une fois le paquetage chargé, la JCVM appelle la méthode **install** pour allouer les ressources requises et enregistrer – via la méthode **register** – un objet persistant, l'instance de *applet*, auprès de l'environnement d'exécution Java Card (JCRE) afin de permettre les futures requêtes à cette *applet*.

Durant sa vie, l'*applet* pourra créer des objets Java Card, voire les détruire dans certains cas comme nous le verrons plus loin.

Sa phase d'utilisation commence lorsque le JCRE invoque sa méthode **select** et elle se termine lorsque le JCRE invoque sa méthode **deselect**.

Par ailleurs, le cycle de vie de l'*applet* se termine lorsque le JCRE reçoit une demande authentifiée d'effacement de l'instance de l'*applet* et/ou du paquetage à laquelle celle-ci appartient. En revanche, si jamais le JCRE ne reçoit pas une telle demande, le cycle de vie de l'*applet* s'étend par défaut jusqu'à la fin du cycle de vie de la Java Card.

Le cycle de vie des objets

Le cycle de vie d'un objet sur la plate-forme Java Card s'étend de l'instant de sa création (par un des *bytecodes* **new**, **newarray** ou **anewarray** ou par l'invocation d'une des méthodes **makeTransientBooleanArray**, **makeTransientByteArray**, **makeTransientShortArray** et **makeTransientObjectArray**) jusqu'à l'instant de sa destruction (soit par un mécanisme

de ramasse-miettes s'il est présent et qu'il n'est plus référencé, soit par l'effacement de l'*applet* qui l'a créé). Si aucun de ces deux cas de destruction ne se présente, alors le cycle de vie de tout objet se termine à la fin du cycle de vie de la carte.

On remarquera que le cycle de vie du contenu des champs d'un objet dépend de la nature de cet objet ainsi que nous le verrons plus loin (*cf.* section 2.5.2).

Interactions entre la Java Card et l'extérieur durant les sessions CAD

La temps qui s'écoule entre le moment où la carte est insérée dans le CAD (*i.e.* mise sous tension) et le moment où la carte est retirée du CAD (*i.e.* mise hors tension) est appelée la session CAD. C'est ainsi que durant une session CAD, le JCRE opère comme une carte à puce classique et gère les entrées/sorties d'APDUs en provenance de l'application hôte (*cf.* FIG. 19).

Lorsque la Java Card est insérée dans un CAD, elle est mise sous tension, puis le JCRE est lancé et met la carte dans un état cohérent avant d'émettre l'ATR¹ de la carte et finalement de se mettre en attente d'un ordre (*i.e.* une commande APDU) sur le port de communication géré par le système d'exploitation sous-jacent.

Lorsqu'une application externe (*i.e.* le client) initie une session avec une *applet* présente sur la carte (*i.e.* le serveur) en envoyant une commande APDU de sélection de l'*applet*, le JCRE invoque la méthode `select` de cette *applet* et la marque comme active si la méthode retourne sans erreur.

Si l'*applet* est active le JCRE transfère toutes les prochaines commandes APDUs envoyées par le client à la méthode `process` de l'*applet* lui donnant ainsi le contrôle, sauf pour les commandes de sélection qu'il interprète lui-même.

Ensuite, l'*applet* traite la commande et prépare la réponse APDU à envoyer au client à la fin de son exécution lorsqu'elle rend le contrôle au JCRE.

Après le traitement de plusieurs commandes, la session de l'*applet* se termine soit parce que la sélection d'une autre *applet* est demandée et dans ce cas l'*applet* active est tout d'abord désélectionnée par le JCRE via sa méthode `deselect`, soit parce que la carte sera retirée du lecteur et dans ce cas il s'agit d'une désélection implicite puisqu'à la prochaine mise sous tension le JCRE désélectionnera l'*applet* précédemment active. Ainsi, une session avec une *applet* dure depuis sa sélection jusqu'à sa désélection.

2.5.2 La nature des objets Java Card

La technologie Java Card a été conçue afin d'apporter aux programmeurs d'applications embarquées sur cartes à puce tous les avantages offerts par le langage Java [131, 69]. De plus, grâce à la technologie des mémoires persistantes qu'elle embarque, la carte à puce peut stocker de façon sécurisée des informations mais aussi les conserver une fois hors tension. Adapter l'environnement Java aux cartes à puce, consistait à répondre à la question : « Comment intégrer Java sur un périphérique de nature persistante ? ».

¹Réponse à la mise sous tension de la carte.

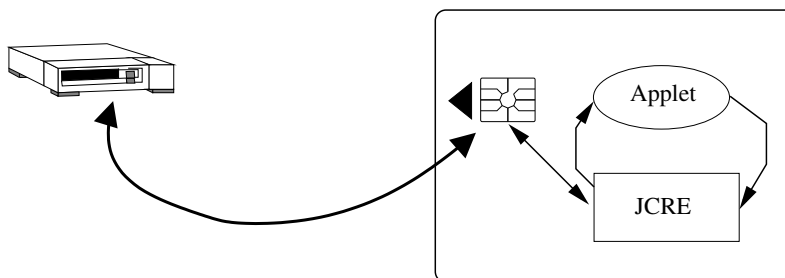


FIG. 19 – Communication APDU.

L'environnement Java est un environnement de programmation temporaire ; pour satisfaire aux contraintes des cartes à puce les spécifications Java Card ont pour leur part choisi d'adopter les concepts de *persistance* et de *transience* issus des systèmes persistants orthogonaux [70]. Ainsi, dans la technologie Java Card, le lieu de stockage des valeurs d'un objet dépend de sa nature. Pour l'instant, nous nous bornerons à décrire les objets *persistants* et *transients* tels qu'ils sont présentés dans les spécifications. Mais dans le chapitre 8, nous montrerons les ambiguïtés de ces spécifications.

On notera que nous avons choisi de créer le néologisme *transience* et ses dérivés pour exprimer les propriétés de temporalité spécifiques à certaines données de Java Card. En effet, la traduction simple du terme anglais *transient* par les termes de « temporaire » ou « transitoire » n'exprimait pas exactement, comme nous le verrons, les concepts sous-jacents.

Les objets persistants

Un objet persistant est un objet qui a été créé par l'opérateur `new` (*i.e.* par un des *bytecodes* `new`, `newarray` ou `newarray`) avant de devenir persistant. Un tel objet conserve ses valeurs au fur et à mesure des sessions avec le lecteur. Chaque mise à jour d'un champ d'un objet persistant est atomique (*cf.* section 2.5.3) et un objet persistant prend part aux transactions (*cf.* section 2.5.3). Un objet persistant peut être référencé par un champ d'un objet transient.

Les valeurs des champs d'un objet persistant sont persistantes et sont donc stockées en mémoire persistante. Les spécifications Java Card ajoutent qu'un objet persistant doit être créé :

- lorsque la méthode `Applet.register` est appelée – ici, l'instance de l'*applet* est rendue persistante – ;
- lorsqu'une référence à l'objet est stockée dans un champ d'un objet persistant ou dans celui d'une classe.

Les objets transients

La notion de *transient* permet d'avoir des objets dont les champs ont un contenu temporaire. Il existe deux types d'objets transients, nommés `CLEAR_ON_RESET` et

CLEAR_ON_DESELECT. Chaque type est associé à un événement qui lors de son occurrence déclenche la réinitialisation des champs de l'objet.

Les objets transients **CLEAR_ON_RESET** sont utilisés pour conserver des données qui doivent exister tout au long d'une session avec le lecteur et donc au delà des sessions de l'*applet*. Ces objets sont réinitialisés lorsque l'événement **CLEAR_ON_RESET** se produit. Il est provoqué par le redémarrage explicite ou implicite (comme à la suite d'une coupure de l'alimentation) du périphérique.

Les objets transients **CLEAR_ON_DESELECT** sont utilisés pour conserver des données qui doivent exister seulement pendant la durée d'une session de l'*applet* (*i.e.* le temps où une *applet* est sélectionnée). Ces objets sont réinitialisés lorsque l'événement **CLEAR_ON_DESELECT** se produit, *i.e.* lors de la sélection d'une autre *applet* ou de l'occurrence d'un événement **CLEAR_ON_RESET**. En effet, un redémarrage du périphérique implique en particulier la désélection implicite de l'*applet* sélectionnée. Les objets transients **CLEAR_ON_DESELECT** sont donc aussi des objets transients **CLEAR_ON_RESET**.

Il est important de noter que ces événements affectent seulement le contenu des champs de l'objet et aucunement son en-tête dont les informations persisteront tout au long de la vie de l'objet. Par exemple, dans le cas des tableaux transients qui sont les seuls types objets transients spécifiés jusqu'à maintenant dans Java Card, l'attribut de taille du tableau n'est pas remis à jour lors de l'occurrence d'un événement **CLEAR_ON_DESELECT** ou **CLEAR_ON_RESET**. En effet, le contenu de ce tableau transient a été réinitialisé mais occupe toujours l'espace mémoire qui lui avait été alloué. On remarquera donc que le terme anglais *transient*, qui veut dire « temporaire » ou « transitoire », est ambiguë pour exprimer la nature réelle de ces objets.

Un objet transient est créé par l'invocation de méthodes de la classe **JCSYSTEM** : **makeTransientBooleanArray**, **makeTransientByteArray**, **makeTransientShortArray** et **makeTransientObjectArray**. La mise à jour d'un champ d'un objet transient n'est pas atomique, et de plus, un objet transient ne participe pas aux transactions. Un champ d'un objet transient peut référencer un objet persistant.

Les objets de nature transiente sont utilisés en raison de leur rapidité et de la sécurité (*cf.* section 8.2.2) qu'ils apportent – *e.g.* pour le stockage de clés cryptographiques.

On notera aussi qu'un tableau transient d'objets est en fait un tableau de références désignant des objets qui peuvent être de nature transiente ou persistante. Ainsi, sur l'occurrence de l'événement associé au tableau les valeurs du tableau sont réinitialisées mais les objets précédemment référencés continuent d'exister en mémoire.

2.5.3 Atomicité et transaction

Afin de garantir l'intégrité des données lors de la mise à jour des objets persistants, les notions d'atomicité et de transaction ont été introduites dans l'environnement Java Card.

Atomicité

Une opération est atomique si elle est soit totalement exécutée soit pas du tout.

L'interpréteur Java Card doit assurer que la mise à jour des champs des objets persistants est atomique afin de palier à tout incident pouvant mettre en péril l'intégrité des données modifiées (perte d'alimentation, redémarrage de la puce après une détection de rayonnements lumineux anormaux [195], injection de fautes [72, 120], etc.). Ainsi, lors d'une opération d'affectation, le champ (d'instance ou de classe) ou l'élément de tableau concerné pourra soit prendre la nouvelle valeur, soit conserver la valeur précédente. En aucun cas une autre valeur n'est admissible.

Les APIs permettent aussi la mise à jour atomique d'une portion d'un tableau persistant grâce à la méthode `Util.arrayCopy`. Les *applets* n'ayant pas ces exigences d'intégrité utiliseront la méthode `Util.arrayCopyNonAtomic`, plus rapide, puisque ne nécessitant pas la sauvegarde des anciennes valeurs.

Transaction

Une transaction représente un bloc d'opérations qui doit être effectué complètement ou pas du tout. Le principe est le même que celui qui existe dans le domaine des bases de données.

En Java Card, le déclenchement, la terminaison ou l'annulation des transactions se fait via des méthodes des APIs (*i.e.* respectivement `beginTransaction`, `commitTransaction` et `abortTransaction` de la classe `JCSys`). Au niveau applicatif, un seul niveau de transaction est autorisé.

Pour restaurer les anciennes données en cas d'échec de la transaction (ou de la mise à jour atomique) ou en cas d'annulation de la transaction via la méthode `abortTransaction`, la Java Card doit posséder un mécanisme de *rollback* des données.

La possibilité de restaurer des données implique la sauvegarde dans un buffer de transaction des anciennes valeurs des champs des objets persistants modifiés. La taille de ce buffer est dépendante de l'implantation et conditionne le nombre de mises à jour possibles au sein d'une transaction. La sauvegarde d'une ancienne donnée correspond à la sauvegarde de sa valeur mais aussi à la sauvegarde d'informations supplémentaires [178] telles que sa position mémoire, son type, etc. La taille et la nature des informations sauvegardées sont dépendantes de l'implantation. Ainsi, un programme pourra fonctionner parfaitement sur une implantation et mal se comporter sur une autre dans les conditions limites.

Il existe de nombreux problèmes liés à l'instanciation d'objets à l'intérieur des transactions annulées [178].

Perte d'alimentation et reset

Lors d'une perte d'alimentation ou d'un reset, toutes les données en mémoire volatile sont perdues. À la remise sous tension le JCRE doit remettre la carte dans un état cohérent. En particulier, il doit annuler les éventuelles transactions en cours avant de pouvoir émettre son ATR et accepter une commande.

2.5.4 Gestion de la mémoire

Allocation

L'interprète alloue :

- les champs statiques des classes au chargement des fichiers CAP ;
- la représentation des objets dans le tas lors de l'exécution :
 - des *bytecodes* `new` pour les objets, `newarray` pour les tableaux classiques de type simple et `anewarray` pour les tableaux classiques de type référence,
 - des méthodes des APIs pour les tableaux transients (*e.g.* `makeTransientByteArray`, etc.).

Initialisation

L'initialisation des champs statiques se fait après le chargement d'un paquetage avec les valeurs que l'on trouve dans le composant `StaticField` du fichier CAP. Par ailleurs, ces initialisations sont soumises à quelques contraintes dans la technologie Java Card. En effet, les champs statiques d'un paquetage d'*applet* peuvent être initialisés à des valeurs constantes de types primitifs ou à des tableaux de constantes de type primitif alors que les champs statiques d'un paquetage de bibliothèque utilisateur peuvent seulement être initialisés à des valeurs constantes de type primitif. Enfin, seuls les champs statiques déclarés dans une classe d'un paquetage peuvent être initialisés par ce paquetage.

Désallocation

Une nouvelle fois, les désallocations se font soit au niveau interprète soit au niveau JCRE. La destruction de différentes structures de données peut se faire au niveau des octets dans les mémoires physiques de différentes manières :

- par passage des octets aux valeurs naturelles de la mémoire volatile lorsqu'elle n'est plus alimentée (*i.e.* `0x00` ou `0xFF` suivant le type de la mémoire) ;
- par passage des octets à des valeurs aléatoires afin d'éviter des phénomènes de rémanence [137].

En Java Card, comme en Java, il n'y a pas de mot-clé pour la libération explicite de la mémoire. Comme la spécification ne rend pas obligatoire la présence d'un mécanisme de ramasse-miettes, il n'y a pas de libération de la mémoire allouée qui n'est plus utilisée, sauf lors de l'effacement d'une *applet* et dans certains autres cas comme cela est implicitement suggéré dans les spécifications (*e.g.* lors de l'échec d'une installation). Ainsi, dès qu'un objet ne sera plus référencé, cet objet ne sera plus jamais accessible et l'espace mémoire qu'il occupe sera perdu. Cependant, depuis sa version 2.2, Java Card prévoit la présence optionnelle d'un mécanisme de ramasse-miettes avec déclenchement à la demande de l'*applet* via la méthode `JCSystem.requestObjectDeletion()`.

On remarquera que le comportement d'une *applet* dépendra donc de la Java Card sur laquelle elle s'exécute, s'éloignant un peu plus de la philosophie Java : « Write once, run anywhere ». Certains constructeurs ont aussi implanté des mécanismes de ramasse-miettes

avec un déclenchement à la demande en provenance de l'application client au travers d'une commande envoyée à la carte.

De toute évidence, on notera qu'un mécanisme de ramasse-miettes nuit aux performances de la carte à cause de la lenteur des écritures en EEPROM mais aussi à sa durée de vie, une fois encore à cause de l'EEPROM et de son nombre de cycles d'écriture limité.

2.6 Les sécurités

Il existe plusieurs types de mécanismes sécuritaires fournis par Java Card :

- ceux qui sont intégrés au langage Java comme les différents niveaux de contrôle d'accès aux méthodes et aux variables (**public**, **protected**, **private**) et le typage fort qui empêche par exemple de construire des pointeurs ;
- ceux qui ont été rajoutés pour pallier l'absence de vérifieur embarqué, comme le pare-feu.

En effet, sur un ordinateur classique les sécurités intégrées au langage et le typage fort sont normalement assurées par la machine virtuelle en association avec le vérifieur de *bytecodes*. Or, historiquement il n'était pas possible d'embarquer de vérifieur à cause des faibles capacités mémoire des cartes à puce. C'est pourquoi, les spécifications Java Card ont introduit un mécanisme orthogonal : le pare-feu.

2.6.1 Le vérifieur

Un des avantages du langage Java est qu'il s'adapte bien à la mobilité car il est possible de vérifier le code intermédiaire. Seulement, en raison des contraintes des cartes à puce (*i.e.* peu de mémoire, microprocesseur peu puissant, etc.) les spécifications Java Card ont décidé de ne pas embarquer le mécanisme de vérification sur la carte.

La vérification de la correction du *bytecode* se fait donc avant le chargement sur la carte, entre l'étape de conversion qui construit le fichier CAP et l'étape de chargement (*cf.* FIG. 20). Dans le cycle de chargement d'une *applet* décrit précédemment, nous avons fait abstraction de cette étape afin de simplifier l'explication mais également parce que des cartes avec un vérifieur embarqué devraient faire leur apparition dans un futur très proche.

Le processus de vérification permet de s'assurer que le fichier CAP n'a pas été modifié pour y introduire un code malicieux. Ce processus consiste à réaliser une interprétation abstraite du *bytecode* de chaque méthode dans le but de vérifier que le flux de contrôle et le flux de données ne génère pas d'*overflow* ou d'*underflow* de la pile d'appels, que les variables n'utilisent pas un type invalide, etc. Cette interprétation utilise des algorithmes d'inférence de type et les concepts introduits dans un article de Gary Kildall [149]. Pour avoir une vue d'ensemble du processus de vérification, le lecteur peut consulter l'article de Xavier Leroy [154]. Bien souvent, les vendeurs de Java Cards fournissent une solution deux en un qui prend en entrée les fichiers **class** et **export** pour réaliser la conversion et la vérification en une seule étape transparente pour l'utilisateur final.

On se rend bien compte qu'il est encore possible de falsifier le fichier CAP après le passage du vérifieur et avant le chargement sur la carte. Or, comme le vérifieur est un

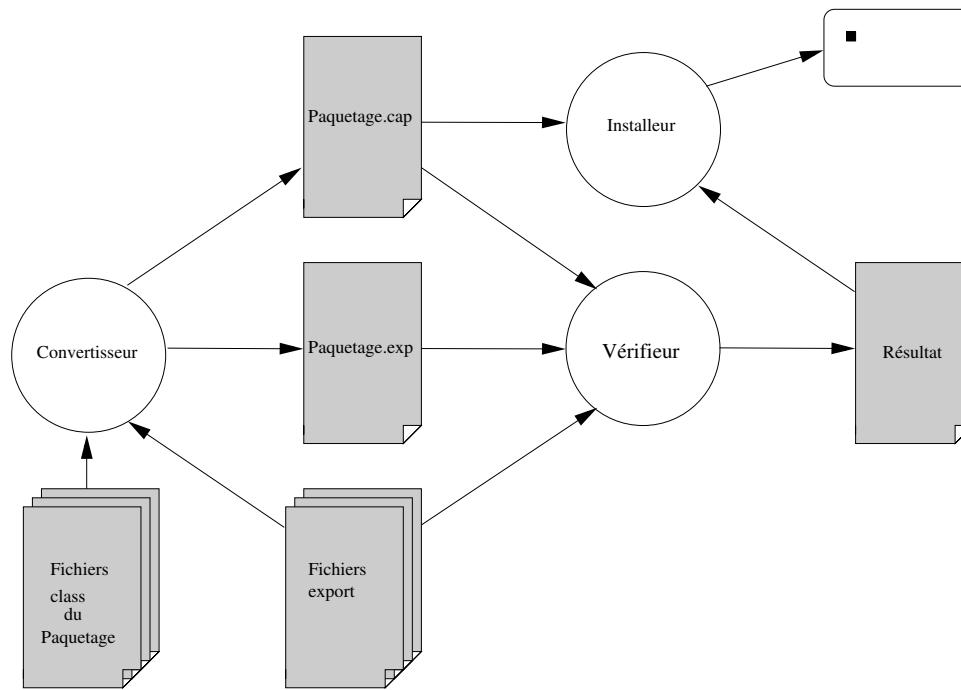


FIG. 20 – Le processus de vérification.

élément essentiel pour assurer la sécurité et la sûreté d'une machine virtuelle [157] il devrait être impossible pour l'utilisateur final de charger une nouvelle application. Comme un des objectifs de Java Card est la multi-application il a fallu trouver des solutions.

La première solution, mais aussi celle qui est la plus largement utilisée industriellement, consiste à signer le fichier **CAP** à charger sur la carte à sa sortie du vérifieur puis à vérifier cette signature sur la carte. Cette solution est efficace et peu coûteuse en temps puisque la vérification sur la carte est faite par le processeur cryptographique. Elle est d'ailleurs reprise par le standard GlobalPlatform que nous présentons plus loin au chapitre 3. En revanche, son principal problème est un déploiement centralisé des applications puisque ce déploiement requiert un tiers de confiance pour les signer. En effet, si tout le monde disposait du droit de signer les applications cette solution serait incohérente pour protéger la carte. Tout cela diminue la flexibilité du système et pose donc quelques problèmes.

Une seconde solution consiste à construire dans la carte une machine virtuelle défensive [95] qui réalisera les opérations de vérification avant l'exécution de chaque *bytecode*. Si cette solution est très sûre, elle n'est pas très efficace à cause de la surcharge en temps et en mémoire que représente la vérification de chaque *bytecode*. Elle a par contre l'énorme avantage d'être autonome et donc de pouvoir accepter le chargement d'applications malicieuses sans risque puisqu'elle les bloquera lors de leur exécution.

Une troisième solution est la technique de « Proof-Carrying Code » [177] qui a été adaptée pour Java [187]. Cette méthode consiste à générer à l'extérieur une preuve pour le programme qui est ensuite vérifiée dans le système embarqué par un vérifieur spécialisé. L'avantage de cette méthode est que la preuve à vérifier sur la carte est plus simple que

le travail qu'aurait dû faire un vérifieur embarqué. Les problèmes de cette approche sont les suivants : d'une part, la taille des applications chargées augmente puisqu'elles incluent la preuve et leur déploiement devient complexe (car l'application doit être traitée dans un générateur de preuves forcément centralisé si l'on veut assurer la fiabilité de la preuve à charger) ; d'autre part, des applications valides peuvent être rejetées si la preuve n'est pas fournie.

Une quatrième solution proposée par Xavier LEROY consiste à normaliser le code hors carte pour que chaque variable manipulée par la machine virtuelle soit d'un seul type [155, 156] (*e.g.* un registre de variable locale ne pourra pas dans le même appel de méthode contenir un `short` puis une référence à un objet). Puis, un vérifieur est embarqué sur la carte pour assurer l'application de cette propriété. Un premier problème est que cette méthode nécessite que les applications passent au travers d'un outil propriétaire [158] fourni par Trusted Logic car sinon elles sont rejetées. Mais le problème le plus important de cette approche est l'augmentation du nombre de variables et une diminution de leur réutilisabilité pour un périphérique où la mémoire est rare. Donc, la carte pourra avoir besoin de plus de ressource mémoire pour exécuter un programme et elle pourrait refuser des codes optimisés.

Enfin, la solution idéale serait un vérifieur embarqué autonome. En effet, cette solution est parfaite puisqu'un tel vérifieur ne refusera jamais une *applet* valide et qu'il est décentralisé. Ses seuls inconvénients sont la complexité en temps et en mémoire. Cette solution a longtemps été considérée comme impossible et mais elle a été récemment réalisée par Damien DEVILLE et Gilles GRIMAUD [105].

Nous verrons dans la partie 3 de cette thèse ce que nous considérons comme une carte à puce multi-applicative ouverte en se basant sur ces différents vérifieurs et nous montrons que malgré tout, il peut exister des attaques qui pourraient nécessiter de revoir les algorithmes de vérification du *bytecode*.

On pourra remarquer que des travaux [75] ont conclu à l'équivalence de la machine virtuelle défensive et de la machine virtuelle offensive (*i.e.* une machine virtuelle ne faisant pas de vérification à l'exécution) si on lui adjoint le vérifieur extérieur – embarqué ou non.

2.6.2 Le pare-feu

Les contextes

Comme nous l'avons déjà évoqué précédemment le pare-feu a pour mission d'isoler les *applets* et les différents objets créés au sein d'espaces appelés contextes. Un contexte est associé avec un paquetage de telle sorte que toutes les *applets* d'un même paquetage partagent le même contexte. Le JCRE possède son propre contexte avec des privilèges spéciaux.

Les concepts de propriétaire, d'accès au objet et de changement de contexte

Si la machine virtuelle exécute un *bytecode* d'invocation et si certaines conditions sont réunies, alors elle fait un changement de contexte en ajoutant sur une pile le contexte

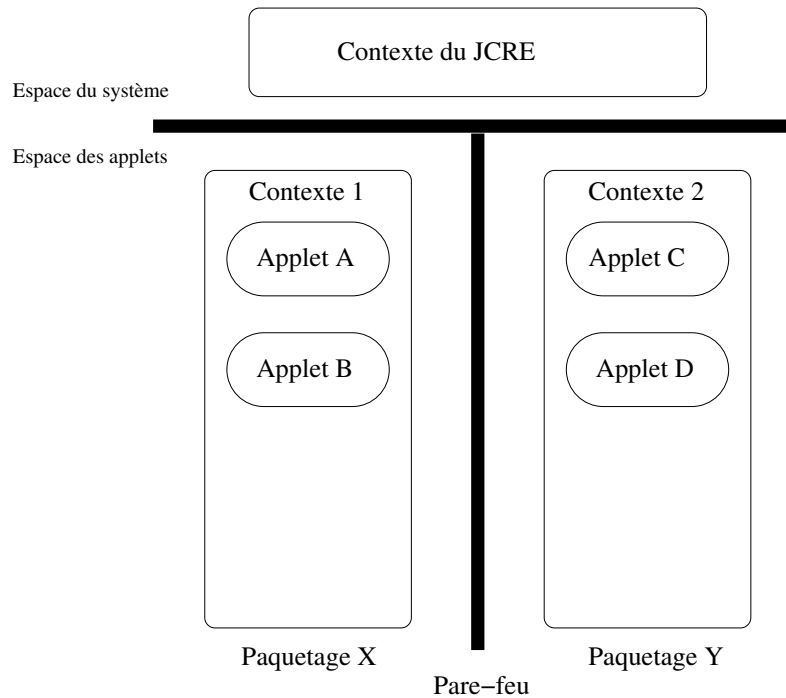


FIG. 21 – Le pare-feu Java Card.

courant et en choisissant comme nouveau contexte courant le contexte du propriétaire de l'objet sur lequel la méthode est invoquée. À la fin de l'exécution de la méthode, le contexte de l'appelant est restauré en l'enlevant de la pile pour en faire le contexte courant.

Ainsi, sur la plate-forme Java Card chaque objet créé appartient soit au contexte d'une *applet* soit à celui du JCRE : le propriétaire de l'objet créé étant l'*applet* du contexte courant. De plus, un objet est accessible seulement depuis le contexte de son propriétaire ce qui évite tout accès non autorisé à cet objet.

Illustrations des changements de contexte

Nous présenterons ici une illustration des changements de contexte (*cf.* FIG. 22). Prenons par exemple deux *applets* A et B qui sont dans le même paquetage X et une *applet* C qui est dans un autre paquetage Y. A et B partagent donc le même contexte, le Contexte 1, et C est dans un contexte différent, le Contexte 2.

Si le Contexte 1 est le contexte actuellement actif et qu'une méthode *m0* d'un objet appartenant à l'*applet* A est appelée, alors aucun changement de contexte ne se produit. Si la méthode *m0* appelle une méthode *m1* d'un objet appartenant à l'*applet* B alors une fois encore aucun changement de contexte ne se produit (malgré le changement de propriétaire de l'objet) et aucune restriction du pare-feu ne s'applique.

Si la méthode *m1* appelle maintenant une méthode *m2* d'un objet appartenant à l'*applet* C, les restrictions du pare-feu s'appliquent et si l'accès est permis (*e.g.* la méthode *m1* possède une référence sur l'objet de C et *m2* est une méthode d'une interface partageable

implantée par l'objet) alors un changement de contexte se produira et le Contexte 2 deviendra le contexte courant. Au retour dans la méthode m1 à la fin de l'exécution de la méthode m2, le contexte de l'*applet* B est restauré et le Contexte redevient le contexte courant.

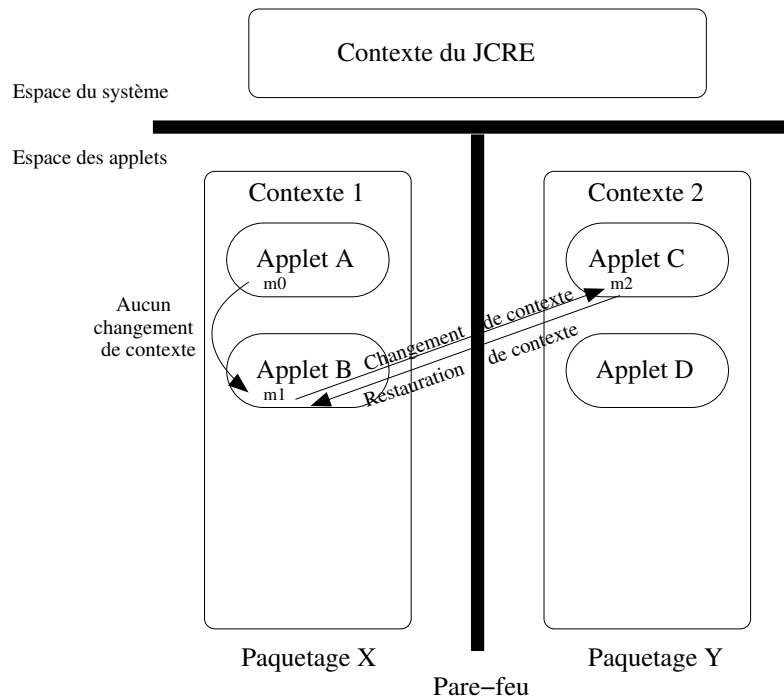


FIG. 22 – Illustration du changement de contexte.

La communication inter-*applet*

Afin de permettre les interactions entre les *applets*, Java Card définit le concept d'interface partageable (*i.e.* shareable interface). C'est un mécanisme qui permet d'invoquer depuis un contexte les méthodes d'un objet qui ne lui appartient pas. On remarquera que les champs et les autres méthodes de cet objet ne sont pas accessibles et que seules les méthodes définies dans l'interface partageable qu'implante l'objet le sont.

Ainsi, quand une méthode d'une interface partageable sera invoquée, un changement de contexte interviendra sous le contrôle du JCRE vers le contexte du propriétaire de cet objet.

C'est grâce à ce mécanisme que l'on peut avoir une communication inter-*applet*.

Les points d'entrée du JCRE

La carte à puce comme tous les systèmes informatiques sécurisés doit fournir un moyen pour une application d'un simple utilisateur (*i.e.* une application qui est restreinte à l'utilisation de quelques ressources basiques) d'utiliser des services systèmes implantés par des

primitives aux droits privilégiés.

Pour cela Java Card a prévu l'utilisation des objets appelés points d'entrée du JCRE (JCRE Entry Point). Ces objets appartiennent au contexte du JCRE mais ils sont marqués comme contenant des méthodes « point d'entrées ».

Le pare-feu protège ces objets de l'accès par les *applets*. La désignation « point d'entrée » signifie que les méthodes de ces objets peuvent être appelées depuis n'importe quel contexte. Lorsque cela se produit, il y a un changement de contexte vers celui du JCRE. Ces méthodes sont donc des passerelles grâce auxquelles les *applets* demandent des services systèmes possédant les privilèges du JCRE. Le service appelé est réalisé par la méthode du point d'entrée une fois vérifié que les paramètres de la méthode sont dans les limites et que tous les objets passés en paramètres sont accessibles depuis le contexte de l'appelant.

Il y a deux catégories de points d'entrée du JCRE :

- Les points d'entrée temporaires.

Comme tous les points d'entrée du JCRE, les méthodes des points d'entrée temporaires peuvent être appelées depuis n'importe quel contexte. Les références à ces objets ne peuvent pas être stockées dans des variables de classe, variables d'instance ou éléments d'un tableau. Grâce à une partie du pare-feu, le JCRE détecte et limite les tentatives de stockage de références à ces objets afin d'empêcher une réutilisation non autorisée. L'objet APDU et toutes les exceptions appartenant au JCRE sont des exemples de points d'entrée temporaires du JCRE.

- Les points d'entrée permanents. Comme tous les points d'entrée, les méthodes des points d'entrée permanents peuvent être appelées depuis n'importe quel contexte. De plus, les références à ces objets peuvent être librement stockées et réutilisées.

Les instances d'AID appartenant au JCRE sont des exemples de points d'entrée permanents.

Seul le JCRE peut construire des points d'entrée et décider si ils sont permanents ou temporaires.

Les tableaux globaux

La nature globale de quelques objets exige qu'ils soient accessibles à partir de n'importe quel contexte. Or, comme par défaut le pare-feu empêche les objets d'être employés d'une façon aussi flexible, la machine virtuelle Java Card permet à des objets particuliers d'être marqués comme global.

Tous les tableaux globaux sont des objets globaux temporaires. Ces objets appartiennent au contexte du JCRE mais peuvent être consultés depuis n'importe quel contexte. Les références à ces objets ne peuvent pas être stockées dans des variables de classe, variables d'instance ou éléments d'un tableau. Ici aussi, grâce à une partie du pare-feu, le JCRE détecte et limite les tentatives de stockage de références à ces objets afin d'empêcher une réutilisation non autorisée. Pour plus de sécurité, seuls les tableaux peuvent être marqués comme globaux et seul le JCRE peut construire des tableaux globaux. Puisque les *applets* ne peuvent pas en créer, l'API ne définit aucune méthode pour cela.

Actuellement, les seuls tableaux globaux requis par les APIs Java Card sont le buffer APDU et le tableau d'octets passé en paramètre d'entrée (*i.e.* `bArray`) à la méthode `install` des *applets*.

Les privilèges du JCRE

Puisque c'est le contexte système, le contexte du JCRE a un privilège spécial. Il peut appeler une méthode de n'importe quel objet sur la carte.

Par exemple, supposons que l'objet X appartienne à l'*applet* A. Normalement, seul le contexte de A peut accéder aux champs et aux méthodes de X.

Cependant le contexte du JCRE possède le droit d'appeler n'importe laquelle des méthodes de X. Pendant une telle invocation, un changement de contexte se produit depuis le contexte du JCRE vers celui de l'*applet* qui possède X.

Le contexte du JCRE est le contexte couramment actif quand la machine virtuelle commence à fonctionner après une remise sous tension de la carte. En quelque sorte, le contexte du JCRE est le contexte racine et il est toujours le contexte couramment actif ou le contexte sauvé en bas de la pile.

2.7 Les inconvénients de la technologie Java Card

Loin d'être un plaidoyer contre Java Card cette section va lister objectivement de multiples inconvénients de cette technologie et va décrire pourquoi Java Card est un langage bien à part et très différent de Java. En effet, introduire des concepts similaires sur la plate-forme Java Card (*e.g.* RMI) à ceux déjà existants sur la plate-forme Java ne suffit pas pour autant à faire converger ces deux langages tant les différences sémantiques sont énormes.

2.7.1 Les problèmes de sémantique

La principale raison pour laquelle Java Card ne sera jamais le langage Java est que les spécifications Java Card sont fait un sur-ensemble d'un sous-ensemble de Java.

Tout d'abord, c'est un sous-ensemble puisque comme nous l'avons vu à la section 2.4.1 les spécifications Java Card n'ont pas repris certains concepts de Java, mais c'est aussi un sur-ensemble car pour répondre aux problématiques des cartes à puce elles ont introduit les concepts de persistance/transience et pour répondre à l'absence de vérifieur embarqué, elles ont introduit un pare-feu. Ces deux mécanismes modifient si profondément la sémantique du langage que si on laisse un programmeur confirmé Java écrire une *applet*, il y a fort à parier que son code ne sera pas efficace et contiendra même de nombreux problèmes.

La persistance

L'introduction de Java qui est un environnement de programmation temporaire sur un périphérique à mémoire persistante et les contraintes de sécurité que doit assurer le

langage Java Card font que le code ci-dessous (*cf.* listing 2.1) a une sémantique totalement différente selon la plate-forme sur laquelle il est exécuté.

Listing 2.1 – Différences sémantiques dues à la persistance.

```
public class A {
    private byte b1;
    private Object o1;

    public void foo() {
        b1 = (byte) 0;
        o1 = new Object();
    }

    public void bar() {
        byte b2 = (byte) 1;
        Object o2 = new Object();
    }
}
```

Imaginons que l'on ait une instance *a* de la classe *A*. Que se passe-t-il si l'on invoque *a.foo()* et *a.bar()* en Java et en Java Card ?

Tout d'abord, en Java puisque l'environnement de programmation est temporaire l'instance sera un objet localisé en RAM et dont les champs seront donc aussi en RAM. Ensuite, l'appel à la méthode *foo()* déclenchera l'affectation de la valeur 0 à la variable d'instance *b1* et d'une référence de valeur que nous appellerons *x* à la variable d'instance *o1*. Par ailleurs, avant l'affectation de la valeur *x* à *o1* l'objet référencé par *o1* aura été créé dans la RAM. Ainsi, on constate que tout (*i.e.* 0, *x* et l'objet) a été créé dans un même espace volatile, la RAM.

Toujours en Java, si maintenant on appelle la méthode *bar()*, puisque nous avons deux variables locales *b2* et *o2* ces variables sont elles aussi en RAM et nous aurons une affectation de la valeur 1 à la variable d'instance *b2* et d'une référence de valeur que nous appellerons *y* à la variable d'instance *o2*. Ici aussi avant l'affectation de la valeur *y* à *o2* l'objet référencé par *o2* aura été créé dans la RAM. On constate donc toujours que tout (*i.e.* 1, *y* et l'objet) a été créé un même espace volatile, la RAM.

Reproduisons les mêmes appels en Java Card. Puisque l'environnement de programmation de Java Card est un environnement persistant l'instance sera un objet localisé en NVM et dont les champs seront donc aussi en NVM. Ainsi, l'appel à la méthode *foo()* déclenchera l'affectation de la valeur 0 à la variable d'instance *b1* et d'une référence de valeur que nous appellerons *x* à la variable d'instance *o1*. Par ailleurs, avant l'affectation de la valeur *x* à *o1* l'objet référencé par *o1* aura été créé en NVM. On constate ici que tout (*i.e.* 0, *x* et l'objet) a été créé dans un même espace persistant la NVM.

Mais maintenant, si on appelle la méthode *bar()*, puisque nous avons deux variables locales *b2* et *o2* ces variables sont elles aussi en RAM et nous aurons une affectation de la valeur 1 à la variable d'instance *b2* et d'une référence de valeur que nous appellerons *y* à la variable d'instance *o2*. De plus, avant l'affectation de la valeur *y* à *o2* l'objet référencé par *o2* aura été créé en NVM. On remarque donc ici qu'il y a une dissymétrie dans la sémantique du langage due à sa persistance puisqu'une partie (*i.e.* 1 et *y*) a été créée dans

un espace volatile, la RAM, alors qu'une autre partie (*i.e.* l'objet) l'a été en NVM.

On notera donc que le programmeur Java doit bien être au courant de ce qu'il fait lorsqu'il alloue un objet et cela même dans une variable temporaire (*i.e.* cas de la méthode `bar()`). En effet, s'il pensait qu'un retrait de la carte du lecteur suivi de sa réinsertion suffirait à désallouer l'objet précédemment créé, il a tort aussi bien dans le cas de la méthode `foo()` que dans le cas de la méthode `bar()` car tous les objets sont alloués en NVM. Le programmeur doit donc sans cesse favoriser la réutilisation d'objets plutôt que l'allocation.

Nous venons d'illustrer le comportement de la plupart des Java Cards, mais nous verrons au chapitre 8 qu'une lecture attentive des spécifications soulève de nombreuses ambiguïtés concernant ces aspects sémantiques. Pour cela, nous introduirons la notion de pré-persistance afin de proposer de nouveaux modèles mémoire et d'expliquer celui déjà existant.

Le pare-feu

L'introduction du pare-feu dans Java Card est sûrement ce qui a le plus dénaturé le langage par rapport à Java. En effet, les contraintes qu'il impose sont tellement fortes qu'il est assez orthogonal aux règles de contrôles d'accès classiques de Java. Prenons l'exemple suivant (*cf.* listing 2.2) où l'on dispose de trois paquetages différents `X`, `Y` et `Z`.

`Z` est un paquetage de bibliothèque qui ne contient pas d'*applet* et seulement deux classes `Foo` et `Bar`. La classe `Foo` ne contenant qu'un champ public statique qui servira d'intermédiaire et `Bar` ne proposant qu'une méthode publique `test()` pour illustrer notre propos. `X` contient une *applet* `A` qui va créer un objet (*i.e.* une instance de `Bar`) dont elle sera donc le propriétaire et elle va le placer dans un champ public `Foo.bar`. `Y` contient une *applet* `C` appartenant donc à un contexte différent de l'*applet* `A`.

Cette *applet* `C` va essayer de récupérer – comme cela fonctionnerait en Java – la référence sur l'instance de `Bar` placée dans le champ statique public `Foo.bar` afin d'invoquer dessus la méthode publique `test()`. Cela fonctionne jusqu'au moment de l'invocation où, l'*applet* `C` et l'*applet* `A` appartenant à des contextes différents, le pare-feu empêche l'opération. Ainsi, l'*applet* `C` peut accéder à la référence sur l'instance de `Bar` mais ne peut rien en faire. On remarquera donc qu'une fois encore la sémantique d'un même code en Java et en Java Card est différente.

Pour résumer, le pare-feu rajoute donc des contraintes supplémentaires à celles dues au contrôle d'accès classique de Java. Le mécanisme de pare-feu permet ainsi d'éviter qu'un programmeur inattentif ne laisse des références sur des objets lui appartenant (*i.e.* des objets à protéger), dans des champs potentiellement accessibles à un attaquant (*e.g.* un champ public statique). En effet, si ce dernier récupère une telle référence le pare-feu l'empêchera de l'utiliser.

Notons que si l'*applet* `A` avait voulu intentionnellement partager une instance de `Bar`, il aurait fallu que cette classe implante une interface partageable ainsi que le propose le listing 2.3 qui devrait remplacer le paquetage `Z` de l'exemple précédent. En effet, dans ce cas l'*applet* `C` aurait pu faire appel à la méthode `test()` et même si ce n'est pas la méthode classique de communication inter-*applet* recommandée cette méthode de partage

fonctionne. Toutefois, il vaut mieux déconseiller son utilisation car, à l'inverse du mécanisme classique de communication inter-*applet*, celui-ci ne permet pas de contrôle d'accès sur la référence. Toutes les *applets* peuvent récupérer cette référence (et utiliser la méthode `test()`), ce qui va à l'encontre de la raison pour laquelle le pare-feu a été introduit dans Java Card : une protection de toutes les références sauf si le partage est explicitement spécifié et ce pour renforcer la sécurité au maximum vis-à-vis des erreurs d'un programmeur inattentif.

On notera aussi qu'il est impossible de stocker les références sur un point d'entrée temporaire ou sur un tableau global dans une variable d'instance, de classe ou dans un élément d'un tableau. En conséquence, on peut seulement stocker ces références dans une variable locale. Donc, à nouveau, on a des limitations sur certains objets ce qui conduit une fois encore à des dissymétries dans la sémantique alors que l'on ne rencontre pas ces problèmes en Java. On se heurte également aux mêmes limitations lorsqu'on veut stocker des arguments de type tableau d'une méthode distante appelée sur la carte (*i.e.* JCRMI). En effet, ceux-ci sont représentés par des tableaux globaux, donc comme ci-dessus ils ne peuvent être stockés que dans des variables locales. Une fois encore, le comportement d'un programme utilisant RMI sur Java ou sur Java Card sera donc différent.

En conséquence, on peut légitimement se demander pourquoi cette solution du pare-feu a été choisie pour dialoguer entre les applications. La solution utilisée par exemple par MULTOS [176] est bien plus élégante puisque la communication entre *applets* consiste tout simplement à envoyer des commandes APDUs à l'*applet* serveur, reléguant la communication inter-*applet* à l'équivalent d'une communication lecteur à carte.

2.7.2 Les problèmes de la politique de développement

Un autre gros inconvénient de Java Card est lié à sa politique de développement qui se fait via le Java Card Forum (JCF). En effet, c'est la seule technologie Java à ne pas avoir de Java Community Process (JCP). Ainsi, les membres du JCF discutent des solutions techniques puis font des propositions à Sun microsystems qui décide ou non de leur intégration.

Or, le problème de ce mode de développement est que seuls les membres du JCF peuvent proposer des solutions, oubliant parfois une partie des recherches menées dans le monde institutionnel.

Par ailleurs, pour des raisons de compatibilité arrière compréhensibles, le JCF n'a pas suivi les solutions proposées pour les mécanismes de pare-feu [173], de transactions [178] et pour les transients [179]. Ceci est vraiment dommage car ces propositions amélioreraient grandement le modèle de programmation Java Card.

2.7.3 Les problèmes d'intégration

Aujourd'hui, alors que Java Card est largement répandu, Sun microsystems n'a toujours pas développé d'outils spécifiques pour cette cible (*e.g.* compilateur, etc.). Pour développer ses applications, l'utilisateur final doit utiliser le compilateur Java standard. Or, comme le type par défaut du compilateur est `int` et comme bien souvent les `int` ne sont pas

disponibles sur les cartes, le développeur doit ajouter des *casts* partout, ce qui donne un code horrible à lire et donc un plus grand risque d'erreurs.

Il suffirait pourtant à Sun microsystem de reconnaître Java Card comme un langage différent de Java et de développer un compilateur/convertisseur spécifique pour cette technologie.

Un autre exemple flagrant des problèmes d'intégration de cette technologie sont les fichiers CAP. En effet, bien que spécifié, plusieurs constructeurs ne suivent pas le format standard, ce qui pose quelques difficultés d'interopérabilité. Pour résoudre ce problème nous avons développé dans la suite JCatools un outil de conversion entre ces différents formats (*cf.* section 7.3).

2.7.4 Les problèmes de sécurité des applications

Enfin, un problème important mais qui n'est pas spécifique à Java Card est celui de la sécurité des applications embarquées.

Si la sécurité de la plate-forme relève de la responsabilité du fournisseur de cartes, celle de l'application est de la responsabilité du fournisseur d'applications. En effet, même si l'application se base sur la sécurité de la plate-forme, c'est à elle de mettre en place les mécanismes sécuritaires qui lui sont spécifiques.

À l'inverse, comme nous l'avons vu précédemment le système d'exploitation, lui, ne doit pas faire d'hypothèse sur les applications et doit donc jouer pleinement son rôle de protecteur des applications vis-à-vis des autres applications dans l'hypothèse où un code malicieux serait chargé (*i.e.* c'est ce que fait le pare-feu).

Ainsi, si la sécurité de la plate-forme Java Card est très bonne, une mauvaise utilisation des mécanismes qu'elle propose peut réduire à néant les efforts de sécurisation (*e.g.* comme nous l'avons vu plus haut, le partage via un champ statique d'un objet implantant l'interface **Shareable**). C'est pourquoi, un des points faible de Java Card, est le manque de guides de développement à l'attention du programmeur naïf souhaitant développer des *applets* sécurisées.

2.8 Les différentes recherches sur Java Card

Dans cette section, nous allons exposer les recherches menées sur Java Card de façon thématique.

Les travaux autour des APIs Java Card concernent pour l'essentiel leur modélisation formelle ou semi-formelle [182, 183, 76, 163]. Ils utilisent des langages tels que JML [63, 182, 15] ou ESC/Java [14, 15] pour traduire les spécifications informelles, puis ils vérifient la correction d'une implantation à l'aide d'un outil de preuve (*e.g.* PVS [52]). Bien évidemment, cette vérification n'est possible que grâce à une traduction préalable des spécifications JML et du code Java en un langage compréhensible par un outil de preuve, avec à des outils comme LOOP [199]. Dans ces travaux différentes propriétés commencent à être introduites pour exprimer des contraintes temporelles [198] et matérielles [143].

Pour la JCVM, les travaux consistent principalement à en établir une description formelle, et à modéliser sa sémantique opérationnelle dans divers langages de preuve (*i.e.* Coq, Isabelle/HOL) pour vérifier la correction de certaines propriétés [74, 193]. Par ailleurs, afin de faciliter les différentes étapes de description, de modélisation et de vérification, un ensemble d'outils, appelé Jakarta, a été développé [73].

Concernant le pare-feu, les travaux portent essentiellement sur l'analyse statique du code pour vérifier si le partage d'objets est bien fait, mal fait, ou indéterminé [81]. Il existe aussi des travaux établissant une spécification formelle [194] et définissant une sémantique opérationnelle pour le pare-feu [161]. Enfin, le reste des travaux se concentre sur le développement de kits de programmation assurant une bonne gestion par les applications du partage d'objets [180].

Dans le domaine des transactions, il existe des travaux concernant le traitement à apporter aux d'objets créés lors des transactions [178]. D'autres travaux présentent une façon d'introduire les notions d'arrachage de carte et de transaction dans des spécifications JML [143].

Pour le vérifieur, nous avons déjà évoqué les travaux connexes dans la section 2.6.1. Si dans ce domaine des travaux ont conclu à l'équivalence de la machine virtuelle défensive et de la machine virtuelle offensive à laquelle on adjoint un vérifieur [75], d'autres ont permis de construire à partir d'une machine virtuelle défensive, une machine virtuelle offensive et le vérifieur associé.

Des travaux ont montré à l'aide de Coq et d'une sémantique opérationnelle abstraite définie pour le fichier **CAP** et les fichiers **class** correspondants, qu'il y avait une relation logique permettant de prouver la correction des optimisations faites lors de la construction d'un fichier **CAP** à partir des **class** initiales [103, 104].

Dans le domaine de la multi-application, nous présenterons les travaux connexes dans la section 11.4.

Dans le domaine de la vérification d'*applets*, on trouve des méthodologies [191] et des outils (*e.g.* JIVE [33]) permettant de vérifier qu'une *applet* suit bien une spécification informelle en utilisant une fois encore des langages comme JML. Par ailleurs, des travaux et des outils (*e.g.* JCAVE [32]) sont également disponibles afin d'établir les interactions entre des *applets* au niveau bytecode en utilisant pour modéliser le comportement des *applets* une logique temporelle linéaire [94]. Enfin, d'autres travaux proposent de réaliser des tests de conformité en générant un jeu de tests à partir d'un modèle UML des *applets* et de leurs communications avec les autres composants, grâce aux outils UMLAUT et TGV [162]. Évidemment, il est également possible de tester une *applet* dans un générateur de preuve en convertissant les fichiers **CAPs** à l'aide d'outils de Jakarta.

S'il est possible de citer encore beaucoup d'autres travaux, nous avons décrit ce que nous pensons être les principales recherches qui ont été menées sur Java Card lors de ces dernières années.

Listing 2.2 – Différences sémantiques dues au pare-feu.

```
package X;

import Z.*;

public class A extends Applet {

    public void process(APDU apdu) {
        ...
        Foo.bar = new Bar();
        ...
    }
    ...
}
```

```
package Y;

import Z.*;

public class C extends Applet {

    public void process(APDU apdu) {
        ...
        Bar bar = Foo.bar;
        bar.test();
        ...
    }
    ...
}
```

```
package Z;

public class Foo {

    public static Bar bar;

}

public class Bar {

    public void test() {
        ....
    }

}
```

Listing 2.3 – Une méthode pour le partage d’objet inter-*applet*.

```
package Z;

import javacard.framework.Shareable;

public interface BarShareableInterface extends Shareable {
    public void test();
}

public class Foo {
    public static Bar bar;
}

public class Bar implements BarShareableInterface {
    public void test() {
        ....
    }
}
```


Chapitre 3

GlobalPlatform

Dans ce chapitre nous évoquerons quelques problématiques liées à la gestion d'applications multiples sur une même carte. Par exemple, comment charger ces applications ? Comment les gérer efficacement sur la carte ? Quels sont leurs cycles de vie ?

En effet, nous avons vu dans la section 2.4.2 que les spécifications Java Card ne traitaient pas ces problèmes et nous verrons ce qu'apportent dans ce domaine les spécifications GlobalPlatform [19].

Enfin, nous expliquerons pourquoi GlobalPlatform a eu un tel succès auprès des décideurs.

3.1 Quelques problématiques de la multi-application

Les problématiques liées à la présence de multiples applications sur une même carte que nous entendons présenter ici sont celles portant sur la gestion de ces applications. En effet, nous verrons quelques problèmes dus aux attaques internes au chapitre 10 mais également de nouveaux problèmes au chapitre 11.

Le premier problème engendré par la cohabitation de plusieurs applications sur la même carte est « Comment et quand les charger ? ». Faut-il des procédures de chargement sécurisé ? Tout dépendra du niveau de confidentialité du code de l'application mais aussi de l'environnement physique dans lequel se trouve la carte.

Ensuite, toujours dans un environnement multi-applicatif, le second problème est d'être sûr que l'application que l'on vient de charger ne va pas nuire aux autres applications ou à la carte. En effet, jusqu'à aujourd'hui très peu de cartes multi-applicatives – voire aucune – n'embarquent de vérifieur permettant d'assurer l'innocuité du code chargé. En conséquence, quels mécanismes faut-il mettre en place pour avoir un niveau d'assurance suffisant ?

D'autres questions se posent, ainsi, « Quel est le cycle de vie d'une application sur la carte ? » et « Quelles opérations une application présente sur la carte peut-elle être autorisée à réaliser sur la carte au risque de modifier l'état global du système ? ».

Enfin il faut se demander « Comment une application utilisatrice hors-carte peut être sûre qu'elle dialogue avec la bonne application sur la carte et vice-versa ? » et « Quelles sont

les procédures à mettre en place pour assurer que chacun est celui qu’il prétend être ? ». En effet, sur les cartes mono-applicatives ce problème n’existait pas puisque lorsqu’une application hors-carte dialoguait avec une carte, c’était forcément avec la bonne application – sauf dans le cas d’une carte contrefaite mais dans ce cas le problème est un peu différent.

L’objectif de GlobalPlatform est de résoudre ces problèmes que l’on rencontre lorsqu’on veut utiliser des cartes multi-applicatives sur une grande échelle.

3.2 Présentation

Initialement développées par Visa International au sein de l’initiative Open Platform [202], ces spécifications ont été transférées entre la version 2.0.1 et la version 2.1 au consortium GlobalPlatform regroupant des industriels du secteur bancaire et des communications, et des acteurs gouvernementaux. Aujourd’hui, c’est lui qui est chargé de maintenir et de promouvoir ces spécifications sous le nom de *GlobalPlatform Specifications*. On notera qu’on retrouve encore souvent dans la littérature les termes OP (OpenPlatform) ou encore VOP (Visa OpenPlatform) pour désigner les spécifications GlobalPlatform. D’ailleurs, la dernière version des spécifications GlobalPlatform (la 2.1.1) consiste principalement en une correction de la version 2.1 sur ces terminologies afin d’aboutir à une plus grande cohérence des documentations.

Le but des spécifications GlobalPlatform était d’apporter un standard qui manquait jusqu’alors au monde des cartes : une spécification pour gérer les cartes de façon indépendante du matériel, des vendeurs et des applications. En effet, le marché des Technologies de l’Information (TI) est excessivement volatile et son marché nécessite souvent d’être capable de déployer rapidement des applications sur les cartes. Seulement, le déploiement via des processus de masquage de nouvelles applications sur de nouvelles cartes est incompatible en terme de délai et de coût avec les besoins du marché. Ainsi, si les cartes multi-applicatives (cf. section 1.9) avaient permis d’apporter la réduction du coût grâce à la possibilité de charger des applications à différents moments du cycle de vie de la carte, mais également grâce à la concurrence que se livrent les différents fabricants de cartes multi-applicatives, aucun standard n’existait pour spécifier une infrastructure permettant la réduction des délais. C’est pourquoi les spécifications GlobalPlatform fournissent une architecture globale de gestion de la sécurité et des cartes.

En d’autres termes, GlobalPlatform définit des spécifications flexibles et efficaces pour que des émetteurs de cartes puissent créer des systèmes utilisant des cartes multi-applicatives pour répondre à l’évolution de leurs besoins industriels. Ainsi, ces spécifications leur permettent de choisir la technologie de cartes multi-applicatives correspondant le plus à leurs besoins actuels tout en s’assurant qu’ils pourront également utiliser, au besoin, une technologie de carte différente dans l’avenir sans que cela ait un impact significatif sur leur infrastructure.

Les spécifications GlobalPlatform actuelles portent sur les aspects relatifs :

- aux cartes, qui incluent principalement :
 - *Card Specification v2.1.1* [122],
 - *Guidelines for Developing Java Card applications on GlobalPlatform Cards v1.0*

[129],

- *Card Security Requirements Specifications v1.0* [121];
- aux terminaux, qui incluent :
 - *GlobalPlatform Device API v2.0* [124],
 - *Device Tools Supporting Documents* [123],
 - *GlobalPlatform Guidelines for Using GPD 2.0 /STIP 2.1 API v1.0* [125];
- aux systèmes, qui incluent :
 - *GlobalPlatform Messaging Specification v1.0* [126],
 - *Key Management Requirements Systems Functional Requirements Specification* [130],
 - *GP Load and Personalization Interface v1.0* [128],
 - *GP Guide to Common Personalization v1.0* [127].

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

- les conditions d'interopérabilité et d'indépendance relativement à la cible pour implanter GlobalPlatform ;
- les communications hors carte avec le terminal et les communications sur la carte pour la gestion des applications, *i.e.* une API et comment l'utiliser pour développer des applications pour la carte.

3.3 L'architecture GlobalPlatform

L'architecture GlobalPlatform comprend un certain nombre de composants permettant de s'abstraire du matériel et du vendeur grâce à des interfaces pour les applications (*i.e.* une API standardisée) et pour le système de gestion hors carte (*i.e.* des APDUs standardisés). La figure 23 montre un exemple de configuration comprenant une application de l'émetteur de carte (*Card Issuer*) et une application d'un de ses partenaires commerciaux appelé fournisseur d'applications (*Application Provider*). Dans le modèle, afin d'être portables, toutes les applications doivent être implantées au sein d'un environnement d'exécution sécurisé utilisant une API indépendante du matériel. Néanmoins, GlobalPlatform n'impose aucune technologie particulière quant à l'environnement d'exécution. Pour l'anecdote, on notera que les spécifications GlobalPlatform lorsqu'elles étaient sous le contrôle de Visa International étaient très orientées vers la plate-forme Java Card car son concurrent direct MasterCard supportait lui le standard MULTOS concurrent de Java Card.

En tout cas, aujourd'hui, GlobalPlatform peut donc être utilisé pour n'importe quelle technologie de carte multi-applicative, mais aussi pour des cartes mono-applicatives même si ce n'est pas là sa cible première.

Le gestionnaire de la carte (*Card Manager*) est le principal composant de GlobalPlatform et il joue le rôle d'administrateur central sur une carte GlobalPlatform. De plus, des applications spéciales, appelées « domaines de sécurités » (*Security Domains*), sont aussi créées pour gérer la sécurité en fonction des privilèges accordés.

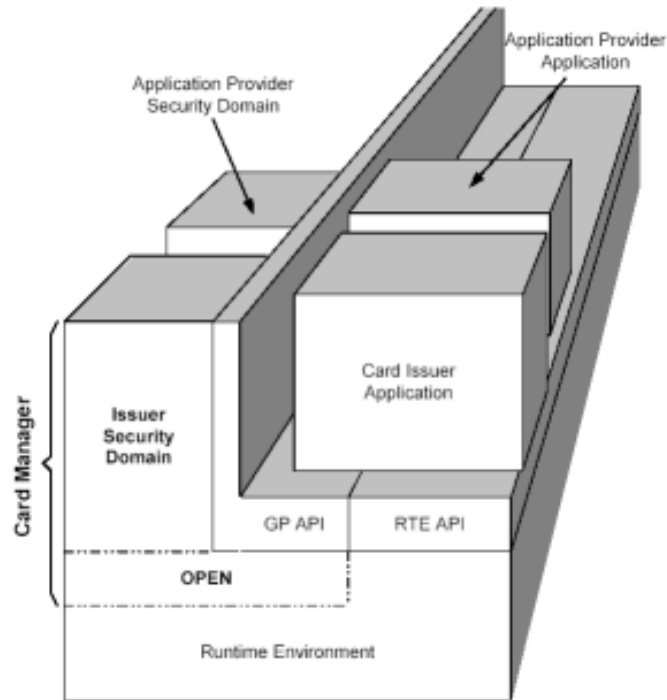


FIG. 23 – Architecture d’une carte GlobalPlatform (source : GlobalPlatform).

3.3.1 L’environnement d’exécution (*RTE*)

GlobalPlatform est prévu pour fonctionner sur tout environnement d’exécution sécurisé pour carte à puce multi-applicative. Cet environnement d’exécution doit fournir une API indépendante du matériel pour les applications. Tout environnement d’exécution basé sur une machine virtuelle est donc le bienvenu. Mais, le RTE doit aussi fournir aux applications un espace de stockage et d’exécution sécurisés afin d’assurer que le code et les données de chaque application pourront demeurer isolés et à l’abri des autres applications présentes sur la carte. Ceci est réalisé à l’aide du mécanisme de pare-feu vu précédemment sur différentes cartes multi-applicatives. Comme nous l’avons déjà souligné, on retrouve dans cette définition du RTE une définition très proche de la cible initiale de GlobalPlatform, à savoir Java Card.

3.3.2 Le gestionnaire de la carte (*Card Manager*)

Le Card Manager en tant qu’administrateur central de la carte a de multiples responsabilités. Il est le représentant de l’émetteur de carte sur cette dernière au travers du domaine de sécurité de l’émetteur (Issuer Security Domain). Certaines de ses autres responsabilités sont parfois implantées par le RTE. Elles sont décrites dans la section ci-dessous.

L'environnement GlobalPlatform (*OPEN*)

Les principales responsabilités de l'environnement de GlobalPlatform (anciennement OPEN pour *Open Platform ENvironment*) sont de fournir une API aux applications, de router les commandes APDUs vers la bonne application, de sélectionner les applications, de gérer les canaux logiques (facultatif), et surtout de gérer le contenu de la carte. OPEN doit fournir ces fonctions si elles ne sont pas déjà disponibles dans le RTE, ou si elles sont disponibles mais d'une façon qui n'est pas compatible avec ces spécifications.

OPEN réalise donc le chargement de code des applications et gère le contenu de la carte. Il contrôle également l'installation des applications chargées sur la carte. En outre, il est chargé d'assurer le respect des politiques de sécurité définies pour le chargement et l'installation. Ces politiques englobent la vérification du code de l'application et que l'autorisation de charger et/ou installer a bien été produite par l'émetteur de carte.

Une autre fonction importante fournie par OPEN est le routage de commandes APDUs et la sélection d'application. Quand une commande SELECT est reçue, OPEN marque l'application référencée dans la commande SELECT comme l'application couramment sélectionnée et les commandes APDU suivantes seront expédiées vers cette application.

La disponibilité des canaux logiques ajoute une dimension supplémentaire à l'architecture de la carte puisqu'ils permettent à plusieurs applications d'être sélectionnées de façon concurrente mais également à une même application d'être sélectionnée plusieurs fois sur différents canaux grâce à un mécanisme appelé *multisélection*. Pour implanter cette fonctionnalité, OPEN pourra se baser sur le RTE si celui-ci offre les mécanismes permettant d'être informé si une application accepte la multisélection ou la sélection concurrente. Même s'il supporte la gestion des canaux logiques, OPEN fonctionne également très bien avec les applications refusant la sélection concurrente et la multisélection (*i.e.* ce sont souvent des applications anciennes). De plus, le support des canaux logiques est facultatif.

OPEN possède et emploie un registre interne, appelé GlobalPlatform Registry, qui contient les informations sur les codes exécutables chargés, les applications présentes, leurs associations avec les domaines de sécurité et leurs privilèges.

De par la nature des opérations que réalise OPEN sur une Java Card, une partie au moins de son code doit généralement être développée en natif et une autre partie peut se programmer en Java Card (*e.g.* le registre GlobalPlatform, etc.).

Le domaine de sécurité de l'émetteur (*Issuer Security Domain*)

Puisqu'il est le représentant de l'émetteur sur la carte, le domaine de sécurité de l'émetteur a la possibilité de charger, d'installer et d'effacer les applications appartenant à l'émetteur mais aussi celles d'autres fournisseurs d'applications. Il est donc très similaire avec les autres domaines de sécurité définis dans la section suivante. On pourra noter que sur une Java Card, cette entité devrait normalement pouvoir être programmée en Java sous la forme d'une *applet* classique car elle ne réalise que très peu d'appels natifs.

3.3.3 Les domaines de sécurité (*Security Domains*)

À l'instar des Issuer Security Domain, les Application Provider Security Domains, aussi appelés plus simplement Security Domains, sont les représentants des fournisseurs d'applications sur la carte. Ils peuvent aussi être le représentant d'une autorité de contrôle (Controlling Authority) qui peut exister pour renforcer la politique de sécurité sur tous les codes chargés sur la carte.

Les domaines de sécurité proposent aux applications de leur propriétaire (émetteur, fournisseurs d'applications, autorité de contrôle) des services de sécurité comme la manipulation de clés, le chiffrement, le déchiffrement, la génération et la vérification de signatures. En effet, pour permettre d'authentifier leur propriétaire avant toute opération (*e.g.* chargement ou effacement d'applications, initiation d'un canal sécurisé, etc.), les applications mais également les domaines de sécurité utilisent des clés cryptographiques. Les domaines de sécurité sont donc les maillons essentiels permettant d'assurer une totale séparation entre les clés des différents fournisseurs d'applications et celles de l'émetteur (celles-ci sont conservées par le *Issuer Security Domain*).

Les domaines de sécurité peuvent être vus comme des applications spécifiques et en possèdent toutes les caractéristiques comme un AID, des privilèges, un état de cycle de vie (l'Issuer Security Domain hérite de l'état du cycle de vie de la carte).

Chaque domaine de sécurité implante un protocole de canal sécurisé (Secure Channel Protocol) qui permet de sécuriser la communication entre l'émetteur de carte, le fournisseur d'applications, ou l'autorité de contrôle et son domaine de sécurité sur la carte.

Le domaine de sécurité offre également une interface aux applications pour leur permettre d'accéder aux services qu'il met en place (*e.g.* canal sécurisé, etc.). Pour s'assurer que tous les domaines de sécurité se comportent de façon cohérente et peuvent être gérés de manière identique par un même système hors-carte, les domaines de sécurité doivent respecter l'interface APDU externe définie par les spécifications GlobalPlatform. La plupart des services et des commandes APDUs spécifiés par cette interface concernant la gestion du contenu de la carte (*e.g.* installation ou effacement d'applications, changement de l'état du cycle de vie d'une application, etc.), le domaine de sécurité interagit de façon très étroite avec OPEN dont c'est le rôle.

Chaque domaine de sécurité étant établi pour le compte de l'émetteur de carte, d'un fournisseur d'applications, ou d'une autorité de contrôle, chacun possède ses propres clés et celles-ci sont complètement isolées de celles des autres domaines de sécurité.

Comme nous l'avons déjà souligné pour l'*Issuer Security Domain*, les domaines de sécurité peuvent être programmés sous la forme d'*applet* classique lorsque la plate-forme visée est une Java Card.

3.3.4 Les APIs GlobalPlatform

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

3.4 Les mécanismes de sécurité

La plupart des mécanismes de sécurité que propose GlobalPlatform sont d'ordre cryptographique. Ils spécifient des méthodes :

- pour sécuriser les communications ;
- pour s'assurer que les applications chargées sont officiellement signées ;
- pour vérifier l'identité du porteur de carte.

3.4.1 La sécurité des communications

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

- d'authentification mutuelle ;
- de canal sécurisé.

Ces services permettent d'obtenir différents niveaux de sécurité lors des échanges selon la sensibilité des informations concernées. Ainsi, le niveau de sécurité de la communication avec une entité extérieure n'est pas nécessairement le même pour chaque message transmis individuellement. Ce niveau peut être dicté par l'environnement au sein duquel les messages sont transmis (*e.g.* un environnement de confiance ne requiert pas de chiffrement alors qu'une utilisation dans un lieu public l'exige). Le concept du cycle de vie de la carte peut également être employé pour déterminer le niveau minimum de sécurité requis pour la communication entre la carte et une entité extérieure (*e.g.* dans la phases d'initialisation les données sont très souvent transmises en clair alors que dans une phase d'utilisation ils sont chiffrés).

L'authentification mutuelle

L'authentification mutuelle est une phase durant laquelle une carte et une entité extérieure vérifient l'identité de leur correspondant en s'assurant qu'ils partagent les mêmes secrets.

Cette étape a pour objectif de garantir que chacun est bien celui qu'il prétend être, *i.e.* l'application sur la carte vérifie que l'entité distante est bien une entité autorisée (*e.g.* le fournisseur d'applications) et l'entité distante vérifie qu'elle dialogue avec la bonne application sur la carte. La figure 24 montre les différentes étapes d'une authentification mutuelle entre un PC et une carte.

En préalable à toute authentification mutuelle, les deux entités doivent partager une même information secrète, ici, un jeu de clés statiques.

- La première étape de l'authentification mutuelle est l'envoi par l'entité extérieure vers la carte d'une commande APDU connue sous le nom de *Initialize Update Command* et contenant un nombre aléatoire appelé *Host Challenge*.
 - À la réception de cette commande, la carte génère un autre nombre aléatoire appelé *Card Challenge*. Puis, en utilisant *Card Challenge*, *Host Challenge* et les clés statiques communes, la carte crée de nouvelles clés appelées clés de session suivant

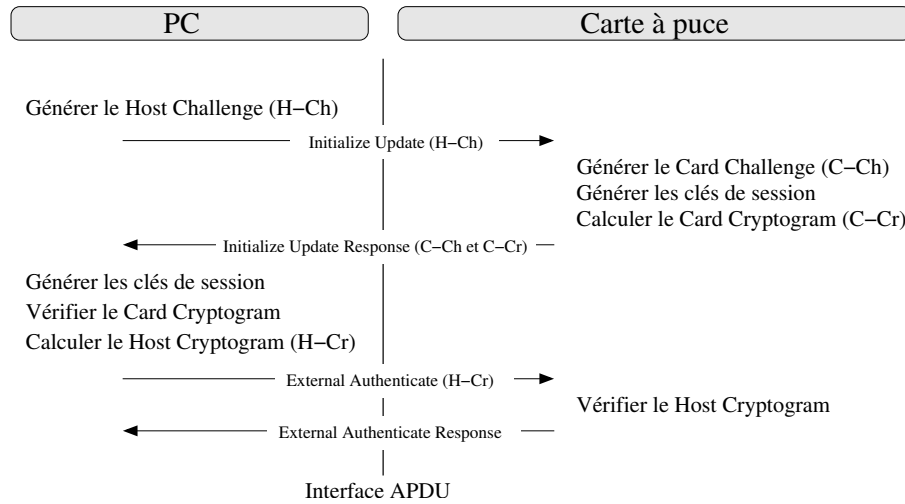


FIG. 24 – Procédure d'authentification mutuelle.

un algorithme connu par la carte et par l'entité extérieure. Ces clés de session sont ensuite utilisées avec un algorithme cryptographique lui aussi connu par l'entité extérieure pour produire une valeur appelée *Card Cryptogram*. Ce *Card Cryptogram*, accompagné du *Card Challenge* et d'autres informations, sont renvoyés à l'entité distante dans une réponse APDU appelée *Initialize Update Response*.

- L'entité extérieure dispose maintenant de toutes les informations que la carte a employées pour produire son *Card Cryptogram*. Or, comme elle connaît l'algorithme pour générer les clés de session et celui utilisé pour obtenir le *Card Cryptogram*, elle est donc capable de produire de son côté un *Card Cryptogram*. Donc, en effectuant une comparaison entre son propre calcul de *Card Cryptogram* et en le comparant à celui obtenu par la carte, elle pourra authentifier cette dernière si la comparaison est réussie.
- Dans une seconde étape, l'entité extérieure génère son *Host Cryptogram* en employant une procédure semblable à celle utilisée par la carte pour générer son *Card Cryptogram*, et l'envoi dans une commande APDU appelée *External Authenticate*. Notons que c'est dans l'en-tête de cette commande que l'utilisateur choisit le niveau de sécurité à appliquer dans le reste de la communication (*cf.* section 3.4.1).
 - De la même façon la carte calculera un *Host Cryptogram* et effectuera une comparaison avec celui reçu afin de lui permettre d'authentifier l'entité distante.
 - Pour valider complètement le processus d'authentification mutuelle, la carte informera l'entité extérieure de la réussite ou non de son authentification.

À la fin de ces étapes, les deux entités se sont donc mutuellement authentifiées et possèdent chacune un jeu de clés de session. Suivant les algorithmes utilisés ces clés de session pourront être des clés symétriques, comme c'est le cas dans la plupart des implantations de GlobalPlatform, ou asymétriques.

La figure 25 illustre comment une application s'exécutant sur la carte utilise, au travers de l'API GlobalPlatform, les services du domaine de sécurité auquel elle est associée pour réaliser l'authentification mutuelle. C'est bien là tout l'intérêt du domaine de sécurité que

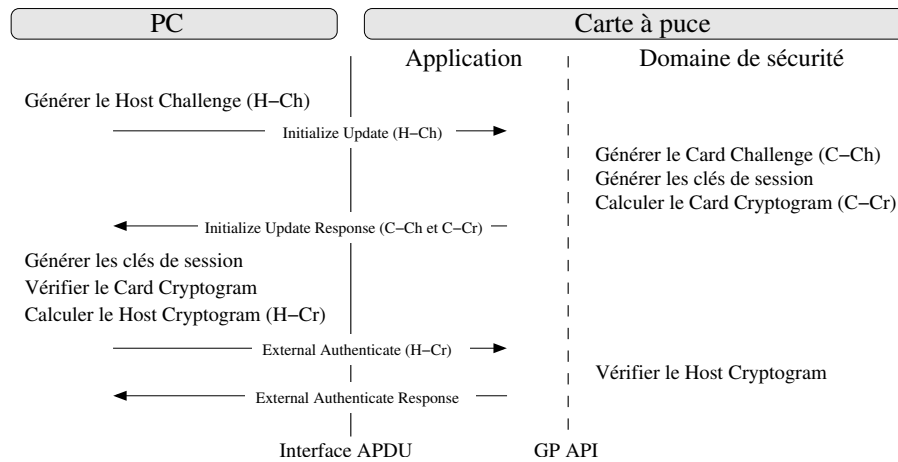


FIG. 25 – Procédure d'authentification mutuelle d'une application avec l'extérieur.

de traiter de façon uniforme et commune tous les aspects liés à la gestion des clés pour différentes applications d'un même fournisseur d'applications.

L'authentification mutuelle constitue un préalable à l'établissement d'un canal sécurisé et bien évidemment si n'importe quelle étape échoue dans ce processus alors le canal sécurisé ne pourra pas être établi.

Le canal sécurisé

Après une authentification mutuelle réussie, GlobalPlatform propose plusieurs niveaux de canal sécurisé :

- le niveau authentification correspond à un canal dans lequel les entités se sont authentifiées via le processus d'authentification mutuelle vu précédemment et sur lequel les messages passent en clair ;
- le niveau authentification et intégrité assure que les messages reçus par une entité proviennent bien de l'autre entité précédemment authentifiée, et que ni l'ordre des messages, ni leur contenu, n'ont été altérés ;
- le niveau authentification, intégrité et confidentialité assure que les messages transmis entre les deux entités authentifiées ne sont pas visibles par une entité non authentifiée et seront intègres.

Nous développons maintenant les méthodes assurant l'intégrité et la confidentialité.

L'intégrité des messages

L'intégrité des messages assure que ni l'ordre des messages ni leur contenu n'ont été altérés. Elle est obtenue en appliquant une fonction cryptographique faisant intervenir une des clés de session générées dans l'étape d'authentification mutuelle, la commande APDU à envoyer et la commande précédente, afin de générer le code d'authentification de message MAC (Message Authentication Code). Lors de la réception du message contenant le MAC, la carte, en utilisant la même procédure et la clé de session correspondante, produira le MAC

correspondant à la commande reçue. Puis en comparant son MAC calculé avec le MAC reçu, elle pourra s'assurer de l'intégrité de la commande APDU.

La confidentialité des données

La confidentialité de données permet de les échanger au travers d'un réseau ouvert sans craindre qu'elles soient interceptées pour une éventuelle analyse. Elle est assurée en appliquant une fonction cryptographique sur le champ de données de la commande APDU en utilisant une des clés de session générées dans l'étape d'authentification mutuelle.

3.4.2 La signature des applications

GlobalPlatform propose de vérifier l'intégrité et l'authentification pour la gestion du contenu de la carte, et surtout lors du chargement d'une application. Une fois encore, la spécification n'impose pas de mécanismes cryptographiques. Toutefois, pour assurer l'objectif d'intégrité lors du chargement d'une application, GlobalPlatform propose d'ajouter aux différents messages correspondants au chargement d'un morceau de code de l'application, un DAP (*Data Authentication Pattern*), *i.e.* une empreinte du morceau de code chargé. OPEN pourra alors en vérifier l'intégrité.

Pour renforcer la sécurité, l'application devra également être signée. Cette phase consiste à signer les différentes empreintes (*i.e.* les DAP) qui auront été envoyées à la carte lors du chargement de l'application. Dans ce cadre, lors du chargement de l'application, c'est le domaine de sécurité associé à l'application qui vérifiera la signature afin d'en déterminer son origine.

On notera que la génération des différentes empreintes et de la signature est généralement réalisée par un outil s'intégrant dans la chaîne de vérification d'une application. Par exemple, dans le cas d'une application Java Card, celle-ci sera normalement passée par le vérifieur hors carte avant d'être signée et protégée en intégrité.

3.4.3 La gestion de la vérification du porteur de carte (CVM)

Le *Cardholder Verification Management* (CVM) est un mécanisme qui permet de vérifier l'identité du détenteur de la carte. Il s'agit pour le porteur de carte de prouver qui il est, en prouvant qu'il connaît un secret contenu dans la carte. C'est notamment le cas du PIN.

Afin d'éviter à chaque application d'avoir à gérer son propre secret et sa propre méthode de vérification, mais aussi que l'utilisateur soit obligé de retenir un grand nombre de secrets, GlobalPlatform a mis en place un mécanisme de CVM partagé et accessible par les applications au travers de l'API GlobalPlatform. En tant que mécanisme global, le CVM est assuré par le CardManager et peut sur une Java Card être programmé en Java. L'avantage de n'avoir qu'une seule méthode de vérification est qu'on peut plus facilement en contrôler la sécurité et que toutes les applications y accèdent de manière uniforme. En revanche, si pour accéder à une application le porteur de carte atteint successivement le nombre maximum d'essais autorisés pour présenter les données nécessaires à la vérification, alors la carte sera bloquée. Heureusement un mécanisme de déblocage pourra être mis en

place. Néanmoins, un déni de service aura été présent pour toutes les applications utilisant le CVM entre l'instant de son blocage et celui de son déblocage (*i.e.* ces applications ne pouvaient pas fonctionner sans le déblocage du mécanisme de CVM).

3.5 Le verrou psychologique versus le verrou technologique

Comme nous venons de le voir au travers de ces différentes sections, GlobalPlatform propose un certain nombre de fonctionnalités qui permettent de résoudre les problèmes de la multi-application que nous avons évoqués.

Pour cela, GlobalPlatform standardise la façon de charger les applications et de les installer. Il définit les états du cycle de vie où de telles opérations sont possibles et il met en place des services permettant d'effectuer un chargement sécurisé si nécessaire. GlobalPlatform définit également la manière dont une application peut accéder à des services globaux de la carte au travers de l'API GlobalPlatform (*e.g.* vérification de l'identité du détenteur de la carte, modification des octets d'historiques de l'ATR, etc.). Il détermine aussi les états possibles du cycle de vie d'une application sur la carte. Enfin, GlobalPlatform met en place des procédures standards pour l'authentification mutuelle des applications embarquées sur la carte et celles hors carte.

Mais le problème le plus essentiel que résout GlobalPlatform est celui de la vérification des applications ou en d'autres termes celui de la confiance qu'on peut avoir en telle ou telle application. En effet, grâce au mécanisme de signature des applications, il garantit que l'application est de confiance et que, par conséquent, elle ne devrait pas tenter de nuire aux autres applications présentes sur la carte si on la chargeait. Effectivement, si l'application a été signée c'est par un partenaire commercial de l'émetteur de carte ou par lui-même et on peut donc raisonnablement supposer qu'il aura fait les vérifications qui s'imposent, soit par l'utilisation d'un programme automatique de vérification du code de l'application, soit par un audit de son code.

En ce sens, GlobalPlatform a réussi à lever le verrou psychologique quant à une utilisation industrielle des cartes multi-applicatives mais pas le verrou technologique puisque pour pouvoir charger du code sur sa propre carte multi-applicative il faut être un partenaire commercial de l'émetteur de carte.

On peut penser que tant que les vérificateurs de code ne seront pas embarqués sur les cartes ou tant qu'il n'aura pas été prouvé à 100% que les attaques ne sont pas possibles sur telle ou telle technologie de cartes multi-applicatives, il y a fort à parier que les industriels choisiront de ne pas ouvrir leur cartes davantage.

Cependant, l'infrastructure GlobalPlatform possède de nombreux atouts qui lui ont permis de prouver son efficacité puisqu'elle a déjà été déployée dans le secteur des cartes SIM et des cartes bancaires.

Chapitre 4

PC/SC : communication entre le PC et la carte

Dans ce chapitre nous allons présenter PC/SC, un standard de communication entre une application et un lecteur de carte à puce et par conséquent entre l'application et la carte contenue dans le lecteur.

Nos travaux du chapitre 13 utilisent une implantation de PC/SC – `pcsc-lite` – que nous avons modifiée pour nos propres besoins et que nous décrirons plus bas. Nous présenterons également quelques applications de PC/SC que nous utilisons pour nos recherches.

4.1 Présentation

Afin de standardiser une architecture pour interfacer la carte à puce avec un PC, un consortium d'industriels a vu le jour en mai 1996, le PC/SC Workgroup [49]. Il regroupait des fabricants de PC, de lecteurs et de cartes à puce parmi lesquels on trouve entre autres Apple, Bull, Gemplus, Hewlett Packard, Infineon, Intel, Microsoft, Schlumberger et Toshiba. Le PC/SC Workgroup a identifié trois parties à standardiser :

- l'interface entre le lecteur et le PC ;
- une API de haut niveau pour accéder aux fonctionnalités de la carte ;
- les mécanismes autorisant plusieurs applications à partager des ressources communes comme la carte ou le lecteur.

Les membres du consortium souhaitaient également une architecture et des spécifications permettant aux fabricants de cartes et de lecteurs de développer leurs produits indépendamment tout en assurant qu'ils puissent fonctionner ensemble.

C'est une approche classique de la standardisation qui vise à laisser le marché ouvert aux diverses parties, plutôt que d'imposer un standard qui force tous les fabricants à avoir les mêmes produits et qui donc ferme le marché en enlevant toute la valeur ajoutée.

En Décembre 1997, le groupe de travail a publié le « Interoperability Specification for ICCs and Personal Computer Systems » [204] qui est aujourd'hui plus communément connu sous le nom de PC/SC (PC/Smart Card). Il s'agissait de la version 1.0 et elle comportait

huit parties détaillant chacune un aspect de l'architecture. En novembre 1999 avait lieu l'annonce d'une future version 2.0 [205].

Finalement, c'est en septembre 2004 que devrait sortir la version 2.0, qui ajoute la prise en charge des cartes sans contact, des lecteurs aux fonctions étendues (*e.g.* avec un clavier, avec un capteur biométrique, avec un écran, etc.) et des cartes synchrones. À l'heure où j'écris ces lignes cette dernière est actuellement en version *draft* [206] depuis juin 2004 pour une revue publique. Elle comporte neuf parties que nous décrirons dans la section suivante.

4.2 L'architecture

La figure 26 présente ce que couvre chacune des neuf parties des spécifications PC/SC 2.0 :

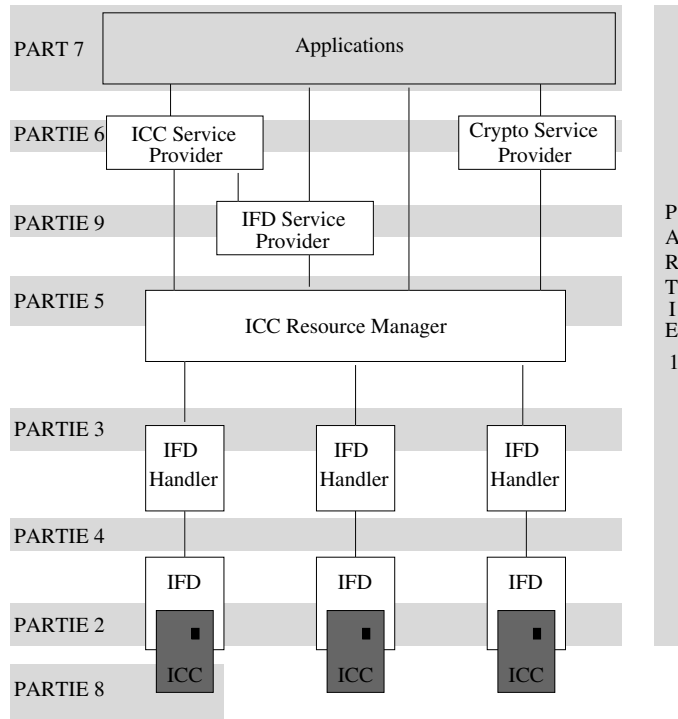


FIG. 26 – L'architecture de PC/SC 2.0.

- La partie 1 donne une vue d'ensemble de l'architecture du système et des composants définis par le groupe de travail.
- La partie 2 détaille les conditions pour assurer l'interopérabilité entre la carte à puce (*i.e.* ICC, Integrated Chip Card) et lecteur (*i.e.* IFD, InterFace Device) :
 - les caractéristiques physiques ;
 - les protocoles de communication et la gestion des erreurs. Il s'agit principalement d'une reprise des ISO 7816 et ISO 14443 afin de gérer les cartes synchrones et asynchrones, avec contact et/ou sans contact.

- La partie 3 décrit l'interface que doit implanter le pilote d'un lecteur (*i.e.* IFD Handler) afin que les couches hautes de PC/SC puissent communiquer avec lui de façon indépendante de la connectique et des protocoles utilisés. En particulier, il doit implanter l'envoi de commandes, la mise sous tension et hors tension, etc.
- La partie 4 présente les différents types de connexion avec un lecteur (*e.g.* PS2, USB, série, PCMCIA, etc.) et fournit un certain nombre de recommandations que les lecteurs devraient respecter.
- La partie 5 décrit les interfaces et les fonctionnalités supportées par le gestionnaire de ressources (*i.e.* Resource Manager). Le gestionnaire de ressources est l'élément central du système PC/SC. En effet, il est responsable de la gestion des ressources systèmes relatives aux cartes (*i.e.* ICC Service Provider) et du contrôle d'accès aux lecteurs, et donc à travers eux, de l'accès aux cartes. En théorie, chaque système d'exploitation fournit son propre gestionnaire de ressources. En outre, il résout trois problèmes pour gérer l'accès à différents lecteurs et cartes :
 - il est responsable de l'identification et de la détection des ressources (*e.g.* insertion/retrait d'une carte, branchement/débranchement d'un lecteur, etc.) ;
 - il est responsable de l'allocation des (ressources) cartes et lecteurs pour les différentes applications en fournissant des mécanismes de partage ou d'exclusivité (sur ces ressources) ;
 - il propose un mécanisme de transactions aux couches hautes de PC/SC permettant de réaliser des opérations complexes tout en assurant que les informations des états intermédiaires n'ont pas été corrompues.
- La partie 6 propose une façon de s'abstraire de la complexité de l'architecture sous-jacente en proposant aux programmeurs d'application des APIs de haut niveau correspondant aux opérations que pourra effectuer la carte. Cette partie :
 - décrit les modèles pour la fourniture de service de façon générale (*i.e.* ICC Service Provider) et pour la fourniture de services cryptographiques (*i.e.* Crypto Service Provider) – cette distinction sur la cryptographie est liée aux restrictions pour son importation et son exportation dans certains pays ;
 - identifie les interfaces requises ;
 - indique comment ces modèles peuvent être étendus aux besoins d'une application d'un domaine spécifique. Nous illustrerons ces concepts plus loin lors de la description de l'implantation de Microsoft.
- La partie 7 présente des méthodes pour la construction des applications et décrit comment utiliser les autres composants.
- La partie 8 décrit les fonctionnalités recommandées pour les cartes qui supportent la cryptographie. Elle contient des conseils sur la façon de gérer l'identification, l'authentification et le stockage sécurisé mais aussi sur la façon d'assurer l'intégrité des informations, leur traçabilité et leur confidentialité dans une solution basée sur la carte.
- La partie 9, qui n'existait pas dans la version précédente de PC/SC, décrit comment gérer des lecteurs aux fonctionnalités étendues (*i.e.* IFD Service Provider) : capteur biométrique, écran, etc.

Pour résumer, dans l'architecture PC/SC les applications sont construites au-dessus d'un ou plusieurs fournisseurs de services et d'un gestionnaire de ressources. Le fournisseur de services encapsule les fonctionnalités attendues pour une carte à puce spécifique et la rend ainsi accessible à travers des interfaces de programmation de haut niveau. Le gestionnaire de ressources gère les ressources relatives aux cartes et aux lecteurs à l'intérieur du système. Il permet via une API simple, au-dessus de laquelle les services de plus haut niveau seront construits, d'accéder aux lecteurs et, à travers eux, aux cartes à puce.

4.3 Les différentes implantations

Au départ implanté exclusivement par Microsoft, membre du consortium, PC/SC permettait d'accéder aux lecteurs de cartes à puce depuis différents langages de programmation seulement sous Windows. OpenCard Framework [47, 138, 139] permettait quant à lui d'accéder aux lecteurs de cartes à puce sur tous les systèmes supportant Java. Donc, même si PC/SC et OpenCard Framework ont aujourd'hui beaucoup de concepts similaires, c'était au départ des standards très orthogonaux [96, 192].

Depuis peu un projet libre, `pcsc-lite` [48], initié par David CORCORAN – et dans lequel je suis partie intégrante – a vu le jour et permet d'accéder aux lecteurs de cartes à puce dans beaucoup de systèmes d'exploitation autre que Windows comme nous le montrerons plus loin.

4.3.1 Microsoft PC/SC

Microsoft en tant que membre du groupe de travail a été le premier à fournir une implantation des spécifications PC/SC. Au départ, il s'agissait d'un programme externe, le gestionnaire de ressources, et d'une bibliothèque de haut niveau, l'API SCard (pour Smart Card) [166] permettant de communiquer avec lui. Cette API est une implantation de l'API suggérée dans la partie 5 des spécifications PC/SC.

Pendant longtemps les premières versions du gestionnaire de ressources ont connu beaucoup de problèmes et, par exemple, on ne pouvait communiquer qu'avec un seul lecteur à la fois. Après avoir résolu ces quelques problèmes, Microsoft a intégré ces composants dans tous ses systèmes d'exploitation.

Dans ce modèle, chaque constructeur de lecteurs, pour chacun de ses lecteurs, devait donc fournir un pilote (*i.e.* IFD Handler) respectant l'API Microsoft pour le support des lecteurs (*e.g.* une API semblable à celle définie dans la partie 3 des spécifications PC/SC). De plus, ce pilote devait passer un ensemble de tests du Windows Hardware Quality Labs afin d'être homologué.

En effet, au niveau de la partie 4 des spécifications, différents lecteurs pouvant utiliser un protocole spécifique même s'ils utilisent le même média (*e.g.* l'USB), il fallait un pilote par lecteur ou tout du moins par famille de lecteur utilisant le même protocole pour le même média. Toutefois, afin de résoudre ce lourd problème d'installation de pilotes, une initiative visant à standardiser le protocole sur le média USB a été lancée au sein de l'USB Implementers Forum [66] et a développé la spécification CCID 1.0 (*i.e.* USB Chip/Smart

Card Interface Device) [112]. Ainsi, Microsoft a implanté cette spécification dans un pilote qu'il distribue avec ses systèmes d'exploitation permettant ainsi à n'importe quel lecteur compatible CCID de fonctionner sans requérir l'installation de pilote supplémentaire lors de son branchement. Bien évidemment, si beaucoup de lecteurs ne suivent pas encore ce standard, il va sans dire que c'est l'avenir puisqu'il prévoit également la prise en compte de lecteurs aux fonctionnalités étendues.

Toujours dans le modèle PC/SC, il appartient donc à l'émetteur de cartes de fournir le Service Provider adéquat (*i.e.* une API de haut niveau – partie 6 des spécifications PC/SC) pour sa carte afin de cacher aux développeurs d'applications la complexité de la pile protocolaire sous-jacente. Par exemple, si un émetteur fournit une carte avec un système de fichiers il est probable qu'il fournira également un Service Provider permettant de lire et d'écrire un fichier, de créer un répertoire, de se déplacer dans l'arborescence, etc. Pour le programmeur d'application qui voudra utiliser ces cartes pour stocker des informations, il sera plus commode d'utiliser cette API de haut niveau plutôt que de communiquer directement avec la carte dans un langage assez abscons, surtout s'il n'est pas familier avec toute la pile protocolaire sous-jacente.

On a vu ici, qu'un ICC Service Provider pouvait se rapporter à une carte particulière d'un fabricant particulier, mais il peut aussi se rapporter à une famille de cartes respectant le même standard et donc à des cartes pouvant provenir de différents fabricants. On peut reprendre l'exemple des cartes possédant un système de fichiers respectant le standard ISO7816-4. Si un ICC Service Provider existe pour ce type de cartes alors le programmeur d'application pourra y accéder de façon transparente sans avoir à se soucier de quel fabricant proviennent les cartes qu'il utilise. Bien entendu les ICC Service Provider peuvent cohabiter sans problème. On peut noter sur cet exemple le souci du groupe de travail de laisser le marché ouvert à tous les acteurs afin de les laisser se concurrencer sainement et permettre ainsi une large adoption du standard PC/SC.

Les Crypto Service Provider ont un rôle similaire à celui des ICC Service Provider. Ils sont d'ailleurs normalisés eux aussi par la partie 6 des spécifications PC/SC. Ce domaine étant, dans certains pays, soumis à des règles d'importation et d'exportation très strictes ils ont été différenciés. Ainsi, il peut y avoir des CSP spécifiques à une carte ou bien des CSP généralistes pour un type de cartes.

Pour résumer les ICC Service Provider et Crypto Service Provider sont fournis en général par les émetteurs de carte ou par des vendeurs de solutions transverses interopérables sur un segment de marché (*e.g.* système de fichiers, etc.).

On notera par ailleurs, que ces CSP sont à la base du standard cryptographique de Microsoft : CryptoAPI ou CAPI [165]. En effet, dans ses logiciels, Microsoft n'emploie pas le standard PKCS#11 [153] utilisé par pratiquement tout le monde.

En conséquence, aujourd'hui il existe deux modules cryptographiques pour fournir des services cryptographiques aux applications : CryptoAPI (sous Windows) et PKCS#11 (sous Windows et sous les autres plates-formes). Ces deux modules fournissent une couche d'abstraction permettant de fournir des services cryptographiques indépendamment de l'utilisation de cartes à puce. Ainsi, comme le montre la figure 27 Microsoft fournit sa propre couche d'abstraction CAPI au-dessus des CSP afin de permettre l'utilisation de CSP ne se basant pas sur PC/SC et/ou sur les cartes à puce.

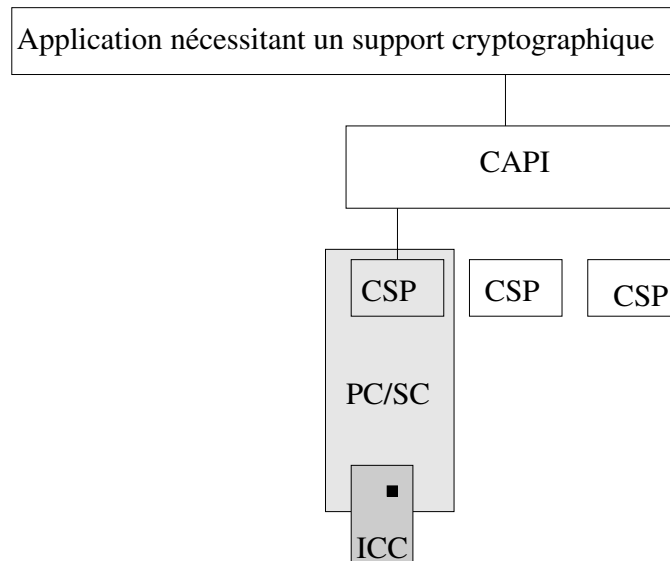


FIG. 27 – L’architecture CryptoAPI.

Par ailleurs, c’est suite au développement de PC/SC que Microsoft avait lancé la WfSC. En effet, son objectif était de fournir une solution verrouillée pour l’authentification sous Windows grâce à sa carte à puce, WfSC, et son propre standard, CryptoAPI. Depuis, cette idée est abandonnée à cause du peu de succès qu’a connu la WfSC.

4.3.2 pcsc-lite

Le projet **pcsc-lite** [48] a été initié en 2000 [97] par David CORCORAN de l’université de Purdue. Ce projet est une implantation libre sous licence BSD du standard PC/SC 1.0. Le but des développeurs de ce projet était de fournir un meilleur support des cartes à puce pour les environnements Unix. Leurs efforts se concentrent d’ailleurs au sein de MUSCLE (Movement for the Use of Smart Card in a Linux Environment) [41].

Afin de fournir aux développeurs une API simple et de faciliter le portage des applications pour cartes développées sous Windows, il a été décidé de reprendre l’API SCard proposée par Microsoft.

Aujourd’hui, **pcsc-lite** fonctionne sur un grand nombre de plates-formes : Linux, Solaris, Mac OS X, BSD, HP-UX, etc. On peut d’ailleurs noter qu’en 2000 Apple a rejoint le groupe de travail PC/SC et a décidé d’adopter **pcsc-lite** comme implantation de référence pour son système Mac OS X.

En revanche, si **pcsc-lite** reprend l’API SCard de Microsoft, il n’en implante pas encore toutes les fonctionnalités. Le fera-t-il un jour ? Est-ce nécessaire ? En effet, comme pour le moment **pcsc-lite** n’implante pas toutes les parties des spécifications de PC/SC 1.0, d’où son nom, « *lite* », par conséquent il ne peut pas implanter toutes les fonctionnalités offertes par SCard de Microsoft. Actuellement, il comprend un gestionnaire de ressources et une implantation de l’API SCard se présentant respectivement sous la forme d’un démon

et d'une bibliothèque partagée. Cette dernière communique avec le gestionnaire au travers de sockets Unix. Celui-ci est capable de gérer jusqu'à 255 lecteurs simultanément ou plus exactement 255 emplacements car un lecteur peut posséder plusieurs emplacements acceptant plusieurs cartes simultanément. Le démon `pcsc-lite` peut aussi gérer par défaut jusqu'à 16 applications mais le logiciel étant open source, il suffit de changer la valeur de la constante avant la compilation pour en gérer davantage.

Le fonctionnement de base du démon lors de son démarrage est le suivant :

- allocation des structures de données pour les lecteurs et création de l'espace public contenant des informations sur l'état des lecteurs (via un mécanisme de mémoire partagée) ;
- lecture du fichier de configuration des lecteurs statiques (*e.g.* lecteurs série) ;
- création du serveur principal et écoute dans l'attente d'une requête client ;
- lancement d'une thread de détection des lecteurs « hot plug » (*e.g.* USB, PCMCIA).

Pour chaque lecteur détecté, si le pilote du lecteur (*i.e.* IFD Handler) le permet, le démon `pcscd` créera une thread pour prendre en charge la gestion des communications et permettre l'accès simultané aux ressources.

Lors de la première requête client (*i.e.* `SCardEstablishContext`) vers le démon, au travers de la bibliothèque partagée `pcsc-lite` et donc au travers des sockets Unix, celui-ci créera une thread qui démarrera un serveur dédié pour prendre en charge le contexte du client. Ainsi, toutes les prochaines requêtes se feront sur ce nouveau socket Unix jusqu'à ce que l'application décide de terminer son dialogue avec le démon (*i.e.* `SCardReleaseContext`) ou qu'elle meurt (*e.g.* problème de segmentation, terminaison, etc.).

Mon principal apport au projet `pcsc-lite` est le support multi-thread dans la bibliothèque cliente et dans le démon gestionnaire de ressources.

Par ailleurs, depuis 2003 le projet `pcsc-lite` joue sur le terrain des grands grâce au travail de Ludovic ROUSSEAU puisqu'il est à même de proposer une implantation libre sous licence GPL d'un IFD Handler pour les lecteurs respectant le standard CCID [17]. Ce pilote qui supporte aussi les lecteurs possédant des fonctionnalités évoluées permet au projet MUSCLE d'être au moins au même niveau que Microsoft pour une utilisation avancée des cartes.

4.4 Quelques applications

Les applications que nous allons présenter ici sont disponibles en open source et ont pour objectif de donner un aperçu de ce qu'il est possible de faire en utilisant PC/SC. Par ailleurs, nous utiliserons une de ces applications (*i.e.* le wrapper JPC/SC) dans la grille de cartes présentée au chapitre 13.

4.4.1 Les wrappers

Les premières applications disponibles pour PC/SC sont en fait des *wrappers* (*i.e.* textuellement des *encapsulateurs*) se présentant sous la forme de bibliothèques intergicielles

(*i.e. middleware*). Elles permettent d'écrire des applications dialoguant avec le gestionnaire de ressources dans un autre langage que celui fourni et imposé par la bibliothèque native. À titre d'exemple `pcsc-lite` ne fournit qu'une bibliothèque implantant l'API SCard pour le C et il ne permet donc que d'écrire des programmes dans ce langage.

Des wrappers sont apparus pour Perl [50], Python [53], Prolog [20], Ruby [56] et Java [31]. Ils utilisent les mécanismes de FFI (Foreign Function Interfaces) [9] de ces différents langages.

Ces intergiciels contribuent à populariser PC/SC en permettant à plus de programmeurs d'accéder simplement et à bas niveau aux fonctionnalités du standard PC/SC.

4.4.2 JPC/SC vs OCFPCSC

Pour accéder à des lecteurs de cartes à puce et donc à des cartes depuis le langage Java, il y a deux solutions.

La première que nous venons de voir ci-dessus est un wrapper appelé JPC/SC et développé par Marcus OESTREICHER de IBM BlueZ Secure Systems. Il s'agit là d'une simple réification des concepts de « *Context* » et de « *Card* » utilisés par l'API SCard avec une utilisation de JNI (Java Native Interface) pour renvoyer les appels de méthode et les arguments Java vers la bibliothèque cliente native fournie avec Microsoft PC/SC ou `pcsc-lite`. Cette solution a l'avantage d'être simple et de bas niveau. Ce dernier point est à la fois un avantage car il laisse une plus grande liberté mais c'est aussi un inconvénient car cela demande la maîtrise d'un certain nombre de concepts sous-jacents pour communiquer avec une carte.

L'autre solution est le pont OCFPCSC [46] *i.e.* un pont entre l'OpenCard Framework (OCF) et PC/SC. En effet, le projet OCF initié en 1997 par le consortium OpenCard qui regroupe Gemplus, Giesecke & Devrient, IBM, Schlumberger, Sun microsystems, Visa International, etc., permet d'accéder en Java à des lecteurs de cartes. Or, pour pouvoir accéder à des lecteurs depuis OCF, il fallait disposer de pilotes (*i.e.* Card Terminal dans ce modèle) spécifiques OCF pour chaque type de lecteurs et ces pilotes sont peu nombreux. De plus, comme pour accéder à un niveau matériel en Java, il faut toujours utiliser une couche d'appels JNI, que cela soit au travers de l'utilisation du port série ou du port parallèle si on utilise `javax.comm`, ou à un autre niveau dans d'autres cas, une solution consistant à utiliser PC/SC a été développée : le pont OCFPCSC. Du point de vue d'OCF, c'est en fait un simple Card Terminal à déclarer dans le fichier de configuration `opencard.properties` et qui via des appels JNI se connecte à la bibliothèque cliente native SCard du système pour détecter la présence des lecteurs et communiquer avec eux. Cette solution a pour inconvénient d'ajouter des couches supplémentaires dans une pile logicielle déjà assez complexe. L'autre problème de l'utilisation d'OCF est l'absence de maintenance puisque depuis février 2000, le projet semble à l'abandon et n'évolue plus du tout. Pour l'instant ce n'est donc pas la solution que nous avons retenue pour communiquer depuis Java avec la grille de cartes que nous décrirons au chapitre 13. En revanche, nous reviendrons peut-être sur ce choix stratégique dans l'avenir car avec l'apparition de RMI dans Java Card 2.2, Sun microsystems fournit un Card Service (*i.e.* une bibliothèque cliente intégrée dans le modèle OCF) pour gérer les communications RMI avec les Java Cards au standard 2.2.

Pour être exhaustif, nous citerons juste l'API `javax.smartcard` [167] de Sun microsystems qui permettait de communiquer avec une carte depuis Java mais elle est aujourd'hui totalement abandonnée.

4.4.3 MuscleCard

Le projet MuscleCard [42] fut initié, comme `pcsc-lite`, par David CORCORAN au sein de MUSCLE. Il a pour ambition de fournir à l'utilisateur final des services cryptographiques pour lui permettre de s'authentifier sur sa machine, de signer ses messages, etc.

MuscleCard a initialement été développé comme une solution pour Mac OS X et pour Linux afin de pallier l'absence d'un support complet de PC/SC dans ces environnements. Depuis, il a aussi été porté dans les environnements Windows de Microsoft.

Aujourd'hui, si son architecture est très similaire à celle décrite dans les spécifications PC/SC, elle est en fait plus exactement un croisement entre la CryptoAPI de Microsoft et les spécifications PC/SC (*cf.* FIG. 28). Les CSP de la CryptoAPI trouvent leur équivalent dans les *plug'ins* de MuscleCard et la CAPI dans la bibliothèque MuscleCard. Le reste des couches basses de l'architecture repose sur l'implantation PC/SC de la machine. La seule vraie différence avec le modèle CryptoAPI interfacé avec Microsoft PC/SC est que ce dernier est plus dynamique puisqu'il dispose d'une API SCard plus complète que celle proposée par `pcsc-lite`. Cela lui permet donc de rajouter ou de supprimer les CSP dynamiquement. Au contraire, dans le modèle MuscleCard les *plug'ins* sont ajoutés de façon statique et non pas au travers de méthodes de l'interface SCard.

Depuis son commencement, un des buts du projet a été d'être indépendant du fournisseur de cartes pour offrir les services cryptographiques. C'est donc tout naturellement que les développeurs ont pensé à utiliser la technologie Java Card. Ainsi, ils ont développé une *applet* Java Card connue sous le nom de MCardApplet et destinée à être chargée sur la Java Card de toute personne souhaitant utiliser la plate-forme MuscleCard.

Cette *applet* permet de stocker des informations de diverses natures : PINs, clés cryptographiques, certificats et toute autre donnée que l'on souhaite garder confidentielle. Bien évidemment, cette *applet* gère également des notions de droit en lecture et en écriture. Enfin, suivant les capacités cryptographiques des Java Cards sur lesquelles elle est chargée, cette *applet* est capable d'accéder à un certain nombre de mécanismes cryptographiques comme le chiffrement ou le déchiffrement de données, la signature ou la vérification de signature, le hachage de données.

Pour résumer, grâce à cette *applet*, le projet MuscleCard touche plusieurs dizaines de modèles de Java Cards, ce qui est loin d'être négligeable.

De plus, le mécanisme de *plug'ins* de MuscleCard a également permis l'émergence de *plug'ins* pour des cartes propriétaires (*i.e.* dont le code source n'est pas disponible et sur lesquelles on ne peut pas charger de code) susceptibles d'intégrer le système comme la CryptoFlex d'Axalto, l'AuthentIC de Oberthur Card Systems ou encore la CAC (Common Access Card) de ActivCard qui a d'ailleurs été adoptée par le département de la défense des États-Unis [3]. Ces *plug'ins* ont pour la plupart été développés par les contributeurs du projet MuscleCard.

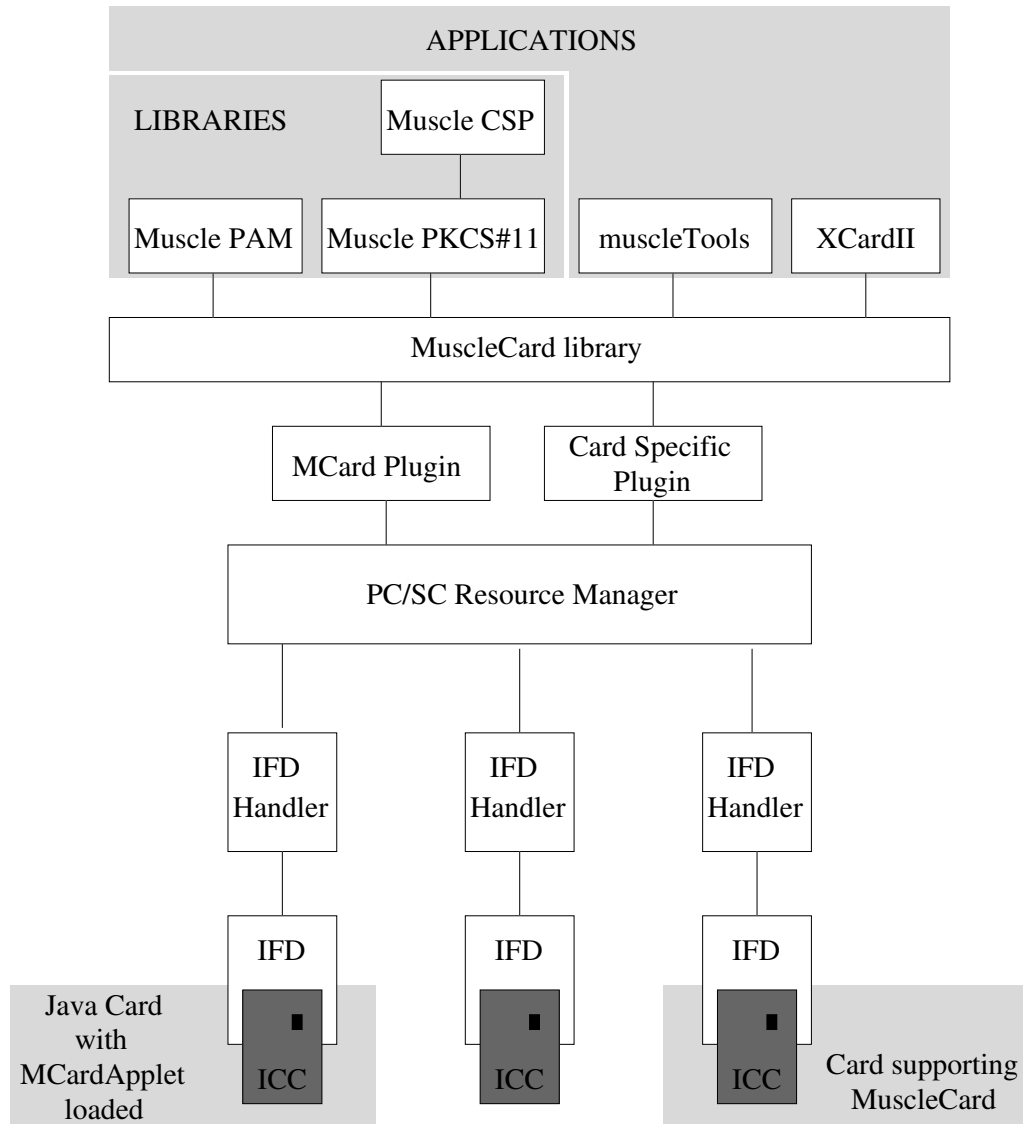


FIG. 28 – L'architecture MuscleCard.

La plate-forme MuscleCard propose également un ensemble d'outils d'administration pour gérer les cartes et les objets qu'elles contiennent *i.e.* :

- sous les environnements Unix : muscleTools et XCardII pour gérer respectivement les cartes en ligne de commande et en mode graphique ;
- sous Mac OS X : CocoaCard ;
- sous Windows : un outil pour l'installation de l'*applet* et la gestion du contenu existe depuis quelques semaines mais il n'a pas encore de nom.

La plate-forme propose surtout, et c'est là tout l'intérêt de MuscleCard, un ensemble de bibliothèques pour traiter divers aspects tels que l'authentification, la signature électronique, etc. Il s'agit, par exemple, d'une bibliothèque PAM pour gérer l'accès aux services de machines sous Unix (*i.e. login*, etc.). Mais il y a également un module PKCS#11 qui permet par exemple de signer ses messages et de s'authentifier auprès de sites web en utilisant Mozilla.

Comme nous l'avons vu précédemment, Windows ne supportant pas le standard PKCS#11 mais son propre standard CryptoAPI, il était impossible de se servir de MuscleCard avec Internet Explorer. Pour pallier ce problème, IdentityAlliance propose depuis quelques semaines un CSP construit au-dessus du module PKCS#11 afin d'offrir un très large support de MuscleCard à travers le monde. Le seul inconvénient de ce CSP est qu'il est, pour le moment, la seule partie non *open source* du projet MuscleCard.

Chapitre 5

La certification

Le processus de certification consiste en une procédure complexe mettant en jeu plusieurs acteurs et dont l'action finale est d'émettre un certificat pour un produit ou un système issus des Technologies de l'Information (TI). La délivrance de ce certificat a lieu à l'issue de l'évaluation sécuritaire du produit réalisée par un tiers indépendant. En conséquence, le certificat d'un produit TI a pour vocation d'établir le niveau de confiance minimum que pourra avoir un utilisateur final dans la sécurité du produit.

Dans ce chapitre, nous présenterons la certification dans un cadre général, puis nous décrirons le schéma français de certification. Enfin, nous examinerons le processus d'évaluation menant à l'obtention d'un certificat dans le cadre particulier des Critères Communs.

Le but de ce chapitre est de bien montrer l'environnement dans lequel s'est déroulée ma thèse afin de mettre en exergue les problématiques sous-tendant une partie de mes travaux. En effet, ma thèse CIFRE a été effectuée dans le cadre d'une collaboration entre le LaBRI et le CESTI de SERMA Technologies. Or, comme nous le verrons dans la section 5.1 les CESTIs sont des tiers essentiels pour la certification.

Comme on peut le voir figure 29, le processus de certification fait intervenir trois tiers indépendants :

- le commanditaire de l'évaluation qui souhaite obtenir un certificat pour son produit ou son système TI ;
- le centre d'évaluation qui conduit les tâches d'évaluation de la sécurité du produit en fonction de la méthodologie choisie par le commanditaire (*e.g.* les critères communs ou ITSEC), du niveau d'assurance choisi par le commanditaire, des éléments de preuve apportés par le commanditaire et de sa propre expertise technique ;
- l'organisme de certification qui revoit la pertinence technique des rapports émis par le centre d'évaluation et qui au vu du Rapport Technique d'Évaluation (RTE) décide ou non de la délivrance du certificat.

On notera qu'à chaque étape des éléments de preuve sont produits afin de garantir le bon déroulement de l'évaluation. C'est la présence d'un tiers indépendant (*i.e.* l'organisme de certification), chargé de vérifier l'exhaustivité et la qualité des travaux menés par le centre d'évaluation grâce aux éléments de preuve, qui garantit le niveau de confiance que l'on peut avoir dans le produit TI. En effet, l'activité d'évaluation de la sécurité est une

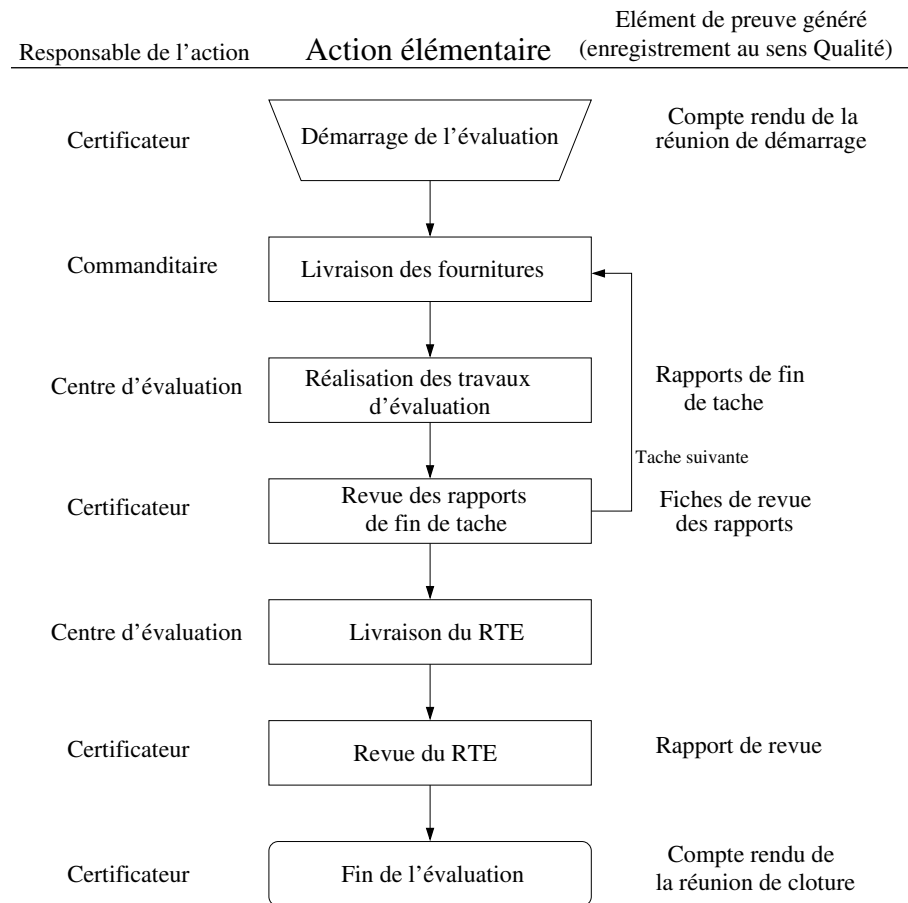


FIG. 29 – Déroulement de l'évaluation.

activité commerciale et à ce titre c'est le commanditaire qui choisit le centre d'évaluation qui va conduire l'évaluation de son produit. Ainsi, s'il n'y avait pas le contrôle extérieur de l'organisme certificateur, on pourrait tout à fait imaginer qu'un commanditaire « achète » son certificat avec la complaisance du centre d'évaluation désireux de ne pas froisser son client. Heureusement, il n'en est rien au sein des différents schémas de certification tripartite mis en place dans les différents pays. D'ailleurs, les gestionnaires de schémas de certification de plusieurs pays ont passé des accords de reconnaissance mutuelle des certificats émis par eux au niveau international afin de développer la certification des produits TI mais aussi pour garantir la confiance dans leur système de certification.

Les avantages de la certification sont donc de garantir un certain niveau de confiance dans la sécurité d'un produit TI (confidentialité, intégrité, disponibilité). Ainsi, les utilisateurs finaux peuvent comparer les différents produits sur le marché de façon objective. Cela permet également aux commanditaires de prouver leur compétence, d'en faire un argument marketing et ainsi d'étendre leurs marchés. Enfin, cela permet aux organismes de certification, qui sont pratiquement toujours des organismes gouvernementaux, d'être sûrs que les produits sur le marché sont sécurisés et que leur pays ne court pas de risque majeur lors d'une attaque contre ses systèmes TI.

5.1 Le schéma français

Le schéma français suit une organisation assez similaire à celle que nous venons de décrire plus haut. Il est présenté sur la figure 30.

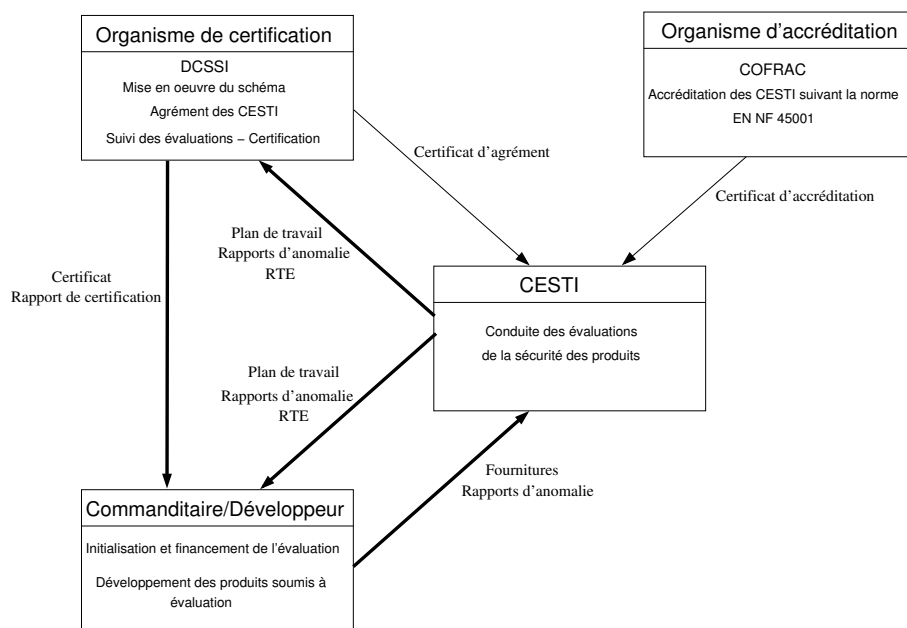


FIG. 30 – Rôles des différents intervenants et échanges d'information.

L'organisme de certification est un organisme gouvernemental dénommé Direction Centrale de la Sécurité des Systèmes d'Information (DCSSI) [8]. Cet organisme appartient en fait au Secrétariat Général de la Défense Nationale (SGDN) qui est un service placé directement sous les ordres du Premier Ministre. En France c'est donc la DCSSI qui délivre un agrément aux centres d'évaluation qui en font la demande. Ces centres sont appelés des Centres d'Évaluation de la Sécurité des Technologies de l'Information (CESTI).

Pour obtenir cet agrément les CESTIs doivent mettre en place des politiques de sécurité très strictes afin de garantir la sécurité des informations qu'ils manipulent. Ainsi, l'accès aux locaux doit être contrôlé, les documents doivent être conservés dans des coffres, les personnels embauchés doivent être déclarés au SGDN en vue de subir d'éventuelles enquêtes de moralité, etc. En d'autres termes, tous les moyens visant à garantir un niveau de sécurité type Confidentiel Défense doivent être mis en œuvre.

De plus pour être autorisés à conduire une évaluation, les CESTIs doivent être accrédités par l'organisme d'accréditation, le COFRAC (COmité FRançais d'ACcréditation) [5], pour tous les aspects liés à la Qualité. En effet, nous avons vu précédemment que lors d'une évaluation, il y avait un certain nombre de preuves qui étaient produites par l'organisme d'évaluation ; afin qu'elles puissent être vérifiées par l'organisme de certification, il est nécessaire que la qualité de ces preuves soit, elle aussi, assurée (*e.g.* il faut que les outils soient étalonnés, sans quoi, les mesures sont fausses et donc les résultats non reproductibles).

Bien évidemment, un centre d'évaluation désirant commencer son activité de CESTI ne peut pas le faire dans un domaine où il n'a aucune expertise ; il commence son activité avec un certificat d'agrément temporaire qui deviendra permanent s'il réussit à mener à bien les évaluations qui lui sont confiées. Ainsi, dans le schéma français un commanditaire ne pourra pas aller voir un CESTI pour lui faire évaluer un produit sur lequel il n'a pas fait ses preuves dans l'espoir d'obtenir plus facilement son certificat car la DCSSI y opposerait son veto. Malgré cela, la DCSSI ne peut pas imposer aux commanditaires le choix du CESTI et c'est donc la loi du marché qui joue le rôle de régulateur. Cependant, le nombre de CESTIs en France est assez faible puisqu'il y a 5 CESTIs agréés et 1 CESTI en formation, chacun possédant des domaines d'expertises différents.

Ainsi, on dispose dans le domaine de l'informatique et des réseaux des CESTIs AQL, Oppida et Algoriel (en formation) et dans le domaine des composants électroniques et des logiciels embarqués des CESTIs CEA - LETI, CEACI (THALES - CNES) et SERMA Technologies.

Les produits certifiés au sein du schéma français bénéficient des accords de reconnaissance mutuelle européens : SOG-IS qui permet la reconnaissance entre les états signataires de l'accord des certificats délivrés par leur autorité de certification. La reconnaissance mutuelle européenne s'applique jusqu'aux niveaux ITSEC E6 et CC EAL7. À l'instar des CC (Common Criteria) [24], les ITSEC (Information Technology Security Evaluation Criteria) [25] sont une autre batterie de critères d'évaluation de la sécurité des systèmes informatiques. Les niveaux E6 et EAL7 correspondent comme nous le verrons plus loin à des niveaux d'assurance que l'on peut avoir dans la sécurité d'un produit. Les niveaux E6 et EAL7 sont les niveaux les plus élevés respectivement des ITSEC et des CC. Tous les produits certifiés par les pays signataires de l'accord émetteurs de certificats bénéficient des accords de reconnaissance mutuelle.

En avril 1999, les pays signataires de l'accord émetteurs de certificats sont : la France, l'Allemagne, le Royaume-Uni. Les pays signataires de l'accord qui n'émettent pas de certificats sont : l'Espagne, l'Italie, la Suisse, les Pays-Bas, la Finlande, la Norvège, la Suède et le Portugal. Si ces pays européens n'émettent pas de certificats reconnus dans le cadre du SOG-IS, ils n'en reconnaissent pas moins la valeur des certificats émis. Il existe également un accord de reconnaissance mutuelle selon les Critères Communs : CC-MRA (Common Criteria Mutual Recognition Arrangement). Il permet la reconnaissance, par les pays signataires de l'accord, des certificats délivrés dans le cadre des schémas de certification selon les Critères Communs. La reconnaissance mutuelle s'applique jusqu'au niveau d'évaluation EAL4. En novembre 2003, les pays signataires de l'accord émetteurs de certificats sont : la France, l'Allemagne, le Royaume-Uni, les États-Unis, le Canada, l'Australie, la Nouvelle-Zélande et le Japon. Les pays signataires de l'accord qui n'émettent pas de certificats sont : l'Autriche, l'Espagne, la Finlande, la Grèce, la Hongrie, Israël, l'Italie, la Norvège, les Pays-Bas, la Suède et la Turquie.

5.2 L'évaluation selon les Critères Communs

Avant de présenter une évaluation Critères Communs à proprement dit nous allons expliciter ce que sont ces critères et quels sont les concepts qu'ils mettent en jeu.

5.2.1 Les Critères Communs

La norme Critères Communs

Les Critères Communs (CC) [24] sont une norme qui a pour vocation d'être utilisée comme base pour l'évaluation des propriétés de sécurité des produits et systèmes des Technologies de l'Information (TI). Ils sont définis par le Common Criteria Interpretations Management Board (CCIMB) [4] et sont depuis quelques années une norme ISO définie comme ISO/IEC 15408 [109, 110, 111].

Avant l'apparition des CC, les pays utilisaient des critères d'évaluation différents :

- l'Orange Book [84] pour les États Unis ;
- les ITSEC [25] pour la France, l'Allemagne, le Royaume-Uni et les Pays-Bas ;
- le CTCPEC [83] pour le Canada.

Quelques définitions

Avant d'aller plus loin dans notre présentation, nous allons définir quelques termes des Critères Communs que nous utiliserons par la suite. Nous emploierons en particulier les abréviations :

- TOE (Target Of Evaluation) pour définir la cible d'évaluation, *i.e.* le produit ou le système TI à évaluer ;
- ST (Security Target) pour désigner une cible de sécurité, *i.e.* un document rassemblant l'ensemble des exigences de sécurité et des spécifications à utiliser comme base pour l'évaluation d'une TOE identifiée ;

- PP (Protection Profil) pour désigner un profil de protection, *i.e.* un document rassemblant l'ensemble des exigences de sécurité valables pour une catégorie de TOE qui satisfait des besoins spécifiques d'utilisateurs, indépendamment de son implantation (*e.g.* il existe des PPs pour l'évaluation de puce nue – *i.e.* sans applicatif –, pour le domaine des applications bancaires).

Champ d'application

Les CC traitent de la protection des informations contre leur divulgation, leur modification ou leur perte d'usage. Les types de protection vis-à-vis de ces trois sortes de défaillances de sécurité sont dénommées dans la pratique respectivement confidentialité, intégrité et disponibilité. Ils s'appliquent aux mesures de sécurité des TI implantées dans du matériel, des microprogrammes ou du logiciel. Ils sont donc utiles en tant que guide pour le développement de TOE. Les CC permettent également de comparer les résultats d'évaluations de sécurité menées indépendamment les unes des autres. En revanche, les CC ne traitent ni de la méthodologie d'évaluation ni du cadre administratif et légal dans lequel les critères peuvent être utilisés par les autorités d'évaluation.

Audience

Trois catégories de personnes sont intéressées de façon générale par l'évaluation des propriétés de sécurité des produits et systèmes TI. Il s'agit des utilisateurs de TOE, des développeurs de TOE et des évaluateurs de TOE.

Les CC ont été rédigés de façon à garantir que l'évaluation répond aux besoins des utilisateurs, car il s'agit bien là du but fondamental et de la justification du processus d'évaluation. Les utilisateurs se servent des résultats des évaluations pour décider si un produit ou un système évalué répond à leurs besoins de sécurité. Ces besoins de sécurité résultent typiquement d'une analyse de risques et de la définition d'une stratégie politique. Les utilisateurs peuvent également utiliser les résultats d'évaluation pour comparer différents produits ou systèmes. La présentation hiérarchique des exigences d'assurance facilite ce besoin. De plus, les CC offrent aux utilisateurs, particulièrement à ceux constitués en groupes et en communautés d'intérêts, une structure appelée Profil de Protection (PP), indépendante de l'implantation, dans laquelle ils peuvent exprimer leurs exigences spécifiques en termes de mesures de sécurité d'une TOE. Toutes les TOE souhaitant être compatibles avec un PP donné planteront au moins dans leur ST toutes les exigences fonctionnelles de sécurité définies dans ce PP. Ainsi, Sun microsystems a obtenu la certification de quatre PPs pour faciliter l'écriture de ST pour les produits Java Card. Le choix du PP avec lequel la ST est compatible dépendra des fonctionnalités implantées par la TOE et de celles que le commanditaire voudra faire évaluer. Ces quatre PPs [172, 168] sont :

- JavaCard System Minimal Configuration Protection Profile version 1.0b [99] ;
- JavaCard System Standard 2.1.1 Configuration Protection Profile version 1.0b [100] ;
- JavaCard System Standard 2.2 Configuration Protection Profile version 1.0b [101] ;
- JavaCard System Defensive Configuration Protection Profile version 1.0b [102].

Les développeurs peuvent également s'appuyer sur les CC pour préparer et aider à l'évaluation de leurs produits ou systèmes et pour identifier les exigences de sécurité que leurs propres produits ou systèmes doivent satisfaire. Par la suite, ils peuvent utiliser les structures des CC pour déclarer que leur TOE est conforme à ses propres exigences grâce à des fonctions de sécurité et des composants d'assurance spécifiés, qui devront être évalués. Les exigences de chaque TOE se trouvent dans une structure qui dépend de l'implantation, appelée Cible de Sécurité (ST : Security Target). En revanche, comme nous venons de le voir les exigences d'une grande partie de la clientèle du développeur peuvent être exprimées au moyen d'un ou de plusieurs PP. Les CC décrivent les fonctions de sécurité qu'un développeur peut inclure dans la TOE. Ils peuvent également être utilisés pour déterminer les responsabilités et les actions contribuant à constituer les éléments de preuve nécessaires à l'évaluation de la TOE. Enfin, ils définissent aussi le contenu et la présentation de ces éléments de preuve.

Les CC contiennent les critères que les évaluateurs doivent utiliser pour juger de la conformité des TOE à leurs exigences de sécurité. Les CC décrivent l'ensemble des actions d'ordre général que l'évaluateur doit mener, ainsi que les fonctions de sécurité auxquelles s'appliqueront ces actions. Il est à noter que les CC ne spécifient pas les procédures à suivre pour mener ces actions.

Organisation des Critères Communs

Les CC se présentent sous un ensemble de trois parties distinctes mais liées entre elles comme nous l'indiquons ci-dessous.

- « La partie 1, Introduction et modèle général » [109], est l'introduction des CC. Elle définit les concepts généraux et les principes de l'évaluation de la sécurité des TI, et présente un modèle général d'évaluation. La partie 1 présente également des structures pour exprimer des objectifs de sécurité des TI, pour sélectionner et définir des exigences de sécurité des TI et pour écrire des spécifications générales pour produits et systèmes. L'utilité de chaque partie des CC est décrite en fonction de chacune des audiences visées.
- « La partie 2, Exigences fonctionnelles de sécurité » [110], établit un ensemble de composants fonctionnels pour exprimer de façon standardisée les exigences fonctionnelles des TOE. Elle propose un catalogue de l'ensemble des composants fonctionnels, des familles et des classes.
- « La partie 3, Exigences d'assurance de sécurité » [111], établit un ensemble de composants d'assurance pour exprimer de façon standardisée les exigences d'assurance des TOE. Elle propose un catalogue de l'ensemble des composants d'assurance, des familles et des classes. La partie 3 définit également des critères d'évaluation pour PP et ST et présente les niveaux d'assurance de l'évaluation qui définissent l'échelle prédéfinie des CC pour coter l'assurance pour les TOE, constituée par les niveaux d'assurance de l'évaluation ou EAL (Evaluation Assurance Levels).

5.2.2 Les concepts

La sécurité a trait à la protection de biens contre des menaces, ces dernières étant classées selon leur potentiel de nuisance envers les biens protégés. La figure 31 présente les concepts de sécurité et leurs relations.

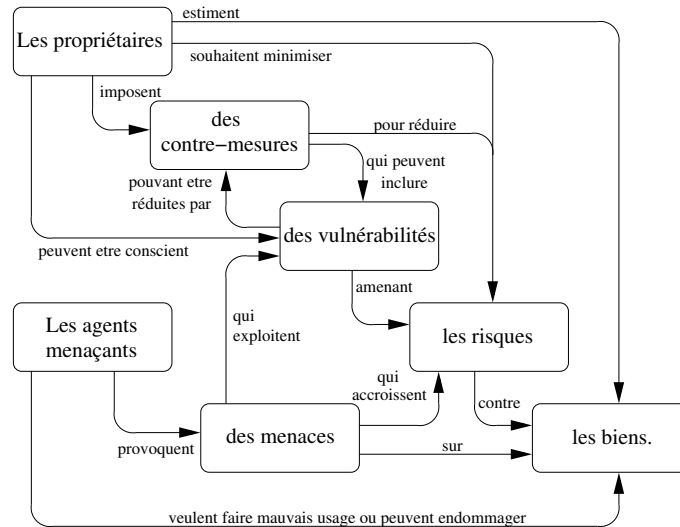


FIG. 31 – Concepts de sécurité et relations.

Ainsi, la sauvegarde des biens dignes d'intérêt est de la responsabilité des propriétaires pour qui ces biens ont de la valeur. Les propriétaires des biens sont les commanditaires de l'évaluation de la TOE ou ses clients.

Prenons l'exemple où la TOE est une carte à puce ; un bien qu'une banque voudra protéger pourra être le numéro de compte du porteur de carte.

Des agents menaçants, réels ou présumés, peuvent aussi attacher de la valeur à ces biens et chercher à en faire un usage contraire aux intérêts du propriétaire. L'agent menaçant pourra par exemple être un pirate souhaitant débiter de l'argent sur le numéro de compte du porteur de carte. C'est pour cela que les propriétaires des biens vont analyser les menaces possibles pour déterminer celles qui s'appliquent à leur environnement. Imaginons maintenant que le numéro de compte stocké sur la carte soit accessible en lecture sans qu'il ne soit nécessaire de s'authentifier, alors une menace sera le vol de la carte (TOE) qui accroîtra le risque de divulgation du bien (perte de confidentialité). En effet, un pirate pourra exploiter une vulnérabilité de la TOE pour lire le numéro de compte du porteur. Bien évidemment l'analyse de risque du propriétaire sera utile au choix des contre-mesures pour minimiser les risques et les ramener à un niveau acceptable.

Des contre-mesures sont alors imposées pour réduire les vulnérabilités et pour satisfaire aux politiques de sécurité des propriétaires des biens (soit de façon directe, soit de façon indirecte en donnant des instructions aux autres parties). Et de fait, dans notre cas, une authentification, par exemple, par PIN, sera mise en place pour réduire le risque

de divulgation. La vulnérabilité précédente sera donc réduite par la contre-mesure d'authentification. Néanmoins, des vulnérabilités résiduelles peuvent persister après la mise en œuvre de contre-mesures. Si par exemple le nombre d'essais du PIN n'était pas limité dans l'implantation mise en place, une nouvelle menace serait d'essayer tous les PINs possibles pour exploiter la vulnérabilité et accéder au bien qu'est le numéro de compte. De telles vulnérabilités peuvent être exploitées par les agents menaçants, constituant ainsi un niveau de risque résiduel à l'encontre des biens. Par conséquent, les propriétaires vont chercher à minimiser ce risque en prenant en compte d'autres contraintes.

Comme on peut le voir figure 32, les propriétaires auront besoin d'avoir confiance dans le fait que les contre-mesures sont adéquates pour contrer les menaces qui pèsent sur les biens avant de permettre que leurs biens ne soient exposés aux menaces spécifiées. Or, les propriétaires peuvent ne pas posséder par eux-mêmes la capacité de juger tous les aspects des contre-mesures et par conséquent peuvent chercher à les faire évaluer. On a supposé plus haut que les propriétaires représentés sur la figure 31 étaient les commanditaires de l'évaluation alors qu'il s'agit souvent plutôt en réalité des concepteurs de la TOE. En revanche, sur la figure 32 les propriétaires sont en général les commanditaires de l'évaluation. Le résultat de l'évaluation sera une déclaration concernant l'étendue de l'assurance que

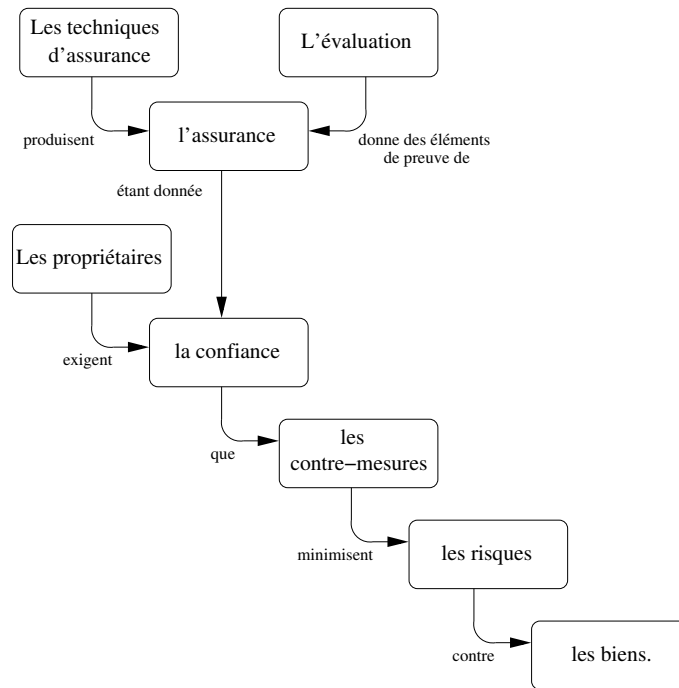


FIG. 32 – Concepts de l'évaluation et relations.

l'on peut avoir relativement à la confiance à accorder aux contre-mesures pour réduire les risques contre des biens protégés. Cette déclaration attribuera une cotation de l'assurance des contre-mesures, l'assurance étant la propriété des contre-mesures qui fonde la confiance dans leur bon fonctionnement. Ainsi, dans notre exemple, le résultat de l'évaluation (*i.e.* la déclaration) sera la cotation de la contre-mesure qu'est le mécanisme d'authentification par PIN et produira une assurance plus ou moins forte selon son implantation. Selon le

résultat de cette déclaration, le propriétaire des biens décidera s'il accepte ou non le risque d'exposer ses biens aux menaces.

Bien évidemment, notre exemple très simpliste ne servait qu'à illustrer le propos.

5.2.3 L'évaluation

Voyons maintenant sur un plan pratique comment les CC sont utilisés dans une évaluation.

Le premier acteur mis en jeu dans le processus d'évaluation est le développeur. Il a en charge de rédiger la ST ou le PP, document fondateur de l'évaluation qui décrit la TOE et son périmètre d'évaluation. Dans le cas d'une carte à puce, le développeur peut par exemple vouloir faire évaluer soit seulement le circuit, soit seulement une application, soit les deux. Il peut aussi vouloir exclure certaines parties, comme des mécanismes cryptographiques qu'il ne souhaite pas voir évalués. On notera donc, que lorsqu'un utilisateur devra choisir entre différents produits évalués pour les fonctionnalités qu'il souhaite, il ne devra pas seulement regarder le niveau d'assurance qu'aura reçu la TOE, mais aussi le périmètre d'évaluation de cette TOE.

La deuxième étape pour le développeur consistera à identifier les biens à protéger dans le périmètre d'évaluation de la TOE. En partant de ces biens et du but de la TOE, il fera une analyse de risque prenant en compte l'environnement physique dans lequel sera utilisée la TOE. En effet, si la TOE est destinée à être placée dans un coffre-fort entouré de gardes armés, les risques sont moindres. À l'inverse, ils sont majorés si elle est destinée à être placée à la portée de tout un chacun. De cette analyse, le développeur établira une liste de menaces, une liste d'hypothèses et mettra en place une politique organisationnelle de sécurité.

En tenant compte des hypothèses et de la politique de sécurité, il dérivera un certain nombre d'objectifs de sécurité afin de contrer les menaces. À ce moment là, le développeur utilisera la partie 2 des CC, *i.e.* le catalogue des exigences fonctionnelles de sécurité, dans lequel il choisira les exigences nécessaires pour réaliser les objectifs de sécurité. Si jamais il ne trouvait pas l'exigence désirée dans cet épais catalogue, ce qui est peu probable, il a aussi la possibilité d'en définir de nouvelles. Une fois les exigences choisies, le développeur doit les réaliser par des fonctions de sécurité qui seront une représentation de ce qui sera implanté dans la TOE pour réaliser les objectifs de sécurité. Ces étapes sont résumées sur la figure 33.

La ST ou le PP devra inclure tous ces éléments de preuve ainsi que leur correspondance et leur complétude. Attention, le PP étant une ST générique, il ne comprendra jamais d'information relative aux fonctions de sécurité qui sont, elles, dépendantes de l'implantation. Souvent, la correspondance et la complétude sont réalisés sous la forme de plusieurs tableaux prouvant que :

- toutes les menaces sont bien couvertes soit par des hypothèses sur l'environnement soit par des objectifs de sécurité ;
- tous les objectifs de sécurité sont bien décrits par des exigences fonctionnelles de sécurité ;

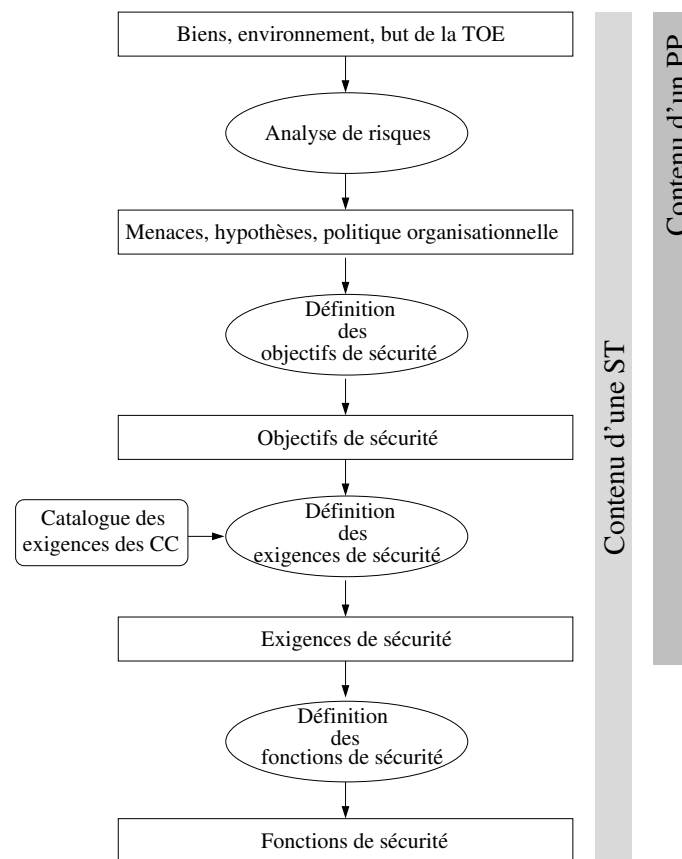


FIG. 33 – Contenu d'une ST et d'un PP.

- toutes les exigences fonctionnelles de sécurité sont bien implantées par une ou plusieurs fonctions de sécurité.

Même si cela n'est pas imposé par les CC, ces étapes devraient normalement être un préalable à la conception d'une TOE. En effet, le développeur écrit fréquemment des spécifications fonctionnelles de sa TOE puis il les implante en suivant une méthodologie sécuritaire assez informelle. Or, cela aboutit bien souvent à l'introduction de vulnérabilités liées à des menaces oubliées lors de la phase de rédaction des spécifications fonctionnelles. En tout état de cause une analyse logique telle que celle que nous avons décrite est beaucoup plus efficace pour fournir les preuves que la TOE fait bien ce qu'elle doit faire.

Les CC suggèrent également au développeur de suivre un modèle de développement représenté figure 34. Il consiste à partir des exigences de sécurité pour obtenir les spécifications fonctionnelles de sécurité puis par raffinements successifs de la conception (*i.e.* design de haut niveau, design de bas niveau) à obtenir l'implantation.

Par ailleurs, suivant le niveau d'assurance qu'il souhaite obtenir pour sa TOE, le développeur devra fournir différents documents correspondant aux spécifications fonctionnelles de sécurité, au design de haut niveau, au design de bas niveau, à l'implantation et nommés respectivement FSP (Functional SPecification), HDL (High Level Design), LLD (Low Level Design) et IMP (IMPlimentation). Si le développeur a bien appliqué depuis le début les conseils de développement suggérés par les CC, il devrait donc être en mesure de fournir les éléments de preuve que lui demandera l'évaluateur afin de s'assurer que les biens sont bien protégés contre toutes les menaces par des fonctions de sécurité efficacement implantées.

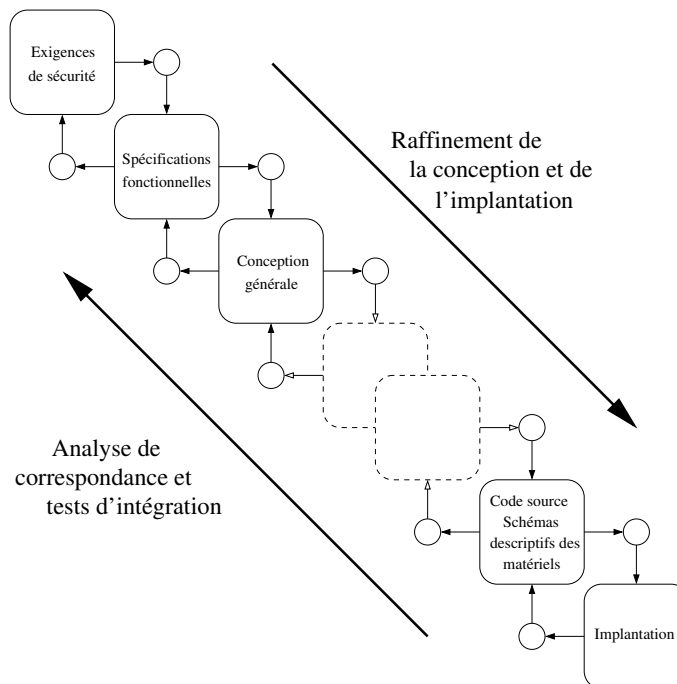


FIG. 34 – Modèle de développement d'une TOE.

Une fois la TOE conçue et sa ST écrite ou une fois le PP écrit, l'évaluateur va commencer l'évaluation du document. Pour l'aider dans son travail, il a à sa disposition la CEM (Common Methodology for Information Technology Security Evaluation) [82], une méthodologie d'évaluation commune qui contribue à la répétabilité et à l'objectivité des résultats. Mais elle n'est cependant pas suffisante par elle-même car la plupart des critères d'évaluation nécessitent des jugements d'expert et une connaissance générale du contexte, d'où les domaines de spécialités des différents CESTIs.

Comme nous l'avons vu précédemment sur la figure 32, le travail de l'évaluateur consiste à donner des éléments de preuve de l'assurance, *e.g.* prouver que ce qui est écrit dans la ST est bien implanté et de manière efficace dans la TOE. Comme le montre la figure 35, l'évaluateur devra donner des éléments de preuve de l'assurance que :

- la ST est correctement et complètement implantée ;
- le processus de développement est adéquatement organisé ;
- la TOE fait ce qu'elle est supposée faire, ni plus ni moins ;
- le client déploie correctement la TOE.

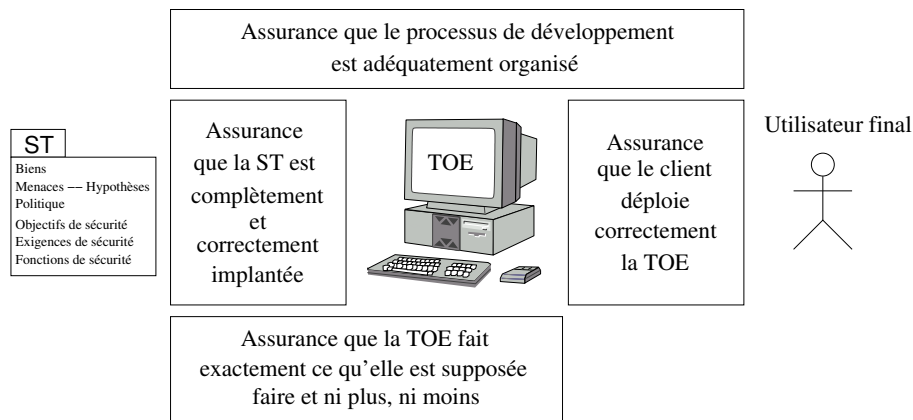


FIG. 35 – L'assurance des Critères Communs.

Pour obtenir et apporter les preuves de cette assurance, l'évaluateur devra vérifier un certain nombre d'exigences d'assurance qui sont décrites dans la partie 3 des CC et qui sont replacées dans leur contexte sur la figure 36. En fait, selon le niveau d'assurance que voudra atteindre le commanditaire de l'évaluation pour la TOE, la tâche de l'évaluateur pour fournir l'assurance sera plus ou moins complexe. En effet, selon le niveau d'assurance choisi, le développeur devra fournir plus ou moins d'éléments de preuve de la bonne implantation de ce qui est décrit dans la ST. Toujours selon le niveau d'assurance, il arrivera que l'évaluateur ait besoin d'auditer les sites de développement afin d'obtenir l'assurance qu'aucune information confidentielle ne puisse sortir dans la nature lors des phases de conception et de fabrication.

Au niveau le plus bas de l'échelle d'assurance des CC, EAL1 (Evaluation Assurance Level 1), le développeur doit seulement fournir à l'évaluateur les documents relatifs aux exigences d'assurance représentées sur la figure 37. Le travail du développeur est donc minime puisqu'il n'a presque aucune information sur la TOE. Donc, la confiance que l'on

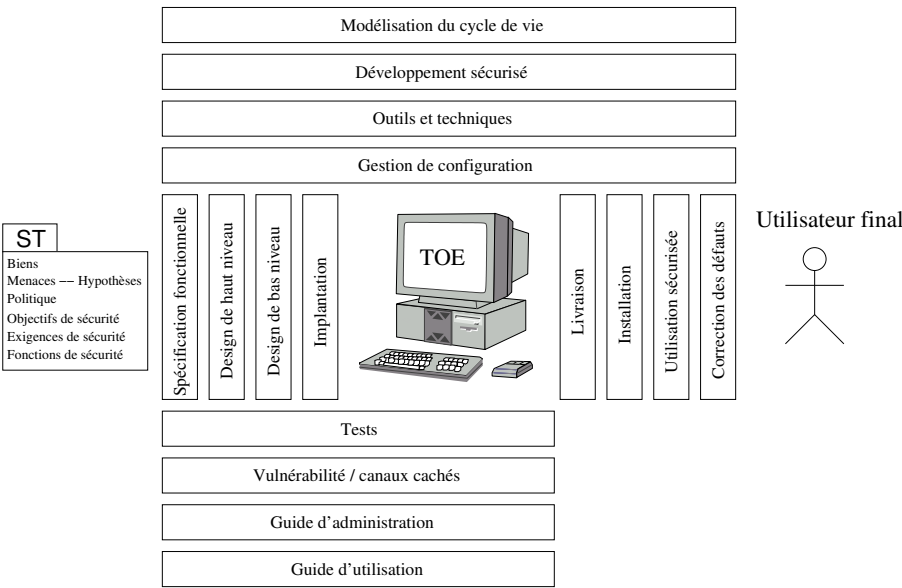


FIG. 36 – Les exigences d’assurance des Critères Communs.

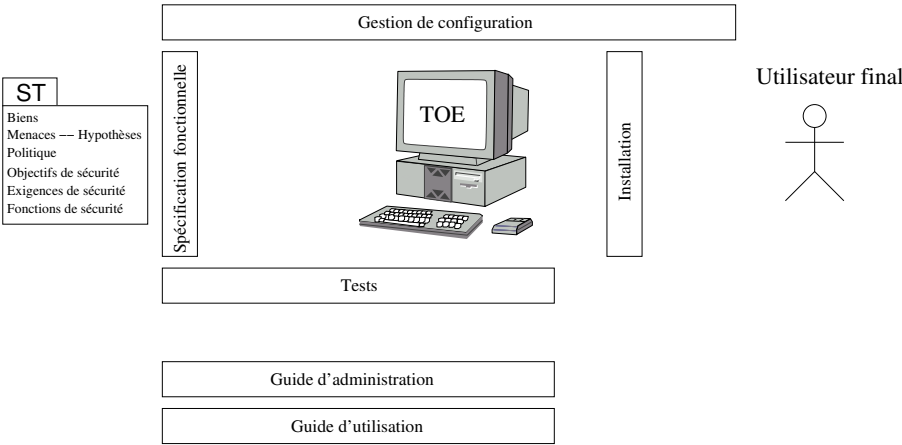


FIG. 37 – Les exigences d’assurance des Critères Communs pour le niveau EAL1.

pourra avoir dans le produit sera elle aussi minime. En revanche, au niveau le plus haut de l'échelle d'assurance des CC, EAL7, le développeur doit fournir à l'évaluateur les documents relatifs à toutes les exigences d'assurance représentées sur la figure 36. Certains de ces documents doivent même fournir des preuves formelles. Ainsi, le niveau de connaissance que l'évaluateur possède de la TOE varie en fonction des différents niveaux d'assurance. Cela passe d'une connaissance boîte noire pour les niveaux les plus bas à une connaissance boîte blanche pour les niveaux les plus élevés (*cf.* FIG. 38).

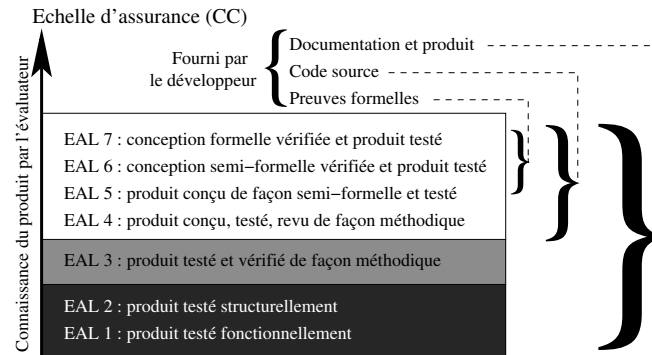


FIG. 38 – L'échelle d'assurance des Critères Communs.

Aujourd'hui, le plus haut niveau d'évaluation des produits est en général EAL4+ ou EAL5+. Bien évidemment, une fois le produit évalué, il doit passer l'étape de la certification dont nous avons parlé précédemment.

5.3 Conclusion

Dans le cadre de ma thèse CIFRE au sein de SERMA Technologies, un des objectifs de mes travaux était d'accompagner le CESTI dans la définition de méthodologies d'évaluation pour les produits implantant la technologie Java Card. En effet, cette nouvelle technologie suscitait beaucoup d'interrogations quant à la façon de conduire une évaluation et aux outils à utiliser. Java étant un langage de haut niveau produisant du code intermédiaire (*i.e.* le *bytecode*), est-ce que l'on pourrait évaluer indépendamment la plate-forme Java Card d'un côté et les applications de l'autre? Java étant également un langage adéquat pour des modélisations formelles, allait-on pouvoir atteindre des niveaux d'assurance élevés (*i.e.* EAL 7)? Mes activités de recherche avaient aussi des buts plus pragmatiques comme l'identification de nouvelles vulnérabilités et la créations de nouvelles attaques utilisables par le CESTI dans le cadre de son activité d'évaluation sécuritaire.

L'étude du domaine de la certification menée ici nous permettra d'avoir une vision précise des acteurs, des risques et des méthodologies d'évaluation pour les appliquer au mieux dans la suite de nos travaux au contexte de la Java Card.

Seconde partie

Un environnement libre pour Java Card

Dans cette partie, nous allons présenter le projet « Sécurité Java Card » issu de la collaboration entre le LaBRI et SERMA Technologies. Ce projet, dont la vocation était d'accroître l'expertise du LaBRI et de SERMA Technologies dans le domaine de la technologie Java Card, a abouti à la réalisation d'un environnement d'expérimentation libre pour Java Card. Nous détaillerons donc l'architecture et les possibilités de cet environnement, appelé JCatools. Par ailleurs, l'élaboration de notre suite d'outils nous a permis de détecter des ambiguïtés dans les spécifications Java Card quant à la modélisation de la mémoire. C'est pourquoi dans le chapitre 8, nous proposons d'introduire le concept de *pré-persistance* afin d'expliquer les différents modèles de mémoire que nous envisageons.

Nos contributions sont le développement de l'environnement d'expérimentation JCatools et l'introduction de la *pré-persistance*.

Chapitre 6

Contexte

Lorsque SERMA Technologies a contacté le LaBRI pour la première fois, en 1999, c'était dans l'objectif de se former dans le domaine de la sécurité sur Java Card. En effet, les premiers produits Java Card commençaient à être développés et les évaluations à des niveaux bas des Critères Communs (*i.e.* EAL1+ ou EAL2+) devaient débiter pour se poursuivre en cas de réussite à des niveaux plus hauts (*i.e.* EAL4+). À cette époque, le CESTI de SERMA Technologies était le seul centre en France habilité par la DCSSI pour mener des évaluations Java Card et jusqu'alors une expertise limitée lui avait permis de répondre aux besoins des évaluations.

Néanmoins, en 2001, comme il était alors impératif que le CESTI s'approprie totalement cette technologie, la société SERMA Technologies m'a recruté comme ingénieur de recherche et développement pour mener une thèse, sous convention CIFRE, autour des nouvelles problématiques apportées par Java Card. Les travaux menés lors de mon DEA (*i.e.* un survey [188] et le mémoire de DEA [189]) m'avaient permis d'acquérir une bonne connaissance de la technologie. En parallèle, le LaBRI et SERMA Technologies se sont associés, en 2001, autour d'un projet de recherche intitulé « Sécurité Java Card », labellisé PROGSI (*i.e.* « PROGramme Société de l'Information ») par le Ministère de l'Économie, des Finances et de l'Industrie. Les objectifs étaient de fournir à SERMA Technologies les compétences et les moyens tant en termes de logiciels, que de méthodologies pour mener à bien ses évaluations sécuritaires de cartes à puce multi-applicatives.

6.1 Les problèmes d'évaluation apportés par Java Card

En 2000, lorsque Java Card a réellement fait son entrée dans les schémas de certification en vue d'obtenir ses lettres de noblesses d'environnement ultra-sécurisé, les responsables lui ont réservé un accueil assez froid. En effet, la multi-application dynamique (*i.e.* chargement ou effacement d'*applet* après l'émission de la carte) qui devait révolutionner le monde de la carte à puce a été perçue comme une fonctionnalité dangereuse pour la sécurité du système. S'il est compréhensible qu'une nouvelle technologie mette du temps à convaincre, aujourd'hui en 2004, nous en sommes essentiellement au même point. L'argument développé par les vendeurs de technologies multi-applicatives, selon lequel on pourrait faire baisser les

coûts d'évaluations en évaluant une fois et une seule la plate-forme, et au coup par coup les applications qu'elle embarquera, n'est plus valable. En effet, les organismes responsables des différents schémas de certification se sont toujours refusé à laisser certifier une application seule. On peut se demander pourquoi puisque dans le cadre d'applications embarquées sur PC ces mêmes organismes acceptent tout à fait la certification d'application (*e.g.* pare-feu).

Toujours est-il que dans le cadre de la certification des cartes multi-applicatives et en particulier Java Card, en France, la DCSSI ne permet pas l'évaluation d'une *applet* seule. C'est d'autant plus dommageable que le langage et les services à la disposition d'une *applet* sont assez restreints et tout à fait adéquats pour donner lieu à des évaluations d'*applets* indépendamment de toute plate-forme. Il serait possible, après leur certification, de les charger sur n'importe quelle plate-forme Java Card déjà évaluée. Par ailleurs, la DCSSI ne permet pas non plus l'évaluation d'une plate-forme Java Card sur un composant non certifié. Cette position est quand même beaucoup plus compréhensible puisque l'environnement Java Card est l'équivalent d'un système d'exploitation. C'est donc un système plus complexe qu'une *applet* et dont une seule faille peut mettre à mal la sécurité de l'ensemble : puce, environnement Java Card et *applets*.

Pour résumer, aux contraintes techniques de Java Card s'ajoutent les contraintes politiques imposées par les organismes de certification. Ainsi, en France, il est seulement possible d'évaluer des produits dont la cible de sécurité décrit un des périmètres d'évaluation suivant :

- la puce seule ;
- la puce et l'environnement Java Card (*i.e.* JCRE, JCVM et API) ;
- l'environnement Java Card sur une puce déjà certifiée ;
- la puce, l'environnement Java Card et une ou plusieurs *applets* ;
- l'environnement Java Card et une ou plusieurs *applets* sur une puce déjà certifiée ;
- une ou plusieurs *applets* sur une plate-forme déjà certifiée (*i.e.* puce et environnement Java Card certifiés).

Alors même que le verrou technologique est en passe d'être levé puisque l'on sait maintenant faire des vérificateurs embarqués ou des machines virtuelles défensives sur les cartes, on constate qu'il y a toujours un réel verrou psychologique chez les industriels et les organismes de certification qui refusent de laisser cohabiter librement plusieurs applications sur le même support. On peut également noter la réticence des utilisateurs à posséder plusieurs applications sur la même carte (*e.g.* carte bancaire et application Monéo). De plus, si vous conjecturez qu'à terme chacun d'entre nous aura le contenu de son porte-feuille sur une seule et même carte (*i.e.* carte bancaire, carte vitale, permis de conduire, carte d'identité, etc.), on vous rétorquera quasiment tout le temps : « Et si je perds ma carte, je fais comment ? Non, moi je ne veux pas ça ! ». Aussi, pour l'instant, afin de rassurer les industriels et les responsables des schémas de certification, les cartes multi-applicatives sont soit fournies avec des mécanismes tels que GlobalPlatform pour sécuriser le chargement, soit utilisées comme plate-forme mono-applicative ou multi-applicative statique. Constatant, cette dernière tendance, Sun microsystems a lancé en 2003 son programme Java Card S [27]. Les Java Card S sont des Java Cards classiques mais ne possédant pas les fonctionnalités de gestion dynamique d'applications. Quel paradoxe pour une technologie qui pourrait

révolutionner le monde de la carte à puce grâce à la multi-application dynamique !

Par ailleurs, l'arrivée de ces technologies multi-applicatives a aussi stigmatisé les lacunes des méthodologies d'évaluation classiques. En effet, il fallait alors prendre en compte les nouvelles menaces dues à la multi-application dynamique qui permet les attaques internes postérieurement à la phase d'évaluation – ce qui n'était pas le cas avant, puisque lors de l'analyse du produit toutes les applications étaient déjà embarquées sur la carte et pouvaient donc être vérifiées in situ. Dans ce cadre nous avons défini de nouvelles procédures d'évaluation des plates-formes multi-applicatives pour le CESTI de SERMA Technologies.

6.2 Le projet *Sécurité Java Card*

6.2.1 Présentation

Lancé en juin 2001, ce projet fait suite à une première collaboration avec le LaBRI initiée en 1999. En effet, lors de leurs premières évaluations sécuritaires Java Card à des niveaux faibles (EAL1), le personnel du CESTI de SERMA Technologies avait suivi au LaBRI une formation au langage Java et une courte collaboration s'était développée pour mettre au point de nouveaux types d'attaques. Ce premier contact a permis de mettre sur pied une collaboration à plus long terme dans le domaine de Java Card : le projet *Sécurité Java Card*.

Son but était de mettre en œuvre une expertise par défaut de sécurité sur les produits Java Card. Les objectifs étaient de développer :

- une bonne expertise de la structure interne des produits répondant aux spécifications Java Card ;
- une gamme étendue d'attaques logicielles, de haut niveau technique, afin de tester et de valider la qualité des implantations de produits sécurisés soumis à une évaluation Critères Comuns ou ITSEC.

Au terme du projet, les partenaires prévoyaient de formaliser les attaques développées sous forme d'un jeu d'*applets* agressives permettant de tester la résistance de divers produits basés sur Java Card. En fonction des faiblesses identifiées, des procédures de vérification systématique de certaines caractéristiques des produits devaient être établies, en collaboration avec la DCSSI. La démarche de ce projet s'inscrivait exactement dans le cadre de l'évaluation et de la certification de la sécurité des Technologies de l'Information, puisque le CESTI de SERMA Technologies était agréé pour réaliser des évaluations sécuritaires de haut niveau. La réussite du projet devait contribuer à améliorer la position du schéma français d'évaluation et de certification (représenté par la DCSSI) au niveau national et international. En effet, le schéma bénéficie naturellement d'une reconnaissance plus forte pour un domaine technologique lorsqu'un des CESTI agréés dispose d'une compétence pointue – et reconnue par les industriels – pour l'évaluation de la sécurité dans ce domaine.

6.2.2 Déroulement

Lors de son lancement le projet se décomposait en trois phases. La phase 1 d'une durée de 5 mois devait permettre d'appréhender et de développer les outils nécessaires aux travaux de recherches ultérieurs. Elle s'organisait ainsi :

- étape 0 : prise en main de l'outil Java Card par le LaBRI et SERMA Technologies ;
- étape 1 : choix d'un micro-contrôleur ;
- étape 2 : étude et développement d'une machine virtuelle.

La phase 2 d'une durée de 5 mois prévoyait l'analyse de la machine virtuelle Java Card afin de permettre d'en élaborer un modèle pour valider les propriétés du système. Elle s'organisait en deux étapes :

- étape 3 : analyse de la machine virtuelle Java Card ;
- étape 4 : élaboration d'un modèle partiel ou complet de Java Card / spécification formelle.

Enfin la phase 3 qui s'étalait sur 5 mois consistait à développer des tests d'attaque sur les produits actuels et sur les futures générations de produits. Ainsi, cette phase suivait trois étapes :

- étape 5 : chargement d'applications agressives ;
- étape 6 : analyse des futures générations de Java Cards dans l'objectif de développer des applications virus ;
- étape 7 : communication et commercialisation.

6.2.3 Les choix et quelques résultats

Malgré un décalage sur le planning initial, la phase 1 a été un succès total tant au niveau de la collecte d'informations qu'au niveau du développement d'une machine virtuelle. La rédaction d'un rapport interne du LaBRI [188] mais aussi de mon mémoire de DEA [189] avait permis de bien commencer le projet (*i.e.* étape 1). Cependant, plutôt que de cibler un micro-contrôleur particulier, nous avons finalement décidé de réaliser un émulateur sur PC (*i.e.* étape 2). Parmi les arguments qui nous ont orientés vers ce choix, celui nous permettant de travailler virtuellement sur les plates-formes et les micro-contrôleurs des différents constructeurs par un paramétrage approprié de notre émulateur a été décisif. L'idée initiale de ce micro-contrôleur s'est donc traduite en contraintes sur la machine virtuelle qui a été développée ; elle reproduit ainsi, le plus fidèlement possible, les conditions effectives du fonctionnement d'une machine Java dans une carte à puce (limitation mémoire, etc.). De plus, il était plus facile d'intégrer au niveau logiciel qu'au niveau matériel des outils de collecte de traces ainsi que d'autres mécanismes nécessaires et utiles dans un environnement logiciel (*e.g.* le débogage). Nous avons également pu fournir avec notre émulateur des outils de haut niveau facilitant la mise en œuvre d'attaques et leur observation afin de concourir au mieux à la réalisation du projet. Les outils et l'environnement que nous avons développés seront détaillés au chapitre 7.

En outre, du fait de la confidentialité des produits sur lesquels SERMA Technologies est amenée à travailler, il était difficile de transmettre au LaBRI un produit développé

par un client. En conséquence, le LaBRI a mené l'étape d'analyse (*i.e.* étape 3) de la machine virtuelle et des spécifications Java Card à partir de son propre développement. Cela a conduit à la rédaction d'un article [88], publié dans une conférence internationale, présentant la suite JCatools et certaines lacunes des spécifications Java Card. Ce dernier point sera d'ailleurs exposé dans le chapitre 8 puisque c'est l'un des résultats de nos travaux de développement de l'environnement Java Card. Nous verrons également que pour combler ces lacunes nous avons proposé l'introduction d'un nouveau concept : la *pré-persistence*. Ce concept a donné lieu à un article sur la formalisation des modèles mémoires Java Card [90]. De plus, à l'issue du développement de notre machine virtuelle, nous possédions une parfaite connaissance de la technologie Java Card et nous avons ainsi rédigé pour SERMA Technologies un rapport de synthèse sur les divers aspects sécuritaires de gestion de la mémoire sur une Java Card.

Même s'il s'agissait de l'étape ultime du projet, dès le début du développement de la JCVM, nous avons commencé à mettre en place des *applets* agressives (*i.e.* étape 5) pour valider les différents mécanismes que nous implantions. D'ailleurs ces jeux de tests ont, pour partie, déjà été utilisés avec succès par SERMA Technologies lors d'évaluations de produits Java Card. C'est également dans ce cadre que nous avons pu mettre en place des outils facilitant le développement de ces attaques afin d'éviter les étapes de modification manuelle de fichiers CAP.

Enfin, l'étape de communication (*i.e.* étape 7) a parfaitement été réalisée tout au long du projet grâce à l'organisation d'une journée de la carte à puce à Bordeaux [34], à la participation à divers actions nationale (*i.e.* Action Spécifique « Sécurité logicielle : modèles et vérification » [2] et Réseau Thématique Pluridisciplinaire « Systèmes embarqués complexes ou contraints » [55]) et à la publication des rapports intermédiaires d'avancement et des articles. Lors de la journée organisée par le LaBRI le 19 décembre 2001 et intitulée « les nouveaux enjeux de la carte à puce » [34] une centaine de personnes ont pu à la fois découvrir les technologies, les produits mais aussi les problématiques de sécurité sous-jacentes. Par ailleurs, des contacts ont pu être établis avec les principaux acteurs du secteur de la carte à puce (Oberthur, Schlumberger, Gemplus, Sun microsystems, etc.). Il semblerait que ces sociétés soient intéressées par la définition d'une suite au présent projet qui impliquerait tout ou partie de ces intervenants.

Pour conclure, il nous faut admettre que nous avons été un peu trop optimiste quant à la modélisation partielle de la machine virtuelle Java Card (*i.e.* étape 4) puisqu'elle n'a finalement pas vu le jour. Une cause majeure est le décalage de cette phase à la fin du projet mais également le manque de ressources pour mener à bien une telle tâche. Les travaux similaires menés dans les projets Formavie sont une preuve, s'il en fallait, du nombre important de personnes nécessaires pour réussir de telles activités. Nous n'avons pas, non plus, tout à fait abouti dans le développement d'application virus (*i.e.* capable de se répliquer) pour les générations Java Card de la version 2.1 à la version 2.2.

6.3 Les outils existants au début du projet

Lors du lancement du projet « Sécurité Java Card » aucun outil susceptible d'aider SERMA Technologies pour ses évaluations Java n'existait.

Sur le marché, seul Sun microsystems proposait gratuitement un kit de développement [28]. Les autres acteurs possédaient des produits payants et peu adaptés. Le kit de Sun microsystems permettait de créer des *applets* et des bibliothèques Java Card, de simuler des programmes dans son simulateur et de les émuler dans un environnement d'émulation totalement fermé. On ne pouvait, par conséquent, tirer aucune information sur le fonctionnement interne (*i.e.* on ne pouvait pas visualiser les interactions entre les *applets* et le reste du système).

Gemplus proposait également un kit de développement GemXpresso RAD [16] basé pour partie sur le kit de Sun microsystems. L'atout de ce produit était un environnement de simulation un peu plus évolué permettant de suivre, par l'intermédiaire d'une interface graphique, l'évolution des valeurs des champs des objets mais seulement entre deux échanges APDUs. Le produit était donc peu intéressant pour SERMA Technologies. De plus, puisqu'il s'agissait d'un simple simulateur, le comportement des *applets* pouvait différer une fois qu'elles étaient effectivement chargées sur la carte.

Giesecke & Devrient dans son kit de développement Sm@rtCafé [59] permettait de déboguer les *applets* dans un émulateur et donc de suivre leur comportement réel. Par contre, le produit étant fermé et propriétaire (*i.e.* il émule seulement les Java Cards de Giesecke & Devrient), il n'apportait qu'une connaissance partielle des interactions entre les applications et l'environnement d'exécution Java Card. C'était donc un outil assez cher pour un intérêt somme toute assez limité.

Le kit Cyberflex Access SDK [6] de Schlumberger et le kit M-Smart JADE de Motorola contenaient également un simulateur avec les mêmes inconvénients que ceux déjà évoqués plus haut.

L'équipe du projet OASIS [45] de l'INRIA Sophia Antipolis avait développé un simulateur Java Card [71] mais l'absence des objets transients, des tableaux de *bytes* et des APIs Java Card nous ont conduit à laisser de côté ce produit.

Enfin les autres kits de développement Odyssey-Lab de Bull, MoKard de Incard, NexSmart Java Card SDK de NexSmart technology et GalactIC de Oberthur Card Systems n'offraient pour leur part que la suite d'outils nécessaires à la compilation, à la conversion et au chargement d'*applets* sur leur carte respective.

Il n'existait donc aucun outil répondant aux critères de SERMA Technologies et à l'extensibilité et l'évolutivité souhaitées par le LaBRI.

6.4 Les outils existants aujourd'hui

Actuellement, on trouve, à quelques exceptions près, les mêmes kits que ceux précédemment cités. Bien évidemment, certains acteurs ont disparu du marché (*e.g.* Bull, Motorola) et d'autres sont apparus (*e.g.* IBM). Parmi les nouveaux outils existants aujourd'hui et qui auraient pu intéresser SERMA Technologies, quelques années auparavant, on peut citer le kit de développement JCOP d'IBM et le projet libre JC Emulator.

Le kit JCOP [23] s'intègre à Eclipse, l'environnement de développement Java, sous la forme de *plug'in*. Il permet de développer des *applets* et de tester leur comportement dans un environnement d'émulation. Malheureusement, une fois encore le code source de cet

outil n'est pas disponible.

Le projet libre JC Emulator [64] propose, lui, ses sources sous licence GPL, mais contrairement à ce que son nom laisse supposer il s'agit en fait d'un outil de simulation et non d'un émulateur. L'intérêt est donc une fois de plus assez limité pour SERMA Technologies. Toutefois il est toujours instructif d'analyser le code pour déterminer comment certains concepts ont été implantés.

Notre émulateur est donc bien aujourd'hui le seul outil intéressant pour SERMA Technologies et le LaBRI et dont les sources soient disponibles.

Chapitre 7

Les outils de la suite JCatools

JCatools [88, 140] est un environnement pour l'étude de plates-formes et d'*applets* Java Card. Il est composé d'un ensemble d'outils destinés à répondre aux besoins de SERMA Technologies pour l'évaluation de produits Java Card et aux besoins du LaBRI pour y adjoindre des extensions en particulier en terme de vérification formelle. Ce développement réalisé en Java comprend trois outils principaux (*i.e.* JCat Emulator, JCat View, JCat Converter) ainsi que quelques utilitaires voués à simplifier les opérations de modification de fichiers CAP. Le tableau 8 résume les principaux objectifs du CESTI de SERMA Technologies et comment les JCatools y ont répondu.

Objectifs de SERMA Technologies	Java Card Attack & Test			
	Emulator	View	Converter	Autres
Connaissance détaillée de la technologie Java Card	X	X		
Définition d'une méthodologie d'évaluation de plate-forme	X			
Création d'une batterie de tests d'attaques	X	X	X	X
Expérimentation de tests d'attaques logicielle & matérielle	X			

TAB. 8 – Réponse des JCatools aux objectifs de SERMA Technologies.

Cette suite d'outils a été conçue, développée et testée principalement par Iban HATCHONDO et moi-même avec le concours des autres membres de l'équipe Systèmes et Objets Distribués.

7.1 JCat Emulator

JCat Emulator est une implantation complète des spécifications Java Card 2.1.1 (*i.e.* VM, JCRE et APIs, sauf les APIs cryptographiques). Cet outil inclut donc le pare-feu, l'atomicité, le mécanisme de transactions et le format standard des fichiers CAP, ce qui en fait un réel émulateur et non pas un simple simulateur. Il est destiné à tester et attaquer des implantations d'*applets* afin de détecter des problèmes dans leur comportement comme, par exemple, une mauvaise utilisation des services offerts par la plate-forme. Il

peut également permettre le débogage d'*applets* et la mise au point d'*applets* agressives grâce à la visualisation des différents états de l'environnement et de la machine virtuelle. Le développement de cet outil a contribué à établir une méthodologie d'évaluation des plates-formes Java Card en listant les points essentiels à vérifier pour assurer la sécurité. Il a aussi permis de répertorier des parties obscures des spécifications qu'il faudra donc vérifier au cas par cas dans les implantations des plates-formes que SERMA Technologies aura à évaluer.

7.1.1 Fonctionnalités

L'émulateur permet l'exécution pas à pas des *bytecodes* tout en autorisant la visualisation des mémoires, des objets, du buffer de transaction et de la pile d'appels (*frame stack*). Il est ainsi possible de suivre l'impact de chaque *bytecode* ou séquence de *bytecodes* sur le système. L'outil se propose aussi d'établir des statistiques sur les *bytecodes* exécutés. Il permet également de modifier toutes les valeurs des différentes zones représentant l'environnement d'exécution (*i.e.* mémoires, objets, buffer de transaction et pile d'appels) afin de simuler les effets d'une attaque physique. Les scripts de commandes envoyées à la carte acceptent aussi deux ordres particuliers pour indiquer à la carte qu'elle est retirée du lecteur et qu'elle y est à nouveau insérée afin de simuler une série de transactions entre le lecteur et la carte. Enfin, l'émulateur possède une fonction simulant l'arrachage de la Java Card pendant l'exécution des *bytecodes*. Il ne faudra pas confondre cette fonctionnalité avec la précédente qui, elle, intervenait seulement après le renvoi de la réponse de la carte au lecteur et donc normalement pas dans un état potentiellement dangereux pour l'application. On notera également la possibilité d'instancier au sein du même environnement plusieurs émulateurs Java Card pouvant s'exécuter en parallèle. Cette fonctionnalité facilite grandement la mise au point et le test d'attaques logicielles qui mettent souvent la carte dans un état bloqué. En effet, il est ainsi possible de modifier en conséquence son attaque ou son test puis, de la lancer dans un nouvel émulateur pour avoir un comportement différent tout en conservant l'exécution précédente dans un autre onglet afin de bien visualiser les différences.

7.1.2 Utilisations possibles

Il est possible d'utiliser JCat Emulator pour :

- Détecter des exceptions et en particulier les exceptions non récupérées au niveau des *applets*. En effet, si elles ne sont pas rattrapées au niveau de l'*applet* elles le seront par la boucle principale de la machine virtuelle.
- Faire des statistiques sur :
 - le nombre d'exceptions levées, leur type, etc. ;
 - l'appel aux méthodes sensibles afin de savoir sur lesquelles il faudrait se concentrer pour réaliser des attaques physiques ou physiques et logicielles couplées ;
 - les *bytecodes* afin de choisir ceux qu'il est préférable d'attaquer ;
 - la mémoire utilisée par une exécution (en terme d'octets utilisés, d'objets créés, dans le buffer de transaction, dans le buffer APDU, dans l'espace transient et persistant).

- Modifier des zones mémoires différentes en cours d'exécution pour, par exemple, simuler une attaque physique et tester le comportement susceptible d'en découler.
- Effectuer des tests d'arrachages systématiques lors d'appels sensibles afin de vérifier si la conception de l'*applet* utilise les mécanismes visant à contrer cette menace.

7.1.3 Limitations

Parmi les utilisations possibles que nous citons section 7.1.2, il nous faut toutefois reconnaître que certaines requièrent une très bonne expertise de Java Card pour être à même d'exploiter les informations fournies par l'émulateur. En effet, l'interface graphique de notre émulateur n'est pas suffisamment *user-friendly* pour permettre d'accéder directement à l'information (*e.g.* accéder au type d'un objet nécessite de savoir naviguer dans les fichiers **CAP** et **export** pour retrouver l'information voulue). Par ailleurs, une autre limitation de JCat Emulator est l'impossibilité de faire un test d'arrachage au milieu d'un *bytecode* ou d'une méthode native. Il est néanmoins possible de simuler des interruptions au milieu de certains *bytecodes* en modifiant des données de la mémoire manuellement. Enfin, en l'absence des APIs cryptographiques, il est impossible de faire fonctionner une *applet* réaliste sur l'émulateur puisque toutes utilisent un minimum de cryptographie. Néanmoins, comme nous le préciserons plus loin, des projets sont en cours avec des étudiants de Master pour réaliser ces APIs.

7.1.4 Aperçu de JCat Emulator

Afin de bien comprendre les diverses fonctionnalités de notre émulateur, la figure 39 montre un aperçu de JCat Emulator à la fin de l'exécution d'une *applet* en réponse à un script d'APDUs.

Si on s'intéresse à cette capture d'écran de façon globale, on peut distinguer plusieurs parties importantes :

- les menus et la barre d'outils pour l'instanciation d'une Java Card, le chargement des paquets, sauvegarde, etc. ;
- une zone de visualisation de paquets chargés sur la carte ;
- la pile d'appels, vide ici puisque l'exécution est terminée et que la carte est « hors-tension » ;
- l'état courant du buffer de transaction ;
- une zone permettant de voir tous les objets résidants en EEPROM et une autre permettant de voir tous les objets résidants en RAM ;
- une zone de visualisation de la trace d'exécution de l'*applet*, des commandes reçues et des réponses renvoyées.

Nous allons maintenant regarder plus en détails les fonctionnalités que nous avons décrites en les illustrant par des captures d'écrans.

La barre d'outils

C'est principalement par son intermédiaire qu'on accède aux différentes fonctionnalités de l'émulateur.

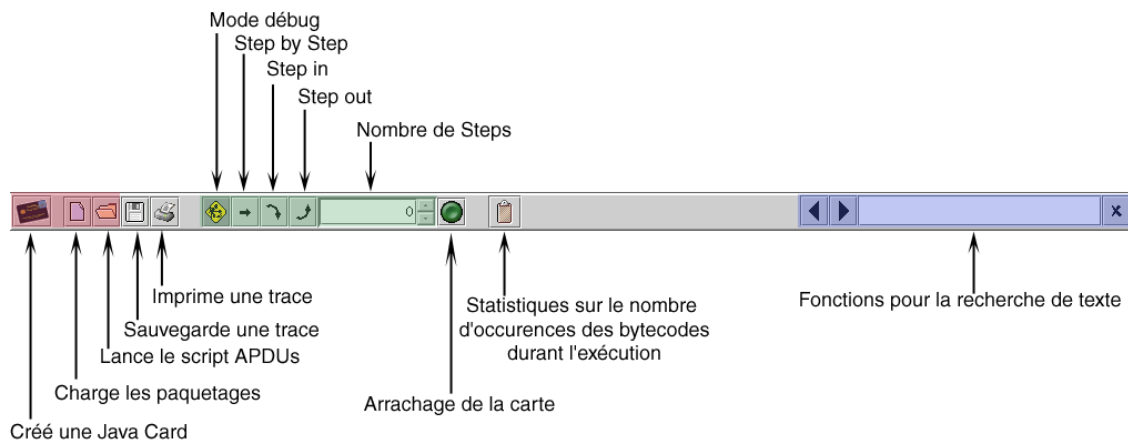


FIG. 40 – Les différentes fonctions de la barre d'outils.

Comme l'illustre la figure 40, elle permet :

- de créer de nouvelles instances de l'émulateur Java Card afin d'exécuter et de tester du code en parallèle ;
- de charger pour chaque instance d'émulateur Java Card un ensemble de paquetages tels que les APIs Java Card, les *applets* et surtout le *bootloader* (point d'entrée par lequel commencera l'exécution de la JCVM) ;
- de lancer le script APDU correspondant à l'installation des *applets* et à leur utilisation ;
- de sauvegarder et d'imprimer les traces d'exécution ;
- de lancer le mode de débogage puis d'exécuter le code pas à pas, de rentrer dans un bloc ou d'en sortir, d'exécuter un nombre fixé d'instructions, ce qui est très pratique lors des phases de débogage nécessaires à la mise au point d'attaques ;
- de faire une attaque physique d'arrachage de carte ;
- de consulter les statistiques sur le nombre de fois que chaque *bytecode* a été exécuté ;
- de faire une recherche textuelle dans les traces d'exécution.

Le mode débogage

Une fois ce mode activé, l'utilisateur a accès à plusieurs fonctionnalités qui étaient inactives dans le mode d'exécution classique. Il peut entre autres exécuter le *bytecode* pas

à pas, entrer dans un bloc de fonction ou au contraire en ressortir à l'aide des trois boutons dédiés de la barre d'outils. Il peut également demander à l'émulateur d'exécuter un certain nombre n de *bytecodes* puis de se mettre dans le mode pas à pas, juste avant l'exécution de ce $n^{ième}$ *bytecode* pour visualiser les différents états. On peut donc assimiler cette opération à la pause d'un point d'arrêt même si elle est en réalité beaucoup plus limitée que la vraie fonctionnalité de point d'arrêt. Afin d'illustrer le comportement en mode débogage de notre émulateur, la figure 41 montre l'état de l'environnement avant l'exécution du 431^{ième} pas, depuis l'initialisation de la carte.

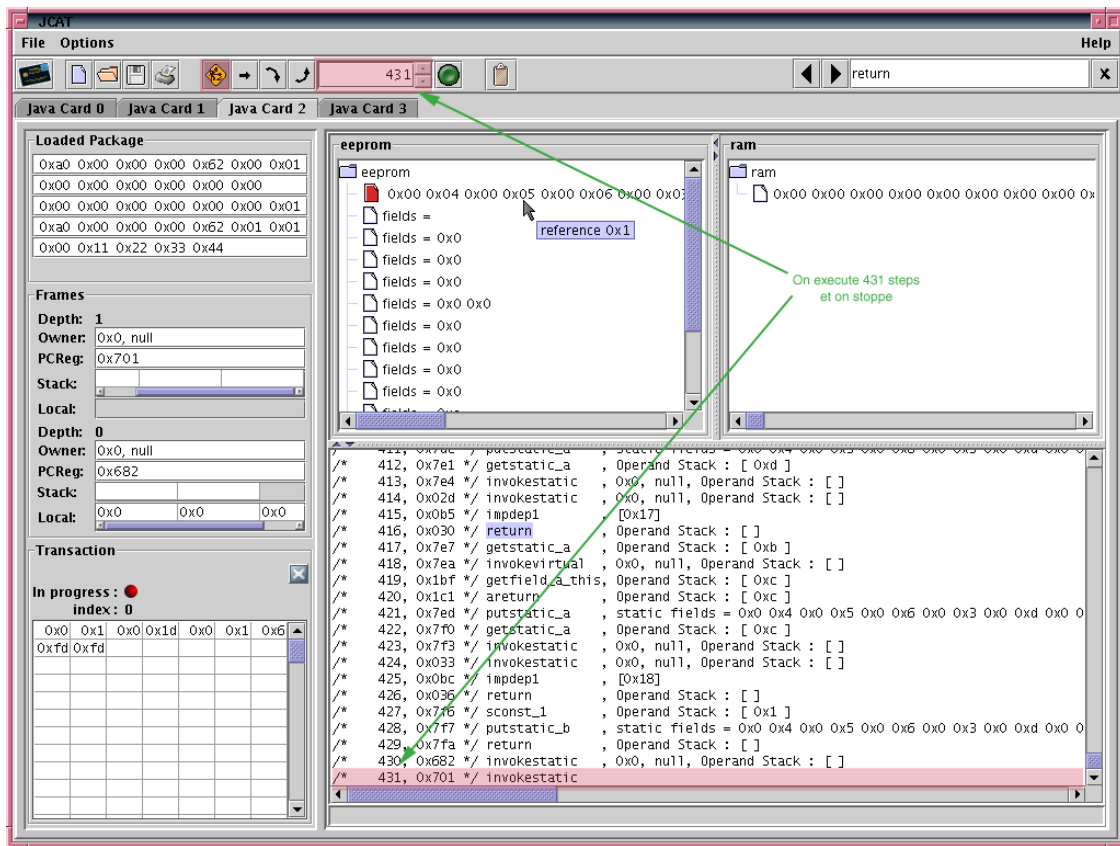


FIG. 41 – Vue de l'exécution en mode débogage.

Après l'exécution de 430 pas, l'émulateur est passé en mode pas à pas et il permet de visualiser et de modifier les différentes parties de l'environnement pour simuler les effets d'une attaque physique. La figure 41 permet de voir l'état de la pile d'appels, qu'aucune transaction n'est en cours, qu'un certain nombre d'objets ont été créés dans les mémoires, etc. Cette capture montre aussi qu'il y a quatre émulateurs Java Card qui fonctionnent en parallèle dans des onglets différents. Nous verrons plus loin comment nous pouvons visualiser l'état des différents objets et les modifier.

L'arrachage

Lorsque l'émulateur est en mode débogage, une fonctionnalité active est l'arrachage. L'arrachage correspond au retrait de la carte du lecteur pendant son exécution ou à une perte de son alimentation. C'est une attaque connue depuis fort longtemps et qui a pour objectif de placer la carte dans un état incohérent, état dont l'attaquant pourra tirer partie. Il se matérialise, dans notre émulateur, par l'appui sur le bouton vert de la barre d'outil qui procède à la simulation du retrait de la carte puis à sa réinsertion de façon automatique. Sur l'exemple, nous avons décidé d'effectuer un arrachage après avoir exécuté les 430 pas précédents. Or, comme nous pouvons le voir sur la figure 42, l'exécution de la machine virtuelle est repartie au début du *bootloader* qui se situe, pour les paquetages que nous avons chargés, à l'adresse `0x67a`. Elle considère donc l'instruction à cette adresse (*i.e.* `getstatic_b`) comme la 431^{ème} instruction qu'elle doit exécuter. On constate d'ailleurs que la machine virtuelle est bien repartie du début puisque la pile d'appels ne contient plus qu'un seul élément contrairement à ce que l'on peut voir sur la capture précédente. On peut aussi constater que nous sommes maintenant juste avant le 433^{ème} pas, parce que nous avons également effectué deux appuis sur le bouton de pas à pas.

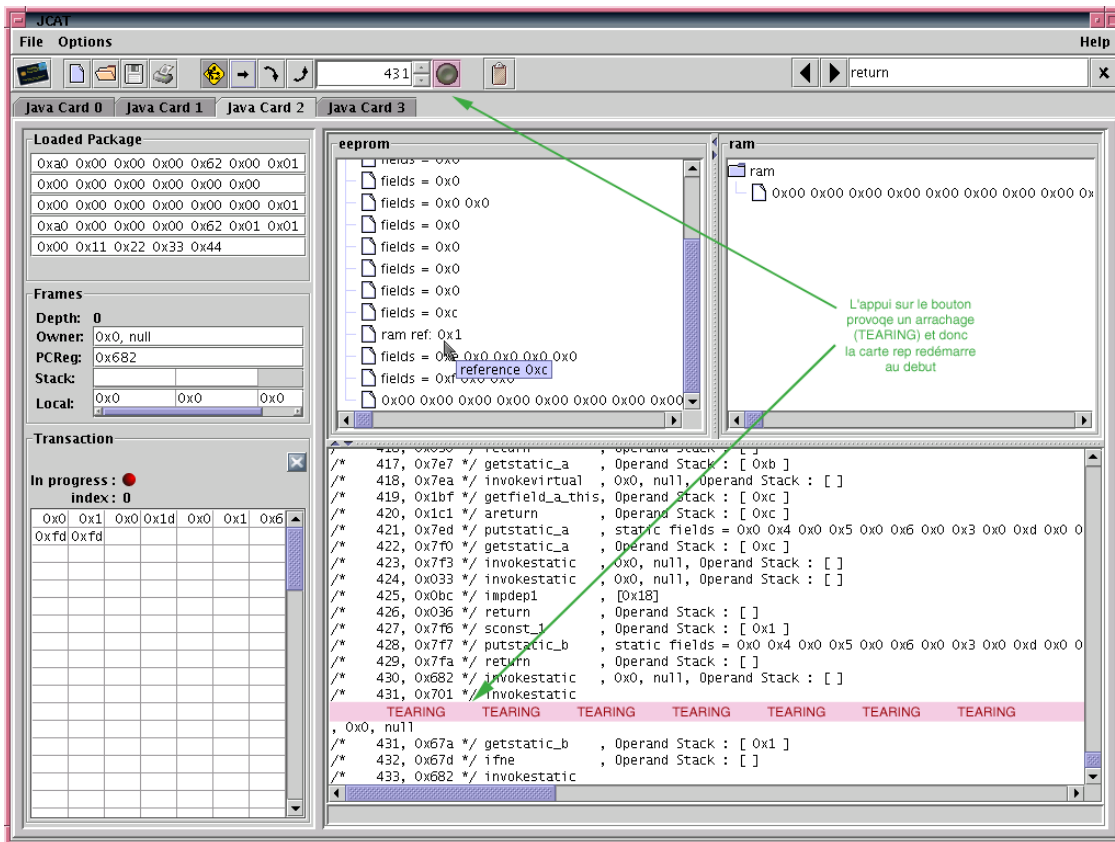


FIG. 42 – Vue de l'exécution après un arrachage virtuel.

Les objets

Comme nous l'avons vu sur les captures d'écran précédentes, les zones mémoires EEPROM et RAM contiennent un certain nombre d'objets dont on a une vue partielle. Ces objets sont rangés dans les mémoires par ordre de création (*i.e.* par référence croissante). Si on veut les observer, de façon plus complète, un double-clic sur l'objet choisi permet d'avoir d'avantage d'information comme l'illustre la figure 43.

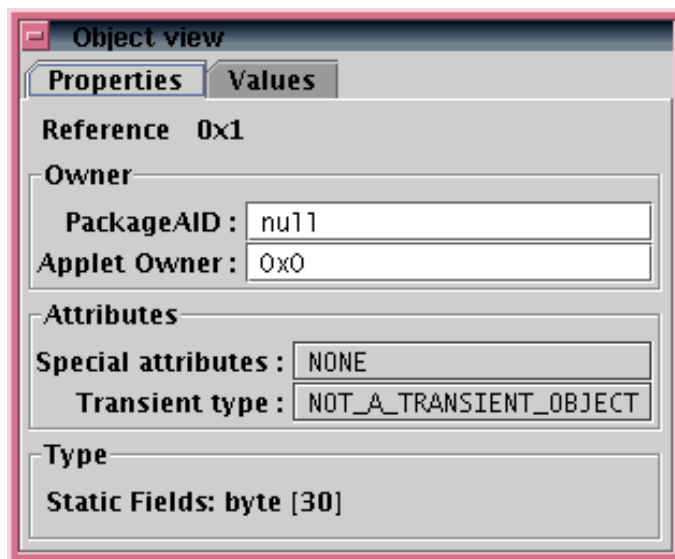


FIG. 43 – Vue d'un objet.

On constate sur cette vue deux onglets : propriétés et valeurs. L'onglet propriétés donne des informations générales sur cet objet comme la valeur de sa référence, le contexte de création et son propriétaire, ses attributs, et son type. Ici, par exemple, il s'agit :

- du premier objet créé car il a la référence `0x1` ;
- d'un objet créé par le JCRE car il a été créé dans son contexte (*i.e.* `null`) et il a pour propriétaire l'objet de référence `0x0` (que nous avons choisi dans notre développement comme se référant au JCRE) ;
- d'un objet persistant classique (*i.e.* pas un objet transient) ;
- d'un tableau statique de type `byte` de taille 30.

On notera qu'on peut éditer les informations relatives au contexte de création et au propriétaire de l'objet.

Si maintenant on clique sur l'onglet valeurs de cet objet, on peut voir les valeurs de ses champs et plus exactement dans notre cas, les valeurs des éléments du tableau (*cf.* FIG. 44). Comme on peut le constater sur la vue, les valeurs aussi sont ici éditables.

Si on essaye maintenant de visualiser des objets un petit peu plus exotiques comme les points d'entrée temporaires, on obtient une vue similaire à celle de la figure 45 et pour des objets points d'entrée permanents, on obtient une vue semblable à la figure 46. On peut d'ailleurs voir sur ces vues la nature peu conviviale de notre interface puisque les

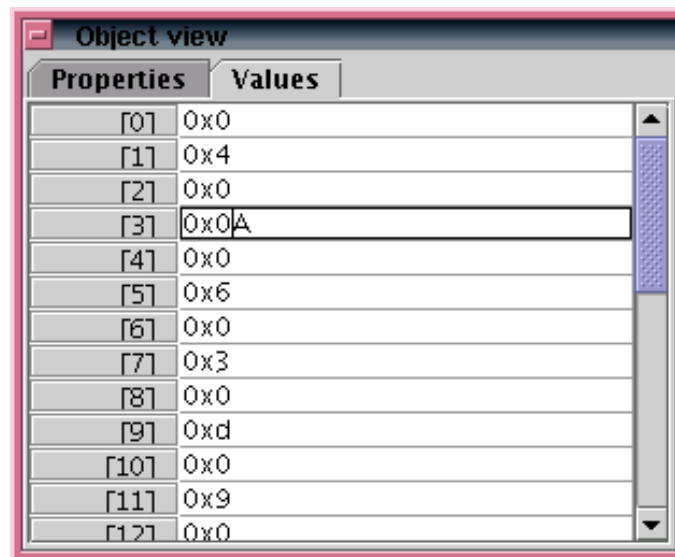


FIG. 44 – Édition des valeurs de l'objet.

informations qu'elle fournit sont assez abscons et nécessitent vraiment de savoir naviguer dans les fichiers `CAP` et `export` correspondants pour retrouver le type de l'objet.

Enfin, il sera aussi possible de visualiser des tableaux globaux transients de type `CLEAR_ON_RESET` ou `CLEAR_ON_DESELECT` comme on peut le voir sur la figure 47. On constate que cet objet n'a pas d'onglet de valeurs car il s'agit juste d'une structure en mémoire persistante qui référence un tableau en RAM. En l'occurrence ce tableau est le buffer APDU.

On notera bien que les objets points d'entrée permanents et temporaires ainsi que les tableaux globaux appartiennent tous au JCRE et ont été créés dans son contexte comme le précisent les spécifications.

Les statistiques

Cette fonctionnalité est activée et accessible seulement lorsque l'émulateur est arrêté *i.e.* lorsqu'il est en mode débogage ou lorsqu'il a fini son exécution. Elle permet de savoir combien de fois chaque *bytecode* a été exécuté et de pouvoir les trier soit par nombre d'exécution soit par ordre alphabétique (*cf.* FIG. 48).

L'impression

Là encore, cette fonctionnalité est activée seulement lorsque l'émulateur est arrêté. Elle permet d'imprimer les traces d'exécution ou tout au moins de générer un fichier PostScript avec une mise en page paramétrable qui permet de faciliter la lecture des *logs* (*cf.* FIG. 49).

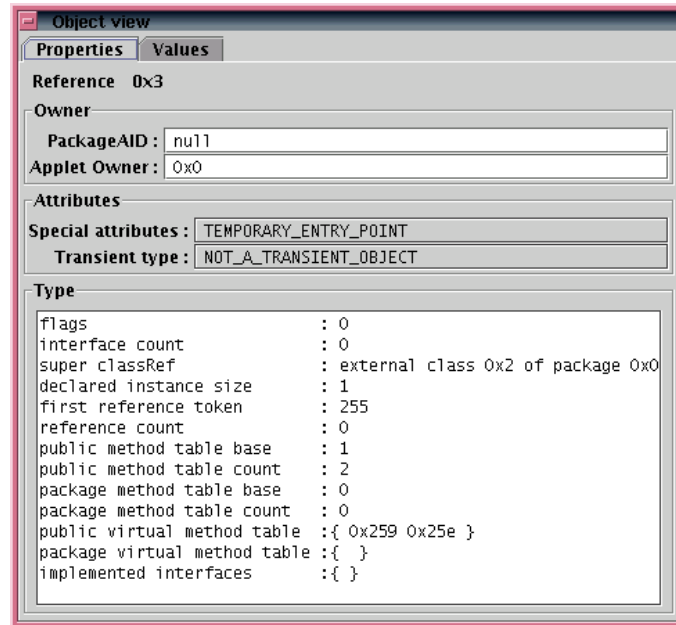


FIG. 45 – Vue d'un objet point d'entrée temporaire.

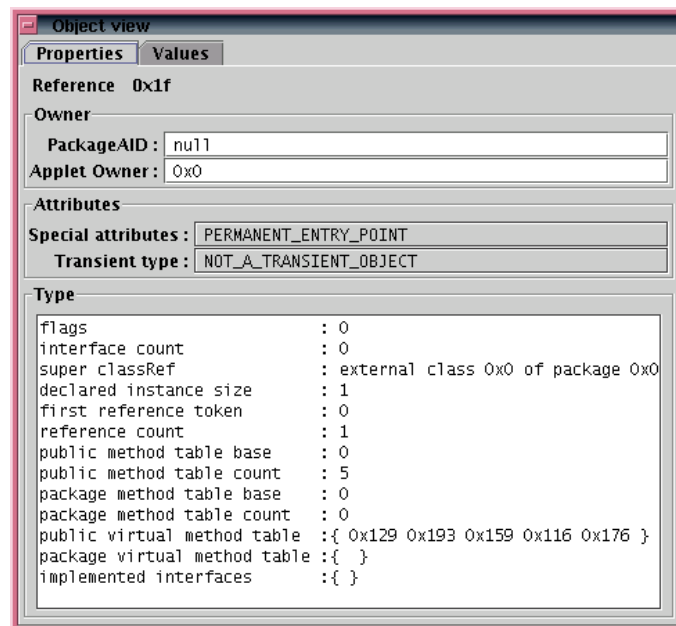


FIG. 46 – Vue d'un objet point d'entrée permanent.

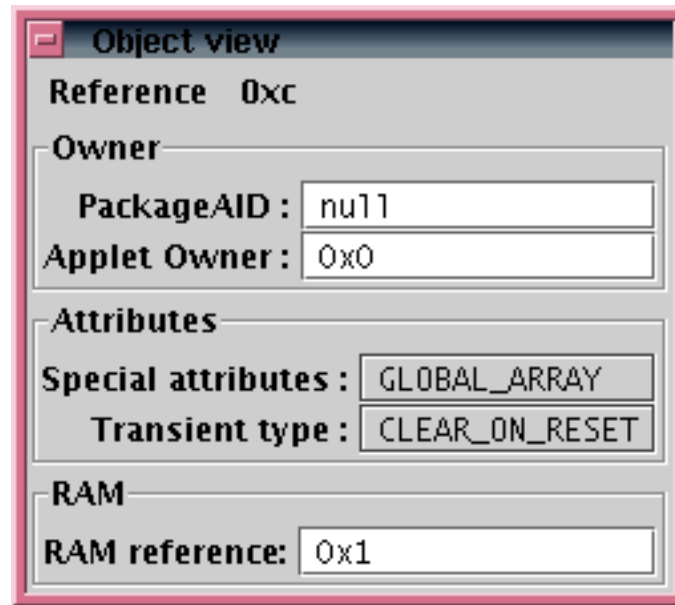


FIG. 47 – Vue d'un tableau global transient de type CLEAR_ON_RESET.

Statistics		
byte code name	count	
aload_0	168	
return	144	
invokestatic	126	
sload_1	80	
invokespecial	63	
sload	62	
sconst_0	61	
sload_3	48	
getstatic_a	48	
impdep1	47	
getfield_a_this	44	
sadd	43	
invokevirtual	43	
aload_2	36	
dup	32	
sload_2	31	
ifnonnull	29	
sconst_1	27	
sreturn	27	
baload	25	
arraylength	22	
putfield_a	18	
putfield_b	18	
getstatic_b	17	
aload_1	16	

FIG. 48 – Présentation des statistiques sur les *bytecodes*.

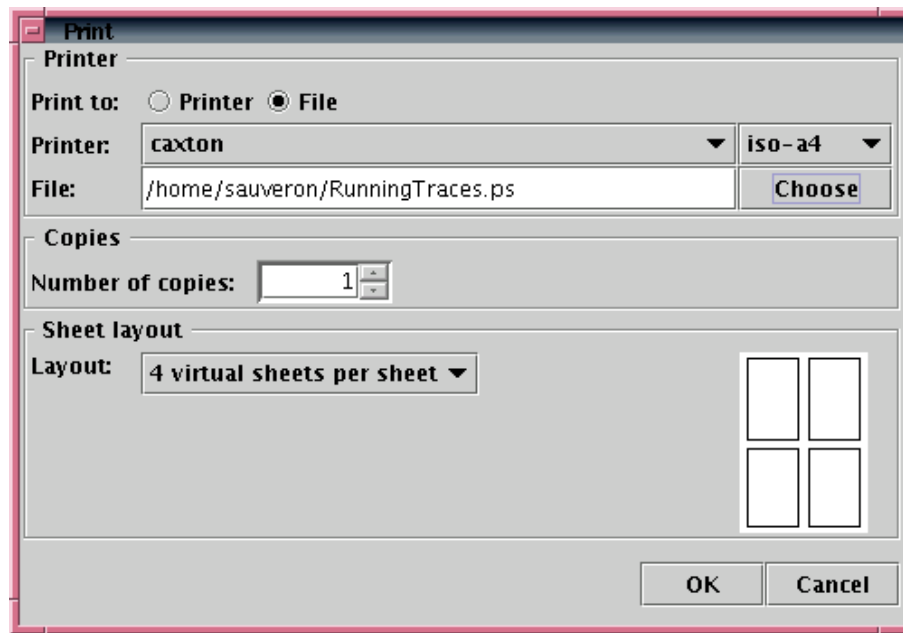


FIG. 49 – Fenêtre des paramètres d’impression.

7.1.5 Travaux en cours et améliorations futures

Le projet étant arrivé à son terme, l’évolution de l’émulateur s’est beaucoup ralentie depuis, mais quelques développements sont en cours et nous avons déjà prévu la liste des fonctionnalités qu’il serait intéressant de continuer à développer.

Les développements en cours ont pour but d’implanter les APIs cryptographiques spécifiées dans Java Card. Deux groupes d’étudiants ont commencé ce développement l’année dernière mais en l’état, il n’est pas fini et exige encore un gros travail pour être intégré dans l’émulateur. D’autre part, nous avons commencé à développer un pilote pcsc-lite (*cf.* chapitre 4) pour notre émulateur afin de pouvoir pleinement tirer partie de la fonctionnalité qui permet l’exécution parallèle de plusieurs Java Card. Ainsi, nous pourrions, par exemple tester des applications distribuées sur une grille de Java Cards faites d’instances de l’émulateur. Cette notion grille de Java Cards, sera présenté au chapitre 13.

Dans le futur, nous savons qu’il nous faudra améliorer la couche simulant la communication avec l’extérieur mais aussi rendre l’interface utilisateur plus conviviale. Nos réflexions nous conduisent à penser qu’il faudra aussi instrumenter la machine virtuelle afin d’avoir plus de statistiques. Nous souhaiterions également améliorer la simulation d’attaques physiques qui est aujourd’hui trop fastidieuse si l’on souhaite réaliser de grosses modifications.

Enfin, il faudra que nous prenions en compte les derniers développements des spécifications Java Card 2.2.1 (*i.e.* JCRMI et les canaux multiples) et que nous intégrions GlobalPlatform.

7.2 JCat View

Historiquement, JCat View est le tout premier outil développé dans le cadre de JCa-tools. Il permet la visualisation des formats de fichiers **CAP** de différents constructeurs. En effet, comme nous l'avons vu dans la section 2.7.3 certains constructeurs ont un format de fichier **CAP** différent et parfois incompatible avec les autres à cause d'interprétations différentes des spécifications Java Card. JCat View est compatible avec les fichiers **CAP** au format 2.1.1 et 2.2. La mise au point de cet outil a permis de découvrir de nombreux bugs dans le format de fichiers de certains constructeurs (*e.g.* IBM BlueZ Secure Systems, Oberthur Card Systems).

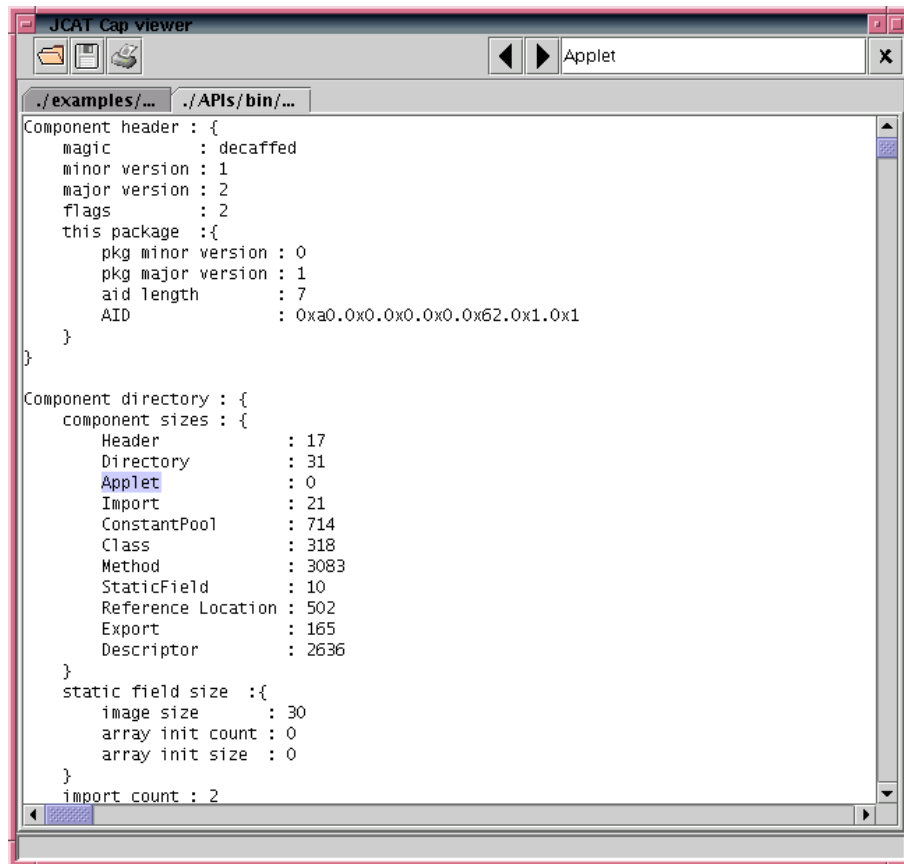


FIG. 50 – Vue de JCat View.

La possibilité de lire des fichiers **export** est envisagée dans les améliorations à apporter.

7.3 JCat Converter

JCat Converter est un outil en mode texte destiné à convertir les fichiers **CAP** d'un constructeur vers le format **CAP** d'un autre constructeur. Il utilise une syntaxe minimaliste et permet ainsi de construire une batterie de tests pour une cible précise puis de la convertir

pour les cartes utilisant un format de fichier CAP différent. Il serait en effet excessivement laborieux de devoir modifier les fichiers CAP pour chaque type de carte utilisant un format différent.

7.4 Les autres JCatools

Nous avons aussi créé deux outils d'aide à la modification de fichiers CAP permettant de faciliter la création de batteries de tests et d'attaques. Un exemple d'utilisation de ces outils sera illustré dans la section 11.6 qui expose des méthodes pour réaliser des tests d'identification de la signature de *bytecodes* en courant électrique et en électromagnétique. L'outil *parseComponent* permet d'extraire le composant Method d'un fichier CAP et de le transformer dans un mode lisible par un humain. À l'aide d'un éditeur de texte l'utilisateur peut alors en modifier le contenu. Enfin, il suffit de faire appel à *methodRewriter*, l'outil de réécriture du composant Method dans le fichier CAP. Ce dernier outil remplace l'ancien composant par le nouveau et modifie également les composants ReferenceLocation et Directory si besoin.

7.5 Les problèmes de licence

Lorsque nous avons voulu mettre la suite d'outils JCatools sous la licence libre GPL afin de faire bénéficier la communauté de nos travaux, nous avons demandé à Sun microsystems l'autorisation en vue de se prémunir de tout problème. Sachant qu'il existait nombre d'implantations de Java et même un simulateur Java Card sous licence libre, nous pensions obtenir ce droit sans trop de formalités. Mais après de longs mois de tractations et plusieurs rendez-vous téléphoniques nous sommes toujours dans l'impasse. L'enjeu économique que représente Java Card est tel que Sun microsystems refuse pour l'instant de laisser se développer ce genre d'initiative comme cela a été le cas avec Java. Par ailleurs, nous voulions également obtenir la certification de notre JCVM, *i.e.* avoir la possibilité de passer la suite de tests officiels de Sun, afin de vérifier que nous implantions correctement les fonctionnalités de Java Card, mais une fois encore Sun microsystems nous l'a refusée. La seule solution qui s'offre à nous est d'acquérir une licence ce qui représente un coût énorme.

Pour résumer, nous avons développé un produit implantant les spécifications Java Card et nous souhaiterions pouvoir le partager avec la communauté afin d'en améliorer la qualité et les fonctionnalités. Mais cela est impossible car le téléchargement des spécifications implique une acceptation de la licence qui nous contraint à prendre une licence de développeur par la phrase “*You acknowledge that any commercial or productive use of an implementation of the Specification requires separate and appropriate licensing agreements.*”. De plus, Sun microsystems ne souhaite pas voir des produits libres venir concurrencer ceux développés par ses licenciés. Si cette dernière position est compréhensible, il faut constater que le milieu institutionnel est pour beaucoup dans le succès de Java Card. Il serait légitime de pouvoir lui donner les moyens de développer ses propres produits dans un cadre de recherche.

Nous sommes donc toujours en discussion avec Sun microsystems dans l'espoir d'obtenir leur accord pour la libération de la suite JCatools.

7.6 L'architecture des JCatools

La figure 51 présente l'architecture de JCat Emulator. Elle est très simple et repose sur l'interpréteur de *bytecodes* qui fait évoluer l'environnement d'exécution (*e.g.* pile d'appels, tas – lors des créations et des mises à jours d'objet –, etc..) en fonction du code qu'il exécute. Comme il peut lui arriver de rencontrer des appels à du code natif, il est en relation avec une bibliothèque de méthodes natives. Nous décrirons d'ailleurs ce mécanisme plus loin dans cette section. Par ailleurs, JCat Emulator comprend aussi un chargeur de fichiers **CAP** pour pouvoir exécuter le code des *applets* et des bibliothèques.

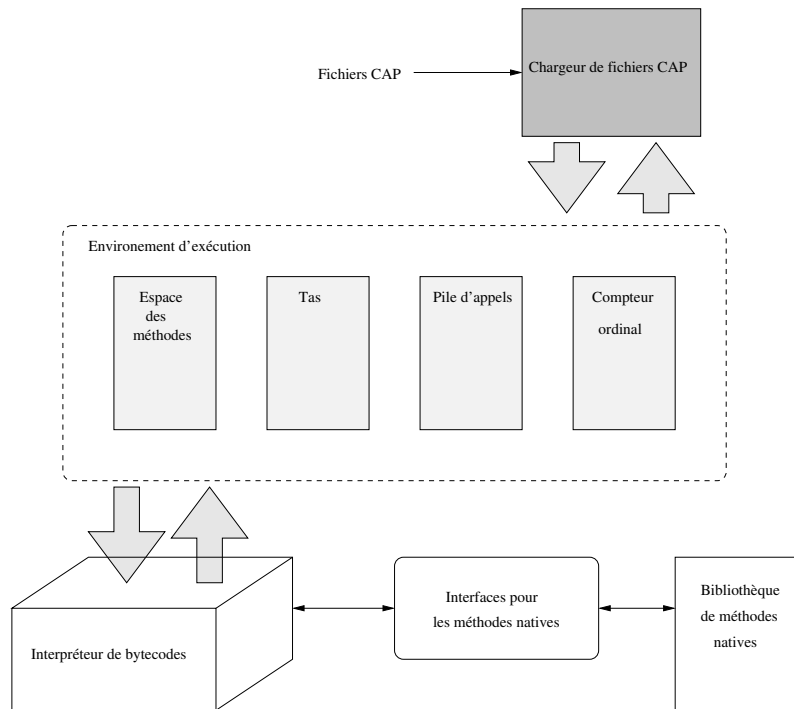


FIG. 51 – L'architecture de JCat Emulator.

Afin de visualiser les changements d'états de la mémoire dans l'interface graphique mais aussi de permettre sa modification par la machine virtuelle nous avons utilisé l'architecture représentée figure 52.

Par ailleurs, nous avons conçu JCat Emulator afin qu'il soit facilement extensible. Il peut, comme le montre la figure 53, supporter des *plug'ins* permettant de changer différentes représentations (*e.g.* la représentation des objets dans la mémoire, la mémoire physique, etc..) afin de s'adapter à des cartes de différents constructeurs. En effet, nous souhaitons que divers constructeurs puissent proposer des *plug'ins* pour notre outil pour émuler leur carte. Ainsi, la communauté disposera d'un outil unique possédant un cœur

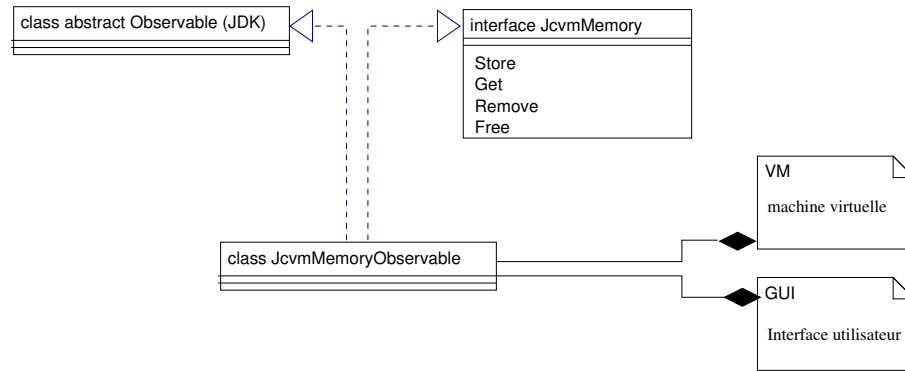


FIG. 52 – Architecture pour l’observation et la modification de la mémoire.

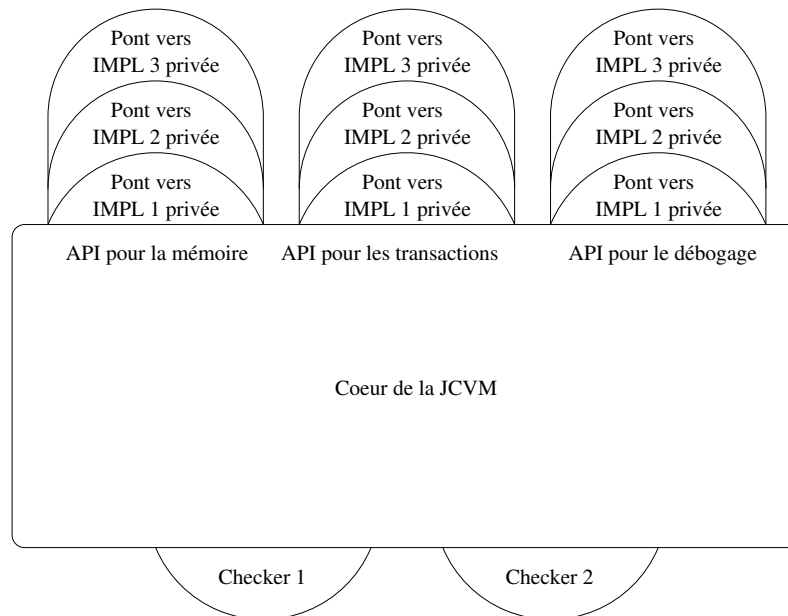


FIG. 53 – Extensibilité de l’architecture.

unique mais permettant d'émuler les cartes de plusieurs constructeurs, sans que ceux-ci n'est à dévoiler les secrets de leur JCVM. Nous souhaitons également permettre aux acteurs institutionnels de greffer leurs outils de vérification sur notre environnement. Le but est d'obtenir une plate-forme commune complète pour la recherche et le développement autour de la technologie Java Card.

7.6.1 L'extensibilité

Dans cette section, nous allons présenter quelques mécanismes que nous avons implantés dans notre JCVM afin de lui permettre d'être extensible et donc de supporter les évolutions futures des spécifications mais également de supporter de nouveaux outils.

L'emploi de *factory design patterns*

Nous avons utilisé plusieurs fois le *design pattern* de *factory* dans notre architecture. Ici, nous illustrons son utilisation dans le cadre du lecteur de fichier CAP : JCat View. La raison en est que ce lecteur doit être extensible, d'une part parce que les spécifications Java Card vont certainement évoluer et que de nouveaux composants vont être ajoutés, et d'autre part, parce qu'un utilisateur peut vouloir ajouter son propre composant et que le lecteur doit donc être capable d'avoir une méthode pour le lire.

Le *design pattern* de *factory* [115] fournit une classe abstraite `ComponentReader` pour créer des familles de lecteurs de composant sans spécifier leur classe réelle. De cette façon, un utilisateur peut créer un nouveau lecteur de composant ou bien en remplacer un déjà existant en donnant son implantation et en l'enregistrant dans le catalogue de la *factory* `ComponentReaderFactory` (cf. FIG. 54). Par exemple, en Java, le code du listing 7.1 permet qu'après la lecture de l'étiquette `FOO_TAG` dans le fichier CAP, la méthode `parse` du composant `FooComponentReader` soit appelée. En effet, le code du lecteur de fichiers CAP appellera la méthode `ComponentReaderFactory.getReader` en lui passant en paramètre l'étiquette lue, *i.e.* `FOO_TAG`. Cette méthode renverra alors le lecteur de composant approprié, *i.e.* ici `FooComponentReader`, et le lecteur de fichiers CAP pourra ainsi appeler sa méthode `parse` pour lire le composant rencontré.

Nous avons utilisé un mécanisme similaire pour l'implantation des différents types de tableaux et d'objets (cf. FIG. 55) mais aussi pour les transactions et les mémoires avec à chaque fois une interface spécifique décrivant le comportement respectif.

Les méthodes natives

Si les spécifications Java Card ne permettent pas au programmeur d'appeler des fonctions natives au niveau utilisateur, l'implantation de certaines APIs Java Card nécessite pourtant leur utilisation, *i.e.* le code ne peut pas être écrit en pure Java Card. Par exemple, les mécanismes de transactions sont très enchevêtrés avec les mécanismes de gestion de la mémoire. Ils sont donc impossibles à implanter avec le langage Java Card. Or, pour pouvoir faire de tels appels natifs, il a été nécessaire d'implanter certaines fonctionnalités supplémentaires au niveau JCVM et en particulier la fonctionnalité de FFI (Foreign Functions

Listing 7.1 – Le *factory design pattern* utilisé pour le lecteur de fichier CAP.

```

public class FooComponentReader
    extends ComponentReader {

    static final int FOO_TAG = 1234;
    static {

        ComponentReaderFactory.addReader(
            FOO_TAG,
            new FooComponentReader());
    }

    public void parse( ... ) {
        ...
    }
    ...
}

```

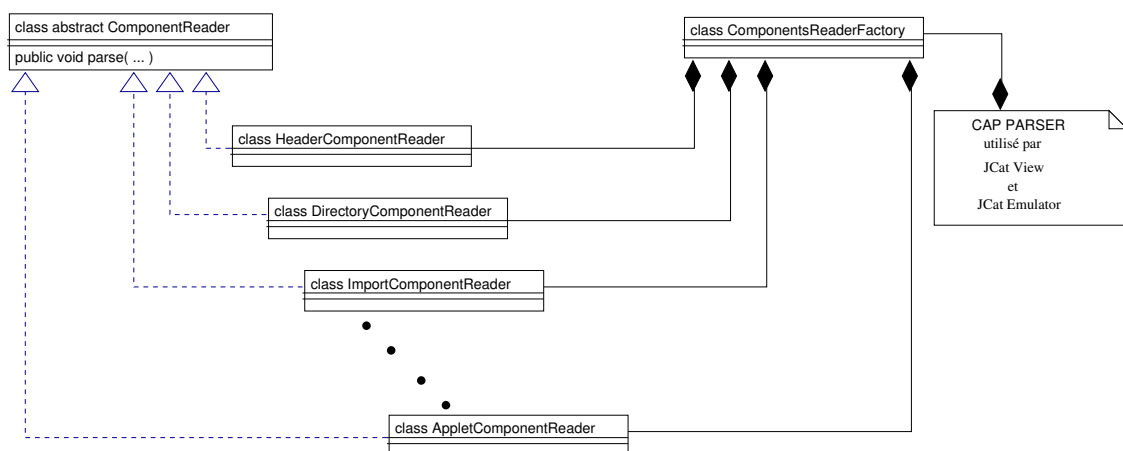


FIG. 54 – Factory de création des différents lecteurs de composant.

Listing 7.2 – Implantation JCtools de la classe OwnerPIN.

```

package javacard.framework;

public class OwnerPIN implements PIN {

    private byte tryLimit = (byte) 0;
    private byte[] triesRemaining;
    private byte[] triesRemainingTemp;

    private byte maxPINSize = (byte) 1;
    private byte[] pin;
    private byte pinSize;

    private boolean[] flag;

    public OwnerPIN(byte tryLimit, byte maxPINSize) throws PINException {

        if (! (tryLimit >= (byte) 1) && (maxPINSize >= (byte) 1))
            PINException.throwIt(PINException.ILLEGAL_VALUE);

        this.tryLimit = tryLimit;
        triesRemaining = new byte [1];
        this.maxPINSize = maxPINSize;
        pin = new byte[maxPINSize];
        pinSize = maxPINSize;

        // it's CLEAR_ON_RESET and not CLEAR_ON_DESELECT (8.1)
        flag = JCSysSystem.makeTransientBooleanArray((short) 1, JCSysSystem.CLEAR_ON_RESET);
    }

    public boolean check(byte[] pin, short offset, byte length)
        throws ArrayIndexOutOfBoundsException, NullPointerException {

        setValidatedFlag(false);
        if (triesRemaining[0] == 0)
            return false;
        // Problem if a transaction is in progress with : triesRemaining[0]-;
        if (triesRemainingTemp == null)
            triesRemainingTemp = JCSysSystem.makeTransientByteArray((short) 1, JCSysSystem.CLEAR_ON_RESET);
        triesRemainingTemp[0] = (byte) (triesRemaining[0] - 1);
        Util.arrayCopyNonAtomic(triesRemainingTemp, (short) 0, triesRemaining, (short) 0, (short) 1);

        if (Util.arrayCompare(this.pin, (short) 0, pin, offset, (short) length) != 0)
            return false;
        setValidatedFlag(true);
        triesRemaining[0] = tryLimit;
        return true;
    }

    public byte getTriesRemaining() {
        return triesRemaining[0];
    }

    public boolean isValidated() {
        return flag[0];
    }

    protected boolean getValidatedFlag() {
        return flag[0];
    }

    protected void setValidatedFlag(boolean value) {
        flag[0] = value;
    }

    public void reset() {
        if (isValidated())
            setValidatedFlag(false);
    }

    public void resetAndUnblock() {
        setValidatedFlag(false);
        triesRemaining[0] = tryLimit;
    }

    public void update(byte[] pin, short offset, byte length)
        throws PINException {

        if (length > maxPINSize)
            PINException.throwIt(PINException.ILLEGAL_VALUE);

        pinSize = length;
        Util.arrayCopy(pin, offset, this.pin, (short) 0, length);
        triesRemaining[0] = tryLimit;
    }
}

```

En effet, même si une transaction est en cours, les mises à jour des états internes tels que le compteur d'essais, le drapeau de validation ne doivent pas y participer lors de la comparaison du PIN. Imaginons que l'implantation de cette classe ne prenne pas en compte ces éléments. Si le programmeur d'une application officielle utilisait le code du listing 7.3, *i.e.* la vérification du PIN à l'intérieur d'une transaction, la sécurité de son application serait en danger. Comme on peut le constater dans le code, si un attaquant présente un mauvais PIN à l'*applet* officielle (*i.e.* il y a une décrémentation du compteur d'essais à chaque mauvaise présentation lors de l'exécution de la méthode de vérification `check`), puis retire la carte juste avant la validation de la transaction `JCSYSTEM.commitTransaction`, à la prochaine mise sous-tension, le JCRE annulera les transactions précédemment en cours et remettra donc le compteur d'essais à sa valeur initiale. Ainsi, l'attaquant possède un nombre illimité d'essais. En revanche, il ignore si le PIN présenté était bon ou mauvais puisqu'il n'a pas le statut final normalement retourné par la carte en fin d'exécution. Il doit donc coupler cette technique avec une méthode d'analyse des canaux cachés (*e.g.* la *timing-attack* que nous présenterons section 9.1.4) pour découvrir le PIN. Globalement, une implantation de la carte telle que nous l'avons décrite ici est facilement attaquable pour des experts en sécurité.

Listing 7.3 – Code d'une application officielle.

```
public class OfficialApplet extends Applet {

    private OwnerPIN pin;

    // Initialisation du pin + diverses méthodes
    ...

    void process(APDU apdu) {

        ...
        // Déclenchement d'une transaction
        JCSYSTEM.beginTransaction();
        ...

        // Comparaison du pin avec la valeur présentée pinToCompare
        pin.check(pinToCompare, (short)0, (short)pinToCompare.length);

        ...
        // Validation de la transaction
        JCSYSTEM.commitTransaction();

        ...
    }
}
```

Inversement, une implantation telle que la notre suit les exigences des spécifications et n'est pas attaquable par une méthode d'arrachage même si le programmeur de l'*applet* officielle a utilisé la méthode de vérification du PIN `check` au sein d'une transaction. En effet, pour programmer le processus de vérification en Java Card, l'astuce consiste à utiliser les tableaux transients qui eux ne participent pas aux transactions. Dans notre implantation, le compteur d'essais `triesRemaining` est un tableau persistant puisqu'il doit conserver la valeur du nombre d'essais restants au-delà des sessions avec le lecteur. Mais

nous utilisons pour faire la décrémentation du compteur un tableau transient intermédiaire `triesRemainingTemp`. Ce dernier recevra donc la valeur du compteur d'essais et fera la décrémentation (*i.e.* `triesRemainingTemp[0] = (byte) (triesRemaining[0] - 1)` ;). Pour la mise à jour du compteur d'essais réel `triesRemaining`, nous utilisons la méthode `Util.arrayCopyNonAtomic(...)` qui permet au sein d'une transaction de faire une copie sans que les tableaux concernés ne participent à la transaction (*i.e.* leur contenu n'est pas sauvegardé dans le buffer de transaction). Ainsi, à la fin de cette opération le compteur d'essais `triesRemaining` a bien été décrémenté sans qu'à aucun moment l'ancienne valeur ne soit stockée dans le buffer de transaction. On va donc maintenant pouvoir faire la comparaison des PINs.

Nous procédons à une pré-décrémentation du compteur d'essais du PIN, car il existe une attaque d'arrachage similaire à celle que nous avons exposée ci-dessus si le compteur est décrémenté après la vérification. En effet, entre la vérification et la décrémentation, l'attaquant pourrait retirer la carte et disposer une fois encore d'un nombre d'essais illimité. Ainsi, dans notre implantation, nous pré-décrémentons le compteur d'essais puis nous l'incrémentons à la valeur initiale si la vérification des PINs est correcte. Si elle est mauvaise, le compteur d'essais reste à la valeur décrémentée.

Par ailleurs, on notera que dans les modélisations des APIs – que cela soit en JML [63] ou ESC/Java [14] –, la nature transiente ou persistante des objets n'est pas prise en compte. Par conséquent, ces modèles ne reflètent pas l'implantation réelle qui doit être faite de certaines classes comme `OwnerPIN`.

Enfin, pour éviter les problèmes dus aux attaques via l'observation du temps d'exécution, la comparaison du PIN doit s'effectuer en temps constant. Ainsi, la méthode `Util.arrayCompare` qui retourne le résultat de la comparaison de deux tableaux, doit réaliser une comparaison totale des deux tableaux, même si elle a détecté que les deux tableaux étaient différents.

Chapitre 8

Résultats

Nous présentons dans ce chapitre un premier bilan de nos travaux. En section 8.1, nous décrivons les résultats issus du développement de la suite JCatools. En section 8.2, nous présentons des propositions relatives à la gestion de la mémoire qui permettent de lever les ambiguïtés que nous avons décelées dans les spécifications.

8.1 Bilan de la suite JCatools

Le bilan du projet *Sécurité Java Card* est plus que positif pour les deux partenaires puisqu'il a permis de développer JCatools. Grâce à cette suite d'outils, SERMA Technologies a pu montrer qu'elle avait acquis un haut degré d'expertise des technologies Java et Java Card. Le CESTI s'est ainsi vu confirmer l'agrément à évaluer des produits basés sur Java Card à de hauts niveaux.

L'équipe *Systèmes et Objets Distribués* du LaBRI a pu, pour sa part, démontrer son savoir-faire dans le domaine des technologies Java et se confronter à de réelles problématiques industrielles. Grâce à une démarche scientifique, nous avons proposé des solutions aux difficultés rencontrées, comme par exemple, le concept nouveau de *pré-persistance* (exposé dans la section suivante) pour résoudre les problèmes dus aux zones d'ombres des spécifications Java Card.

Par ailleurs, suite au développement de JCatools, le LaBRI et SERMA Technologies ont publié deux articles :

- « *JCAT : An environment for attack and test on Java Card* » [88] ;
- « *Modèles de mémoire en Java Card. Introduction du concept de pré-persistance.* » [90].

Conformément au cahier des charges défini initialement, les JCatools ont permis de doter SERMA Technologies des compétences et des moyens tant en termes de logiciels, que de méthodologies pour mener à bien ses évaluations sécuritaires. En effet, la suite d'outils a permis de définir une méthodologie d'évaluation pour les applets et les plates-formes Java Card. Nous avons également développé diverses batteries de tests d'attaques logicielles. Mais surtout, nous sommes restés proches de la réalité (*i.e.* de l'aspect matériel d'une puce – sûrement grâce au passé d'expertise matérielle de SERMA Technologies)

en permettant, dans l'émulateur, la simulation d'attaques matérielles et logicielles, voire même d'attaques couplées. C'est d'ailleurs cette même volonté forte de rester proches du matériel qui se traduit par les résultats présentés dans le chapitre 11 relatif à la mise en évidence de nouvelles vulnérabilités des cartes à puce multi-applicatives.

Dans le futur, les évolutions de JCatools permettront peut-être à SERMA Technologies :

- d'améliorer la qualité de ses évaluations par l'utilisation de preuves formelles ;
- de soumettre une méthodologie à la DCSSI et ainsi de renforcer encore plus le schéma français de certification.

8.2 Interprétations des spécifications

Comme nous l'avons déjà vu en section 2.5.2, les spécifications Java Card ne précisent pas la localisation des structures de données dans les mémoires physiques. En effet, Sun microsystems a fait le choix de laisser toute liberté au développeur, permettant ainsi d'adapter la technologie Java Card à plusieurs types de périphériques (*i.e.* cartes à puce, iButtons [30], etc.). Ceci se traduit par la présence de zones d'ombre, voire mêmes de contradictions, ce qui laisse une large place à l'interprétation. Cela risque d'aboutir à des implantations qui n'ont pas du tout le même niveau de sécurité, ni les mêmes fonctionnalités.

Dans la suite plusieurs interprétations possibles de cette partie des spécifications vont être détaillées et comparées. Les périphériques considérés seront naturellement les cartes à puce, cibles privilégiées de la technologie Java Card.

8.2.1 Le problème de la définition du tas

Lorsqu'on veut étudier la description de la mémoire dans les spécifications Java Card, la première difficulté consiste à trouver des définitions cohérentes et homogènes des différents termes tout au long des documents. Les spécifications définissent le tas tantôt comme une zone de mémoire libre utilisable par le programme, tantôt comme un ensemble d'objets stockés dans une mémoire persistante. Pour éviter toute confusion, nous emploierons la définition du tas largement répandue qui consiste à considérer celui-ci comme la zone mémoire qui comprend l'ensemble des objets alloués et de la mémoire libre. Ainsi défini, le tas aura une taille fixe tout au long de la vie de la Java Card et seules les tailles totales de l'espace alloué aux objets et de la mémoire libre évolueront proportionnellement et en sens inverse dans le temps.

Bien que les spécifications ne fassent pas état de la notion de tas statique, nous appellerons l'espace contenant l'ensemble des champs statiques des paquetages présents sur la carte. Évidemment, il aurait été possible de considérer le tas statique et le tas comme une seule et même entité, mais les travaux de formalisation de la JCVM [74] introduisaient cette notion que nous avons souhaité conserver car elle nous paraît apporter une certaine cohérence. Cet espace contient donc des structures de données Java Card (*i.e.* de type primitif et référence) mais en aucun cas les objets Java Card désignés par les champs de type référence. Par essence les données et les structures du tas statique doivent exister au travers des sessions avec le lecteur. Aussi le tas statique doit être localisé dans une mémoire

persistante, et donc en EEPROM dans le cas des cartes à puce.

8.2.2 Le problème du contenu du tas

Le tas et les objets persistants

Tout naturellement, les objets persistants sont localisés dans le tas persistant. Comme nous l'avons indiqué ci-dessus, cette partie du tas se situe dans une mémoire persistante telle que l'EEPROM.

Le tas et les objets transients

Les spécifications sont très précises concernant la localisation des objets transients. Les informations sur leur structure sont localisées en mémoire persistante, donc, dans l'espace persistant, alors que les valeurs de leurs champs sont localisées dans l'espace transient. Les spécifications déconseillent fortement de localiser l'espace transient dans une mémoire persistante. Il devrait donc être localisé pour les cartes à puce en RAM. L'écriture dans les champs des objets transients ne pose pas de problème de performance car le cycle d'écriture de la RAM est beaucoup plus court que celui de l'EEPROM. Les propriétés de ces objets (*i.e.* rapidité et sécurité) découlent des propriétés de la RAM.

La notion de pré-persistence

La première imprécision des spécifications Java Card concerne la localisation des structures des objets Java Card lors de leur création. En effet, les spécifications du JCRE (*cf.* la section 2 du document [170]) énoncent clairement que le développeur *rendra* les objets persistants lorsque la méthode `Applet.register` est invoquée ou lorsqu'une référence à l'objet est stockée dans un champ d'un objet persistant ou celui d'une classe. Par ailleurs, ces spécifications précisent aussi qu'un objet est créé depuis le tas lors de :

- l'exécution d'un *bytecode* `new`, `anewarray` et `newarray` ;
- l'invocation d'une méthode de l'API `makeTransientBooleanArray`, `makeTransientByteArray`, `makeTransientShortArray` et `makeTransientObjectArray`.

Aussi quelle est la nature des objets entre le moment de leur création et celui de leur affectation à un champ d'un objet persistant ?

Pour aborder ce problème, nous avons choisi de désigner par *espace pré-persistant* l'espace mémoire où sont créés les objets après l'exécution d'un *bytecode* `new`, `anewarray` ou `newarray`. Le nom de cet espace n'implique cependant pas que les objets qu'il contient deviendront forcément persistants. Toutefois dans le glossaire des spécifications 2.2.1 du JCRE se trouve une définition des objets persistants qui décrit les objets comme étant persistants par défaut. Or cette définition va à l'encontre de l'exigence énoncée par la spécification. Quelle partie des spécifications faut-il donc implanter ? Le développeur de plate-forme Java Card doit donc choisir entre deux implantations quant au contenu du tas (*cf.* FIG. 56) :

- une, allouant directement les objets créés dans l’espace persistant et se conformant ainsi au glossaire ;
- l’autre, possédant la notion d’espace pré-persistant introduit ci-dessus et donc appliquant les spécifications.

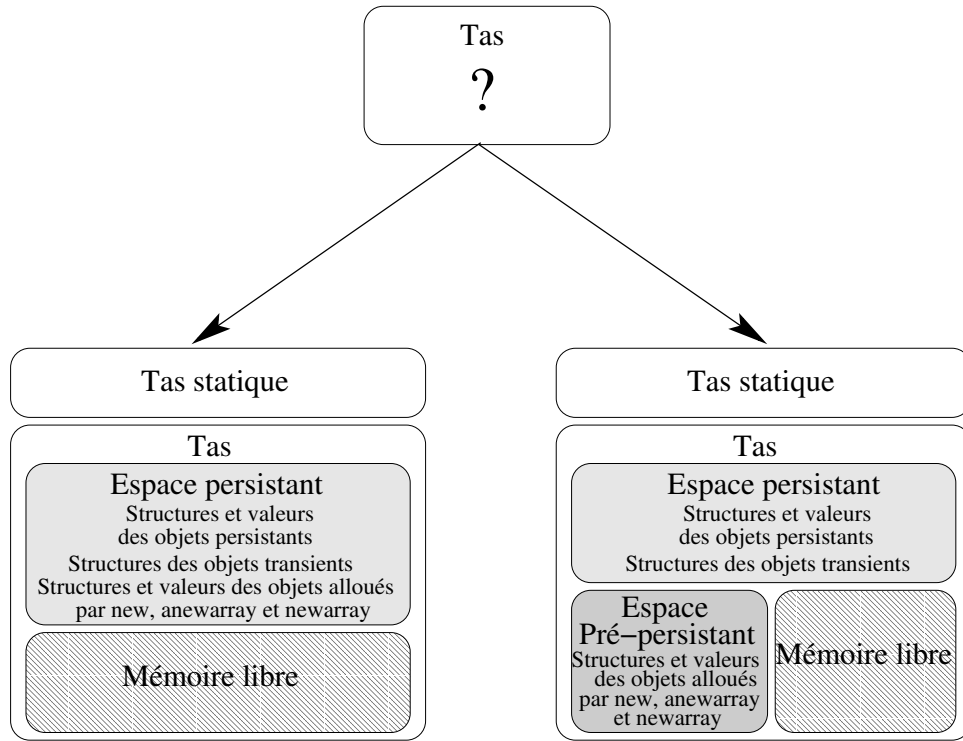


FIG. 56 – Le problème du contenu du tas.

Quels sont les avantages et les inconvénients de ces deux solutions ? La solution sans espace pré-persistant permet une implantation aisée puisqu’il suffit d’allouer simplement tous les objets dans l’espace persistant sur l’exécution des *bytecodes* de création ou des méthodes des APIs.

La solution avec l’espace pré-persistant offre aussi plusieurs avantages :

- la rapidité lors des opérations d’écriture, puisque seuls les champs des objets persistants sont touchés par l’atomicité et les transactions ;
- une implantation facile d’un mécanisme de ramasse-miettes dans l’espace pré-persistant en libérant cet espace à la demande ou à chaque redémarrage du péri-phérique ;
- la possibilité de l’utiliser pour créer un Environnement Transient [179], qui offre au langage un plus grand pouvoir d’expression, et pour résoudre des problèmes liés à la création d’objets dans une transaction annulée [178].

Son inconvénient principal réside dans l’implantation des mécanismes de copie des objets de l’espace pré-persistant dans l’espace persistant. En effet, ces deux espaces peuvent être localisés dans des mémoires de types différents comme nous le verrons plus loin. De

plus, la recopie doit être réalisée au sein d'une transaction et nécessite la mise en œuvre d'algorithmes de parcours d'objets afin de déterminer les objets à rendre persistants. Par exemple, dans le cas où des objets pré-persistants se référencent entre eux, tous devraient devenir persistants. L'opération d'affectation à un champ d'un objet persistant ou à un champ de classe serait donc contaminante.

L'imprécision relevée plus haut implique également qu'un objet transient dont la référence serait affectée à un champ d'un objet persistant ou à celui d'une classe deviendrait persistant. Seuls les objets transients référencés par des variables locales, par la pile d'opérandes ou par des objets non persistants pourraient donc garder leur transience. Dans les autres cas de référencement, ces objets transients deviendraient des objets persistants et ils verraient donc les valeurs de leurs champs migrer de l'espace transient vers l'espace persistant. Un autre avantage de la solution utilisant l'espace pré-persistant est donc que les objets de cet espace pourraient eux aussi référencer un objet transient sans en changer la nature.

Pour résumer, un objet pré-persistant :

- serait créé lors de l'exécution d'un *bytecode* `new`, `anewarray`, `newarray` ou de l'invocation d'une méthode de l'API `makeTransientBooleanArray`, `makeTransientByteArray`, `makeTransientShortArray`, `makeTransientObjectArray`;
- ne participerait pas aux mécanismes d'atomicité et de transactions;
- pourrait référencer des objets persistants, transients et pré-persistants;
- pourrait être référencé par des objets transients et pré-persistants, mais pas par des objets persistants, sans quoi il risque de devenir persistant lui-même.

8.2.3 Le problème de la localisation du tas

La seconde imprécision des spécifications concerne toujours le tas. En effet, les versions antérieures aux spécifications 2.2 ne donnaient aucune définition de la notion de tas et, aujourd'hui encore, malgré les précisions qui ont été ajoutées, la liberté est laissée au développeur du JCRE de choisir l'étendue et la localisation du tas.

Le tas doit-il être localisé (*cf.* FIG. 57) seulement en EEPROM, ou bien est-il localisé en EEPROM et en RAM ? La seule certitude sur le tas est qu'une partie se situe en EEPROM puisqu'il doit contenir les objets persistants (*i.e.* structures et valeurs) et les structures des objets transients.

8.2.4 Pré-persistence et organisation du tas

De ces propriétés sur le tas peuvent être déduites des propriétés sur la nature des objets au moment de leur création. Il faut donc étudier le problème global qui résulte de la combinaison des deux imprécisions des spécifications (*cf.* FIG. 58). Nous avons dégagé quatre solutions d'implantation différentes du tas avec des avantages et des inconvénients pour chacune.

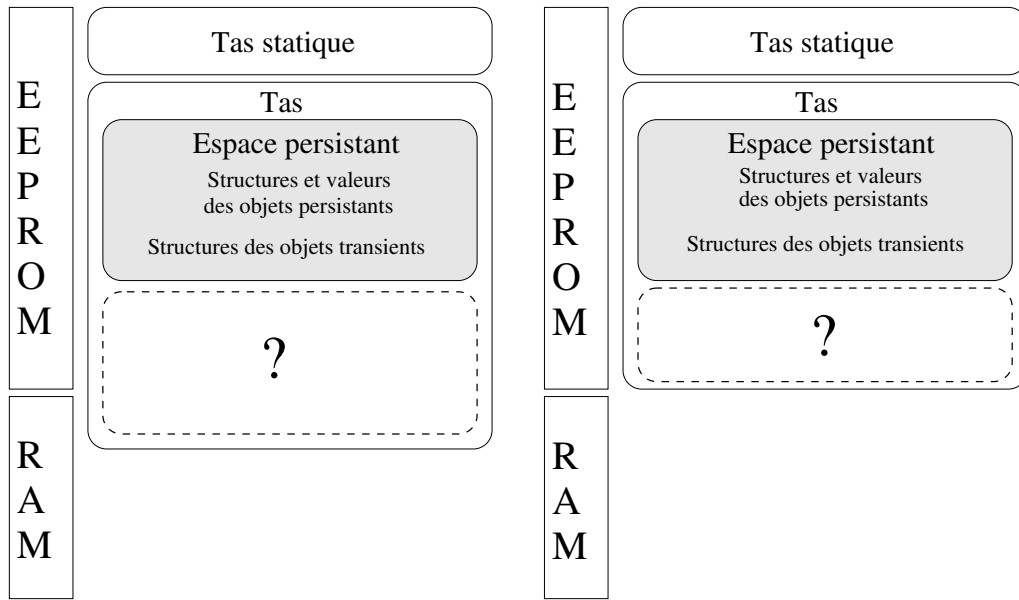


FIG. 57 – Le problème de l’étendue du tas.

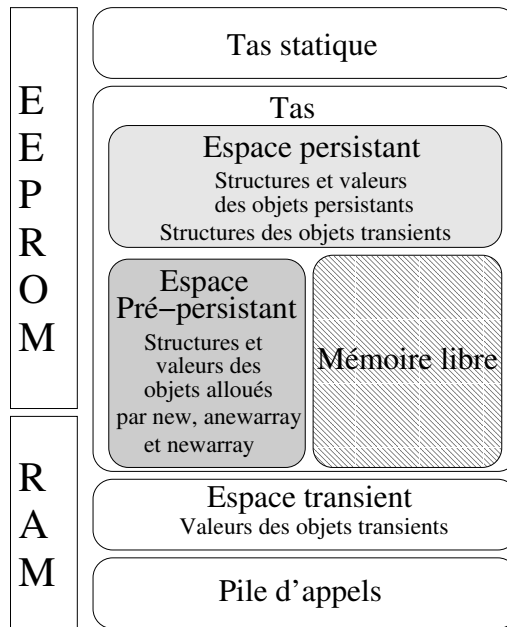


FIG. 58 – Le problème global de l’étendue et du contenu du tas.

La solution n°1 (cf. FIG. 59) est celle retenue dans la plupart des implantations Java Card. Son principal avantage est sa facilité d’implantation au niveau de la gestion de l’allocation des objets. Ses inconvénients sont :

- une sémantique différente de celle du langage Java pour l’allocation des objets puisque les objets sont alloués dans un tas en mémoire persistante (*i.e.* EEPROM) et non en

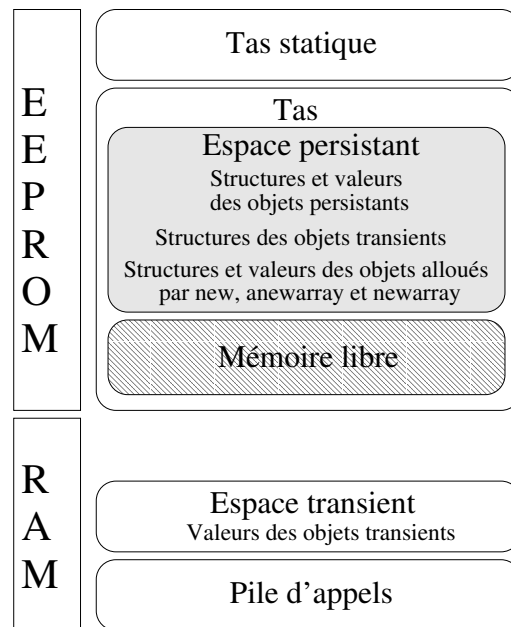


FIG. 59 – La solution n°1 : la solution sans la pré-persistence.

mémoire volatile comme en Java (*i.e.* RAM du PC) ;

- une certaine lenteur à l'exécution causée par les mécanismes d'atomicité et de transaction qui sont plus lents en EEPROM qu'en RAM.

En revanche, dans cette solution on peut vraiment se demander l'intérêt des tableaux transients d'objets. Le listing 8.1 illustre la création d'un tableau transient d'objets de type `CLEAR_ON_RESET`, `tab`, comprenant deux éléments. La méthode `foo` de ce même listing affecte à chaque élément de ce tableau la référence d'un objet nouvellement créé en EEPROM – de type `Object1` pour l'un et `Object2` pour l'autre. Dans cette solution, si la carte est retirée du lecteur après l'appel à la méthode `foo` le contenu du tableau transient `tab` situé en RAM (*i.e.* les références sur les objets) sera effacé – et au prochain démarrage, pour plus de sécurité, le contenu de ce tableau sera remis à ses valeurs par défaut, *i.e.* `null`. En revanche, les deux objets précédemment créés, respectivement de type `Object1` et `Object2`, sont maintenant inaccessibles en EEPROM. Aussi, on voit mal, dans cette solution, une utilisation pertinente des tableaux transients d'objets. Cela est d'autant plus vrai que l'on a tendance à croire qu'un tableau transient d'objets est un tableau d'objets temporaires. Il n'en est rien et il faut bien prendre conscience que ce n'est qu'un tableau de références dont les valeurs sont réinitialisées lors de l'occurrence de l'événement auquel elles sont associées (*e.g.* `CLEAR_ON_RESET`, etc.).

La solution n°2 (*cf.* FIG. 60) est un peu plus difficile à implanter en raison de la gestion du passage des objets de l'espace pré-persistant à l'espace persistant lors des opérations définies dans les spécifications du JCRE (*cf.* la section 2 du document [170]). En revanche par rapport à la solution n°1 le gain en rapidité est appréciable en raison de l'absence des mécanismes d'atomicité et de transaction pour l'espace pré-persistant. L'implantation d'un mécanisme de ramasse-miettes partiel peut être assez facile. Il suffit à chaque redémarrage

Listing 8.1 – Tableau transient d’objets.

```

public class A {
    private Object[] tab;

    public A() {
        tab = JCSystem.makeTransientObjectArray(2, JCSystem.CLEAR_ON_RESET);
    }

    public void foo() {
        tab[0] = new Object1();
        tab[1] = new Object2();
    }
}

```

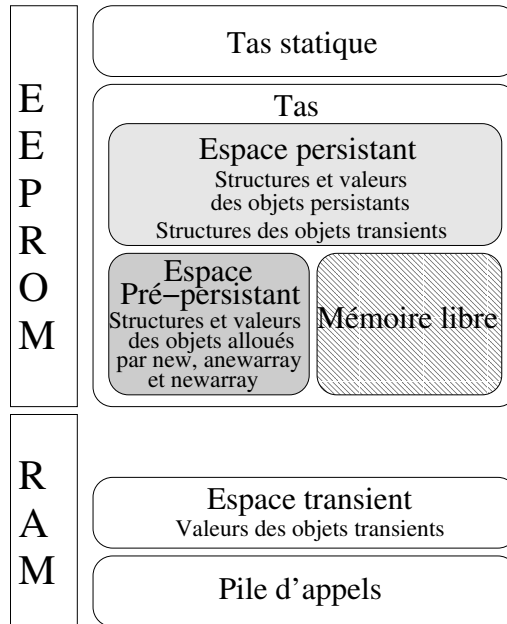


FIG. 60 – La solution n°2 : la solution avec l’espace pré-persistant en EEPROM.

du périphérique de libérer l’espace pré-persistant.

La solution n°3 (cf. FIG. 61) est sûrement la plus performante en terme de sécurité et de rapidité. La rapidité est accrue grâce à la localisation de l’espace pré-persistant dans la RAM. Mais il faut nuancer ce gain par la faible taille de la RAM. La sécurité est accrue par la possibilité de créer des objets dans la RAM. Avec cette solution, les tableaux transients d’objets ont un réel intérêt puisque les objets créés peuvent être dans une mémoire volatile et seront donc détruits physiquement entre deux sessions avec le lecteur (cf. section 2.5.2). En effet, si l’on se réfère au listing 8.1, lors d’un retrait de la carte après un appel à la méthode `foo`, les objets de type `Object1` et `Object2` seront détruits. Cette propriété obtenue avec cette implantation semble plus naturelle et conforme à l’idée qu’on peut se faire d’un tableau transient d’objets. Un dernier avantage de cette solution est

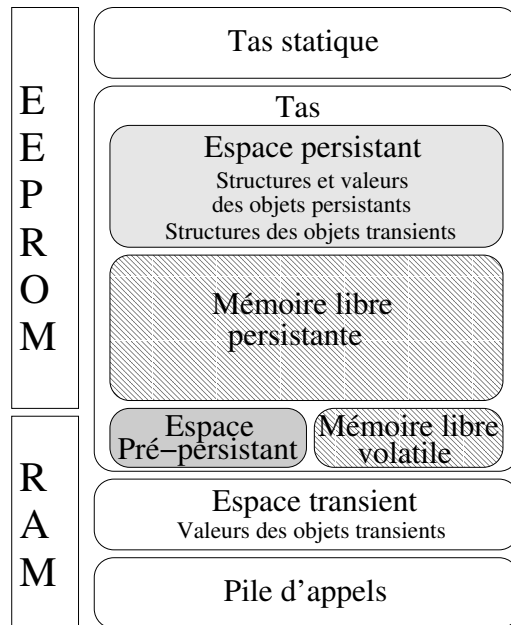


FIG. 61 – La solution n°3 : la solution avec l'espace pré-persistant en RAM.

le mécanisme de ramasse-miettes naturel de la RAM à chaque mise hors-tension, reléguant l'implantation du ramasse-miettes à un simple nettoyage de la table d'allocation de la RAM. Effectivement, rappelons que les objets transients ont vu leur contenu disparaître mais qu'ils existent toujours (leur structure est persistante). Par conséquent, ils requièrent donc toujours de la mémoire RAM qu'il faut marquer comme déjà occupée dans la table d'allocation. Comme nous l'avons expliqué précédemment, le principal problème réside dans la recopie des objets de l'espace pré-persistant en RAM dans l'espace persistant qui est lui en EEPROM.

La solution n°4 (cf. FIG. 62) offre un bon compromis entre la rapidité et la taille des ressources mémoires. En effet, la RAM de l'espace pré-persistant peut être utilisée pour créer les objets par défaut et, en cas de mémoire libre volatile insuffisante, l'objet sera alloué dans l'EEPROM. Son inconvénient est la diminution de la rapidité d'exécution et de la sécurité quand l'espace pré-persistant en RAM est totalement occupé. Les autres propriétés de cette solution sont les mêmes que celles des solutions n°2 et n°3. On notera qu'il est possible d'appliquer différentes politiques d'allocation pour ce modèle.

8.2.5 Travaux connexes

Nos propositions se différencient de celles faites en 1999 par IBM et Schlumberger [179] par plusieurs aspects. Tout d'abord, leurs travaux se plaçaient dans le cadre de la solution n°1, *i.e.* sans espace pré-persistant et avec le tas en EEPROM. Ensuite ils s'intéressaient exclusivement à expliquer les différentes solutions possibles pour définir la transience. C'est dans ce cadre qu'ils ont introduit la notion très séduisante d'Environnement Transient explicite. Ce concept semblait s'intégrer si bien dans la philosophie du langage Java Card

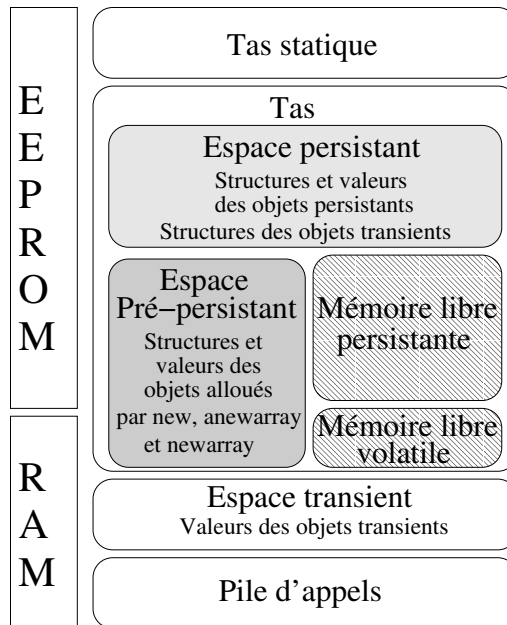


FIG. 62 – La solution n°4 : la solution avec l'espace pré-persistant en RAM et en EEPROM.

qu'on pourrait se demander pourquoi le Java Card Forum et Sun microsystems n'ont pas mis en œuvre leur proposition. En fait, ils n'ont probablement pas voulu changer la sémantique des spécifications. Quoiqu'il en soit si nos travaux se concentrent principalement sur les problèmes issus de la nature persistante des objets, ils sont aussi complémentaires quant à la notion de transience. En effet, la notion de pré-persistence que nous avons introduite permet elle aussi la mise en œuvre des concepts d'Environnements Transients implicite et/ou explicite.

8.2.6 Conclusion et perspectives

Dans ce chapitre, nous avons identifié deux points ambigus dans les spécifications Java Card, introduit la notion de *pré-persistence* et décrit les quatre modèles mémoires possibles, leurs avantages et leurs inconvénients.

S'il est vrai que parmi les différents modèles de mémoire évoqués, il semble que tous les développeurs aient choisi la solution basée sur un tas localisé en EEPROM et sans espace pré-persistant (*i.e.* la solution n°1), il nous paraîtrait souhaitable que le Java Card Forum réexamine la gestion de la mémoire afin de garantir une vraie interopérabilité entre toutes les implantations. La prochaine étape consiste à implanter et tester ces différents modèles sur notre plate-forme Jcatools. Cela nous permettra d'étudier plus en profondeur l'impact effectif des différentes stratégies que nous avons décrites sur des cas réels. Par ailleurs, faute de temps et dans le contexte d'une thèse sous convention CIFRE nous n'avons pas poussé plus avant une formalisation des concepts de persistance, transience et pré-persistence, mais il est certain qu'à terme nous souhaitons proposer un modèle formel.

Troisième partie

Les problèmes de sécurité des cartes multi-applicatives ouvertes

Dans cette partie de la thèse, nous allons lister toutes les attaques possibles sur les cartes à puce multi-applicatives ouvertes. Mais tout d'abord, nous donnerons notre définition de ces cartes.

Nous avons décrit les différentes technologies de cartes multi-applicatives dans la section 1.9. Nous avons également évoqué dans la section 2.6.1 que Java Card (comme les autres technologies), n'a pas été prouvée totalement sûre grâce à des méthodes formelles. En conséquence, il n'est pas sain de laisser fonctionner des applications non certifiées qui pourraient provenir d'un attaquant souhaitant s'en prendre aux biens de la plate-forme et à ceux des applications. C'est pourquoi, afin d'éviter ces problèmes, les émetteurs de cartes suivent des standards comme GlobalPlatform [19] que nous avons décrits au chapitre 3. Si ce standard est utilisé pour se prémunir contre le chargement d'applications non-autorisées, il possède un gros inconvénient pour le déploiement d'applications. En effet, il suit un modèle centralisé, dû au tiers de confiance qui signe les applications, ce qui diminue la flexibilité.

En outre, les travaux autour du vérifieur (*i.e.* vérifieur autonome [106] et machine virtuelle défensive [95]) laissent penser que l'avenir de la carte multi-applicative est de permettre à tout un chacun de charger librement une application – *i.e.* sans authentification préalable. Toute tentative pour charger une application malicieuse sera rejetée soit au chargement par le vérifieur autonome, soit à l'exécution par la machine virtuelle défensive.

Dans cette partie, nous considérerons donc, comme **cartes à puce multi-applicatives ouvertes**, les cartes basées sur une de ces deux techniques. Par ailleurs, puisque nous souhaitons présenter la liste des attaques possibles sur ces cartes, nous présenterons les attaques existantes sur les cartes à puce classiques fermées – les cartes à puce multi-applicatives ouvertes n'étant qu'une catégorie particulière de cartes à puce classiques –, puis celles, déjà existantes, spécifiques aux cartes multi-applicatives et enfin celles, nouvelles, que nous proposons dans le cadre des cartes à puce multi-applicatives ouvertes [91].

Nos contributions sont la définition de ce que devrait être une carte multi-applicative ouverte (*i.e.* réellement ouverte) mais également, dans ce cadre, l'identification de nouvelles attaques.

Chapitre 9

Les attaques classiques

Nous avons vu dans la première partie de cette thèse les protections qu’offre la carte à puce ; nous allons maintenant aborder les différentes attaques connues. Nous verrons qu’il en existe une large gamme et que les attaquants n’ont pas manqué d’imagination dans ce domaine.

Le but de toutes ces attaques est d’accéder aux secrets (*e.g.* le PIN, une clé secrète) contenus sur la carte de façon directe (*e.g.* récupération des données ou du code) ou de façon indirecte (*e.g.* modification des données ou du code). On peut les classer en deux catégories distinctes : les attaques non-invasives et les attaques invasives. Les premières n’affectent pas l’intégrité du composant et ne nécessitent donc pas une mise en œuvre lourde alors que les secondes impliquent des modifications au niveau du câblage de la puce.

9.1 Les attaques non-invasives

L’objectif de cette section est de présenter la majorité des attaques de cette catégorie.

9.1.1 Les attaques hors conditions opérationnelles

Une première classe d’attaques consiste simplement à faire fonctionner la puce en dehors de ses conditions opérationnelles. Elles portent sur la tension et la fréquence d’alimentation (V_{cc} , F) mais aussi sur la température. L’idée est de mettre la carte dans un état de fonctionnement anormal. Par exemple, les attaquants testent souvent le générateur de nombres aléatoires à différentes températures afin de voir s’ils ne peuvent pas modifier son entropie afin d’obtenir un générateur prédictible.

Les attaques sur la fréquence d’alimentation peuvent aussi permettre à un attaquant de mieux comprendre les opérations faites dans la puce en les combinant avec des procédés d’observation des canaux cachés (*cf.* section 9.1.4).

Si ces attaques sont très simples, elles se révèlent souvent peu efficaces en raison des différents détecteurs de conditions anormales installés sur les puces. Même s’il est possible de désactiver ces détecteurs grâce à des modifications de circuiterie, il est peu probable que l’état de fonctionnement anormal de la puce soit exploitable pour l’attaquant.

9.1.2 Les attaques par rayonnement

On a vu se développer une seconde classe d'attaques qui consiste à exposer la puce à un rayonnement afin de modifier l'exécution des applications embarquées. En effet, un rayonnement lumineux est un rayonnement électromagnétique et donc, à ce titre, il possède certaines propriétés pouvant interagir avec un circuit électronique. Pour que le rayonnement puisse atteindre sa cible, ces attaques, contrairement à celles vues précédemment, nécessitent d'avoir un accès direct à la puce. Il faut donc faire ce que l'on appelle une « ouverture », que cela soit par la face avant de la puce (côté dos de la carte) ou par la face arrière (côté contacts de la carte). Ces ouvertures nécessitent l'utilisation de procédés physico-chimiques (*i.e.* extraction à chaud en utilisant des acides) pour retirer le plastique et la colle sans altérer le circuit. Une fois la puce à nue, l'attaquant a accès au côté de la puce qu'il désire et il peut donc l'exposer avec le rayonnement voulu.

L'effet est comparable à ce qui se passe avec des mémoires PROM qui s'effacent en présence de rayonnements UV (Ultra Violet). L'attaquant espère que le rayonnement ira corrompre une partie du contenu de la mémoire afin d'obtenir un fonctionnement inhabituel de la carte. Ces attaques peuvent se faire pendant le fonctionnement mais aussi hors-fonctionnement. Si l'attaque est faite pendant le fonctionnement on peut espérer modifier l'exécution des applications.

Plusieurs types de rayonnements peuvent être utilisés : UV, rayons X, lumière blanche, IR, etc.

Bien sûr, s'il existe également des détecteurs pour ces rayonnements, ils peuvent eux aussi être désactivés par l'attaquant. Par ailleurs, ce type d'attaque même localisée sur des zones bien précises n'est pas des plus performants. En effet, le temps d'exposition est souvent trop long pour causer une erreur seulement locale. La carte se retrouve ainsi fréquemment dans un état inexploitable.

9.1.3 Les attaques par injection de fautes

Une troisième classe d'attaques que constitue l'injection de fautes [72, 195, 120, 150] se révèle souvent très efficace. Elle consiste, durant une exécution, à perturber de manière très brève l'environnement.

On pourra faire des *glitches* (*i.e.* de brèves variations) positifs ou négatifs, par exemple sur la tension d'alimentation. On pourra aussi, à l'aide de laser, envoyer des impulsions lumineuses de courte durée et de différentes natures (IR, lumière blanche, faisceau d'électrons, ...) sur la puce à nue.

En général, l'injection de fautes est réalisée sur plusieurs exécutions du même programme à différents moments dans le temps afin de couvrir la zone intéressante et cela en faisant varier la durée d'exposition de la puce à la perturbation.

Pour obtenir de bons résultats, il faut souvent essayer différents types de perturbation et parfois même les combiner.

Pour se convaincre de la puissance de ces attaques, on peut regarder les travaux d'attaques d'une JVM [132] sur un ordinateur de bureau avec une simple lampe de bureau

convenablement orientée.

9.1.4 Les attaques par canaux cachés

Une quatrième classe d'attaques est celles des attaques par canaux cachés. Cela consiste tout simplement, à observer d'une façon intelligente le fonctionnement du circuit électronique. Ainsi, nous allons voir qu'il existe plusieurs types de canaux cachés :

- le temps d'exécution ;
- la consommation en courant ;
- les émissions électromagnétiques.

Les attaques via le temps d'exécution

Le premier des canaux cachés que nous allons étudier est le temps d'exécution. On sait que toute instruction élémentaire du microprocesseur a une certaine durée, *i.e.* qu'elle dure un certain nombre de cycles, et que cette durée peut être différente de celle d'une autre instruction. Le principe de base de l'attaque sur le temps d'exécution est donc d'observer le temps d'exécution d'un algorithme pour en déduire des informations sur les opérations et/ou les opérandes. Ce sont des attaques qui nécessitent souvent un grand nombre d'exécutions à messages choisis ainsi qu'un traitement statistique des résultats obtenus.

Exemple du PIN

Voici, par exemple, une attaque sur le PIN en utilisant la méthode du temps d'exécution. Prenons le codage simpliste qui consiste à comparer un à un les 8 octets d'un `pinPresente` avec le `pinCarte` qui lui est stocké comme valeur secrète dans la carte. Supposons que nous

```
for(i = 0; i <= 7; i++)  
    if (pinCarte[i] != pinPresente[i])  
        return false;  
return true;
```

ayons droit à un nombre d'essais illimité.

En attaque par force brute sur le `pinCarte` nous devons faire 256^8 essais au maximum pour être certain de trouver le bon PIN, puisque chaque octet a 256 valeurs possibles et que le PIN est un tableau de 8 octets.

Or, l'attaque sur le temps permet ici de diminuer le nombre d'essais à un maximum de 2048. En effet, le problème de cet algorithme de comparaison de PIN est que lorsque l'octet comparé de `pinPresente` est différent de celui de `pinCarte`, la carte sort immédiatement via un saut avec l'opération `return false` ; alors que s'il est identique la carte fait un saut au début de la boucle `for`, incrémente le compteur puis passe à l'opération de comparaison suivante. Ainsi, dans ce dernier cas où le bon octet a été présenté le temps d'exécution est bien différent de celui où il est faux puisqu'il s'agit d'un saut suivi de plusieurs autres opérations (incrémentations puis nouvelle comparaison).

La méthode à appliquer pour arriver à une telle solution est la suivante :

- Présenter sous la forme $(n, 0, 0, 0, 0, 0, 0, 0)$, les n valeurs possibles de `pinPresente[0]` (256 valeurs).
- Mesurer la durée d'exécution τ de la commande, pour les n valeurs.
- Calculer $\tau[n_0]$ le maximum des $\tau : \tau[n_0] = \max(\tau[n]), n = 0, \dots, 255$.
- On en déduit que n_0 est la solution pour `pinCarte[0]`.
- Itérer sur tous les `pinPresente[i]` en utilisant les octets précédemment découverts pour `pinCarte[0], ..., pinCarte[i-1]`.

Ainsi, pour chaque octet de `pinCarte`, on peut prédire la bonne valeur avec un maximum de 256 essais et donc le nombre d'essais pour découvrir `pinPresente` est bien de $8 * 256 = 2048$ contre 256^8 en force brute !

Bien évidemment, il ne s'agissait ici que d'un exemple simple mais de nombreux articles [151, 175] montrent la possibilité de telles attaques sur des algorithmes beaucoup plus complexes.

Les attaques via la consommation en courant

Le second canal caché que l'on va étudier est la consommation en courant. On le doit à Paul KOCHER [151, 152, 98] qui lors de sa découverte a fait trembler le petit monde de la carte à puce. En effet, jusqu'à ce jour les fabricants de cartes, les banquiers et les experts en sécurité avaient fait leur analyse de risque et ils étaient tous certains que personne n'arriverait à mettre en défaut la sécurité leurs produits sans utiliser des moyens tellement coûteux que cela ne pourrait être rentable. Malheureusement pour eux, avec l'article de KOCHER [152] le mythe s'est brisé. Effectivement, grâce à lui, il existe aujourd'hui plusieurs types d'attaques s'étendant de la SPA (Simple Power Analysis) en passant par la DPA (Differential Power Analysis) et ce jusqu'à la HODPA (High Order DPA).

Le principe de base de ces attaques est que les modifications rapides de la tension et de l'intensité du courant au sein du même composant sont à la base des émissions du circuit car ils conduisent des courants RadioFréquences à l'intérieur et à l'extérieur de la puce. Par ailleurs, le matériel nécessaire pour réaliser de telles attaques est somme toute assez sommaire et disponible partout. Il faut un oscilloscope numérique, un lecteur de cartes à puce et un ordinateur équipé de cartes d'acquisition et de logiciels mathématiques pour le traitement des données (cf. les figures 63, 64 et 65). On peut aussi utiliser une sonde CEM (de Conformité ÉlectroMagnétique) si l'on veut étudier les émissions électromagnétiques comme on le verra plus loin.

La SPA

La SPA utilise le fait que des instructions différentes ne consomment pas la même quantité de courant puisqu'elles n'utilisent pas les mêmes parties de la puce.

Ainsi, on devrait idéalement pouvoir observer la signature des instructions avec leur consommation en courant (cf. FIG. 66).

De la même façon, dans le cas idéal, on devrait pouvoir obtenir une trace de consommation en courant identique à celle présentée figure 67 pour un chiffrement DES. En effet sur la figure 67, on distingue nettement la permutation initiale suivie des 16 tours du DES.



FIG. 63 – Une station d'étude de la consommation en courant (source : SERMA Technologies).

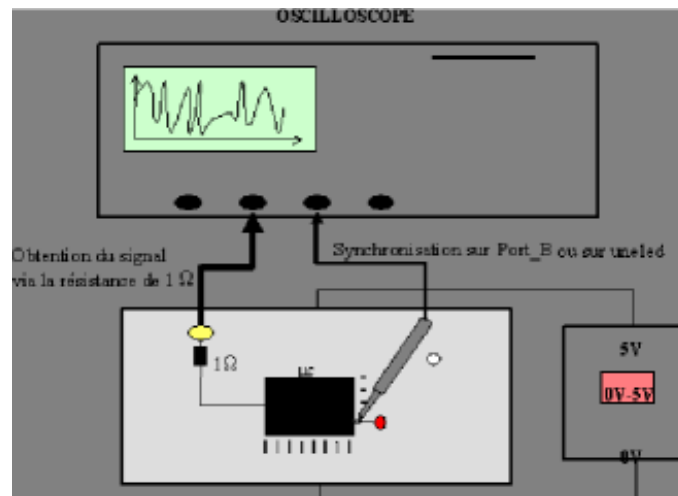


FIG. 64 – Schéma de câblage de l'oscilloscope afin de pouvoir observer la consommation en courant (source : SERMA Technologies).

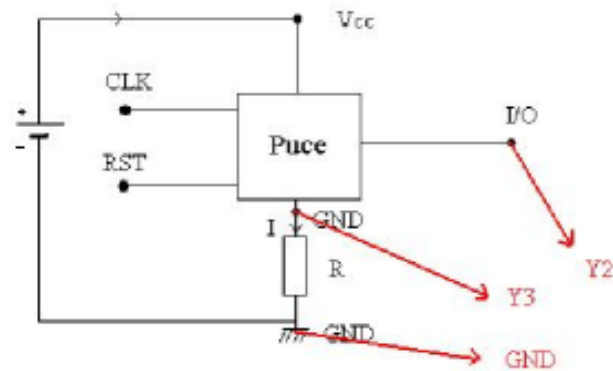


FIG. 65 – Schéma de câblage pour pouvoir observer la consommation en courant aux bornes de la puce (source : SERMA Technologies).

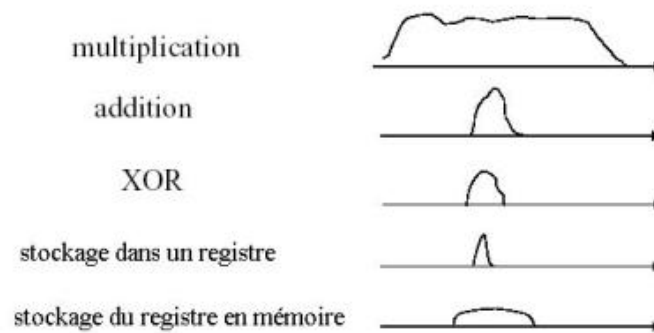


FIG. 66 – La consommation de différentes instructions (source : SERMA Technologies).

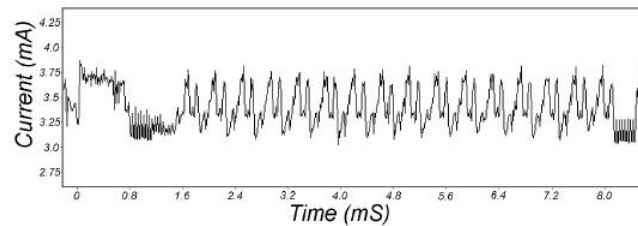


FIG. 67 – La signature en courant d'un chiffrement DES (source : SERMA Technologies).

Exemple de la signature RSA : Prenons l'exemple d'une signature RSA $y^a \bmod n$ où y est le message à signer, n est public et a , l'exposant, peut être considéré comme la clé secrète. Imaginons maintenant que l'algorithme de signature soit codé simplement par le programme suivant où L est la longueur en bits de l'exposant secret. Ce qu'il faut remarquer

```
s = 1;
for(i = L - 1; i >= 0; i--) {
    s = s*s mod n;
    if (a[i] == 1)
        s = s*y mod n;
}
```

sur cet algorithme c'est que, pour chaque tour de boucle **for**, on fait un carré modulaire ($s = s*s \bmod n$;) puis, si le bit i de l'exposant secret a est égal à 1 on fait une multiplication modulaire et sinon on ne fait rien.

Imaginons que l'on connaisse les traces de l'opération de carré modulaire et de multiplication modulaire.



FIG. 68 – La consommation des opérations modulaires (source : SERMA Technologies).

En enregistrant la trace de consommation en courant correspondant à l'exécution de ce code, on peut retrouver tous les bits de la clé secrète simplement en regardant les motifs. En effet, si on retrouve juste une opération de carré modulaire non suivie d'une multiplication modulaire c'est que le bit est égal à 0 et sinon il est égal à 1.

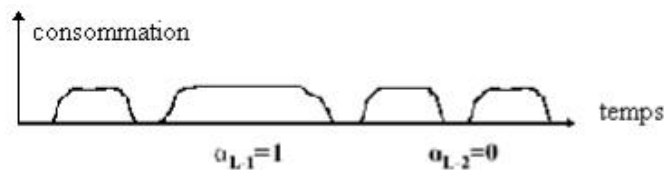


FIG. 69 – La consommation de l'algorithme RSA présenté (source : SERMA Technologies).

On notera donc qu'un tel algorithme n'est pas sécurisé et que souvent les développeurs rajoutent une opération de multiplication modulaire même lorsqu'elle n'est pas nécessaire *i.e.* quand $a[i] == 0$ afin d'empêcher l'attaquant de trouver la clé aussi simplement.

La DPA

La DPA [152, 98, 164] se base sur le principe que la consommation dépend des opérations effectuées mais aussi des opérandes. En effet, la manipulation par une même instruction de deux opérandes ayant un poids de Hamming différent ne donne pas la même consommation

de courant puisque le nombre de bits à 1 et à 0 varie. Or, comme la valeur d'un bit est représenté par un état électrique il semble évident que leur manipulation n'engendrera pas la même consommation.

La DPA utilise des **fonctions statistiques** adaptées à l'algorithme visé pour faire ressortir des **corrélations entre un bit** intermédiaire a (ne dépendant que d'un fragment K_r de r bits de la clé et du message d'entrée M) **et la consommation de courant**.

Un préliminaire à cette attaque est donc d'avoir une fonction $f()$, appelée « fonction de sélection », qui peut être déduite de l'algorithme cryptographique connu à attaquer de telle façon que $a = f(K_r, M)$. Dans cette équation K_r est inconnu, mais M est connu car envoyé par l'attaquant. Lorsque le bit a apparaît à l'entrée d'une instruction I , plus la dépendance D_I entre la consommation de courant pour cette instruction P_I et a sera forte, plus la DPA aura de chances de réussir (cf. FIG. 70).



FIG. 70 – Dépendance de la consommation en courant de l'instruction I (source : SERMA Technologies).

Étapes de la DPA : Maintenant que nous avons expliqué le principe de la DPA, voici ci-après les différentes étapes de l'attaque.

Il faut tout d'abord collecter les données. Pour cela on exécute l'algorithme sur la carte N fois avec N messages d'entrée aléatoires : $M_1, M_2, \dots, M_N$. Pendant les exécutions, on enregistre les consommations de courant $P(K_r, M_1), P(K_r, M_2), \dots, P(K_r, M_N)$.

Ensuite on passe à l'étape d'analyse de données. Pour chacun des 2^r fragments de clé K_r^* possibles, on fait tourner l'algorithme N fois avec les **mêmes messages** que précédemment sur une implantation personnelle, de sorte à pouvoir « observer » le bit a . Puis, il suffit de séparer les courbes en deux paquets de telle sorte que :

- $E_0 = \{P(K_r^*, M_i) \text{ tel que } f(K_r^*, M_i) = 0\}$
- $E_1 = \{P(K_r^*, M_i) \text{ tel que } f(K_r^*, M_i) = 1\}$

Si le fragment de clé K_r^* est correct ($K_r^* = K_r$), quand l'instruction I est effectuée, il y aura une différence notable D_I entre les $P(K_r^*, M_i)$ de E_0 et ceux de E_1 . Donc, si on trace $g(t)$ telle que définit par l'équation 1 on obtiendra un graphique avec un pic significatif en cas de succès (cf. FIG. 71) et un graphique avec du bruit en cas d'échec (cf. FIG. 72).

$$g(t) = \langle P(K_r^*, M_i)(t) \rangle_{P(K_r^*, M_i) \in E_0} - \langle P(K_r^*, M_i)(t) \rangle_{P(K_r^*, M_i) \in E_1} \quad (1)$$



FIG. 71 – DPA effectuée avec succès (source : SERMA Technologies).



FIG. 72 – DPA ayant échouée (source : SERMA Technologies).

Pour résumer, on essaye toutes les possibilités pour le fragment de clé K_r^* en regardant à chaque fois le graphe de la fonction g . Le fragment donnant une fonction g présentant un pic significatif est généralement le bon, *i.e.* les bits de K_r^* sont les bits de la clé.

La HODPA

La HODPA ou DPA d'ordre n est elle aussi basée sur une étude statistique de la consommation de courant de la carte. Cependant, sa grande différence avec la DPA classique est qu'elle utilise des corrélations entre la consommation de courant et n variables intermédiaires ne dépendant que d'un fragment de la clé et du message d'entrée.

Trouver la fonction de sélection adéquate est donc beaucoup plus difficile à réaliser, mais c'est une attaque beaucoup plus puissante que la DPA classique.

Afin de contrer ce type d'attaque, les fondeurs ont installé sur leur puce des mécanismes de pompe de charge. Néanmoins, ces mécanismes peuvent être désactivés de façon matérielle et ne sont pas toujours suffisants pour contrer la DPA.

Les attaques via les émissions électromagnétiques

Le troisième canal caché que nous allons étudier est relatif aux émissions électromagnétiques [184, 116]. Les attaques utilisant ce canal se basent sur le fait que les courants qui circulent dans la puce induisent des champs électromagnétiques susceptibles de donner le même type d'information que le courant. La grosse différence est que l'information est locale alors qu'avec la consommation en courant l'information était globale. On peut donc déplacer une micro-sonde électromagnétique au-dessus de la zone pour laquelle l'on souhaite obtenir de l'information (*e.g.* le co-processeur cryptographique).

Ainsi, l'avantage indéniable de cette technique est qu'elle est insensible aux contre-mesures physiques tels que l'ajout de bruit en sortie ou le lissage de la consommation globale de courant. En revanche, c'est une technique où la reproductibilité des mesures est difficile si l'on n'est pas extrêmement précis.

Pour exploiter ce canal, un attaquant peut chercher à localiser les zones qui émettent le plus de rayonnements électromagnétiques et, pour cela, il va établir une cartographie (*cf.* FIG. 73).

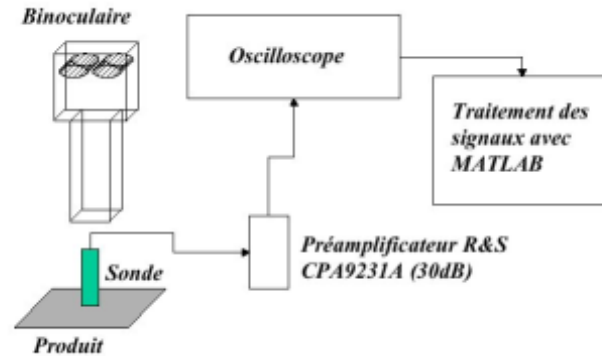


FIG. 73 – Dispositif de cartographie électromagnétique (source : SERMA Technologies).

Les figures 74, 75, 76 et 77 sont des cartographies électromagnétiques pour deux algorithmes effectuant des opérations différentes. Pour chaque cas, on présente la cartographie des valeurs maximales et des valeurs moyennes des rayonnements électromagnétiques enregistrés.

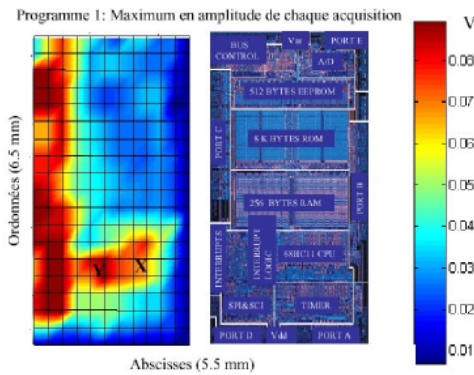


FIG. 74 – Cartographie des valeurs maximales pour l'algorithme 1 (source : SERMA Technologies).

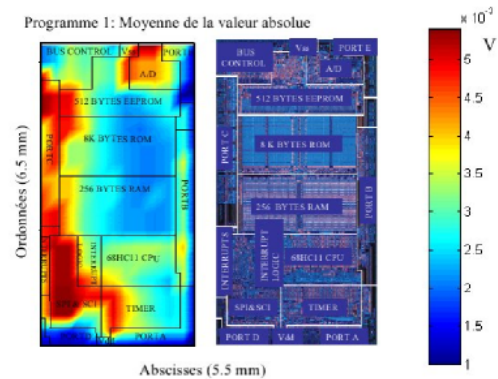


FIG. 75 – Cartographie des valeurs moyennes pour l'algorithme 1 (source : SERMA Technologies).

On constate bien qu'on a une information locale puisque le rayonnement n'est pas partout uniforme et qu'on peut localiser des points chauds (*e.g.* X et Y). Par ailleurs, les attaques possibles en EM sont très semblables à celles que l'on peut réaliser en utilisant la consommation de courant. Ainsi, on peut faire de la SEMA (Simple EM Analysis) qui est l'équivalent de la SPA, ou de la DEMA (Differential EM Analysis) qui est l'équivalent de la DPA. Néanmoins, la DEMA possède le gros avantage de nécessiter un nombre d'acquisitions inférieures à la DPA. En effet, le signal étant moins bruité (*i.e.* l'information est locale et non globale) il y a besoin de moins d'acquisitions pour éliminer, par traitements statistiques, le bruit généré par le reste du composant.

La visualisation du signal EM et sa récupération peuvent également servir, en première

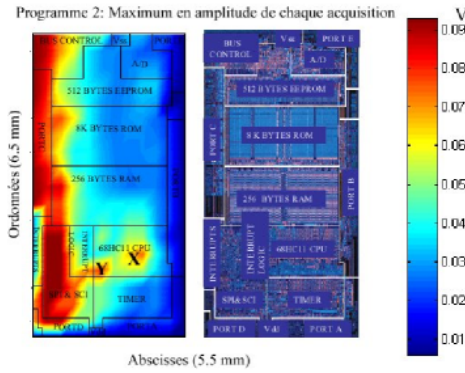


FIG. 76 – Cartographie des valeurs maximales pour l'algorithme 2 (source : SERMA Technologies).

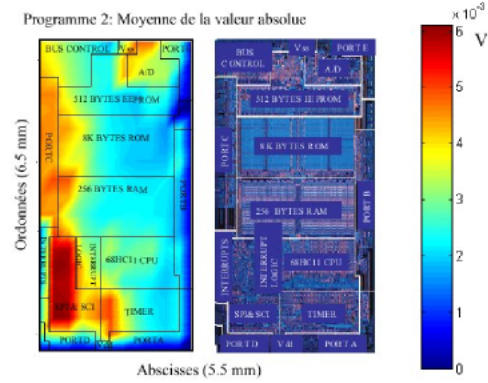


FIG. 77 – Cartographie des valeurs moyennes pour l'algorithme 2 (source : SERMA Technologies).

approche, à resynchroniser les acquisitions de consommation en courant – *i.e.* à trouver les pics locaux en EM qui vont permettre de mettre en phase les signaux de consommation en courant – pour effectuer la DPA qui reste encore aujourd'hui beaucoup plus accessible. Effectivement, si réaliser une DEMA est plus facile théoriquement, dans la pratique cela reste quelque chose de difficile en raison des conditions opératoires qui doivent être extrêmement précises.

Une utilisation des canaux cachés : la rétro-conception de code

Nous allons maintenant présenter une application directe de l'utilisation des canaux cachés qui a pour objectif de faire de la rétro-conception de code. Dans notre description, nous n'évoquerons que l'utilisation de l'EM pour faire cette rétro-conception mais l'utilisation de la consommation en courant est elle aussi, tout à fait possible.

Le principe est très simple et la question peut être posée en ces termes : puisqu'une instruction possède une signature physique différente d'une autre instruction, pourquoi ne pas établir un « dictionnaire » de toutes les instructions et ensuite enregistrer la trace d'une exécution afin de retrouver les instructions qui ont été exécutées ? SERMA Technologies, au travers des travaux de Sébastien Garcia, a essayé de répondre à cette question en utilisant le rayonnement électromagnétique puisqu'en théorie plus local et donc plus précis que la consommation en courant.

Les figures 78 et 79 sont issues du dictionnaire qui avait été construit dans une représentation temps-fréquence. En effet, on représente habituellement les traces dans un mode temps-amplitude mais SERMA Technologies et ses partenaires avaient, dans ce cas précis, voulu étudier le mode temps-fréquence afin d'observer s'ils ne récupéraient pas plus d'information. Globalement le principe général reste cependant le même.

En résultat de l'exécution de l'enchaînement des deux instructions ils auraient souhaité obtenir la trace de la figure 80. Or, s'ils ont bien obtenu un résultat approchant ce n'était pas exactement ce que prédisait la théorie. À cela, il y avait plusieurs raisons : tout d'abord,

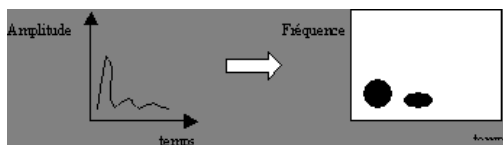


FIG. 78 – Consommation en courant de l’instruction 1 (source : SERMA Technologies).



FIG. 79 – Consommation en courant de l’instruction 2 (source : SERMA Technologies).

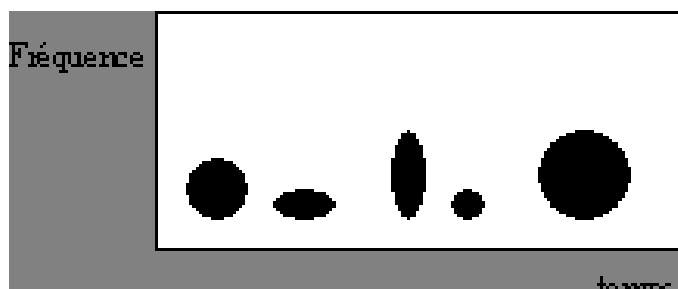


FIG. 80 – Consommation en courant de l’instruction 1 et instruction 2 (source : SERMA Technologies).

ils ont découvert que l’exécution d’une instruction avait une influence sur la suivante ; ensuite, ils ont aussi découvert que sur le composant utilisé, qui n’avait qu’un seul bus pour manipuler les instructions et les données, il y avait une influence des données sur la signature des instructions. Par ailleurs, il y avait aussi des problèmes de précisions à cette époque où l’exploitation des signaux électromagnétiques n’en était qu’à ses balbutiements. Pour le moment, ces recherches sont toujours en cours.

Par ailleurs, si nous avons parlé des attaques électromagnétiques, il ne faut pas oublier que la plupart des composants disposent d’un blindage pour éviter les émissions. Hélas, dans certains cas, les grilles sont contre-productives et facilitent même la propagation des signaux électromagnétiques. Cependant, lorsque ces grilles sont gênantes, les techniques proposées dans la section 9.2.2 permettent de les ôter afin de mieux observer le composant.

9.1.5 Les attaques logicielles

L’accès pour un utilisateur extérieur à la carte étant limité au niveau logiciel à la seule interface de communication, il n’existe pas à proprement parler d’attaques logicielles. Bien évidemment, l’apparition des cartes multi-applicatives sur lesquelles on peut charger du code permettra de réaliser de attaques purement logicielles ou couplées (*i.e.* logicielle et matérielle) comme nous le verrons au chapitre 11. Néanmoins, il est possible d’avoir des attaques de type débordement de tampon suivant l’architecture de la puce. Cela peut être le cas si, par exemple, la méthode de recopie de l’APDU reçu dans la RAM est mal implantée.

9.1.6 Les attaques par erreur

Il peut aussi exister des attaques qu'on appelle « misuse » et qui sont causées par une mauvaise utilisation du produit (souvent involontaire) de la part de l'utilisateur. Ces utilisations peuvent mettre la carte ou l'application dans un état non prévu par les spécifications. Les causes de ces vulnérabilités sont souvent des bugs du programmeur de l'application ou du système d'exploitation.

9.2 Les attaques invasives

Dans le domaine des attaques invasives il y a trois types d'attaques :

- le *microprobing* ;
- la modification de circuit ;
- la rétro-conception matérielle.

9.2.1 Le *microprobing*

La plus simple des attaques invasives est le microprobing [150]. Elle consiste à poser des micro-sondes sur certains bus du circuit pour espionner ou modifier l'information qui circule dessus. S'il est vrai que ces bus peuvent être enfouis dans les différents niveaux de métallisation, nous verrons dans la section suivante qu'il existe des solutions pour arriver à les atteindre sans détériorer la puce.

9.2.2 La modification de circuit

Une seconde attaque consiste à modifier la schématique du circuit [150]. Cela permet de l'étudier ou de lui faire réaliser des opérations particulières permettant de l'attaquer plus facilement.

Dans cette entreprise, le micro-électronicien s'appuie sur le FIB (Focused Ion-Beam). Cet outil dont le principe de fonctionnement est présenté figure 81 permet de modifier à loisir la schématique d'un circuit (*cf.* FIG. 82).

Il permet d'atteindre les différents niveaux de métallisation et donc éventuellement de « poser » un plot pour permettre le micro-probing d'un bus enfoui (*cf.* FIG. 83). Afin de prouver l'efficacité de ce matériel, SERMA Technologies avait fait un test qui consistait à graver son nom sur une piste d'un circuit électronique (*cf.* FIG. 84).

Encore récemment, les attaques invasives n'étaient pas un réel problème car elles exigeaient des outils, comme le FIB, qui coûtaient très cher. Seulement cet outil n'étant pas exclusivement destiné à faire des attaques sur des cartes à puce mais aussi à faire de l'analyse de défaillance ou du prototypage de nouveau circuit, il s'est démocratisé. Ainsi, il est aujourd'hui possible de le louer à la journée, à un prix raisonnable, pour faire ses modifications de circuits.

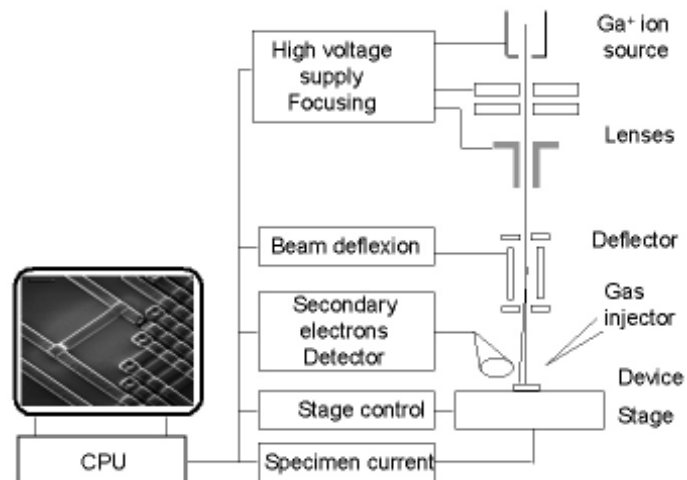


FIG. 81 – Le schéma d'un FIB (source : SERMA Technologies).

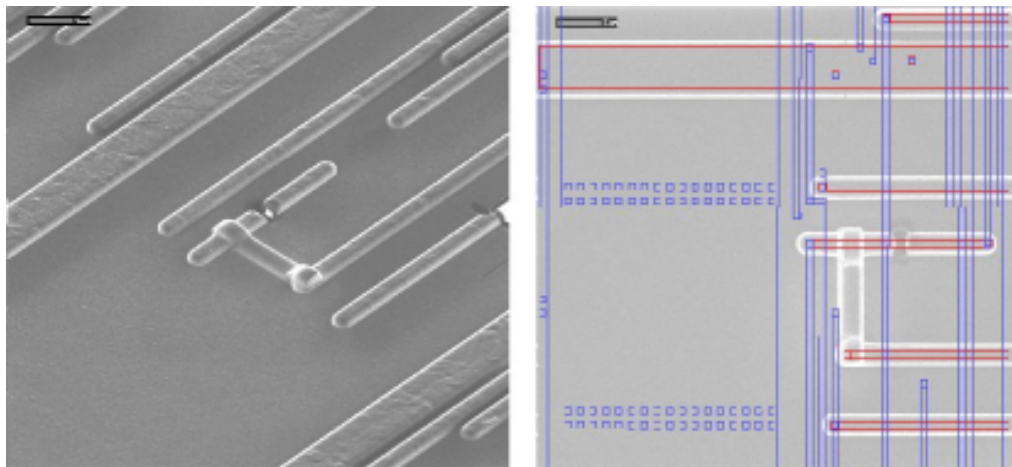


FIG. 82 – Un circuit modifié au FIB et sa schématique (source : SERMA Technologies).

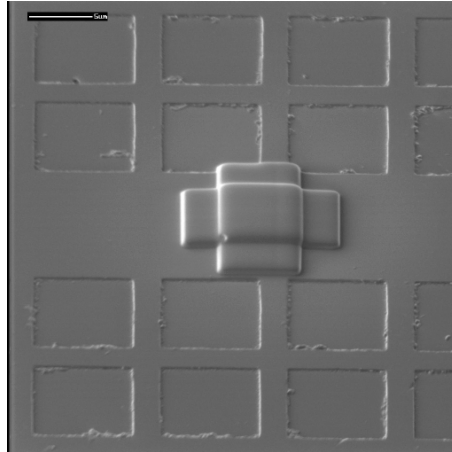


FIG. 83 – Un plot de micro-probing réalisé à l'aide du FIB (source : SERMA Technologies).

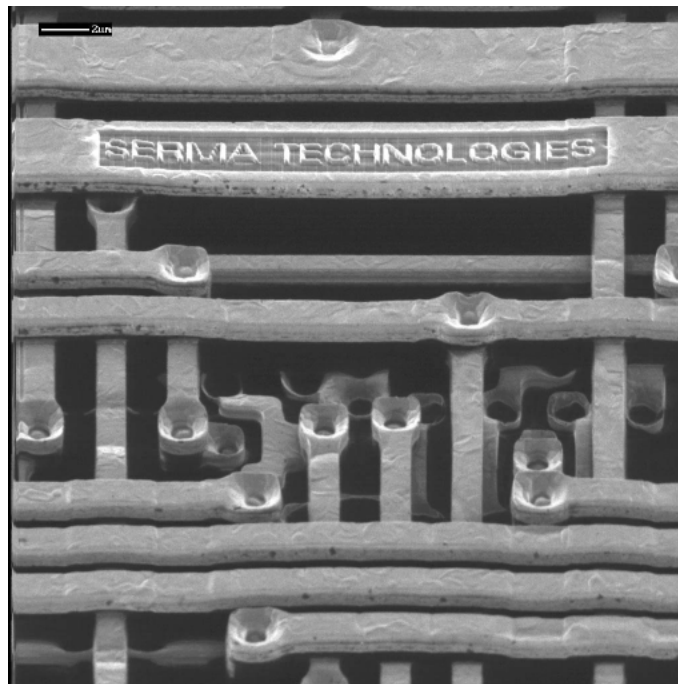


FIG. 84 – Un circuit sur lequel SERMA Technologies a gravé son nom au FIB (source : SERMA Technologies).

9.2.3 La rétro-conception matérielle

Enfin, la dernière catégorie d'attaques consiste à faire de la rétro-conception de la puce afin d'en connaître tous les secrets et donc de pouvoir mieux l'attaquer, voire même la cloner. Par exemple, il est possible en utilisant les analyses physiques de reconstruire toute la circuiterie de la puce, ce qui permet de connaître tous les points sensibles où il est possible d'espionner les informations (*e.g.* bus de données).

Chapitre 10

Les attaques internes

De façon évidente, un des principaux problèmes des cartes multi-applicatives ouvertes est la possibilité de créer des attaques internes grâce à la possibilité de charger des applications contenant du code malicieux. De telles applications pourraient tenter :

- d’identifier les services présents sur la carte et de déduire le comportement probable d’une *applet* officielle¹ (en effet elle ne peut utiliser que les services présents sur la carte ce qui réduit le champ d’investigation de l’attaquant) ;
- de collecter des informations sur la carte pour préparer des attaques matérielles et logicielles ;
- d’attaquer la machine virtuelle et le pare-feu.

10.1 L’identification de service

Le listing 10.1 présente une application qui essaye d’utiliser tous les services cryptographiques définis dans les spécifications Java Card[169, 170, 171] afin de déterminer s’ils sont disponibles ou pas sur la carte.

10.2 La collecte d’information

Le listing 10.2 montre comment il est possible de détecter toutes les applications présentes sur une Java Card. Si ces applications elles proposent des services, il montre également comment essayer d’obtenir une interface `Shareable` pour les utiliser. Ce code présume que les *applets* cibles ne réalisent qu’une authentification basée sur la valeur de `parameter` passée à la méthode `getAppletShareableInterfaceObject(AID serverAID, byte parameter)`. Cette hypothèse forte n’est pas très réaliste. Mais si les *applets* cibles exigent une authentification plus réaliste et basée seulement sur l’AID de l’*applet* client (*i.e.* `ScannerApplet` ici) il est encore possible sur certaines implantations de Java Card d’obtenir un AID valide. Cette attaque connue sous le nom de « *AID Impersonation* » [173] est rendue possible à cause de certaines contradictions sur les règles de nommage des

¹une *applet* officielle est une *applet* installée par un organisme officiel, *e.g.* une *applet* bancaire.

Listing 10.1 – La recherche des services cryptographiques.

```

public class TestCryptoAlgo extends Applet {
    ...
    public void process(APDU apdu) throws ISOException {
        byte[] buffer = apdu.getBuffer();
        // Essaye d'obtenir une instance de tous les algorithmes disponibles dans les spécifications.
        switch ( buffer[ISO7816.OFFSET_INS] ) {
            case 0x02: isValidAlgorithm(Cipher.ALG_DES_CBC_NOPAD);
            case 0x04: isValidAlgorithm(Cipher.ALG_DES_ECB_NOPAD);
            case 0x06: isValidAlgorithm(Cipher.ALG_RSA_PKCS1);
            case ...
        }
    }

    public void isValidAlgorithm(byte bAlgo) throws ISOException {
        try {
            // Essaye de créer une instance d'un algorithme de type bAlgo.
            Cipher cipherInst = Cipher.getInstance(bAlgo, true);
        } catch (CryptoException e) {
            // Si le type n'est pas implanté, le Status Word = 0x6FFF est envoyé sur la ligne d'I/O
            // sinon le SW = 0x9000 signifiant réussi est envoyé par la carte elle-même à la fin.
            if (e.getReason() == CryptoException.NO_SUCH_ALGORITHM)
                ISOException.throwIt((short) 0x6FFF);
        }
    }
    ...
}

```

applets dans les standards Java Card et GlobalPlatform. En effet, GlobalPlatform permet d'installer une *applet* avec l'AID que l'on désire alors que Java Card restreint cela à un AID basé sur le RID du paquetage. Donc, selon les choix d'implantation, il sera possible de mener à bien cette attaque ou non.

Il est également possible d'ajouter un glitch (voir la méthode décrite dans la section 11.1) juste avant d'accéder aux données des *applets* cibles afin de synchroniser une attaque physique (*e.g.* injection de fautes [72, 120, 195]) dans l'espoir de passer outre les vérifications du pare-feu grâce aux perturbations créées sur le composant matériel.

10.3 Les attaques contre la machine virtuelle, le pare-feu, les APIs

Évidemment, il existe beaucoup d'attaques contre la VM et le pare-feu, mais nous ne les décrirons pas en détails car beaucoup d'autres travaux les ont déjà évoquées. Par exemple, deux attaques contre le mécanisme de pare-feu (*AID impersonation* et le *cast* illégal de référence pour avoir accès à toutes les méthodes d'interface d'une classe) sont décrites dans [173]. Il y a également des attaques issues de problèmes dans les spécifications comme celles présentées dans [78]. Des attaques contre la VM existent aussi et elles peuvent être dues à une mauvaise spécification ou à une mauvaise implantation.

Par exemple, le programme du listing 10.3 qui fabrique un pointeur devrait être rejeté par une carte multi-applicative ouverte soit au chargement pour celles utilisant le vérifieur autonome soit à l'exécution pour celles utilisant une machine virtuelle défensive.

Listing 10.2 – Le code pour scanner toutes les applications et essayer de récupérer une interface shareable.

```
public class ScannerApplet extends Applet {
    ...
    public void process(APDU apdu) throws IOException {
        byte[] tab = new byte[16];
        ...
        // Mettre ici le code pour tester toutes les valeurs possibles pour tab
        ...
        for(byte i=5; i <= 16; i++) {
            AID aid = JCSYSTEM.lookup(tab, (short) 0, i);
            if (aid != null) tryToGetServices(aid);
        }
    }

    void tryToGetServices(AID aid) {
        for(byte i=0; i <= 255; i++) {
            Shareable sio = JCSYSTEM.getAppletShareableInterfaceObject(aid, i);
            if (sio != null) callMethodOfSIO(sio);
        }
    }

    void callMethodOfSIO(Shareable sio) {
        // Mettre ici le code pour appeler les méthodes sur cette interface
    }
    ...
}
```

Malgré tout, un attaquant peut toujours essayer de charger de tels codes malicieux afin de déterminer s'il y a des problèmes d'implantation ou non.

Le morceau de code du listing 10.3 (fourni dans un langage intermédiaire) parcourt toute la table d'allocation et essaie de lire un à un les octets de l'objet en le considérant comme un tableau. Il utilise l'hypothèse forte que la pile n'est pas typée (*e.g.* c'est le cas d'une machine virtuelle offensive ou d'une mauvaise implantation d'une machine virtuelle défensive) afin de faire de l'arithmétique sur les références.

On notera que si la référence est représentée par une adresse mémoire (*i.e.* l'équivalent d'un pointeur) dans l'implantation de la machine virtuelle, alors ce code parcourra toute la mémoire.

Listing 10.3 – Le code pour balayer toutes les références.

```
sconst_0 // From reference 0
:loop
    dup
    sconst_0
    baload // Essaie d'obtenir un octet et le pose sur la pile (mais on peut imaginer faire d'autres opérations)
    invoke do_something // Méthode qui retire l'octet lu et ne retourne rien
    sconst_1
    sadd // Incrémente la valeur de la référence par pas de 1
    goto loop

void do_something(byte value@address) {
    // Récupère une opérande de type byte sur la pile (i.e. lit l'octet) et la sauvegarde quelque part
}
```


Chapitre 11

De nouvelles classes d'attaques

11.1 Les méthodes d'aide à l'identification physique

Une fois que les services offerts par une carte ont été identifiés par l'une des méthodes exposées précédemment, il peut être intéressant d'observer leur signature physique. Comme nous l'avons déjà évoqué les principaux signaux physiques observables sont issus des canaux cachés : consommation en courant, émission électromagnétique ou durée du temps d'exécution. Par exemple, on peut utiliser un service déterminé comme présent sur la carte afin d'essayer de rechercher les caractéristiques physiques des signaux émis durant son exécution (*e.g.* durée dans le temps, localisation spatiale des meilleures émissions électromagnétiques, consommation en courant, etc.). Cette technique de caractérisation pourra se focaliser sur l'utilisation d'un service complet mais également sur des opérations élémentaires, *e.g.* l'interprétation d'un seul *bytecode* ou d'une séquence de *bytecodes*.

Afin de déterminer les signatures de façon précise et efficace, nous proposons deux méthodes. La première, la plus simple, se fait par l'utilisation de *glitches* sur le canal d'entrée/sortie de la carte. La seconde consistera pour sa part à répéter plusieurs fois le motif à observer.

Dans les prochaines sections, nous allons donc présenter ces deux approches avec leurs avantages et leurs inconvénients. Nous proposerons également une méthode hybride qui lèvera les problèmes en utilisant le meilleur de ces deux méthodes.

11.1.1 La méthode à base de *glitches*

Cette méthode consiste à entourer le motif à observer par des événements visibles pour un observateur situé à l'extérieur de la carte. La figure 85 montre la trace de l'exécution normale d'une *applet* contenant le motif que l'on veut observer. On peut constater la difficulté dans un tel cas d'isoler le motif à observer de tous ceux composant la trace. C'est pourquoi il nous faut mettre en place un moyen de le faire ressortir. Or, le seul événement visible que la carte peut produire pour un observateur extérieur est l'envoi d'octets sur la ligne de communication. Ainsi, si l'exécution est glitchée en utilisant cet événement il sera alors facile de retrouver le motif dans la trace (*cf.* FIG. 86). Bien évidemment, on

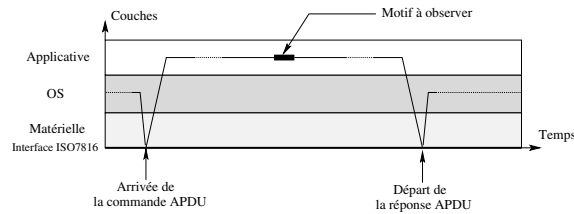
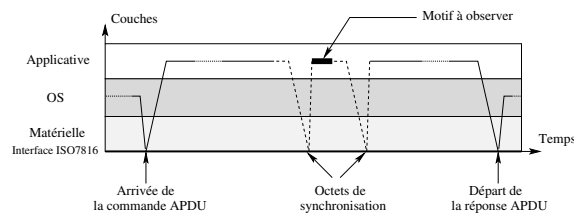


FIG. 85 – Trace d'une exécution normale vue au travers des différentes couches.

FIG. 86 – Trace d'une exécution *glitchée* vue au travers des différentes couches.

notera que l'ajout du code de déclenchement des *glitches* augmente le nombre d'opérations totales (*i.e.* il y a une surcharge d'opérations due au glitch) et ainsi le motif à observer n'apparaîtra pas exactement au même instant dans la première trace et dans la seconde. En plus de permettre de localiser facilement le motif, un autre avantage de cette approche est que la trace à sauvegarder est plus courte (*i.e.* on ne sauvegarde que la partie entre les deux *glitches*) que celle d'une exécution normale. Il est donc possible d'obtenir un meilleur échantillonnage du signal pour une même quantité de données.

Utilisation des requêtes de délai supplémentaire

Le standard ISO 7816-3 [108] définit un mécanisme spécial qui permet à la carte à puce de demander un délai supplémentaire au lecteur, *i.e.* elle l'informe qu'il devra attendre un peu avant de recevoir un résultat afin d'éviter un *timeout*. Ce mécanisme dépend du protocole de transmission (*e.g.* T=0 ou T=1) et permet au lecteur de savoir que la carte n'est pas muette et par conséquent qu'elle fonctionne encore. Par exemple, pour le protocole T=0 ce mécanisme consiste à envoyer l'octet procédural NULL (*i.e.* l'octet 0x60) au lecteur.

En Java Card, la méthode appelée pour utiliser ce mécanisme est `apdu.waitForExtension()`. Le listing 11.1 présente un exemple utilisant les *glitches* produits par `waitForExtension()` pour entourer un chiffrement.

Dans les récentes implantations de beaucoup de fabricants, la méthode `waitForExtension()` est désactivée au niveau utilisateur. Dans ce cas, les demandes de délai supplémentaire sont automatiquement gérées par la plate-forme et il est donc impossible d'utiliser cette méthode d'aide à l'identification physique.

Listing 11.1 – Chiffrement entouré par des *glitches* de `waitExtension()`.

```

public void process(APDU apdu) {
    ...
    // Demande un délai supplémentaire qui envoie des données sur la ligne d'I/O (glitch 1).
    apdu.waitExtension();
    cipherLength = cipher.doFinal(clearData, (short) 0,
                                  clearData.length,
                                  cipherData, (short) 0);
    // Demande à nouveau un délai supplémentaire (glitch 2).
    apdu.waitExtension();
    ...
}

```

Utilisation du modèle classique de communication

La seconde solution utilise les méthodes de communication classique. Par exemple, il est possible de détourner le mécanisme de communication pour générer un événement correspondant à un *glitch* en envoyant la réponse de la carte par morceaux.

Le modèle classique de communication d'une carte à puce vers le lecteur consiste à envoyer toute la réponse en une fois. Dans notre méthode, nous proposons d'envoyer une pseudo réponse en deux fois : une première partie, avant le motif à observer, et une seconde après. Le listing 11.2 présente un exemple utilisant cette méthode basée sur des *glitches* de réponse pour entourer un chiffrement.

Listing 11.2 – Chiffrement entouré par des *glitches* de donnée.

```

public void process(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
    ...
    buffer[0] = (byte)0xFF;
    apdu.setOutgoing();
    apdu.setOutgoingLength((short) 2);
    // Envoie un octet de synchro signifiant que l'on est au début du traitement (glitch 1)
    apdu.sendBytes((short) 0, (short) 1);
    cipherLength = cipher.doFinal(clearData, (short) 0,
                                  (short) clearData.length,
                                  cipherData, (short) 0);
    // Envoie un octet de synchro signifiant que l'on est à la fin du traitement (glitch 2)
    apdu.sendBytes((short) 0, (short) 1);
    ...
}

```

Cette méthode a été testée avec succès sur beaucoup de cartes et elle devrait fonctionner avec n'importe quelle carte compatible ISO7816-3-4. Souvent, on croit que cela ne peut pas marcher car on raisonne au niveau APDU et on pense que les envois (*e.g.* `sendBytes(...)`) sont regroupés. Mais comme nous avons pu le voir sur la figure 5, les APDUs (*i.e.* ISO7816-4) sont simulés par une séquence d'échanges de TPDUs (*i.e.* ISO7816-3). Or, en Java Card, la classe APDU n'est pas une représentation d'un APDU mais seulement une émulation des APDUs au niveau TPDU (*e.g.* avec les cartes T=0 l'appel obligatoire à `getBuffer()` envoie une réponse TPDU au lecteur pour lui signifier qu'il peut envoyer un TPDU avec le champ de données de la commande APDU à la carte). Ceci entretient la confusion pour un utilisateur non expert dans les protocoles de communication sous-jacents.

Améliorations possibles

En utilisant la méthode ci-dessus, il est possible d'observer des motifs de taille importante mais on souhaite parfois améliorer la précision de cette observation.

Prenons l'exemple suivant. La séquence de *bytecodes* du listing 11.3 correspond au résultat de la compilation puis de la conversion des trois dernières lignes du listing 11.2. Il semble évident en regardant ce code qu'il est possible d'améliorer la précision de l'ob-

Listing 11.3 – *Bytecodes* générés pour l'invocation du chiffrement entouré par les *glitches*.

```

aload_1
sconst_0
sconst_1
invokevirtual    0x0 0x6 // glitch 1 (le glitch est réellement envoyé ici)
getfield_a_this 0x0
getfield_a_this 0x1
sconst_0
getfield_a_this 0x1
arraylength
getfield_a_this 0x2
sconst_0
invokevirtual    0x0 0xa // motif à observer (le chiffrement est réellement fait ici)
sstore_2
aload_1
sconst_0
sconst_1
invokevirtual    0x0 0x6 // glitch 2 (le glitch est réellement envoyé ici)

```

servation en travaillant au niveau du *bytecode*. La séquence de *bytecodes* présentée dans le listing 11.3 peut être modifiée pour donner le code du listing 11.4 et garder le même comportement global mais avec une meilleure précision pour l'encadrement du chiffrement par les *glitches*. On remarquera que cette amélioration est aussi possible quand on utilise

Listing 11.4 – Amélioration de la précision de l'encadrement.

```

aload_1
sconst_0
sconst_1
getfield_a_this 0x0
getfield_a_this 0x1
sconst_0
getfield_a_this 0x1
arraylength
getfield_a_this 0x2
sconst_0
aload_1
sconst_0
sconst_1
invokevirtual    0x0 0x6 // glitch 1 (le glitch est réellement envoyé ici)
invokevirtual    0x0 0xa // motif à observer (le chiffrement est réellement fait ici)
sstore_2
invokevirtual    0x0 0x6 // glitch 2 (le glitch est réellement envoyé ici)

```

la méthode avec les `waitExtension()` si la plate-forme le permet.

Un des problèmes lorsque l'on veut travailler au niveau du *bytecode* est qu'une modification même mineure dans le fichier CAP implique souvent des modifications de beaucoup d'autres parties. Ainsi, des changements dans le tableau des *bytecodes* du composant Method du fichier CAP impliquent fréquemment de modifier les informations relatives à la taille maximale de la pile et au nombre maximum de variables locales de la fenêtre d'appel de la méthode modifiée. Le composant ReferenceLocation et sa taille devront également souvent être modifiés. Et enfin, ces modifications impliquent de changer des informations dans le composant Directory. Pour simplifier toutes ces opérations, nous avons développé un outil fourni dans la suite JCatools [88, 140] permettant de modifier de manière cohérente le composant Method d'un fichier CAP. Nous fournirons plus d'informations sur son utilisation en section 11.6.

Protections contre ces attaques

Quelques solutions pour empêcher, ou tout au moins gêner ces attaques existent. Ainsi, il est possible :

- d'introduire un délai aléatoire sur la ligne d'entrée/sortie. Mais, c'est une mauvaise solution puisqu'après beaucoup d'essais il devrait être possible de supprimer le bruit causé par l'aléa.
- de stocker toutes les données dans un tampon jusqu'à la fin de l'exécution avant de les envoyer sur la ligne d'entrée/sortie. Mais, cela exigerait probablement un second tampon (en plus de celui représentant le *buffer* APDU). En effet, le *buffer* APDU – dont la gestion pour l'envoi des messages est par ailleurs laissée au soin de l'utilisateur – a souvent une taille très limitée car implanté en RAM pour des raisons de sécurité ; il est donc potentiellement incapable de contenir toutes les données à renvoyer. Cependant, la taille maximale exigée par l'ISO 7816 pour ce second tampon serait trop importante pour rester acceptable sur une carte (*i.e.* la taille d'un APDU en mode étendu peut atteindre jusqu'à 64 Ko et la taille de l'EEPROM est souvent inférieure à cette valeur) ; ce n'est donc pas non plus une bonne solution.
- de stocker les données dans un tampon jusqu'à ce que leur taille atteigne un certain seuil fixé avant de les envoyer sur la ligne d'entrée/sortie. Dans ce cas, l'attaquant pourra instrumenter son *applet* pour générer la quantité exacte de données nécessaire pour obtenir une réponse sur la ligne d'entrée/sortie. Il pourra ainsi connaître le début du motif à observer.
- de stocker les données comme précédemment dans un tampon mais avec un seuil aléatoire. Malgré cela, l'attaquant pourra encore passer outre cette sécurité en utilisant des techniques probabilistes.

Pour toutes ces raisons, nous croyons que l'utilisation conjointe d'un délai aléatoire avec le stockage dans une mémoire tampon à seuil aléatoire est la solution la moins mauvaise à choisir.

On notera cependant que même si les fabricants trouvaient de meilleures contre-mesures, il serait encore possible de localiser un motif en l'entourant avec des bouts de codes qui induisent des fortes consommations en courant (*e.g.* un mécanisme cryptographique, etc.) pour produire des événements visibles de l'extérieur.

11.1.2 Méthode basée sur la répétition de motifs

Puisque les fabricants peuvent ajouter les contre-mesures proposées ci-dessus, nous allons décrire une autre méthode qui permet encore de capturer la signature des opérations réalisées sur la carte. Cette solution consiste tout simplement à répéter plusieurs fois une séquence identique pour localiser plus facilement la zone à étudier et donc, par exemple, pour identifier des motifs élémentaires.

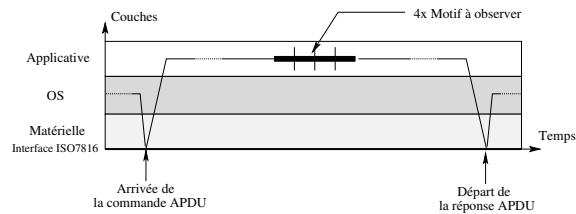


FIG. 87 – Trace d’une exécution de multiples motifs vue au travers des différentes couches.

Par exemple, l’insertion du code du listing 11.5 dans une *applet* Java Card permet de trouver plus facilement le motif élémentaire `dup`. Les émanations physiques correspondant

Listing 11.5 – La méthode des motifs.

```
...
// considère qu'il y a au moins un élément sur la pile
dup    // motif à observer
dup    // motif à observer
dup    // motif à observer
dup    // motif à observer
...
```

à l’exécution du programme ci-dessus contiendront alors quatre fois le motif correspondant à l’opération `dup`.

Solutions contre cette attaque

La première solution est d’ajouter des contraintes au vérifieur embarqué afin qu’il vérifie, par exemple, que le code à charger ne contient pas une séquence de deux *bytecodes* `dup`, *i.e.* `dup dup`. En effet, un convertisseur aurait normalement dû produire le *bytecode* `dup2` et pas cette séquence. Néanmoins il n’est pas réalisable de faire de telles vérifications pour tous les *bytecodes* possibles. En effet, cela exigerait que le vérifieur soit capable de comprendre la sémantique de la séquence de *bytecodes* au niveau applicatif (*i.e.* ce que l’application veut faire) et c’est évidemment impossible. Le vérifieur peut seulement comprendre la sémantique des séquences de *bytecodes* au niveau de la machine virtuelle (*i.e.* si la séquence de *bytecodes* est valide). Une machine virtuelle défensive peut également faire le même type de vérifications, mais le problème est le même. Une meilleure solution serait d’utiliser des contre-mesures matérielles (*e.g.* une horloge interne variant de façon aléatoire ou des processeurs asynchrones [174, 145] qui rendent très difficile la resynchronisation des acquisitions des émanations physiques, etc.) pour perturber les émanations physiques.

11.1.3 La méthode hybride utilisant les *glitches* et les motifs

Cette méthode amortit l'effet de la surcharge causée par la transition entre les différentes couches (*i.e.* applicative, système d'exploitation et matérielle) due à la génération des *glitches*. Elle permet de localiser facilement la séquence de motifs identiques dans la trace globale et il est facile de trouver les motifs élémentaires dans cette séquence.

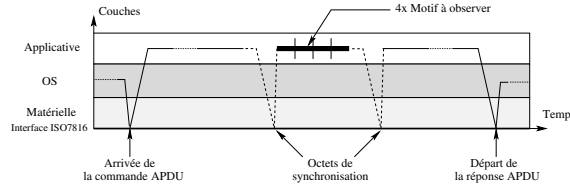


FIG. 88 – Trace d'une exécution *glitchée* de multiples motifs vue au travers des différentes couches.

Le listing 11.6 présente un exemple d'utilisation de la méthode hybride pour observer le *bytecode* `sxor`.

Listing 11.6 – Exemple de code utilisant la méthode hybride.

```
...
aload_1
sconst_0
sconst_m1
sconst_m1
sconst_m1
sconst_m1
sconst_m1
sconst_1
aload_1
sconst_0
sconst_1
invokevirtual 0x0 0xc // glitch 1
sxor // motif à observer
sxor // motif à observer
sxor // motif à observer
sxor // motif à observer
invokevirtual 0x0 0xc // glitch 2
...
```

11.2 Les applications des méthodes d'aide à l'identification physique

Dans cette section, nous décrivons deux applications possibles des méthodes d'aide à l'identification physique expliquées précédemment :

- les attaques par couplage qui permettent de faire de la rétro-conception de code d'une *applet* officielle ;
- les attaques physiques qui permettent de déterminer rapidement la faisabilité d'une attaque contre une *applet* officielle.

11.2.1 Les attaques par couplage

Une telle attaque consiste à coupler le signal identifié pour un motif dans une *applet* malicieuse avec le signal d'une *applet* officielle pour faire de la rétro-conception. La première étape nécessaire pour mettre en place cette attaque est de construire un dictionnaire qui fait correspondre les motifs d'un signal avec les *bytecodes*. La seconde étape est d'identifier les motifs du dictionnaire dans une trace d'exécution d'une *applet* officielle pour connaître les opérations qu'elle réalise.

C'est évidemment une tâche très difficile mais des recherches sont en cours pour la reconnaissance automatique de motifs dans un signal [185]. Il est important de remarquer que si cette attaque met en péril la confidentialité du code de l'*applet* officielle (*i.e.* le code peut être connu), elle n'assure pas qu'une attaque sera possible contre l'exécution de ce code.

Cependant, dans les évaluations Critères Communs, le code d'une *applet* officielle est bien souvent un bien à protéger et une telle attaque sera donc un problème à prendre en compte. De plus, la connaissance du code permet de mener des analyses détaillées pour rechercher d'éventuelles vulnérabilités afin de déterminer si une attaque est possible et à quel moment dans l'exécution du code. Enfin, il est aussi possible de coupler la connaissance du code avec les attaques que nous décrirons ci-dessous.

11.2.2 Les attaques physiques

Grâce aux méthodes d'identifications que nous avons présentées, un attaquant peut mettre en place des attaques physiques [72, 120, 195] sur le motif identifié dans sa propre *applet*. En effet, il peut se placer dans les meilleures conditions pour attaquer facilement et rapidement l'implantation d'une carte (*e.g.* essayer de passer outre les conditions d'accès ou perturber un algorithme cryptographique) évitant ainsi de mener beaucoup d'essais inutiles et donc par la même, gagner beaucoup de temps.

Si son attaque contre sa propre *applet* réussit, il peut alors ensuite attaquer l'*applet* officielle ou même la plate-forme.

Par exemple, s'il connaît une attaque contre un algorithme cryptographique utilisé par une *applet* officielle, il pourra attaquer directement cet algorithme en l'appelant depuis sa propre *applet*. Ainsi, en entourant l'appel de *glitches*, il pourra tester rapidement son implantation.

11.3 Expérimentations

Il y a quelques années, Sébastien Garcia¹ avait fait des expériences similaires visant à réaliser des attaques par couplage pour le CESTI de SERMA Technologies. Il travaillait alors au niveau assembleur et il avait obtenu des résultats intéressants, même s'il est vrai qu'avec le produit étudié il n'avait pas réussi à obtenir un bon dictionnaire. En effet,

¹Laboratoire IXL de l'université de Bordeaux

certaines intructions avaient le même signal et par conséquent il était difficile de les différencier.

Nous sommes dans une situation différente puisqu'un *bytecode* est en fait une séquence d'instructions assembleur qui constituent globalement un motif bien plus gros qu'une unique instruction assembleur. De plus, chaque *bytecode* correspond à une séquence d'instructions assembleur vraiment différente des autres *bytecodes* et c'est pourquoi leurs signatures devraient être également très différentes. C'est pourquoi nous croyons possible de construire un dictionnaire de *bytecodes* Java Card.

Nous avons donc mené des expériences sur des Java Cards au CESTI de SERMA Technologies en utilisant les méthodes d'aide à l'identification physique que nous venons de décrire pour développer une méthodologie visant à faire de la rétroconception d'applications officielles. Elles ont été réalisées sur des Java Cards de la famille des GemXpresso Pro publiquement disponibles et déjà certifiées au niveau EAL5+ des Critères Communs. Mais c'était certainement une mauvaise idée que de commencer nos expériences sur ces produits car nous disposions seulement de deux jours pleins pour les mener à bien sans support pour éventuellement enlever la grille. Comme nous avons déjà utilisé avec succès les attaques décrites dans la section 11.2.2 lors d'évaluations, nous avons voulu travailler directement sur les tests les plus intéressants mais aussi les plus difficiles : l'identification physique de *bytecodes*.

Hélas, nous avons échoué dans la construction d'un dictionnaire simple en utilisant les radiations électromagnétiques. La cause principale est que les puces récentes exigent des techniques très coûteuse pour réaliser une recherche complète et systématique (*e.g.* retrait du bouclier anti-radiation électromagnétique, besoin de temps pour obtenir des réglages de bonne qualité et reproductibles, etc.).

Évidemment, certaines attaques présentées ici ou des versions améliorées sont utilisées avec succès par le CESTI de SERMA Technologies pour tester et attaquer des plates-formes Java Card, des *applets* et leurs services lors des évaluations Critères Communs.

11.4 Travaux connexes

Des travaux visant à détailler les problèmes des cartes à puce multi-applicatives ont également été menés en 1999 par les équipes de Gemplus [118, 119, 117]. Ils soulevaient plusieurs problèmes relatifs au chargement, à la machine virtuelle, au partage d'objets et au flux d'information entre *applets* mais aucun n'identifiait les attaques physiques et les attaques par couplage que nous avons présentées en section 11.2.

Plus récemment, d'autres travaux [79, 190] sur les cartes multi-applicatives n'ont pas non plus identifié les problèmes que nous décrivons. Les travaux de Pierre Bieber *et al.* [79] proposent de permettre à l'émetteur de cartes de vérifier qu'une nouvelle *applet* interagira de façon sécurisée avec les autres *applets* déjà présentes sur la carte. Ceux de Gerhard Schellhorn *et al.* [190] présentent un modèle de sécurité générique pour les systèmes d'exploitation de cartes multi-applicatives qui formalise les principaux aspects de sécurité : confidentialité, intégrité, communication sécurisée inter-applications et chargement sécurisé de nouvelles applications.

Par ailleurs, nous n'avons trouvé aucun travaux connexes basés sur les méthodes d'identification physique pour caractériser une plate-forme.

11.5 Conclusion et perspectives

Nous avons expliqué ce que nous considérons comme une carte multi-applicative ouverte et les problèmes qu'elle engendre. Parmi tous les problèmes que nous présentons une partie n'a jamais été publiée jusqu'à maintenant (*i.e.* sections 11.1 and 11.2) et nous espérons que le partage de nos résultats aidera à sécuriser les prochaines cartes à puce ouvertes.

Par ailleurs, même si les problèmes que nous soulevons et si les contre-mesures que nous proposons semblent simples, il faut savoir que nous avons exploité ces vulnérabilités lors d'évaluations de produits réels pour mettre en place plus rapidement des attaques. Aussi, nous pensons que cela ouvre de nouveaux aspects de recherche et nous pensons dans le futur continuer nos expérimentations afin d'obtenir un dictionnaire de *bytecode*.

11.6 Méthodologie pour modifier un fichier CAP

Cette section détaille la méthode à utiliser pour construire une trace permettant l'identification des opérations en travaillant au niveau du *bytecode*.

- Tout d'abord, il faut écrire soit du code qui utilise le motif que l'on veut observer soit un code Java factice que l'on remplacera ultérieurement par le motif à observer. Dans ce dernier cas, la séquence factice devra générer une séquence de *bytecodes* assez large pour pouvoir facilement être remplacée par le motif à observer. Ainsi, pour observer le `sxor`, il faut, par exemple, écrire le code Java factice du listing 11.7.

Listing 11.7 – Code Java factice.

```
...
    apdu.setOutgoing();
    apdu.setOutgoingLength((short) 2);

    apdu.sendBytes((short) 0, (short) 1); // glitch 1
    // Introduire ici le motif intéressant à observer
    short s = (short) 0; // Code
    s ^= (short) 1; // factice
    apdu.sendBytes((short) 0, (short) 1); // glitch 2
...
```

- Il faut ensuite compiler et convertir ce code pour produire le fichier CAP. Ensuite, il faut utiliser l'outil *parseComponent* de la suite JCatools afin d'obtenir le composant Method dans un format compréhensible par un humain. La partie intéressante du code factice du fichier obtenu grâce au *parseComponent* est détaillée listing 11.8.
- Puis, il faut modifier la séquence générée pour améliorer la position des *glitches* ou pour remplacer la séquence factice par les bons *bytecodes*. Le listing 11.9 présente comment nous avons modifié le programme 11.8 pour entourer le *bytecode* `sxor` par des *glitches* avec une meilleure précision.

Listing 11.8 – *Bytecodes* générés par le code factice.

```

...
aload_1
sconst_0
sconst_1
invokevirtual    0x0 0xc // glitch 1
sconst_0        //
sstore_3        // Les
sload_3         // bytecodes
sconst_1        // factices
sxor            // générés
sstore_3        //
aload_1
sconst_0
sconst_1
invokevirtual    0x0 0xc // glitch 2
...

```

- Il faut également, modifier la taille maximum de la pile et le nombre maximum de variables locales pour la méthode concernée du composant.
- Ensuite, il faut utiliser l’outil *methodRewriter* disponible dans la suite JCatools pour reconstruire le composant Method dans le fichier CAP. Cet outil fera de lui-même les modifications des composants ReferenceLocation et Directory si cela est nécessaire.

Listing 11.9 – Motif à observer entouré par des *glitches*.

```

...
nop            // Le bytecode nop ne fait rien
nop            // et nous l'utilisons pour avoir la même
nop            // taille pour le tableau des bytecodes que
nop            // précédemment afin de simplifier le processus de modification.
aload_1
sconst_0
sconst_0       // ASTUCE : le résultat de sxor
sconst_1       // sera le troisième argument
aload_1
sconst_0
sconst_1
invokevirtual    0x0 0xc // glitch 1
sxor          // motif à observer
invokevirtual    0x0 0xc // glitch 2
...

```

Le fichier CAP résultant est alors toujours valide pour le vérifieur et pour la machine virtuelle défensive. Par conséquent, il peut donc être chargé sur la Java Card.

Le listing 11.9 montre que, dans certains cas, il est possible avec un peu d’astuce d’isoler un seul *bytecode*. Ici par exemple, nous utilisons une astuce pour éviter l’ajout d’un *bytecode* (e.g. **pop**, **sstore**, etc.) juste après le **sxor**. En effet, il devrait normalement y avoir un retrait de la pile de la valeur de type **short** résultant de l’opération de **sxor**. Pour empêcher cela, nous avons fait en sorte que cette valeur puisse servir à la génération du *glitch* comme le montre une analyse détaillée du listing.

Quatrième partie

Le projet Java Card Grid

Dans la dernière partie de cette thèse, nous exposerons comment l'association de certains travaux des divers membres de l'équipe « Systèmes et Objets Distribués » a permis l'émergence d'une solution visant à protéger des code mobiles pour faire du calcul distribué. En effet, un enjeu actuel essentiel dans le domaine du calcul distribué, et en particulier du calcul sur la grille, est celui de la sécurité. Dans cette partie, nous présenterons les grilles de calcul et les problèmes de sécurité associés. Puis, nous proposerons deux applications de démonstration décrivant comment nous avons solutionné ces problèmes en utilisant des cartes à puce multi-applicatives. Enfin, dans une partie plus prospective, nous décrirons les évolutions envisagées pour nos travaux.

Dans cette partie, notre contribution est la preuve de concept de l'architecture proposée, avec la mise en œuvre d'une infrastructure de gestion de matériels sécurisés (*i.e.* de cartes à puce) et de deux applications de démonstrations.

Chapitre 12

Contexte

Ces dernières années, avec l’augmentation du trafic sur le web et l’explosion du nombre de périphériques et des bases de données reliées les unes aux autres, les besoins en capacité de traitement informatique se sont accrues. Pour satisfaire les besoins de leurs clients, les sociétés doivent renforcer leur capacité de traitement ou acheter de la puissance de calcul selon le principe du *Grid* [113]. Ces concepts, inventés et utilisés depuis longtemps par les organismes institutionnels, avaient initialement pour but de mettre à la disposition des chercheurs la puissance de calcul indispensable à l’exécution d’algorithmes complexes nécessitant souvent un fort degré de parallélisme.

Par exemple, c’est ainsi que, pour accélérer le développement de nouvelles thérapeutiques, IBM, l’AFM (Association Française contre la Myopathie) et Génomining, une jeune société de bio-informatique située à Génopole [18] (localisé à Evry, au cœur de la région Ile-de-France), se sont associés, au sein du projet Décrypton [7], pour construire une grille au service de la recherche biologique. Dans cet exemple, grâce à 100000 ordinateurs particuliers contribuant chacun à 100 heures de calcul et où chaque participant a mis à la disposition des chercheurs la puissance inutilisée de sa machine, seulement quelques mois ont été nécessaires pour recueillir les résultats des calculs dans une base de données, là où avec un seul ordinateur, il aurait fallu 417000 jours de calcul, soit 1140 années. Aujourd’hui, grâce à sa puissance inégalée, le *Grid* fait de plus en plus d’adeptes. On assiste ainsi à une démultiplication des capacités de stockage et de traitement délocalisées.

Dans ce chapitre nous montrons les problèmes engendrés par la grille. Pour résoudre ces problèmes, nous nous proposons de développer une plate-forme sécurisée à base de cartes à puce qui sera décrite dans les prochains chapitres.

12.1 Objectifs

L’idée est simple et se base sur le constat suivant : les ordinateurs dispersés dans le monde entier ne sont pas utilisés au maximum de leurs possibilités. Pourquoi donc ne pas fédérer la puissance de calcul de tous ces systèmes, comme s’il s’agissait d’une seule et gigantesque machine, afin de répartir la charge applicative sur un réseau ? Par ailleurs, le *Grid* permet d’utiliser et de vendre à quiconque a besoin d’une puissance de

calcul massive, les moments d'inactivité de centaines ou de milliers de serveurs. Prenons le cas d'une grande banque, dont les ordinateurs ne fonctionnent pratiquement pas la nuit. Leur puissance de calcul pourrait donc être utilisée par une autre société située à l'autre bout du monde pour traiter différentes tâches. En contrepartie, la banque pourrait avoir accès à des capacités supplémentaires lorsqu'elle en a besoin. Les scénarii possibles sont nombreux (calculs, partage d'applications, etc.) et aucun *Grid* ne ressemble à un autre. S'il est séduisant, ce concept n'est simple qu'en apparence. Sa complexité tient à de nombreux facteurs, tels que la gestion du nombre de détenteurs de ressources ou du nombre de sites reliés. La solution doit donc permettre d'affecter des tâches à chacun des postes disponibles, de rassembler ensuite les résultats, et ce, bien entendu de manière sécurisée, car le *Grid* est une cible formidable pour les pirates.

12.2 Les différentes grilles

Lorsque l'on parle de grilles, toute une terminologie bien spécifique vient naturellement à l'esprit : *e.g. Grid Computing*, P2P (Peer-to-Peer, *i.e.* d'égal à égal), *Internet Computing*, *Métacomputing*, etc. En effet, il existe différentes sortes de grilles que l'on peut classer ainsi :

- les grilles d'information dont l'objectif est de partager la connaissance ;
- les grilles de données dont l'objectif est le stockage de données sur une grande échelle ;
- les grilles de calcul dont l'objectif est de réunir les puissances de calcul des différentes unités constitutives.

Ces différentes grilles sont mises en œuvre au travers de systèmes distribués. Or, selon la définition de A. Tanenbaum [197], un système distribué est un ensemble d'ordinateurs indépendants qui apparaissent à l'utilisateur du système comme un seul et même ordinateur. Ces systèmes distribués peuvent suivre deux modes de déploiement.

- Un mode client-serveur dans lequel l'architecture de contrôle du système peut soit être centralisée sur un seul serveur soit distribuée sur plusieurs serveurs. Afin d'éviter les congestions au niveau des serveurs, on utilise souvent des techniques de caches. Dans ce mode de déploiement, l'information est centralisée.
- Un mode d'égal à égal (P2P) dans lequel chaque machine est à la fois client et serveur. Il y a une distribution de la charge dans le réseau si la symétrie est respectée, *i.e.* si tous les noeuds ont les mêmes caractéristiques. Dans ce mode de déploiement, l'information est distribuée sur l'ensemble du système.

12.2.1 Modèle client-serveur

Nous présentons dans cette section des exemples des différentes grilles existant dans ce modèle.

La grille d'information

Des exemples typiques de ce genre de grilles sont les sites web. D'ailleurs il s'agit sans doute de la première incarnation du concept de grille. En effet, le client a accès à

l'information centralisée sur un serveur ou répartie sur plusieurs, à partir d'une adresse http ou d'un moteur de recherche. Par ailleurs, l'accès à l'information est transparent pour l'utilisateur qui ne sait pas toujours d'où elle provient.

La grille de données

On peut illustrer ces grilles par l'exemple particulier de Napster [43]. S'il est vrai que Napster se situe à la frontière du P2P et du modèle client-serveur, il est sans doute plus proche du fonctionnement de ce dernier. En effet, l'accès aux données se fait via un site unique qui partage l'index du contenu des différentes machines composant le réseau. Ainsi, même si les échanges se font ensuite entre les machines clientes du réseau, si le serveur hébergeant l'index est absent, le réseau ne peut plus fonctionner.

Ce type de grille est attaquable par les tribunaux comme on a pu le constater lors de la fermeture du site principal ordonné par la justice américaine. Mais plus grave, le site principal représente également une cible privilégiée pour les pirates souhaitant mettre la grille dans un état de panne.

La grille de calcul

Il existe plusieurs approches des grilles de calcul que nous décrivons ici.

L'Internet Computing

Son principe est basé sur l'utilisation des cycles processeurs inutilisés sur les machines (*i.e.* la moitié du temps en moyenne dans une entreprise selon une étude d'Omni Consulting Group). Ainsi, les utilisateurs des machines reliées à Internet peuvent télécharger et installer un programme se présentant la plupart du temps sous la forme d'un économiseur d'écran. Dès lors, quand la machine se mettra en veille celle-ci effectuera du calcul sans pénaliser l'utilisateur – ce dernier n'étant plus devant sa machine.

C'est le mode de fonctionnement des projets Seti@home [57], Décrypton [7] et RSA-155 [54], consistant respectivement à rechercher des signaux extra-terrestres, à établir une carte des 500000 protéines du vivant et à casser des codes cryptographiques.

Le Métacomputing

Ici, le principe est d'acheter du temps de calcul à des centres de calcul spécialisés reliés à Internet et louant l'utilisation de leur parc de calculateurs et des applications pré-installées. Il suffit à l'utilisateur d'envoyer son programme sur les machines du parc, de le lancer et d'attendre le résultat.

Des plates-formes proposant de tels services de calcul sont par exemple Netsolve [44] de l'université du Tennessee ou encore DIET [10] de l'ENS-Lyon/INRIA.

Le Grid Computing

Le principe du *Grid Computing* est de virtualiser l'ensemble des machines constituant la grille en un supercalculateur parallèle virtuel. Dans ce mode de fonctionnement, la

transparence pour l'utilisateur est totale et il peut ainsi exécuter facilement ses applications sur des ressources distantes.

Les exemples de plates-formes de *Grid Computing* les plus connus sont Globus [61], Legion [38] et Unicore [65].

12.2.2 Modèle d'égal à égal

Dans cette section, nous décrirons les différents types de grille existant pour ce modèle à l'aide d'illustrations.

La grille d'information

Une illustration des grilles d'information est le système de recherche décentralisée mis en place par Google pour retrouver les informations stockées sur son système de fichier GFS [62]. Un équivalent libre à GFS est apparu depuis peu : DRBS (Distributed Replicated Blob Server) [11].

La grille de données

Quelques exemples de grilles de données en mode d'égal à égal sont Gnutella [21], Freenet [60], Kazaa [36], JXTA [35]. Gnutella, Freenet et Kazaa sont des plate-formes bien connues pour échanger des fichiers entre internautes. Contrairement à Napster qui était relié à un serveur central, il n'existe pas de tel serveur pour Gnutella et chaque participant fait partie intégrante du réseau et ce réseau se modifie au cours des connexions et des déconnexions des utilisateurs. Dans un tel réseau, chaque nœud est indépendant et donc presque insensible aux pannes. JXTA propose quand à lui un ensemble de protocoles ouverts qui permettent à tout périphérique connecté sur un réseau (des téléphones cellulaires et PDAs jusqu'aux PCs et serveurs) de communiquer et de collaborer sur un réseau virtuel.

La grille de calcul

Dans le domaine du CGP2P (Calcul Global Pair à Pair), de nombreux travaux sont en cours et il existe encore peu de plates-formes. XtremWeb [51] est une des toutes premières plate-formes de ce type. Elle est en outre utilisée par l'ACI Grid CGP2P [1]. C'est une plate-forme de recherche *open source* utilisée pour étudier :

- le Calcul Global (extensibilité, organisation des serveurs, etc.) ;
- le Calcul Pair à Pair (volatilité, communication inter nœud, sécurité) ;
- la fusion des systèmes de Calcul Global et Pair à Pair

Elle a pour ambition d'être une plate-forme de recherche (*i.e.* étudier de nouveaux mécanismes) mais aussi une plate-forme de production (*i.e.* identifier les problèmes des utilisateurs).

12.3 Problèmes de sécurité sur les grilles

12.3.1 Sécurité du système vis à vis des applications et des infrastructures de calcul

Un premier problème de sécurité évident dans l'*Internet Computing* réside dans la confiance que l'on accorde au programme de calcul à installer sur la machine cliente. En effet, celui-ci pourrait contenir un virus ou un cheval de Troie. Une première solution se base sur l'utilisation de la technique du *sandbox* (*i.e.* bac à sable) dans lequel s'exécutera l'application téléchargée. Cette dernière n'aura accès qu'aux services autorisés par le propriétaire de la machine. Une seconde solution est de mettre le code de l'application en *open source* afin que l'utilisateur puisse vérifier son fonctionnement. Hélas, ce processus est lourd à réaliser puisqu'il nécessite de la part de l'utilisateur une bonne expertise dans l'analyse du code, du temps, mais aussi dans les techniques de développement d'une application (*e.g.* compilation, etc.). Un tel procédé de distribution d'applications pour l'*Internet Computing* ne peut que repousser bon nombre d'utilisateurs. Une troisième solution consiste à laisser faire le travail d'expertise du code par une autorité en laquelle l'utilisateur pourra avoir confiance. Une fois son expertise terminée, cette autorité peut signer l'application afin que l'utilisateur vérifie son intégrité et par la-même son innocuité.

Les mêmes problèmes se posent dans le cadre du *Grid Computing* et du *Métacomputing* et on emploiera donc une des solutions évoquées ci-dessus.

Par ailleurs, dans le cadre du *Grid Computing*, se pose également la question de la confiance à accorder à l'infrastructure constituant le supercalculateur virtuel. En effet, les utilisateurs désirant participer au calculateur virtuel doivent installer des logiciels de support de l'infrastructure (*e.g.* Globus [61]) sur leur machine. Une fois encore, rien ne leur garantit la sécurité de celle-ci. Par conséquent, on utilisera aussi une des solutions évoquées ci-dessus.

Nous pensons que les cartes sont une solution car leur système d'exploitation sécurisé est évalué et certifié par des tiers indépendants qui sont garants de sa sécurité. De plus, les cartes à puce mettent en œuvre des mécanismes sécurisés de chargement des applications. En outre, les applications s'exécutent au sein d'un environnement cloisonné depuis lequel elles ne peuvent pas mettre en danger la sécurité de la carte et des autres applications présentes.

12.3.2 Confidentialité du code de l'application

Un second problème dans le cadre des grilles de calcul, quelles soient de type *Internet Computing*, *Métacomputing* ou *Grid Computing*, est celui de la confidentialité du code de l'application. Les propriétaires de codes peuvent ne pas vouloir dévoiler aux propriétaires des ressources de calcul ce que font leurs codes. Par exemple, le projet Décrypton ne distribue le code de son économiseur d'écran que sous forme de binaire afin d'éviter que des pirates l'étudient et envoient de faux résultats (faux positifs mais également faux négatifs). Dans les cas où la confidentialité du code de l'application sera requise, il sera donc impossible d'appliquer la solution *open source* proposée plus haut pour assurer la sécurité du système.

Par ailleurs, dans le cadre de l'*Internet Computing*, du *Métacomputing* et du *Grid Computing*, il sera toujours possible de mettre en péril la confidentialité du code. Par exemple, dans le cadre de l'*Internet Computing* et du *Grid Computing*, les pirates pourront toujours faire la rétro-conception du code binaire. Dans le cadre du *Métacomputing*, le même type d'attaques pourra s'appliquer sauf qu'il ne s'agira plus d'un pirate, mais d'un administrateur indélicat. En effet, comme ce dernier a accès aux différentes machines, il a également accès au binaire de l'application.

Comme ces exemples le prouvent, la confidentialité du code de l'application est très difficile à garantir. Souvent, l'hypothèse faite est de faire confiance au propriétaire des ressources de calcul, tout spécialement dans le cadre du *Métacomputing*. Il suffit alors de mettre en place un mécanisme d'authentification mutuelle du propriétaire des ressources de calcul et du propriétaire du code pour assurer cette confiance. Seulement, s'il est possible d'avoir confiance dans le propriétaire des ressources de calcul (*i.e.* être sûr que l'administrateur ne tentera rien pour découvrir le code), rien ne pourra jamais garantir qu'une personne malhonnête ne puisse avoir physiquement accès aux ressources de calcul pour récupérer le binaire de l'application (par exemple, par un mécanisme d'observation au niveau matériel).

Il est donc quasiment impossible, dans le cadre d'une grille de calcul, de garantir la confidentialité du code.

Nous pensons que les cartes sont une solution car, une fois l'application chargée, il est pratiquement impossible de déterminer son code. Le binaire est inaccessible même si on possède un accès physique à la carte grâce aux protections qu'elle met en œuvre.

12.3.3 Sécurité de l'exécution de l'application

Un troisième problème que l'on retrouve dans tous les types de grilles de calcul est celui de la sécurité de l'exécution. En effet, les propriétaires des codes de calcul souhaitent que leurs applications s'exécutent et ne soient pas perturbées ou modifiées. Par exemple, dans le cas de l'*Internet Computing*, il arrive fréquemment que des pirates s'amuse à modifier les résultats de l'exécution de l'application, soit postérieurement au calcul, soit en modifiant l'application. Ainsi, le projet Seti@home [57] reçoit de temps à autres des faux résultats positifs de pirates affirmant la découverte de signaux extra-terrestre. Si ces cas sont faciles à gérer, grâce à une simple procédure de recalcul à partir des données envoyées à l'utilisateur, c'est parce qu'ils sont peu nombreux et que les coordinateurs du projet ont assez de puissance de calcul locale à leur disposition. En revanche, les faux résultats négatifs sont, eux, plus délicats à traiter. En effet, les coordinateurs du projet ne peuvent pas vérifier tous les résultats négatifs localement sinon cela signifierait qu'ils disposent d'assez de puissance de calcul pour faire tous les calculs eux-mêmes. C'est pourquoi au sein du projet Seti@home les mêmes calculs sont réalisés par plusieurs utilisateurs différents afin de recouper les informations et d'être certains qu'aucun pirate n'a modifié l'exécution de l'application pour cacher (*sic.*) l'éventuelle existence d'extra-terrestres.

Par ailleurs, il existe beaucoup d'attaques contre l'exécution du code d'une application dans le but de lui faire retourner des résultats invalides, ou de découvrir de l'information sur son fonctionnement. Ce sont des problèmes classiques que nous avons déjà présentés

dans le cadre des attaques contre les cartes à puce à la section 9.1.3.

Nous pensons que les cartes sont une solution car l'exécution du code des applications y est protégée par les protections physiques qui rendent presque impossible la génération de faux résultats. De plus, les nombreux détecteurs installés sur les cartes permettent de détecter de telles attaques et de bloquer définitivement l'application si nécessaire.

12.3.4 Sécurité des communications

Enfin, un problème classique du calcul distribué est celui de la sécurité des communications. Cette sécurité concerne principalement la confidentialité et l'intégrité des messages échangés. Or, dans ce domaine, un grand nombre de systèmes existent et fonctionnent très bien sous l'hypothèse que l'on a confiance dans le propriétaire des ressources de calcul et dans la sécurité qu'il met en place. Cependant, dans le cadre d'un système à base de chiffrement asymétrique, par exemple, les clés privées déployées sur les ressources devraient se trouver dans des matériels sécurisés et non accessibles à un éventuel attaquant. Ceci est rarement le cas et donc il est possible de mettre en place des attaques pour récupérer ces clés privées.

Nous pensons que les cartes sont une solution car leur matériel sécurisé permet de conserver les clés privées véritablement secrètes.

12.4 Exemple d'un système de sécurité existant

Comme on vient de le voir, il semblerait que la sécurité et la confiance se prêtent mal à l'approche distribuée d'un ordinateur virtuel. Par exemple, une personne appartenant à une université française peut démarrer, sur le serveur d'une société finlandaise, un calcul qui accédera aux ressources d'une dizaine de serveurs dans des universités ou des entreprises à travers le monde. Or, toutes ces entités appartiennent à des domaines différents et sont soumises à des contraintes de sécurité également différentes. Assurer la sécurité dans ces conditions peut sembler utopique. Toutefois, le chiffrement asymétrique offre une solution.

Ainsi, l'initiative Grid Security Infrastructure (GSI) [22] du projet Globus [61] décrit une architecture de sécurité de type PKI qui authentifie les serveurs, les utilisateurs et les processus au niveau du Grid, et s'adapte aussi aux modèles locaux de sécurité. Mieux, elle permet aux processus de s'authentifier eux-mêmes lorsqu'ils ont besoin d'accéder à des ressources distantes, afin d'éviter à leur propriétaire de rester en ligne pour la durée des calculs. À la base de la solution se trouve un certificat numérique X.509 [67] standard, délivré à chaque utilisateur et à chaque machine du Grid. Bien que spécifique au Grid Globus, celui-ci doit être signé par une autorité de certification reconnue (le Global Grid Forum envisage parallèlement l'établissement d'un gridCA - Autorités de Certification distribuées). Ce certificat contient le nom unique de l'utilisateur ou du service pour lequel il a été délivré. Sur le Grid, chaque ordinateur dispose de tables de correspondance entre ces noms uniques (DN) et les utilisateurs locaux. Les connexions sont alors établies par SSL et l'authentification réalisée sur la foi du certificat numérique. Le système sollicité recherche dans sa table de correspondance à quelle personne appartient le DN présenté par le certificat. Puis, il procède à l'authentification afin d'autoriser le lancement d'un

processus.

En revanche, comme l'illustre cet exemple, le système de sécurité ne prend pas en compte la sécurité au niveau matériel et il est ainsi possible de réaliser des attaques sur la confidentialité du code des applications distribuées et de leurs exécutions pour peu que l'on puisse obtenir un accès physique au matériel.

C'est dans ce contexte que nous allons proposer au chapitre suivant des solutions pour sécuriser le calcul sur la grille. Notre approche repose sur l'utilisation de cartes à puce.

Chapitre 13

Le projet Java Card Grid

Comme nous l'avons vu, une des motivations principales pour mettre en place une grille ou interconnecter des ressources, est de permettre à des personnes ou à des entreprises d'utiliser des unités de calcul mises à disposition par une tierce partie. Le problème majeur est que l'utilisateur de la grille doit faire confiance aux propriétaires des ressources de calcul qui exécuteront son code. Même si certaines sécurités existent au niveau logiciel, rien ne peut empêcher les opérations malveillantes dues à un accès physique à l'unité de calcul : les données confidentielles peuvent être espionnées, les calculs peuvent être perturbés et les résultats peuvent être modifiés.

Toutefois, nous pensons que l'utilisation de cartes à puce permettra d'éviter ces problèmes. En effet, les protections physiques qu'elles possèdent assurent, à l'inverse des processeurs classiques, l'impossibilité de comprendre, dans un temps raisonnable, ce qui est stocké à l'intérieur ou ce qu'il se passe en interne. Par ailleurs, on sait que la puissance des cartes ne cessant d'augmenter, ces dernières pourront à terme faire du calcul, même s'il est bien évident que l'on ne pourra pas parler de haute performance.

Pour résoudre ces problèmes de sécurité, nous proposons une grille de Java Cards destinée au déploiement d'applications distribuées pour les utilisateurs ayant besoin d'effectuer des calculs dans des environnements où il est impossible de faire confiance aux propriétaires des ressources de calcul.

13.1 Présentation

En se basant sur la fonctionnalité multi-applicative des Java Cards et sur notre expérience dans le calcul distribué, nous avons mis en place un cluster de cartes à puce et une infrastructure logicielle pour développer et gérer des applications sécurisées sur ce matériel [86, 87, 89].

À terme, cette infrastructure logicielle proposera des APIs pour le développeur et des outils pour l'utilisateur final et l'administrateur. Non obstant du fait qu'elle aurait pu s'appuyer sur des systèmes pré-existants pour le calcul distribué, comme RMI [133], JavaParty [181], JiniCard [148], Jason [80], OrbCard [85], nous avons choisi d'utiliser Mandala [92]. Mandala est une infrastructure généraliste pour le calcul distribué qui a été développée

dans l'équipe Systèmes et Objets Distribués du LaBRI. Il fournit entre autres une abstraction pour l'appel de méthodes à distance. Il possède également des caractéristiques, que nous pensons utiles dans le contexte de notre projet, comme le concept de *conteneur actif* sur lequel il se base ou l'*asynchronisme* qu'il procure pour les appels de méthodes à distance.

L'infrastructure matérielle que nous avons envisagée est présentée figure 89. Il s'agit d'un ensemble de cartes à puce formant une sorte de grille ou plus précisément un cluster. Ces cartes sont alimentées en énergie au travers de lecteurs reliés entre eux par des *hubs* USB qui peuvent être connectés sur différentes machines.

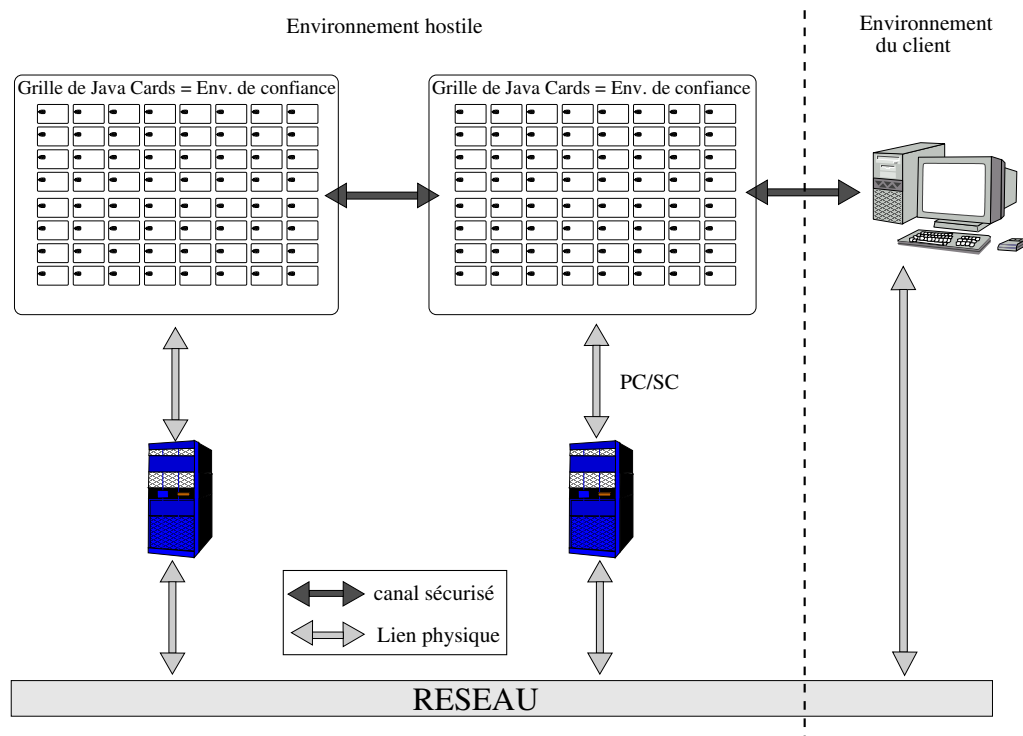


FIG. 89 – Une solution pour le calcul avec des Java Cards.

13.2 Infrastructure

Elle a été développée par Achraf Karray¹ et moi-même avec le concours des membres de l'équipe Systèmes et Objets Distribués.

13.2.1 Infrastructure matérielle

Actuellement, l'infrastructure matérielle de notre projet est constituée de 2 PCs, 9 lecteurs USB CCID, 2 hubs USB auto-alimentés avec chacun 7 connecteurs. Nous pouvons

¹Étudiant en master à l'ENIS de l'université de Sfax ayant effectué son stage au LaBRI.

donc insérer 9 Java Cards sur lesquelles nous distribuons nos calculs. Aujourd'hui, notre plate-forme de démonstration ressemble à la photo 90.



FIG. 90 – La grille de Java Cards.

Avant la fin du mois novembre 2004, nous devrions recevoir une armoire *rackable* de 20 unités destinée à accueillir :

- 2 PCs (2 unités pour chacun) dédiés pour gérer notre infrastructure logicielle ;
- 4 racks (2 unités pour chacun) contenant chacun 8 lecteurs USB CCID (soit 32 lecteurs au total) ;
- 1 rack (2 unités) contenant 6 *hubs* USB auto-alimentés avec 7 connecteurs pour relier les lecteurs aux PC.

13.2.2 Infrastructure logicielle

Actuellement, les PCs fonctionnent sous un environnement GNU/Linux, mais cela devrait théoriquement fonctionner tout aussi bien sous les systèmes de Microsoft. Par ailleurs, aujourd'hui, la couche haute de l'infrastructure logicielle s'appuie sur le *framework* Mandala développé dans l'équipe Systèmes et Objets Distribués. Or, comme ce *framework* est développé en Java, nous avons utilisé JPC/SC [31], un *wrapper* JNI de PC/SC, pour contrôler les lecteurs et ainsi déployer les applications sur les cartes (*cf.* FIG. 91). Comme nous l'avons vu au chapitre 4, PC/SC [49] est un standard qui fournit une API de haut niveau pour communiquer avec les cartes à puce.

Si nous avons choisi JPC/SC plutôt que OCF [47], c'est principalement parce qu'il est plus facile à utiliser mais également parce que nous maîtrisons plus en profondeur la technologie puisque nous avons contribué au développement du projet *pcsc-lite* [48]. De plus, OCF n'est plus maintenu et ne gère pas autant de modèles de lecteurs que PC/SC. Bien sûr, une solution basée sur OCF et le pont OCF/PCSC aurait été possible pour gérer autant de modèles que PC/SC mais nous n'avons pas souhaité ajouter une couche supplémentaire dans une architecture logicielle déjà assez complexe.

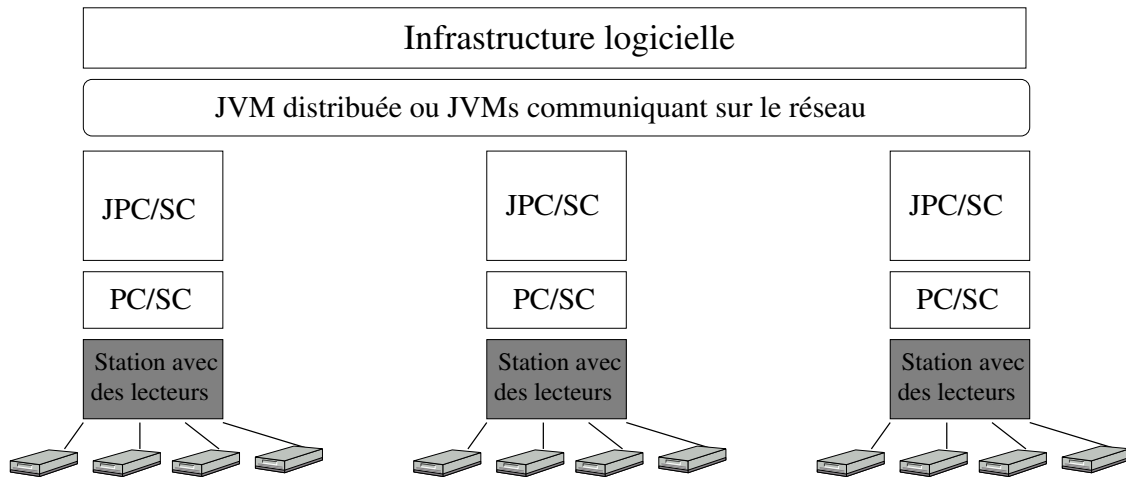


FIG. 91 – Infrastructure logicielle.

Comme nous l'avons vu à la section 4.2, au sein de l'architecture PC/SC, chaque lecteur communique avec le *middleware* PC/SC via un pilote (*cf.* FIG. 92). L'utilisation de lecteurs

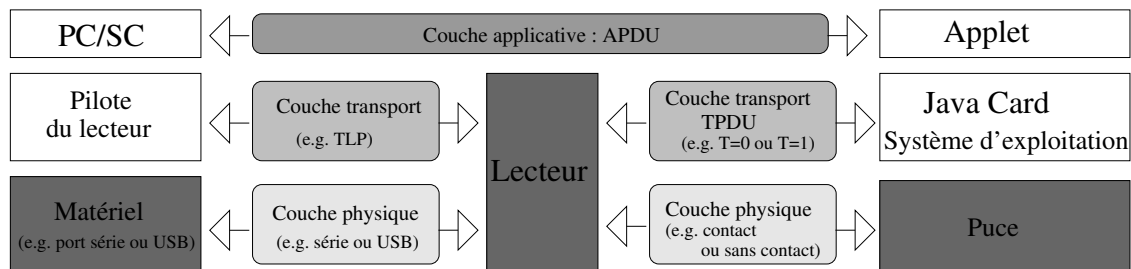


FIG. 92 – La solution PC/SC.

compatibles PC/SC nous permet d'utiliser des lecteurs de différents vendeurs sans avoir à connaître les différents protocoles sous-jacents. Certains lecteurs utilisent des pilotes propriétaires, mais nous avons choisi de n'utiliser que des lecteurs utilisant des pilotes *open source* afin de pouvoir corriger nous-même rapidement les éventuels *bugs*. Plus précisément, nous utilisons uniquement des lecteurs compatibles CCID, car un pilote *open source* est disponible au sein du projet `pcsc-lite` et car ce standard nous permet d'utiliser le même pilote pour accéder aux différents lecteurs CCID de différents fabricants.

Bien évidemment, la dernière brique logicielle pour interagir avec les Java Cards consiste à implanter des *applets* qui contiennent le code métier, puis les installer sur les cartes. Ensuite, il faut développer un proxy qui récupérera les appels en provenance de l'application sur le PC et qui les traduira en appel à destination des *applets* sur les cartes.

13.3 Les défis

Si mettre en place une infrastructure à base de cartes à puce pour faire du calcul distribué peut sembler assez simple, certains défis restaient à résoudre. Tout d'abord, la carte possédant des mécanismes de communication assez primitifs, les APDUs, nous avons voulu mettre en place une solution plus transparente. Ensuite, les cartes à puce étant par nature des serveurs, il nous fallait développer une solution technique nous permettant de la rendre proactive. Par ailleurs, la sécurité de notre grille reposait sur du matériel sécurisé (*i.e.* les cartes à puce) mais pour obtenir une sécurité totale, les communications entre les entités de la grille devaient également être sécurisées. Au même titre, la communication entre la grille et l'utilisateur devait aussi être sécurisée.

Nous allons donc présenter dans cette section les solutions que nous avons proposées pour résoudre ces trois problèmes.

13.3.1 L'appel de méthodes à distance

Un des freins à l'utilisation massive de cartes à puce dans les systèmes distribués est probablement leur interfaçage. En effet, jusqu'à très récemment, le programmeur était obligé de passer par les APDUs. Aujourd'hui, s'il est vrai que la dernière version des spécifications Java Card propose le mécanisme d'invocation de méthodes à distance Java Card (JCRCMI), très peu de produits disponibles l'implémentent. Nous souhaitons cependant acquérir au plus vite ce genre de produit car, grâce à ce mécanisme, le développement des applications est simplifié et par conséquent beaucoup plus rapide qu'en utilisant les APDUs.

En attendant, afin de mieux incorporer les Java Cards dans les systèmes distribués, nous avons défini, côté carte, notre propre mécanisme pour répondre à l'invocation de méthodes distantes.

Dans notre modèle, une *applet* sur la carte se voit associer une classe squelette appelée **Skeleton** traduisant les appels de méthodes en provenance des applications extérieures vers les méthodes de l'*applet*. Le *Skeleton* représente donc une couche logicielle permettant d'interpréter les invocations de méthodes de façon transparente pour le programmeur d'*applets*, *i.e.* il lui cache les détails du code qui réalise le mécanisme de traduction des APDUs en appel de méthode. Nous avons développé un outil permettant de générer automatiquement la classe **Skeleton** à partir d'une interface qui représente l'*applet* et qui définit les méthodes qui peuvent être invoquées à distance. Le programmeur d'*applets* peut ainsi se concentrer sur l'écriture du code métier de son application et ne pas perdre son temps sur la traduction des APDUs en appels aux méthodes concernées. Évidemment, pour être reconnues par la classe **Skeleton**, il est nécessaire que les requêtes APDUs respectent un format commun. Nous ne le décrivons pas en détails ici mais sachez qu'il suit le format TV (Tag, Value) pour les arguments et contient une valeur pour indiquer la méthode à appeler et une autre pour indiquer le nombre de paramètres. Comme l'illustre la figure 93, quand une *applet* reçoit une APDU via sa méthode **process**, elle le transmet à la méthode **invoke** de la classe **Skeleton** qui réalise la vérification du format de la requête et appelle ensuite la bonne méthode avec les bons arguments remis en forme grâce à l'utilisation

d'une bibliothèque de reconstruction de type nommée **ParseAndBuildParameters**.

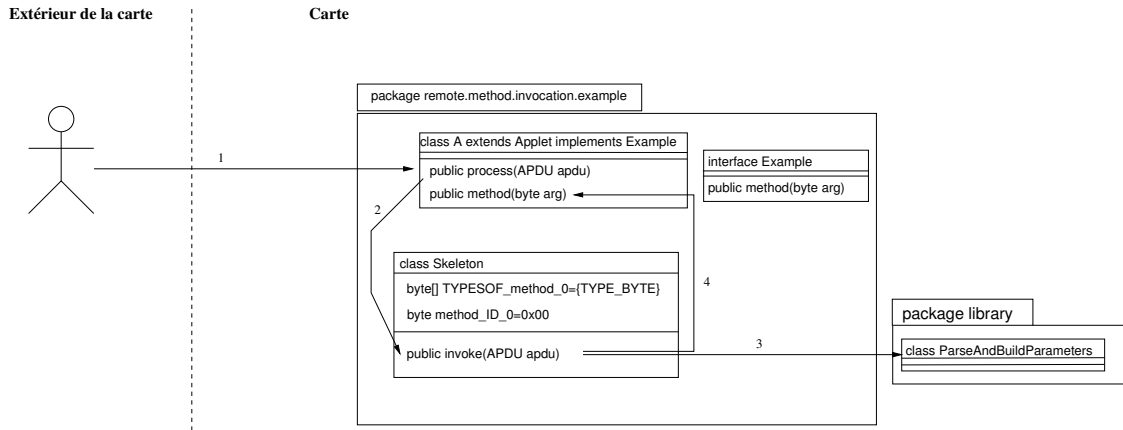


FIG. 93 – Mécanisme d'invocation d'une méthode.

Aujourd'hui, ce mécanisme est seulement à l'état de test et il n'est pas encore intégré à notre infrastructure. D'ailleurs, il ne le sera peut-être jamais puisque les Java Cards proposant JCRMI permettent de faire au moins la même chose et même mieux.

Par ailleurs, on notera que l'apparition de JCRMI nous incitera peut-être à revoir nos choix en matière de communication avec le lecteur. En effet, depuis peu, Sun microsystems propose un service OCF pour communiquer de façon transparente avec des Java Cards supportant JCRMI. Nous pourrions donc remplacer JPC/SC par le couple OCF et OCF/PCSC mais sous la condition de n'utiliser que des Java Cards récentes.

13.3.2 La proactivité des cartes

Les cartes sont par essence des périphériques de nature passive, *i.e.* serveur. Or, nous pensons que les applications utilisant notre infrastructure pourraient avoir besoin d'un modèle où la carte serait proactive, *i.e.* capable d'émettre une requête et donc d'appeler un service présent sur une autre carte. Dans ce modèle d'exécution, la carte ne serait donc plus seulement serveur, mais également client.

Pour pallier cette lacune, nous avons donc mis en place le mécanisme simple consistant à interroger la carte pour savoir si elle désire émettre une requête. Pour cela, comme l'illustre la figure 94, nous envoyons une commande APDU et si la carte souhaite émettre une requête, elle la place dans une réponse APDU. C'est un mécanisme identique qui est utilisé dans les cartes SIM des téléphones portables [147, 136].

En revanche, les cartes étant de nature serveur, elles n'exécutent le code de l'application qu'entre la commande APDU et la réponse APDU. Suivant notre modèle, si une carte demande un service dans une réponse APDU, elle arrête l'exécution du code de l'application entre temps. Il s'agit donc d'un modèle d'appel de méthode synchrone qui peut être gênant pour certaines applications.

Par ailleurs, se pose le problème de la reprise de code. En effet, Java Card suit un modèle d'exécution où toutes les commandes APDUs sont traitées par l'environnement d'exécution

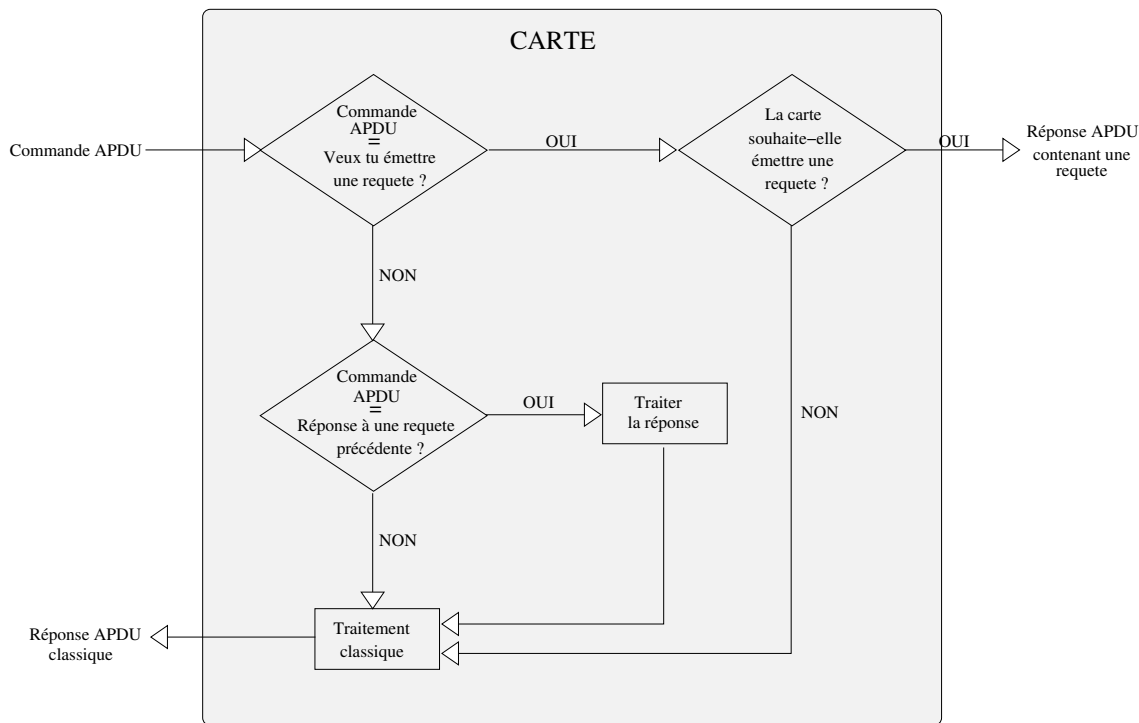


FIG. 94 – Modèle d'exécution d'une carte proactive.

(JCRE) avant d'être envoyées à la méthode `process` de l'*applet*, point d'entrée unique d'une application Java Card (cf. FIG. 95). De la même façon, toutes les réponses APDUs de l'*applet* sont passées à l'environnement d'exécution qui les renvoie immédiatement sur l'interface de communication. En revanche, il n'enverra le *Status Word* (SW) indiquant le statut de l'exécution (*i.e.* bonne exécution ou éventuelles erreurs) qu'à la fin de l'exécution de la méthode `process`. Ainsi, entre une réponse APDU et la commande APDU suivante, c'est l'environnement d'exécution qui est actif. Par ailleurs, comme il est impossible de recevoir deux commandes APDUs à la suite. Une *applet* ne peut donc pas demander à effectuer un service à l'extérieur puis se mettre en attente d'une réponse de ce service qui serait une deuxième commande APDU. C'est pourquoi, nous avons également dû mettre en place un mécanisme de reprise de code dans l'application pour rendre la carte proactive.

Nous avons utilisé avec succès ce mécanisme de cartes proactives dans l'application FBI – Air France que nous décrirons plus loin.

Appel de méthodes entre cartes

Suite à une commande APDU la réponse envoyée par la carte est traitée par un PC qui joue le rôle d'un routeur. En effet, si la réponse reçue par le PC est dans un format particulier, que nous ne détaillons pas ici (car inutile à la compréhension et susceptible d'évoluer), il s'agit d'une demande d'invocation de méthode sur une autre carte. Toutefois, on notera que la réponse envoyée par la carte cliente doit contenir toutes les informations

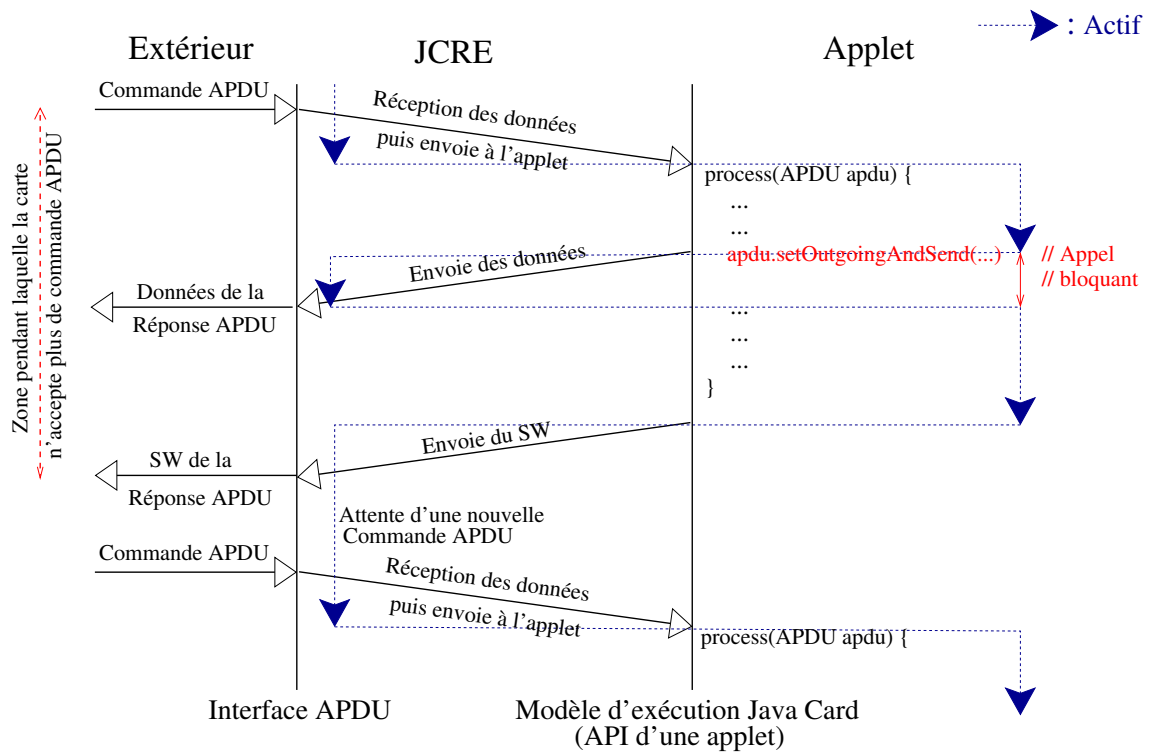


FIG. 95 – Modèle d'exécution classique d'une Java Card.

qui permettent au PC de déterminer l'application et la carte concernée par cette demande d'invocation. En effet, pour faire la distinction entre les cartes, nous avons implanté une *applet* Java Card permettant de nommer, de renommer ou de récupérer le nom d'une carte. Muni de ces informations, le PC se charge d'envoyer vers la carte cible la commande APDU encapsulée dans la réponse APDU de la carte cliente.

Par ailleurs, au niveau de la carte cliente, nous avons mis en place une classe souche appelée *Stub*. Cette classe permet de construire plus facilement les réponses APDUs qui sont envoyées à la machine intermédiaire. Le *Stub* représente donc une couche logicielle permettant de réaliser le mécanisme d'invocation d'une façon transparente pour le programmeur (cf. FIG. 97). Comme c'est le cas de la classe *Skeleton*, la classe *Stub* est générée automatiquement à partir (principalement) de l'interface qui représente le service de l'*applet* sur la carte serveur, du nom de cette dernière et de l'AID de l'*applet*.

La figure 96 illustre le mécanisme de l'invocation des méthodes, sur une carte, depuis une autre carte. Un utilisateur envoie via le réseau ouvert Internet, une commande APDU

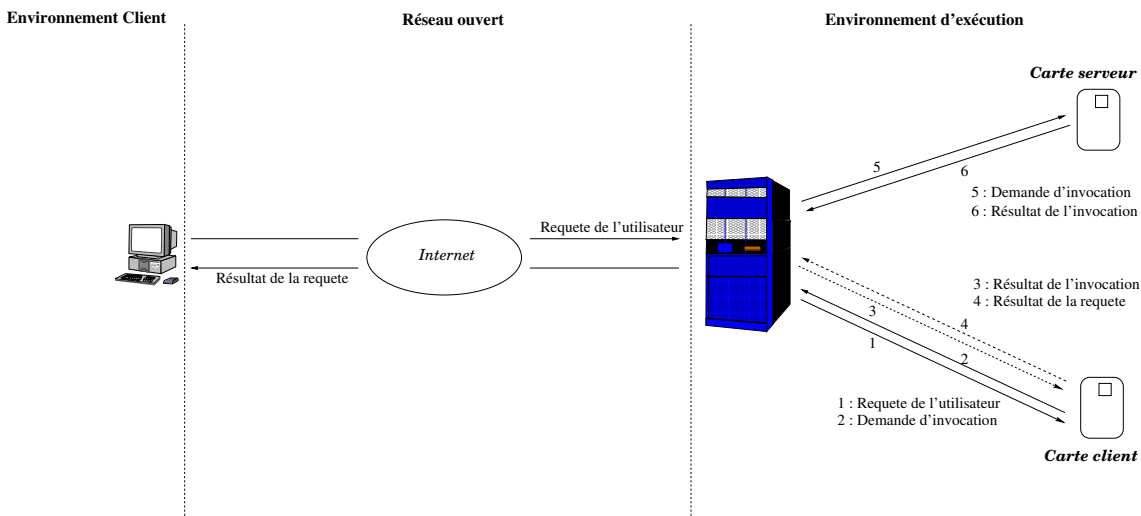


FIG. 96 – Invocation de méthode depuis une autre carte.

à la carte **client** par l'intermédiaire de la machine qui relie les différentes cartes à puce. Si l'exécution de la commande de l'utilisateur nécessite l'invocation d'une méthode distante sur une autre carte (*i.e.* carte **serveur**), la carte **client**, grâce à la classe *Stub*, génère une réponse APDU dans un format particulier qui encapsule la commande APDU destinée à l'appel de la méthode distante. En examinant le format de la réponse APDU, la machine intermédiaire constate qu'il s'agit d'une réponse contenant une commande destinée à une autre carte. Par conséquent, il analyse la réponse APDU afin de déterminer le nom de la carte cible et l'AID de l'*applet* appropriée. Puis, après avoir sélectionné l'*applet* sur la carte **serveur**, le PC intermédiaire lui envoie la commande APDU contenue dans la réponse de la carte **client**. La carte **serveur** traite alors la commande reçue grâce à la classe *Skeleton* comme présenté figure 97. Enfin, le PC intermédiaire récupère la réponse de la carte **serveur** et l'envoie à la carte **client**. Ainsi, cette dernière peut continuer son exécution jusqu'à la fin, où elle envoie la réponse à la requête de l'utilisateur via la machine

intermédiaire.

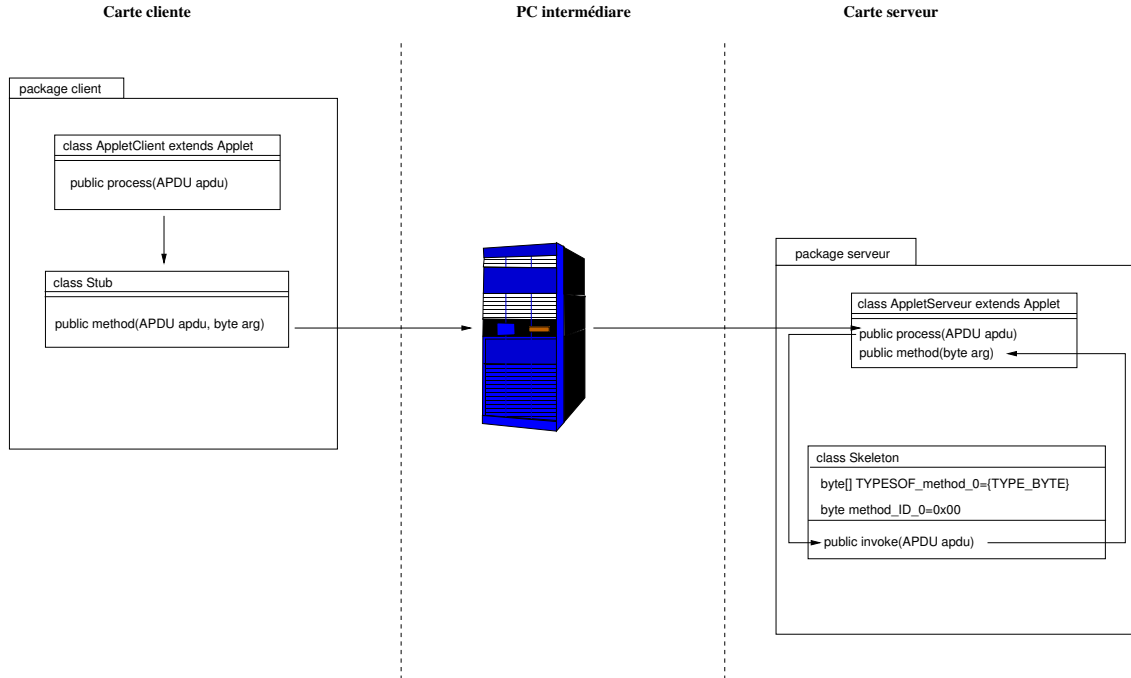


FIG. 97 – Détails de l’invocation de méthode depuis une autre carte.

Reprise de code

Comme nous l’avons déjà souligné, du fait de la passivité de la carte, nous sommes confrontés au problème de reprise de code sur la carte `client` après avoir reçu la réponse de la carte `serveur`. Autrement dit, pour pouvoir continuer son exécution normalement, la carte `client` doit se souvenir de l’endroit où elle s’est arrêtée avant d’invoquer la méthode sur la carte `serveur`.

Nous avons mis en place une solution se basant sur la construction `switch` en distinguant les deux phases : avant et après l’invocation comme le montre le code du listing 13.1.

13.3.3 Le canal sécurisé

Afin d’assurer la sécurité des communication entre les diverses entités de notre plateforme, nous devons créer des canaux sécurisés. Naturellement, puisque notre cible de calcul était la Java Card, nous avons immédiatement pensé à utiliser les mécanismes de GlobalPlatform pour créer nos canaux sécurisés. En effet, ces mécanismes sont présents en standard sur quasiment tous les modèles de Java Cards. De plus, ils sont implantés pour certaines parties en natif et permettent donc d’obtenir de meilleures performances. Toutefois, comme nous l’avons vu au chapitre 3, un préalable à l’établissement d’un canal sécurisé est une phase d’authentification mutuelle. Or, si cette authentification est adaptée pour sécuriser la communication entre un utilisateur et une carte (à laquelle il a un accès

Listing 13.1 – La reprise de code.

```

// b est une variable globale initialement initialisée à 0

switch(b)
  case 0x00 :                                // Point d'entrée classique : AVANT

      instruction 1                          // Instructions
      ...                                    // à effectuer
      instruction n                          // avant l'appel au service

      appel de la classe stub               // Formatage de la requête
                                          // et envoi à l'environnement d'exécution

      incrementation de b                   // Mémorisation d'un appel en cours

      break;                               // Pour sortir de la méthode process
                                          // et envoi effectif sur l'interface

  case 0x01 :                                // Point de reprise du code : APRÈS

      recuperation du resultat              // Résultat de la requête

      instruction n+1                       // Instructions
      ...                                    // à effectuer
      instruction n+m                       // après l'appel au service

      remise de b à 0                       // Mémorisation que le traitement est fini

      break;                               // Pour sortir de la méthode process
                                          // et envoyer la réponse APDU

```

physique – cette propriété permet d'être certain qu'il s'agit bien d'une carte), elle ne l'est pas dans le cadre de notre grille comme nous allons le démontrer.

Pour mémoire, cette étape consiste à prouver l'identité de chaque protagoniste en établissant des échanges à base de challenge/réponse prouvant qu'ils partagent le même secret et donc qu'ils peuvent se faire confiance. Par ailleurs, lors de cette procédure, les entités communicantes génèrent des clés de session qui serviront à assurer l'intégrité et la confidentialité du canal sécurisé pour les échanges suivants.

Dans GlobalPlatform, le secret commun initial que partagent les interlocuteurs (*i.e.* la carte et l'utilisateur) sont des clés statiques qui sont dérivées en clés de session. Ces dernières seront ensuite utilisées comme clés de chiffrement/déchiffrement dans des algorithmes symétriques. Du moins, c'est l'implantation de référence qui est suggérée dans GlobalPlatform pour l'authentification mutuelle. En effet, rappelons que si GlobalPlatform n'impose aucun algorithme particulier, il propose une implantation de référence (*i.e.* les algorithmes et les procédures) pour les mécanismes :

- d'authentification mutuelle nécessaire à l'établissement du canal sécurisé ;
- d'intégrité du canal sécurisé ;
- de confidentialité du canal sécurisé.

Ainsi, pour des raisons d'interopérabilité, toutes les Java Cards implantent ces mécanismes suggérés dans l'implantation de référence. Or, la sécurité d'un algorithme à clé symétrique repose intégralement sur la non-divulgateion de la clé : si celle-ci est dévoilée, n'importe qui peut chiffrer ou déchiffrer les messages. Aussi, si cette procédure semble adéquate

pour une communication où la carte est reliée directement à la machine de son utilisateur, elle pose problème dans le cadre d'une grille de cartes à puce faisant intervenir plusieurs clients à travers un réseau ouvert. En effet, étant donné que la génération des clés de session repose sur des clés statiques connues par plusieurs utilisateurs et sur des données échangées en clair, un intrus (*i.e.* un utilisateur ou un administrateur mal-intentionné de la grille), connaissant les clés statiques et écoutant une communication depuis son début, pourra générer les clés de session et donc déchiffrer les commandes APDUs – voire les modifier. Plus grave encore, un client mal-intentionné, connaissant les clés statiques, peut également se faire passer pour une carte d'une façon totalement transparente auprès d'un client honnête communiquant via un réseau ouvert. En effet, ce dernier croit communiquer avec une carte alors qu'en réalité il s'agit d'un autre utilisateur qui usurpe l'identité d'une carte pour récupérer, par exemple, le code ou les données de l'application que l'utilisateur envoie.

Ces limites découlent de l'utilisation des clés symétriques diffusées et de l'échange en clair de données qui servent pour authentifier la carte et pour générer les clés de session.

C'est pourquoi, nous avons proposé un autre moyen permettant d'identifier la carte de façon formelle et garantissant, par la suite, que les commandes APDUs ne pourront être déchiffrées que par la carte – et non par un utilisateur mal-intentionné usurpant l'identité d'une carte.

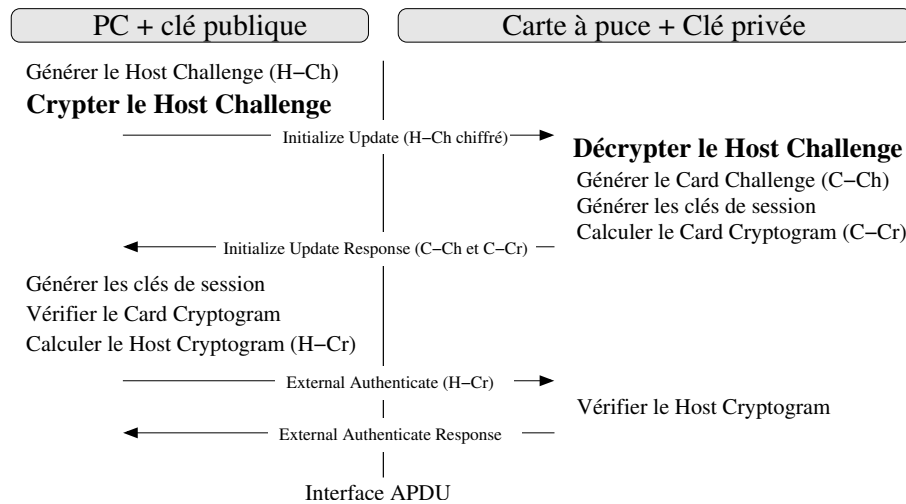
Solution à base de clés asymétriques

Les algorithmes à clés asymétriques ou clés publiques sont conçus de telle manière que la clé de chiffrement est différente de la seule clé de déchiffrement. Par ailleurs, dans ce système, la clé de déchiffrement ne peut pas être calculée à partir de la clé de chiffrement. Ainsi, n'importe qui peut utiliser la clé de chiffrement pour chiffrer un message mais seul celui qui possède la clé de déchiffrement peut déchiffrer le message chiffré. On appelle donc la clé de chiffrement, clé publique et la clé de déchiffrement, clé privée. Alors que dans les algorithmes à clé symétrique, tout repose sur la non-divulgateur d'une clé commune, la cryptographie à clé publique résout ce problème en permettant de distribuer la clé publique et en conservant secrète la clé privée.

Dans un premier temps, nous avons choisi d'intégrer un tel algorithme lors de l'envoi du *Host Challenge* dans le mécanisme d'authentification mutuelle de GlobalPlatform. Nous avons donc pris comme hypothèse de départ que l'utilisateur de la grille possède la clé publique de la carte et que la carte possède la clé privée correspondante. La procédure à mettre en œuvre est alors la suivante :

- L'utilisateur chiffre son *Host Challenge*, en utilisant la clé publique de la carte, avant de l'envoyer.
- Après avoir reçu et vérifié la réponse *Initialize Update*, il pourra être sûr qu'il s'agit bel et bien d'une carte du fait que le *Host Challenge* ne peut être déchiffré que par le détenteur de la clé privée, *i.e.* la carte.
- La suite de la procédure est celle préconisée par GlobalPlatform.

La figure 98 montre l'ajout du chiffrement du *Host Challenge* dans la procédure d'authentification mutuelle telle que définie par GlobalPlatform. L'utilisateur de la grille aura ainsi

FIG. 98 – Procédure d’authentification mutuelle avec chiffrement du *Host Challenge*.

la certitude qu’il communique réellement avec une carte, et que les commandes APDUs suivantes ne seront déchiffrées que par la carte. En effet, la sécurisation du canal sera assurée par les clés de session. Or, celles-ci vont être générées, entre autres, à partir du *Host Challenge* échangé de façon chiffré avec la clé publique de la carte. Ce *Host Challenge* ne peut donc être connu que de la carte et de l’utilisateur et personne hormis ces deux entités ne pourra calculer ou connaître les clés de session. Les clés de session peuvent alors être considérées comme secrètes et ne devront pas être dévoilées si on veut garantir la sécurité du canal de communication.

L’avantage de cette solution est qu’elle répond au problème d’usurpation d’identité présenté précédemment. Par ailleurs, les algorithmes utilisés pour sécuriser le canal restent les mêmes que ceux de l’implantation de référence de GlobalPlatform. Les changements sont donc mineurs.

En revanche, la question qui se pose maintenant est de savoir comment l’utilisateur obtient la clé publique de la carte. Une solution triviale consiste à envoyer une commande APDU qui aurait comme réponse la clé publique de la carte. Mais, puisque l’identité du correspondant est anonyme, c’est-à-dire qu’on n’est pas sûr de communiquer avec une carte, lors de la demande de la clé publique, l’utilisateur ne peut pas être certain qu’il ne s’agit pas d’une attaque du type *man-in-the-middle* ou qu’il ne s’agit pas non plus d’un PC placé côté carte pour tromper le client.

Une autre solution consiste à ce que la carte diffuse sa clé publique. Si dans une petite communauté, il est envisageable de générer sa paire de clés localement et d’échanger les clés publiques hors ligne, qu’en est-il pour une communication internationale où les échanges concernent des milliers d’utilisateurs ? Une authentification automatique des clés publiques est indispensable pour garantir qu’une clé appartient réellement à une entité. Nous avons alors choisi d’utiliser des certificats.

Solution à base de certificat

Le certificat [39] est un document électronique attestant qu'une clé publique appartient réellement à un individu ou à une organisation. Il est possible de transmettre une clé publique en authentifiant son propriétaire, mais tous les destinataires doivent faire confiance à l'autorité de certification (CA, Certification Authority) émettrice du certificat. Ce dernier permet donc d'assurer l'identité du propriétaire de la clé publique qu'il contient. L'utilisation des certificats consiste à ce que tous les intervenants d'un échange fassent confiance au CA, qui se charge de vérifier pour eux et une fois pour toute, l'identité du propriétaire de la clé publique, de délivrer et de signer le certificat assurant la relation de propriété entre le détenteur et sa clé publique.

Pour répondre au problème de distribution de la clé publique, nous avons choisi d'intégrer la récupération d'un certificat puis de reprendre la solution précédente toujours en suivant ensuite les étapes décrites par GlobalPlatform. Maintenant, avant de procéder à l'authentification mutuelle, l'utilisateur de la grille envoie une requête demandant le certificat de la carte, comme présenté figure 99. Ensuite, à l'aide de la clé publique du CA, l'utilisateur vérifie la validité du certificat reçu. Puis, si celui-ci est authentique, l'utilisateur chiffre son *Host Challenge* à l'aide de la clé publique contenue dans le certificat avant de l'envoyer à la carte.

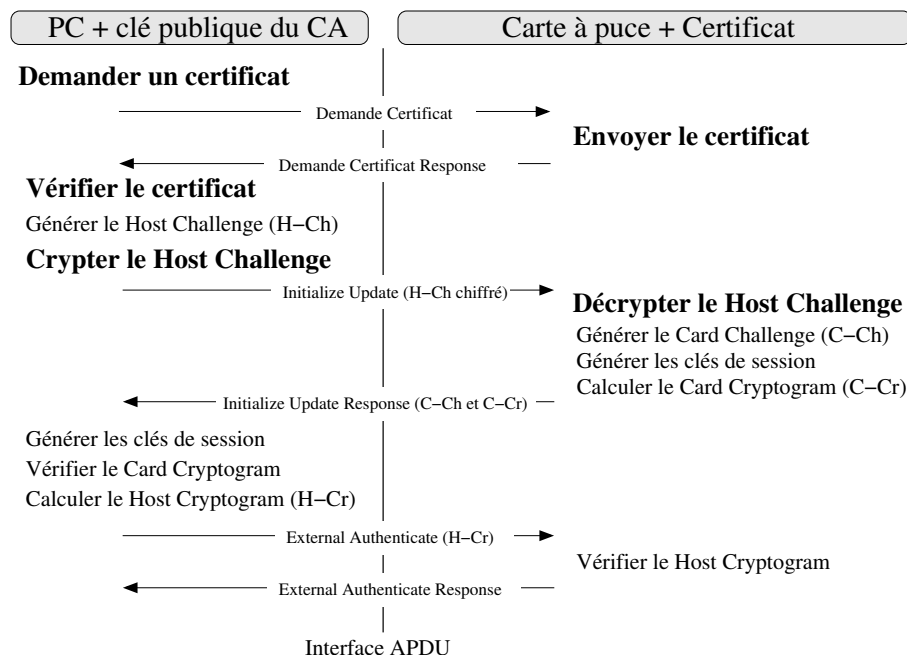


FIG. 99 – Procédure d'authentification mutuelle avec l'utilisation d'un certificat.

L'inconvénient de cette solution est bien évidemment que la clé publique du CA doit être connue au préalable par les utilisateurs afin qu'ils puissent vérifier la validité du certificat reçu. Par ailleurs, une autre question importante se pose. Comment, et par qui est installé le certificat sur la carte ? Si c'est par le propriétaire de la carte (*i.e.* de la ressource de calcul), on retombe dans un cas classique où il faut faire confiance à ce dernier et il est

donc inutile de déployer un tel attirail pour en revenir au point de départ. Ainsi, nous avons donc identifié que le certificat devrait être émis lors des phases de fabrication, d'évaluation et de certification de la sécurité des cartes, *i.e.* avant l'achat par le propriétaire de la ressource de calcul.

Nous nous trouvons donc, pour l'instant, devant un problème non résolu puisqu'il n'existe pas de telle carte. En revanche, nous savons le résoudre car il suffit tout simplement de faire fabriquer de tels produits. Néanmoins, le coût étant trop important pour nous, on considérera par la suite que nous avons de tels produits. En effet, si on disposait de tels produits, on serait désormais sûr que les messages APDUs envoyés de l'utilisateur vers la carte sont bien sécurisés et que personne hormis la carte ne pourra les déchiffrer. Toutefois, puisque nous n'avons pas de telles cartes, nous avons simplement mis en place la solution à base de clés asymétriques lors de la phase d'authentification mutuelle dans l'application de calcul de la fractale de Mandelbrot que nous décrirons ultérieurement.

Les réponses APDUs

Jusqu'à présent, seule la sécurisation des commandes APDUs a été évoquée, qu'en y est-il des réponses APDUs ? GlobalPlatform propose une solution pour sécuriser les réponses APDUs dans la dernière version des spécifications. Mais l'implantation disponible sur les cartes du commerce n'étant pas la dernière version, nous ne disposons d'aucun niveau de sécurité pour les réponses APDUs échangées via GlobalPlatform.

Nous avons donc mis en œuvre une procédure assez similaire à celle fournie par GlobalPlatform pour sécuriser les commandes APDUs. Cette procédure assure la confidentialité des données et l'intégrité des messages.

Nous ne détaillerons pas cette solution ici car dans le futur, nous pensons utiliser exclusivement des cartes compatibles avec la dernière version de GlobalPlatform.

Néanmoins, nous avons utilisé notre solution dans l'application de calcul de la fractale de Mandelbrot que nous présenterons plus loin.

13.4 Les applications de démonstration

Afin de valider la faisabilité de notre projet et avant de développer une plate-forme plus large et l'infrastructure associée, nous avons prototypé, sur une petite grille de carte à puce, deux applications de démonstration qui consistent :

- à calculer la fractale de Mandelbrot ;
- à faire du *data mining* dans le cadre d'une collaboration hypothétique entre le FBI et Air France.

Ces prototypes nous ont permis de résoudre une partie des problèmes techniques qui se présentaient à nous, mais également d'effectuer des tests de performance sur notre preuve de concept.

13.4.1 La fractale Mandelbrot

Dans cette section, nous décrivons les détails de l'application de calcul de la fractale de Mandelbrot.

Présentation

Notre application de démonstration calcule la fractale de Mandelbrot en utilisant le projet DJFractal [200] développé au sein de l'équipe Systèmes et Objets Distribués du LaBRI. DJFractal est un générateur de fractale qui utilise Mandala [201] pour distribuer les calculs sur un ensemble de CPUs pouvant aller de la station de travail à la Java Card.

DJFractal utilise plusieurs abstractions pour faire le calcul. Tout d'abord, il définit le *fractal data*, fd , comme une sous-région rectangulaire $fd = (x, y, width, height)$ de la zone totale à calculer – avec certaines informations supplémentaires. Ensuite il définit un *fractal computer* comme un objet capable de fournir un *fractal result* si on lui donne un *fractal data*. La zone initiale est donc découpée en plusieurs *fractal data* (cf. section 13.4.1) dont chacun est ensuite donné à un des différents *fractal computers*² possibles. Afin de réaliser cette tâche, un ordonnanceur est utilisé pour décider quel *fractal computer* devra calculer le prochain *fractal data*. Par conséquent, l'ordonnanceur doit connaître la totalité des *fractal computer* disponibles. Pour cela, il utilise une collection – au sens du packaging `java.util.Collection`³ – de *fractal computers*.

Afin d'améliorer les performances, nous calculons plusieurs *fractal data* en parallèle. Basé sur l'infrastructure logicielle Mandala qui permet à n'importe quelle méthode publique de n'importe quel objet d'être invoquée à distance et de façon asynchrone, DJFractal utilise l'appel asynchrone de méthodes aussi bien sur les *fractal computers* locaux que sur ceux distants.

Prérequis : définition de la fractale de Mandelbrot

Considérons le plan complexe $\mathcal{P}$, où

$$\forall (x, y) \in \mathbb{R}^2, P(x, y) \in \mathcal{P} \equiv z = x + yi, z \in \mathbb{C}$$

Pour un $z \in \mathbb{C}$ donné, considérons la suite $M(z)$:

$$M(z) = \begin{cases} z_0 = z, \\ z_{n+1} = z_n^2 + z, \quad n \in \mathbb{N} \end{cases} \quad (2)$$

La fractale de Mandelbrot est l'ensemble des $z \in \mathbb{C}$ pour lesquels $|M(z)|$ reste borné. Il peut être prouvé que pour n'importe quel z donné pour lequel $|z| \leq 2$, si il existe un compteur (*count*) n tel que dans $M(z)$, $|z_n| \geq 2$ alors $|M(z)| \rightarrow \infty$ et z est donc en

²Bien que non requis, il y a en général un seul *fractal computer* par machine

³Nous pourrions utiliser `java.util.Set` qui représente mieux la notion mathématique d'ensemble (i.e. une collection sans aucun élément dupliqué), mais cela impose des contraintes comme l'impossibilité d'utiliser `java.util.ArrayList` comme ensemble.

dehors de la fractale de Mandelbrot. Si ce critère peut ne pas paraître vraiment valable car fonctionnant seulement quand $|z| \leq 2$, il est pourtant connu que la totalité de la fractale de Mandelbrot se trouve à l'intérieur du disque centré sur l'origine et de rayon 2. Aussi, seules ces valeurs de z sont à considérer.

L'algorithme

La méthode utilisée par DJFractal s'inspire de l'algorithme décrit par Peter Alfeld [68]. Cet algorithme calcule et, bien évidemment, dessine la structure intéressante de la fractale de Mandelbrot dès que possible. De façon informelle, la structure intéressante est représentée par l'ensemble des points situés à la bordure de la fractale de Mandelbrot.

Pour chaque point P du plan complexe, DJFractal doit déterminer s'il se situe à l'intérieur de la fractale de Mandelbrot ou non. Pour cela, l'algorithme calcule le compteur (*count*) n du point dans $M(z)$. Remarquez que si z est dans la fractale de Mandelbrot, cela signifie qu'aucun compteur (*count*) n n'a été trouvé pour lequel $|z_n| \geq 2$. Ainsi, un *bailout*, $B \in \mathbb{N}$ est défini au début du calcul afin que si $|z_B| < 2$ nous considérerons que z est dans la fractale de Mandelbrot. Par conséquent, la fonction $count(z)$ est définie comme :

$$count(z) = \begin{cases} n, & \text{si } \forall i < n, |z_i| < 2 \text{ et } z_n \geq 2 \\ B, & \text{si } \forall i \leq B, |z_i| < 2 \end{cases}$$

Afin de dessiner la fractale, nous assignons à chaque point $P(x, y)$ d'affixe $z = x + yi$ dans le plan complexe une couleur $color(z)$ qui dépend de $count(z)$. En particulier, si $count(z) = B$, alors $color(z)$ est positionnée sur noir.

Pour le calcul d'un *fractal data* donné fd , i.e. un ensemble rectangulaire de points, l'algorithme distingue deux cas dépendants de la taille de sa surface $size(fd)$:

- si $size(fd) > S$, où S est une constante connue avant le calcul, alors fd est découpé en quatre nouveaux *fractal data*. Le niveau d'intérêt de chaque région doit ensuite être déterminé en utilisant la valeur *discrepancy* (i.e. d'anomalie), δ , définie comme suit :

$$\begin{aligned} \delta(fd) &= \max(K_{fd}) - \min(K_{fd}) \\ K_{fd} &= \{count(lt_{fd}), \\ &\quad count(lb_{fd}), \\ &\quad count(rt_{fd}), \\ &\quad count(rb_{fd})\} \end{aligned}$$

où $lt_{fd}, lb_{fd}, rt_{fd}$ et rb_{fd} correspondent respectivement au coin en gauche-haut, gauche-bas, droit-haut et droit-bas du *fractal data* fd . Par exemple, considérons un calcul où $B = 100$, et un *fractal data* où les compteurs (*counts*) de coin, K sont $K_{fd} = \{10, 65, 78, 100\}$. Alors $\delta(fd) = 100 - 10 = 90$. Le fait que la *discrepancy* soit si haute suggère que le *fractal data* contient une structure significative.

Après le calcul de $\delta(fd)$, chaque point de la région fd reçoit une couleur $c(fd)$:

$$c(fd) = color(average(K_{fd}))$$

et la sous-région est dessinée à l'écran.

- sinon, $size(fd) \leq S$, et chaque point z de la région reçoit séquentiellement sa couleur $color(z)$;

L'algorithme prend toujours le *fractal data* du haut de la pile des plus grandes *discrepancies* – il y a une pile par valeur de *discrepancy* –, demande à l'ordonnanceur de décider à quel *fractal computer* ces données doivent être envoyées et selon la taille de la zone du *fractal data*, invoque de façon asynchrone (et distante si le *fractal computer* est distant) la méthode `discrepancy()` ou `calculateAll()` comme illustré figure 100. Il démarre avec la région entière sur la pile. Puis, l'algorithme raffinerait toujours prioritairement un *fractal data* avec la *discrepancy* la plus haute. Les *fractal data* avec une *discrepancy* basse, *e.g.* zéro, *i.e.* ceux à l'intérieur de la fractale de Mandelbrot seront traités à la fin.

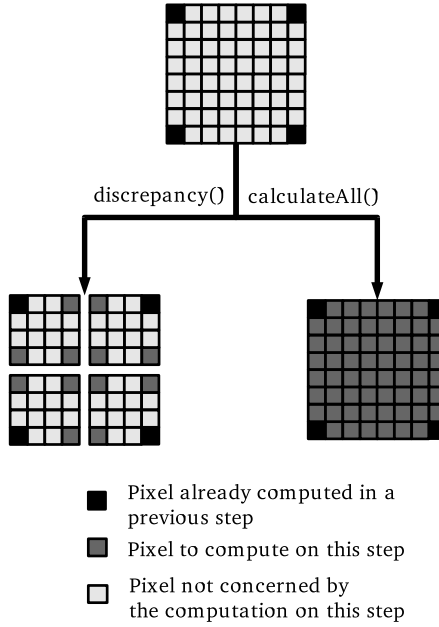


FIG. 100 – Une illustration de l'algorithme utilisé dans DJFractal.

Si t est le nombre d'instructions nécessaires pour calculer une itération de $M(z)$, alors puisque $count(z) \leq B$, la complexité pour un *fractal data* est :

$$C(fd) \leq t.B.size(fd) \quad (3)$$

L'ordonnanceur

L'application DJFractal peut utiliser divers ordonnanceurs prenant en compte les caractéristiques de l'infrastructure matérielle et logicielle sous-jacente. Si, dans cette partie,

nous ne présentons pas une description complète de l'architecture (*cf.* FIG. 101), nous décrivons cependant brièvement l'ordonnanceur utilisé pour les *benchmarks* présentés dans la section 13.4.1.

L'algorithme que nous utilisons pour calculer la fractale de Mandelbrot est construit de telle manière qu'il est possible d'effectuer des calculs en parallèle et d'utiliser plusieurs ordinateurs. Pour tirer partie de cet avantage, nous avons besoin d'un ordonnanceur qui choisit pour chaque tâche à réaliser la meilleure machine. Bien entendu, nous utilisons des Java Cards dans notre cas particulier et toutes ces unités de calcul ont les mêmes performances. Pourtant, nous ne pouvons pas nous baser sur cette hypothèse de performances homogènes puisque nous pouvons utiliser des Java Cards hétérogènes.

Notre ordonnanceur doit choisir l'unité de traitement pour chaque calcul. Il est dédié pour le calcul de la fractale de Mandelbrot et nous ne pouvons pas le réutiliser dans un autre contexte. Les principales caractéristiques à prendre en compte pour calculer la fractale de Mandelbrot sont l'irrégularité imprévisible du calcul et le fait que le travail est généré par le travail précédent.

Deux types d'opérations sont concernés dans le calcul de la fractale de Mandelbrot : `discrepancy()` et `calculateAll()`. Comme nous l'avons vu dans la section 13.4.1, `discrepancy()` requiert le calcul de trois points et génère quatre nouveaux calculs à faire pour plus tard. Par ailleurs, `calculateAll()` requiert le calcul de plus de points, *i.e.* tous les points de la zone correspondante. Ceci est la première source d'irrégularité. De plus, la quantité de calcul nécessaire pour calculer la valeur d'un point est totalement imprévisible. Cependant nous avons décidé d'implanter un ordonnanceur simple basé sur les longueurs des files *fifo* (*first in, first out*) : chaque *fractal computer* possède une file *fifo* où les requêtes en attente sont stockées. En se basant sur l'hypothèse qu'un *fractal computer* qui aura un petit nombre de requêtes en suspens, est plus efficace que les autres, ou qu'on lui a soumis des calculs plus faciles, nous soumettrons les calculs à effectuer au *fractal computer* qui aura le plus petit nombre de requêtes en attente. Nous avons utilisé cet ordonnanceur dans les *benchmarks* présentés dans la section 13.4.1. Il a amélioré les performances de façon significative sur le temps total mis pour les calculs par rapport à la version précédemment utilisée qui se basait sur un algorithme de *round robin*.

Le principal avantage de cette solution est qu'elle se base seulement sur la taille de la file de requêtes en suspens pour choisir parmi les *fractal computers*. Quand tous les *fractal computers* ont la même quantité de requêtes en attente, aucune différence n'apparaît et l'ordonnement revient alors à faire du *round robin*. Si, durant une période de temps, tous les *fractal computers* sont impliqués dans un calcul, alors, durant cette période, l'ordonnanceur enverra les prochains calculs aux *fractal computers* avec les files d'attente les plus courtes. Après un certain temps, toutes les files d'attente auront la même taille. Nous appelons t l'intervalle de temps entre cet instant et l'instant où un *fractal computer* termine son calcul courant et réduit la taille de sa file. Durant t , l'ordonnanceur soumet les prochains calculs suivant un algorithme de *round robin*. Supposons qu'un des *fractal computers* soit vraiment plus lent que les autres. Alors, durant cet intervalle t , il obtiendra le même nombre de calculs à effectuer que les autres, même s'il montre de mauvaises performances. Ce *fractal computer* peut calculer plus lentement car il utilise un matériel moins puissant, ou parce que la complexité des calculs qu'il a en attente peut être plus

importante que celle de ceux soumis aux autres. Les faibles performances des Java Cards conduisent à une multiplication des intervalles de temps où tous les *fractal computers* sont occupés, *i.e.* toutes les files d'attente ont la même taille, et cela nous a conduit à repenser notre ordonnanceur.

Nous avons alors introduit la notion de file d'attente bornée. Nous avons décidé d'une taille maximum pour les files d'attente. Quand tous les *fractal computers* ont leur file d'attente pleine, nous arrêtons de soumettre des requêtes. De cette façon, nous évitons de faire un ordonnancement *round robin*. Quand certains *fractal computers* réduisent le nombre de requêtes dans leur file d'attente alors, nous soumettons les requêtes non envoyées toujours en se basant sur la taille des files d'attente.

Une autre caractéristique intéressante de cet algorithme est le problème de famine. À la fin d'un calcul, certains *fractal computers* peuvent être en attente de travail alors que d'autres font encore des calculs et ont des files d'attente non vides. Dans ce cas, il pourrait être intéressant de récupérer les requêtes en attente pour les soumettre à nouveau aux *fractal computers* inactifs. Par ailleurs, l'algorithme que nous utilisons pour calculer la fractale de Mandelbrot peut montrer d'autres problèmes de famine⁴. Puisque le travail est généré par le travail, il peut y avoir des situations où il n'y a pas de travail à faire avant que certaines `discrepancy()` aient été calculés. Plusieurs *fractal computers* sont inactifs pendant que d'autres ont des files d'attente non vides et les *fractal computers* inactifs doivent attendre jusqu'à ce que de nouveaux calculs soient générés. L'ordonnanceur borné limite ce problème mais n'est pas suffisant et nous travaillons actuellement sur ce point.

Les *benchmarks*

Pour les besoins de notre application de démonstration, nous avons implanté des *applets* Java Card capables de calculer `discrepancy()` et `calculateAll()` pour un *fractal data* donné. Par ailleurs, sur le PC, nous avons installé un proxy qui traduit les appels à destination des *fractal computers* en APDUs à envoyer aux *applets* cibles (*i.e.* les *fractal computers*) comme nous pouvons le voir sur la figure 101.

Nous avons fait tourner notre application de démonstration pendant plusieurs heures en utilisant à chaque fois neuf cartes de différents fabricants. Si on peut se demander pourquoi neuf cartes et pas plus ou pas moins, c'est tout simplement à cause des caractéristiques matérielles de notre infrastructure (*cf.* section 13.2.1). Nous avons également testé notre application sur un mélange⁵ de cartes de différents fabricants. La figure 102 est une capture d'écran de ce qu'affiche notre application de démonstration lors de son fonctionnement. À cause d'un bug dans notre implantation d'une bibliothèque de Double, il y a quelques points erronés sur la figure. Mais, cela n'est pas très gênant pour les *benchmarks* puisque le bug sera présent de façon identique sur toutes les cartes. Par ailleurs, on notera que ce bug concerne seulement notre implantation Java Card de l'interface `FractalComputer` et que DJFractal est une implantation sans bug qui utilise les types natifs `double` ou `java.math.BigDecimal` qui ne sont pas disponibles sur les Java Cards.

⁴Ce cas apparaît de façon triviale au début puisqu'il existe une seule zone qui conduit ensuite à la création de quatre nouvelles zones à calculer.

⁵3 GemXpresso Pro R3 E32PK, 2 JCOP31bio, 2 GemXpresso Pro R3 E64PK and 2 SmartC@fé Expert

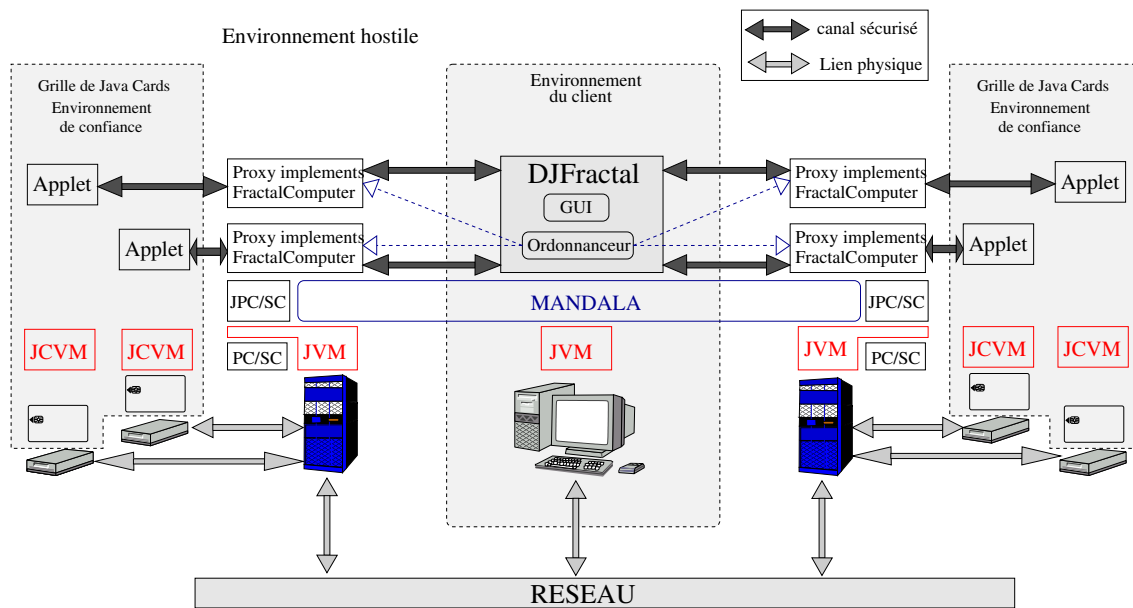


FIG. 101 – Architecture de l'application de calcul de la fractale de Mandelbrot.

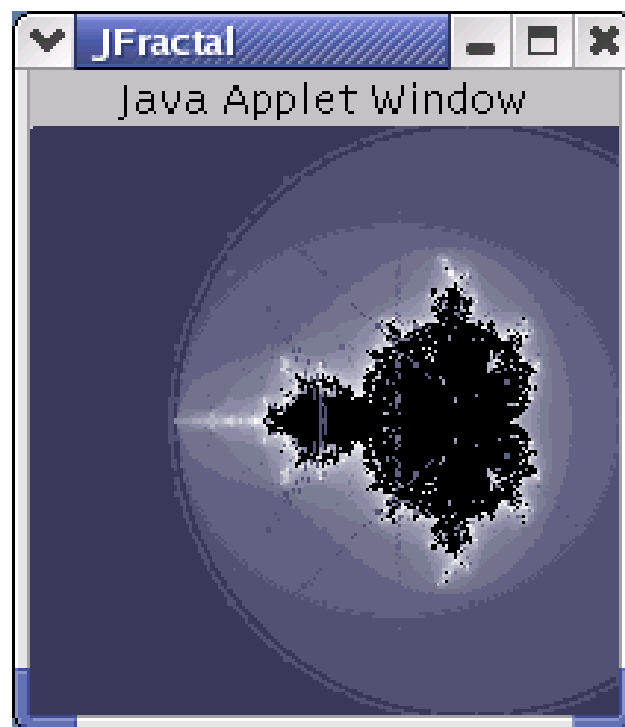


FIG. 102 – Capture de la fractale de Mandelbrot calculée sur les cartes.

Les résultats des mesures sont présentés dans le tableau 9. Le *fractal data* initial possède une surface de 200x200 points définie par ses coins haut-gauche et bas-droit du plan complexe $(-3, -2); (1, 2)$. Les paramètres sont $B = 20, S = 100$ et l'ordonnanceur utilisé est celui décrit dans la section 13.4.1.

Modèle de Java Card	Temps d'exécution (min)
9 Java Card sur un composant Fujitsu mb94r215b (FRAM)	122
9 GemXpresso Pro R3 E32PK	202
9 JCOP31bio	253
9 GemXpresso Pro R3 E64PK	376
9 SmartC@fé Expert	389
Le mélange	344

TAB. 9 – *Benchmark* sans canal sécurisé.

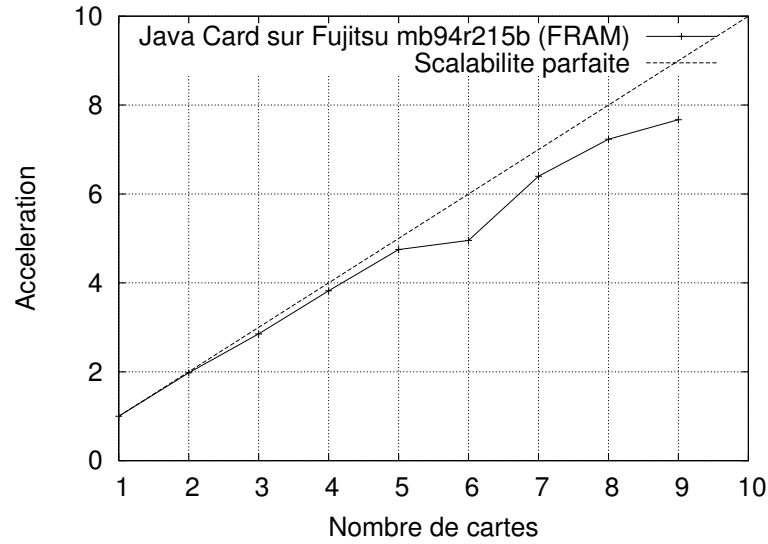


FIG. 103 – Tests sans canal sécurisé.

La figure 103 montre une courbe représentant quelques tests de *scalabilité* effectués sur les cartes à base de composants Fujitsu mb94r215b et une courbe représentant la scalabilité parfaite. Comme nous n'avons pas de version séquentielle de l'algorithme de calcul de la fractale de Mandelbrot, l'accélération (*i.e.* le speedup) a été calculée en considérant le temps séquentiel comme le temps de notre algorithme parallèle mais s'exécutant sur une seule Java Card. La forme de la courbe n'est pas très surprenante (excepté pour 6 cartes, où nous aurions pu attendre de meilleures performances – mais il y a certainement eu un problème que nous n'avons pas détecté à la fin du calcul, *e.g.* une carte en panne ou muette qui ne calculait plus) : l'accélération initiale est vraiment très bonne puisque proche de la *scalabilité* parfaite, puis, l'accélération atteint un seuil lorsque le nombre de cartes augmente et donc la *scalabilité* commence à décroître.

En revanche, nous n'avons pas encore mesuré la surcharge due à notre infrastructure,

qui consiste à envoyer tous les APDUs directement aux cartes, plutôt que de les faire générer et envoyer au travers du *framework*.

Par ailleurs, nous avons réalisé quelques tests en utilisant un canal sécurisé et la surcharge due à la mise en place de cette sécurité est vraiment négligeable. En effet, sur les puces récentes que nous avons testées, le calcul du DES est d'environ $100\mu s$ car exécuté par un crypto-processeur matériel dédié. Or, comme le canal sécurisé est un triple DES de toutes les commandes APDUs (du lecteur vers la carte) et comme le nombre de messages est limité, la surcharge devrait être vraiment faible. D'ailleurs, même si on décide de chiffrer les réponses avec la solution que nous avons proposée plus haut, section 13.3.3, le surcoût est négligeable.

Bien évidemment, nos résultats ne sont pas ceux de calculs haute performance mais la caractéristique qui nous intéresse n'est pas la rapidité mais bien la sécurité. Par ailleurs, les résultats présentés ici ne signifient en rien que les cartes les plus lentes ne doivent pas être utilisées car elles pourraient se révéler des cartes plus sécurisées. En effet, bien souvent, la vitesse d'exécution diminue quand la sécurité est accrue. Ainsi, nous avons juste fait ces *benchmarks* afin de voir les performances de différents produits de nos partenaires. De plus, cela nous a permis de découvrir quelques problèmes et de mieux cibler les produits que nous souhaiterions utiliser pour la mise en place de notre infrastructure (*cf.* section 13.5.1).

Par ailleurs, les excellents résultats obtenus sur les puces de Fujitsu sont principalement dus à leur architecture 32 bits et à leur technologie de mémoire non volatile qui est de la FRAM. En effet, cette dernière est beaucoup plus rapide que l'EEPROM utilisée par toutes les autres cartes que nous avons testées. D'ailleurs, à la vue des mauvais résultats de ses cartes, Giesecke&Devrient a souhaité obtenir le code de notre *applet* et nous a fourni une analyse détaillée expliquant que notre code testait principalement les performances des accès à la mémoire non volatile. Ces *benchmarks* nous ont donc confirmé que l'utilisation de cartes utilisant de la mémoire EEPROM devait être proscrite à cause de leur destruction. En effet, comme nous l'avons déjà vu dans la section 1.4, ces mémoires supportent seulement 10^5 cycles d'écriture alors que la FRAM en supporte un minimum de 10^{12} . Aussi, dans le futur, nous utiliserons autant que possible des cartes utilisant de la FRAM ou de la Flash.

13.4.2 FBI – Air France

L'objectif de cette application est de mettre en évidence les capacités de fouille de données, *data mining*, sécurisées de notre plate-forme grille de cartes à puce.

Présentation

Depuis le 11 septembre 2001, pour des raisons de sécurité, le FBI (Federal Bureau of Investigation) souhaite pouvoir effectuer certaines recherches dans le fichier des passagers des compagnies aériennes, par exemple, Air France, pour les vols à destination des États Unis. Seulement, si Air France ne souhaite pas communiquer son fichier au FBI pour des raisons de confidentialité des données de ses passagers, ce dernier pourrait annuler l'autorisation d'Air France d'atterrir sur le territoire américain, mettant un terme à son

activité commerciale vers les États Unis.

Par ailleurs, les lois en vigueur dans la Communauté Européenne ont été récemment renforcées par des directives visant à contraindre les compagnies aériennes à préserver la confidentialité des informations personnelles des passagers qu'elles transportent.

Ainsi, on se retrouve dans une situation où les compagnies doivent à la fois préserver l'accès aux fichiers de passagers, comme la loi le leur impose, mais également les partager avec le FBI si elles veulent continuer leurs activités vers les États Unis.

Une solution pourrait être que le FBI fournisse à Air France son application. Ainsi, Air France pourrait lancer l'application sur les fichiers de passagers et ne retourner au FBI que la liste des passagers potentiellement suspects. Mais plusieurs problèmes se poseraient alors. En effet, le FBI ne souhaite pas dévoiler son algorithme, *e.g.* les critères discriminants mis en œuvre dans son analyse des passagers. Or, une analyse de rétro-conception du programme permettrait de les découvrir. Par ailleurs, le FBI ne souhaite pas non plus voir l'exécution de son code espionnée ou modifiée. Donc, soit, le code devrait être exécuté soit chez le FBI soit sur une plate-forme de confiance localisée n'importe où. Enfin, si la compagnie aérienne acceptait de faire tourner le programme du FBI chez elle, elle ne pourrait pas être sûre que ce dernier n'a pas placé un cheval de troie dans le code afin d'avoir accès à toutes les données. Ainsi, si la compagnie acceptait de faire tourner le code, cela pourrait seulement être sur une plate-forme de confiance dans laquelle le code ne pourrait pas faire plus que ce que le propriétaire de cette plate-forme lui a permis.

L'objectif de cette application est de montrer comment nous pouvons assurer la confidentialité du code et des données, quand ceux-ci appartiennent à deux entités distinctes ne se faisant pas confiance.

Par conséquent, il se dégage la nécessité d'avoir une plate-forme de confiance sur laquelle sera déployée le fichier de passagers et l'exécution du code de fouille de données du FBI.

Principe

Pour résumer, les caractéristiques de l'application sont les suivantes :

- elle est de type *data mining*, *i.e.* on a un jeu de données que l'on souhaite analyser pour extraire des informations ;
- les données ou une partie d'entre elles doivent rester confidentielles (*i.e.* les données d'Air France) ;
- le code manipulant les données doit rester confidentiel (*i.e.* le code du FBI).

Le prototype que nous avons implanté prend donc en compte ces contraintes.

Nous avons tout d'abord construit une application pour la saisie des passagers d'Air France qui distribue ces informations sur un ensemble de Java Cards. Chaque passager se voit associer une clé unique **key** qui permet à Air France de l'identifier mais qui permet également une utilisation par le FBI en lui cachant le nom réel du passager. Sur chaque carte, une API minimale a été mise en place – supposée développée par Air France – pour permettre à un code chargé sur les cartes d'avoir accès aux données non confidentielles concernant les passagers.

Le code du FBI est ensuite distribué sur les cartes et grâce à l'API fournie par Air France, il analyse le fichier de passagers. Lorsque le critère d'analyse est positif, l'application du FBI renvoie la clé **key** représentant le passager suspect. Il est alors possible pour le FBI de demander à Air France de lui fournir plus d'informations sur le passager suspect, ce qu'Air France peut faire ou non.

Mise en œuvre

Nous avons développé une interface graphique (*cf.* FIG. 104) pour la compagnie Air France qui permet de saisir la liste de ses passagers et la déploie sur les cartes.

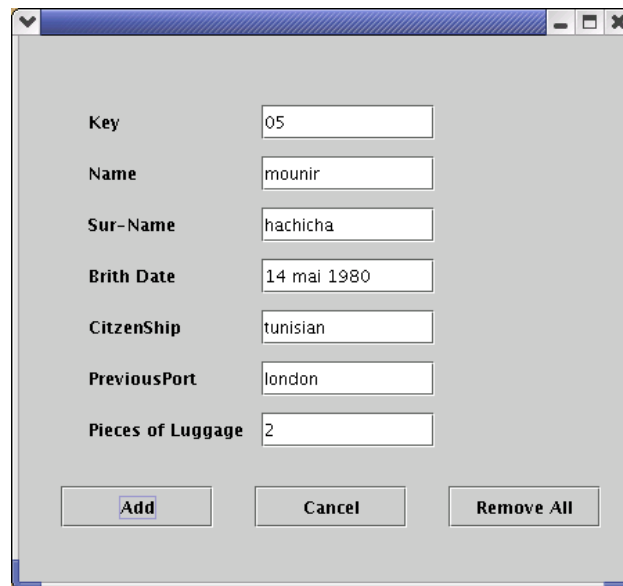


FIG. 104 – Saisie des informations sur les passagers.

Dans cette application, un passager est représenté sur une carte par la classe **Passenger** contenant toutes les informations le concernant et notamment son nom, son prénom, sa nationalité, etc. Sur chaque carte, nous disposons également d'une *applet* appelée **PassengerManager** qui permet à Air France de gérer la liste de ses passagers (*cf.* FIG. 105).

Cette *applet* **PassengerManager** fournit également une interface réduite permettant au FBI de récupérer les informations spécifiées publiques par Air France. Pour cela, la classe **PassengerManager** implante une interface *Shareable* qui donne seulement un accès restreint aux fichiers des passagers et ne permet pas d'identifier un passager (*cf.* FIG. 106). Ainsi, Air France peut partager les données de ses passagers avec le FBI au sein d'un mécanisme sécurisé et contrôlé par le JCRE.

Après que tous les passagers aient été saisis, le FBI essaye de déterminer quels sont les passagers qui apparaissent comme suspects. Pour cela, le FBI spécifie les critères utilisés pour décider si tel passager est suspect ou non. Ceci est réalisé par l'intermédiaire d'un outil graphique (*cf.* FIG. 107).

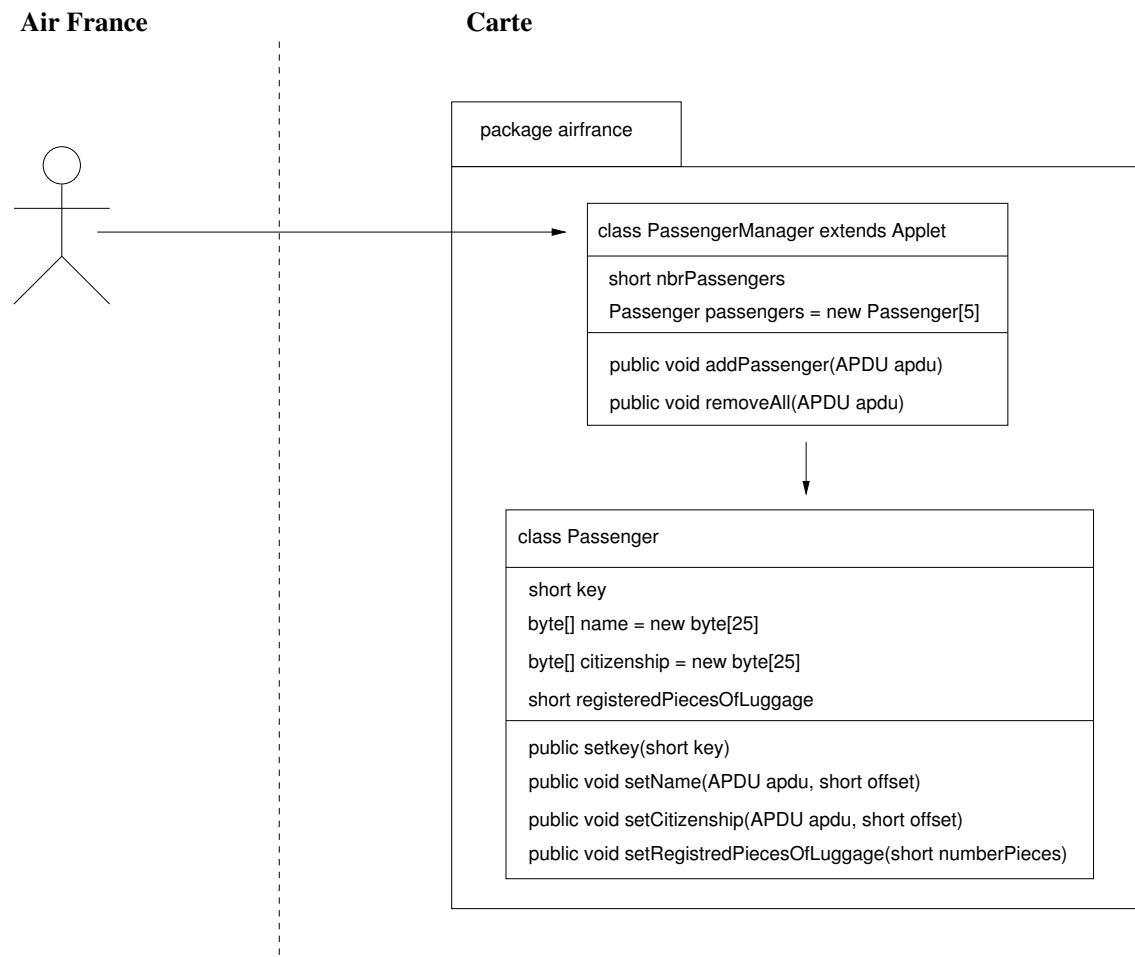


FIG. 105 – Gestion des passagers sur la carte.



FIG. 106 – Partage de données.

citizenship

Previous Port

Pieces Luggage

OK Cancel

FIG. 107 – Saisie de critères de recherche.

Après leur saisie, ces critères sont transmis aux *applets* **CheckPassengers**, représentant le FBI sur les différentes cartes (*cf.* FIG. 108). On notera que pour pouvoir appeler les méthodes d'analyse sur toutes les cartes, nous avons dû définir un mécanisme indiquant à la machine intermédiaire qu'elle doit diffuser l'appel à toutes les cartes (*i.e.* un nom de carte particulier assimilable à une adresse de diffusion pour les réseaux).

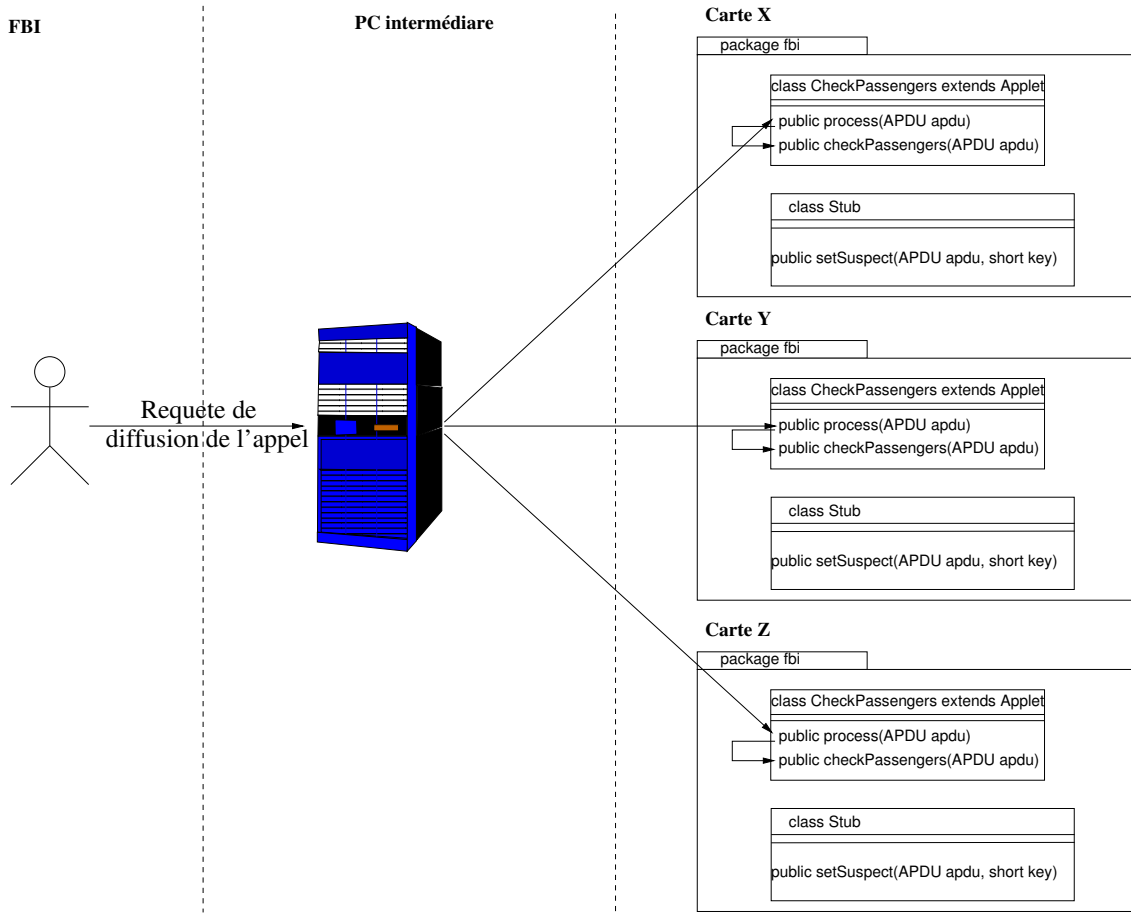


FIG. 108 – Diffusion de la requête d'analyse aux *applets* **CheckPassengers**.

Les *applets* **CheckPassengers** analysent alors la liste des passagers disponibles afin de trouver les passagers suspects correspondant aux critères de recherche. Chaque fois qu'un suspect est trouvé, la carte récupère son identifiant **key** et l'envoie à une *applet* spéciale, appelée **FbiServer** qui centralise tous les résultats. Cette étape avait seulement pour objectif de mettre en évidence le fonctionnement du mécanisme de carte proactive (*cf.* FIG. 109).

Enfin, quand toutes les cartes ont fini la recherche, le FBI demande à l'*applet* **FbiServer** la liste de tous les suspects trouvés puis revient ensuite vers Air France pour demander les noms et prénoms des passagers suspects correspondant aux **keys** trouvés.

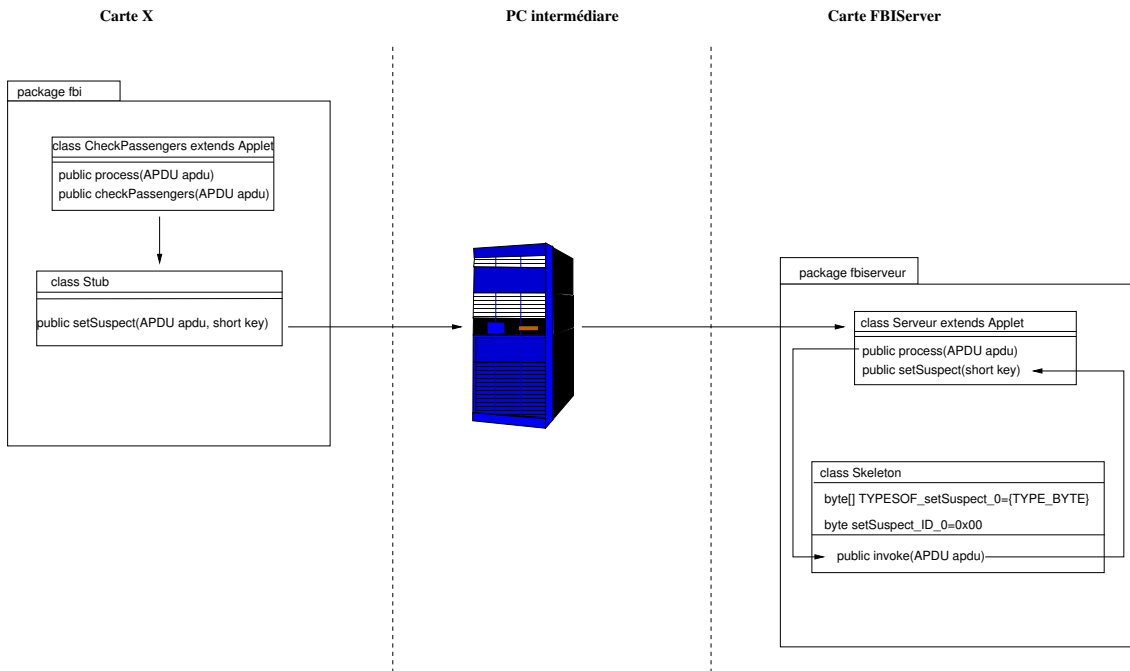


FIG. 109 – Retour des résultats.

13.5 Bilan

Pour mettre en œuvre notre infrastructure et déployer les deux applications de démonstration nous avons dû faire face à un certain nombre de problèmes. Certains sont complètement résolus, d'autres le sont seulement partiellement et d'autres encore sont en attente de solution.

13.5.1 Défis résolus

Nous avons résolu les problèmes suivants.

- Pour la démonstration de l'application de Mandelbrot et puisque le type `double` n'est pas disponible en Java Card, nous avons implanté une classe `Double`. Or, il y a beaucoup de pièges à éviter pour implanter une telle bibliothèque. Par exemple, en Java Card, la pile d'appels est limitée, donc, on ne peut pas traiter le problème en utilisant plus de 3 ou 4 niveaux d'appel imbriqués. De plus, comme cette bibliothèque est destinée à être appelée par un programme client, suivant la position de l'appel la bibliothèque pourra fonctionner ou provoquer une levée d'exception. Toutefois ce problème est insoluble, car d'ordre général et inhérent à toute bibliothèque (*i.e.* quel que soit le langage de programmation et la plate-forme utilisés). Par ailleurs, la mémoire étant très limitée, il faut choisir des représentations adaptées pour les divers objets. Enfin, comme les objets alloués persistent durant toute la vie de la carte, il faut faire une implantation basée sur la réutilisation d'objets.
- Nous avons ajouté de nouvelles fonctionnalités à `pcsc-lite` comme :

- une gestion des lecteurs connectés simultanément à la même machine (jusqu'à 255 lecteurs supportés en même temps) ;
 - une nouvelle gestion des requêtes envoyées par les clients aux lecteurs afin que les appels soient traités en parallèle ;
 - le support de la nouvelle version des spécifications de pilotes `pcsc-lite` ce qui permet, par exemple, d'utiliser des lecteurs sécurisés (*e.g.* lecteur avec un clavier, etc.) ;
 - quelques contrôles de sécurité comme la vérification que seule la *thread* qui possède le contexte PC/SC peut le détruire, etc.
- Nous avons identifié des types de mémoire et les processeurs que nos cartes devaient posséder, *i.e.* des cartes avec des mémoires non volatile FRAM ou Flash et des processeurs 32 bits.

13.5.2 Défis partiellement résolus

Parmi les problèmes que nous avons à résoudre, nous avons proposé certaines solutions pour :

- Les communications. En effet, les canaux de communications doivent être sécurisés entre les clients et la grille de cartes à puce mais également entre les cartes elles-mêmes. Nous avons proposé un début de solution dans l'application de démonstration de calcul de la fractale de Mandelbrot pour sécuriser les communications entre le client et les cartes en utilisant pour les commandes APDU la solution de GlobalPlatform et pour les réponses APDU une solution dérivée de GlobalPlatform. Par ailleurs, nous avons testé une version avec des canaux sécurisés entre les cartes dans l'application FBI – Air France. Aussi, pour l'instant, nous avons donc seulement commencé à intégrer cet aspect dans notre infrastructure.
- Le modèle d'exécution. Pour être efficace, notre infrastructure doit offrir de l'asynchronisme et proposer que la carte puisse fonctionner aussi bien comme serveur, mode de fonctionnement naturel d'une carte, que comme client. Nous avons travaillé sur une méthode pour appeler du code d'une carte depuis une autre carte dans l'application FBI – Air France et nous intégrerons ce mécanisme dans l'infrastructure très prochainement.

13.5.3 Défis à résoudre

Certains problèmes restent en suspens pour obtenir une infrastructure plus flexible.

- La taille mémoire des Java Cards est un réel problème, notamment pour les applications traitant des gros volumes de données. Pour passer outre cette contrainte, nous pourrions, par exemple, chiffrer et stocker les résultats intermédiaires dans des mémoires standards (*i.e.* non sécurisées), rapides et de grandes capacités (*e.g.* clés USB, ...). Nous pourrions également construire une mémoire virtuelle distribuée sur un ensemble dédié de cartes à puce, ce qui permettrait de garder les données secrètes non chiffrées, même si, bien évidemment, pour les échanger il faudrait à nouveau les chiffrer pour passer à travers les canaux non sécurisés reliant les cartes. Ceci dit les

projections prévoient que les cartes à puce auront jusqu'à 1 Go de mémoire dans les 5 ans à venir. Donc, les technologies de fabrication pourraient apporter des réponses satisfaisantes.

- Nous devons également proposer une solution pour traiter de l'hétérogénéité et du déploiement des applications. En effet, pour être pleinement utilisable, notre infrastructure devra :
 - gérer de façon transparente l'hétérogénéité du matériel (cartes, lecteurs, et PC) ;
 - fournir des mécanismes de tolérance aux pannes ;
 - gérer le déploiement des applications, *i.e.* la terminaison des calculs, la localisation des codes sur les cartes, etc. Si, nous avons déjà mis en place un mécanisme de détection de pannes dans l'application de calcul de la fractale de Mandelbrot, celui-ci est excessivement basique puisqu'il ne traite que les problèmes de cartes muettes en tentant de leur envoyer plusieurs fois un signal de *reset*. Par ailleurs, un projet étudiant a permis d'obtenir une première bibliothèque de chargement d'applications sur des cartes compatibles GlobalPlatform. Hélas, celle-ci est loin d'être utilisable tel quel et son intégration nécessitera une reprise complète du code.
- Enfin, il persiste le problème très important de la méthode d'authentification nécessaire à l'établissement d'un canal sécurisé entre la carte et le PC, ou entre les cartes. En effet, cette étape est importante puisque c'est notamment après le passage celle-ci que l'on peut charger une application en toute sécurité sur la carte et communiquer avec elle. Or, pour l'instant, le standard GlobalPlatform propose de réaliser une authentification mutuelle de la carte et du PC à l'aide de clés partagées par chacune des entités. Seulement, comme nous souhaitons mettre notre plate-forme à disposition de tous, nous avons besoin de cartes où l'authentification se fait grâce à des certificats (*cf.* section 13.3.3).

13.6 Travaux connexes

Des *frameworks* tels que JiniCard [148], Jason [80] ou OrbCard [85] ont déjà été développés pour communiquer de manière transparente et sécurisée avec les services des cartes à puce. Il est de fait possible de les utiliser pour faire du calcul distribué sur des cartes. Toutefois, comme ce n'est pas leur paradigme principal, il manque à tous ces *frameworks* des mécanismes utiles dans le contexte des applications réparties telle que l'invocation asynchrone de méthode [107, 146] requise dès que l'on utilise des ressources informatiques lentes comme les cartes à puce.

Des approches plus formelles pour protéger des codes mobiles exécutés dans des environnements d'exécution classiques (*i.e.* pas de confiance) ont également été développées. Par exemple, il existe une solution originale basée sur une extension des fonctions de cache utilisant les codes correcteur d'erreurs [160, 159]. Seulement, bien qu'utilisant des bases solides, il semble difficile d'utiliser ces solutions dans un cas général en raison des hypothèses fortes qu'elles imposent sur les programmes à exécuter pour être applicables.

Par ailleurs, dès 1995, Swisscom et BT Cellnet ont mené des travaux dans le domaine de la téléphonie mobile afin de rendre les cartes proactives [147, 136]. Ces travaux ont

abouti à la norme ETSI TS 102.221 SIM qui a aujourd'hui évolué pour laisser place à la norme ETSI TS 102.223 et au *Card Application Toolkit* dont l'API générique est spécifiée dans ETSI TS 102.240 (*UICC Application Programming Interface*) [13]. Les cartes SIMs de nos téléphones portables sont donc des cartes proactives ce qui leur permet par exemple de configurer elles-mêmes les téléphones. Les autres travaux [77] pour rendre la carte proactive utilisent des techniques similaires à celle que nous proposons.

13.7 Partenaires

Notre projet a reçu un accueil très favorable des industriels du monde de la carte à puce. Ainsi, beaucoup de fondeurs, de fabricants de cartes et de lecteurs suivent de près l'évolution de nos travaux.

Les principaux partenaires sont :

- Gemplus (pour les cartes) ;
- IBM BlueZ Secure Systems (pour les cartes) ;
- SCM Microsystems and SmartMount (pour les lecteurs) ;
- Sun microsystems (pour la totalité de la plate-forme).

Nous pouvons aussi citer, dans une moindre mesure, Fujitsu, Giesecke&Devrient, Oberthur Card Systems et Sharp pour les échantillons de Java Cards qu'ils nous ont fournis. Enfin, sans le projet `pcsc-lite` et le pilote CCID, nous aurions eu beaucoup plus de difficultés à proposer une solution et par conséquent, nous pouvons remercier David Corcoran et Ludovic Rousseau pour leurs développements.

Chapitre 14

Perspectives

Après avoir illustré les moyens de sécuriser le calcul sur la grille, beaucoup de perspectives s'offrent à nous quand au déploiement d'une infrastructure similaire pour les Terminaux Mobiles Communicants (*i.e.* téléphones, PDAs, ultra-portables). En effet, il semble possible de mettre en œuvre une telle plate-forme de confiance pour les TMCs si on les associe à des processeurs sécurisés type cartes à puce. Toutefois, si les TMCs vont complexifier les problèmes déjà rencontrés, nous croyons fortement qu'une telle infrastructure permettra d'en résoudre bon nombre d'autres. En effet, comme nous le verrons dans la section 14.1, diverses applications réparties nécessitant un haut degré de sécurité pourraient tirer partie de l'utilisation d'une telle architecture.

Par ailleurs, comme nous le montrerons dans la section 14.2, nous pensons qu'il sera ainsi possible de déployer, dans cet environnement de confiance, un ensemble de services appelés micro-services voués à la réalisation de tâches bien spécifiques. Une illustration de ce que nous entendons par micro-service pourrait être l'application de *data mining* entre le FBI et Air France présentée lors de la section 13.4.2. Nous développerons cette notion plus en profondeur dans la deuxième partie de ce chapitre.

14.1 Une plate-forme de confiance pour les TMCs

Aspects matériels

S'il est légitime de se demander pourquoi déployer une telle infrastructure pour les TMCs, Terminaux Mobiles Communicants (téléphones mobiles, ultra-portables, assistants personnels, ...), il suffit d'étudier leurs caractéristiques pour se rendre compte qu'une plate-forme de confiance est nécessaire si on veut les faire collaborer. En effet, les TMCs offrent la mobilité du traitement, ainsi que la possibilité de communiquer en tout lieu : soit directement via des liaisons sans fil (WiFi et Bluetooth principalement) où les liens de communication résultent seulement d'un échange effectif entre deux TMCs (*MANET*, *Mobile Ad hoc Network* [40]) ; soit par un réseau sans fil dit à infrastructure (point d'accès uniquement). Tout ceci fragilise naturellement toute architecture qui les exploite. Gérer cette fragilité est encore compliqué à cause de leurs caractéristiques :

- une extrême volatilité : chaque nœud du réseau peut apparaître et disparaître de

manière rapide ;

- un échange de messages (routage) par voisinage : chaque nœud n’a qu’une vue partielle du réseau, et doit passer par l’intermédiaire d’un voisin pour atteindre un autre nœud ;
- une absence de services centralisés : les nœuds peuvent s’échanger des données ponctuellement ou bien selon un modèle *peer-to-peer* (*i.e.* en partageant collectivement des ressources et l’accès à ces ressources) ;
- une large étendue de puissance de traitement entre les nœuds : il existe de fortes différences entre la puissance d’un téléphone mobile et celle d’un PDA ;
- une capacité mémoire des différents nœuds limitée : chaque TMC peut disposer de plus ou moins de mémoire (de 32 Ko à plusieurs dizaines de Mo) ;
- des liens de communication hétérogènes : WiFi, Bluetooth, IrDA, GSM/GPRS, UMTS ;
- des passerelles de sortie vers l’extérieur du réseau : un TMC peut disposer d’une connexion internet par l’intermédiaire d’une liaison GSM/GPRS.

Mise en œuvre de la confiance

Dans un réseau classique, il existe une infrastructure de services (serveur de fichiers, serveur d’authentification, etc.) à laquelle l’utilisateur a accès de par son appartenance à l’organisme (entreprise, laboratoire, etc.) qui met en œuvre ce réseau. Cette infrastructure permet d’établir des liens de confiance entre les utilisateurs. Dans un réseau de TMCs, la confiance s’appuie : soit sur l’acceptation humaine et visuelle des interlocuteurs (appariement d’appareils par Bluetooth par exemple) ; soit sur l’utilisation de certificats délivrés dans une structure différente et réunissant les différents interlocuteurs.

Mise en œuvre des services

La mise en œuvre d’un réseau de TMCs est liée aux besoins des utilisateurs pour le partage d’information. Ainsi, il existe deux catégories de besoins :

- les besoins explicites, qui correspondent à une communication directe entre deux interlocuteurs pour un échange ponctuel, de documents par exemple ;
- les besoins implicites, qui vont nécessiter l’utilisation transparente d’un ou plusieurs autres TMCs pour joindre un tiers ou récupérer une donnée dont on ne sait pas si elle est mémorisée a priori sur un TMC joignable.

14.1.1 Les besoins explicites

Dans le cas explicite, nous voulons proposer une sécurisation des échanges de données lorsque ceux-ci correspondent à l’obtention d’un résultat déterminé en fonction des informations présentes sur plusieurs TMCs, sans jamais avoir à les partager entre les interlocuteurs. Ainsi, il sera possible d’offrir aux usagers des TMCs des traitements spécifiques sécurisés appelés micro-services. Une illustration pourrait être la définition d’un rendez-vous entre deux utilisateurs en mettant en rapport leurs agendas respectifs, mais sans qu’aucun des deux n’ait accès à l’agenda de l’autre. Nous en présenterons d’autres dans la section 14.2.1.

14.1.2 Les besoins implicites

Dans le cas implicite, il est nécessaire d'exploiter une plate-forme de confiance destinée à exécuter des applications distribuées de façon transparente pour l'utilisateur. Une parfaite illustration du type d'applications que l'on peut exécuter sur cette plate-forme pourrait être l'application FBI – Air France qui nécessite, par exemple, la totale collaboration de la machine de routage des APDUs pour pouvoir fonctionner. En effet, dans le but de préserver cette architecture de confiance, il est nécessaire de garantir :

- la mise en commun des capacités des différents TMCs (mémoire, traitement, connexions externes GSM, Internet, etc.) ;
- leur coopération (profit du groupe par rapport à l'individu, rétribution au service rendu, etc.).

Ainsi, les moyens à employer que nous envisageons passent par :

- l'utilisation d'environnements d'exécution sécurisés dans les TMCs (processeurs sécurisés de type cartes à puce et environnement logiciel sécurisé existant de type Java Card) ;
- la définition d'un modèle commun d'organisation des données entre TMCs (pour faciliter les échanges et les comparaisons de ces données) ;
- la définition d'un langage restreint pour l'écriture de code mobile sécurisé capable de traiter ces données ;
- l'utilisation de liaisons sécurisées entre les différents TMCs pour l'échange de codes mobiles et de données.

Une fois cet objectif atteint, on disposera dans ces réseaux des mêmes possibilités que dans un réseau avec une infrastructure de services.

D'autre part, des micro-services pourront également être développés sur cette architecture (*cf.* section 14.2.2). Par exemple, un micro-service de *data mining* semblable à celui de l'application FBI – Air France pourra être utilisé.

14.2 Les micro-services

Conformément à ce que nous avons présenté dans la section 14.1, les micro-services que nous envisageons ici nécessiteront suivant les cas :

- l'utilisation d'une architecture de confiance dans le cas d'un besoin implicite ;
- que les TMCs soit équipés d'un matériel de confiance (*e.g.* une carte à puce) dans le cas d'un besoin explicite.

14.2.1 Les micro-services dans le cadre des besoins explicites

Dans le cadre du développement de micro-services pour les besoins explicites, nous souhaiterions démontrer que l'utilisation de code mobile ainsi que de moyens cryptographiques embarqués est l'une des voies de sécurisation des communications sans fil. Ainsi, nous pourrions développer des applications concrètes montrant cette nécessité, comme le présentent les scénarii suivants :

- Prise de rendez-vous : lors d’une conférence, je souhaite rencontrer l’orateur qui est en pleine présentation de son article. Je définis l’heure de rendez-vous que je souhaiterais obtenir et avec le code mobile, les deux TMCs se mettent en relation et définissent si cette plage horaire est valable. Comme l’orateur est très loin de moi et hors de portée de mon TMC, cette relation a lieu grâce à des nœuds qui ne sont rien d’autres que les TMCs de personnes présentes dans la salle mais dont la réunion des zones d’émission recouvre la distance jusqu’à l’orateur (de proche en proche). La communication et les informations transitant doivent être sécurisées afin qu’aucun des nœuds intermédiaires ne puisse capturer le contenu de l’information.
- Lors de cette même conférence, je cherche à rencontrer une personne que je ne vois pas. Le TMC va interroger les autres TMCs de la salle (limitation volontaire de zone) et chercher à joindre la personne désirée (Est-elle présente ? Est-elle disposée à me rencontrer ?).
- Authentification par mail : un utilisateur veut s’assurer de la validité d’une adresse électronique qu’un destinataire lui a donnée (c’est-à-dire que l’adresse existe bien et que le destinataire peut effectivement relever le courrier reçu à cette adresse). L’utilisateur envoie un mail à l’adresse indiquée par une connexion Internet ainsi que du code mobile sur le TMC de la personne. Le code mobile va vérifier que le destinataire reçoit le mail envoyé sans donner d’autre information sur les autres courriers personnels et sans intervention du destinataire. Cette méthode est plus fiable que le simple accusé de réception d’un serveur de mail qui pourrait être faux (*e.g.* si le serveur est piraté).

14.2.2 Les micro-services dans le cadre des besoins implicites

Dans le cadre du développement de micro-services pour les besoins implicites, *i.e.* basés sur une plate-forme de confiance, nous envisageons différents exemples de micro-services :

- la recherche et l’accès aux données stockées dans les TMC : je recherche une page Web, mais peut-être a-t-elle été déjà chargée et mémorisée par un autre TMC auquel je peux avoir accès. Le problème est d’autoriser un code mobile à rechercher parmi les données d’un TMC pour satisfaire la requête mais sans divulguer le contenu des autres données de l’utilisateur ;
- le partage d’un accès à l’extérieur du réseau ad’hoc (connexion vers internet) : je veux envoyer un courrier, je ne dispose pas de connexion internet moi-même, mais peut-être existe-t-il un autre TMC que je peux joindre et qui peut me prêter ponctuellement sa connexion ;
- la gestion du réseau : je veux bien relayer les communications entre TMCs, mais j’aimerais a contrario profiter d’autres services ;
- la capacité de traitement : je recherche un TMC dont la puissance de calcul me permet d’effectuer un traitement qui sinon me serait impossible.

14.2.3 D'autres micro-services

Enfin, voici une liste non exhaustive des autres micro-services auxquels nous avons songé :

- la recherche sécurisée de documents dans une mémoire collective, pour partager des ressources limitées ;
- la protection des droits d'auteurs et la consultation restreinte ;
- l'authentification des usagers pour la confidentialité des documents partagés ;
- la création d'un rapport par comparaison de données utilisateurs masquées (*i.e. data mining*) ;
- le partage d'accès à des ressources : connexions réseaux, données, etc. ;
- la modification collaborative de documents ;
- un système d'agents logiciels à intelligence collective pour le contrôle et l'application d'une politique de sécurité à l'échelle du réseau (détection et prévention des actes malveillants).

Conclusion

Nos travaux sur la technologie Java Card ont permis d’approfondir l’expertise du LaBRI et de SERMA Technologies dans ce domaine et donc de renforcer la crédibilité de l’expertise du CESTI à conduire des évaluations sécuritaires Java Card de haut niveau. Prochainement, nous espérons pouvoir mettre les JCatools à disposition de la communauté afin donner encore davantage de visibilité à nos recherches. Par ailleurs, dans un avenir très proche, nous souhaitons approfondir, modéliser et implanter la notion de *pré-persistence* pour la présenter au Java Card Forum comme une solution alternative aux ambiguïtés des spécifications.

Dans le cas général des cartes à puce multi-applicatives ouvertes, nous avons contribué à identifier de nouveaux problèmes de sécurité et nous avons proposé des solutions pour les résoudre. Ces cartes étant l’avenir des cartes à puce, tant en terme de flexibilité que de coût, nos travaux concourent à informer les développeurs des vulnérabilités auxquelles elles sont sujettes.

Enfin, nous nous sommes intéressés à sécuriser des systèmes plus larges en exploitant la sécurité offerte par les puces et par les environnements multi-applicatifs des cartes de type Java Card. Dans ce cadre, nous avons mis en place une preuve de concept pour faire du calcul distribué sécurisé sur une grille de Java Cards. Cela nous a permis de résoudre la plupart des problèmes de sécurité classiques rencontrés dans le domaine du calcul sur la grille. Dans le futur, nous souhaitons appliquer nos travaux au cadre de la mobilité (*i.e.* une grille où les nœuds de calculs sont mobiles) afin de sécuriser des réseaux ad’hoc.

Ainsi, tout au long de cette thèse, nous avons développé un tout cohérent dans le domaine des cartes à puce multi-applicatives en cherchant tout d’abord à identifier leurs vulnérabilités intrinsèques et en proposant des solutions pour s’en prémunir. Ensuite, nous avons cherché à tirer avantage des sécurités offertes par ces nouvelles technologies de cartes pour sécuriser des systèmes plus complexes.

Glossaire

AID (Application Identifier) : Nombre unique assigné à une application, à certains types de fichiers, à une instance d'une *applet* ou à un paquetage Java.

APDU (Application Protocol Data Unit) : Unité de données échangée entre la carte à puce et le terminal au niveau de la couche application. Un APDU contient soit une commande, soit une réponse.

Applet : Application écrite pour une plate-forme Java Card.

ATR (Answer To Reset) : Suite d'octets envoyée par la carte lors de sa mise sous tension.

CAD (Card Acceptance Device) : Périphérique qui est utilisé pour communiquer avec une carte à puce. Il peut aussi fournir l'alimentation et la fréquence à la carte à puce.

Canal logique : Un canal logique, vu du côté carte, fonctionne comme un lien logique pour une application embarquée. Il établit une session de communication entre une *applet* sur la carte et le terminal. Les commandes émises sur un canal logique spécifique sont transférées par le système à l'*applet* active sur ce canal logique. Pour plus d'information, voir l'ISO 7816-4 [108].

CAP (Converted APplet) : Le format de fichier standard pour la compatibilité binaire de la plate-forme Java Card. Un fichier CAP contient une représentation binaire exécutable des classes dans un paquetage Java.

Carte à puce Java : Carte à puce supportant la technologie Java Card.

Contexte : Espace protégé associé au paquetage de chaque *applet* ou au JCRE. Tous les objets détenus par une *applet* appartiennent au contexte du paquetage de l'*applet*.

COS (Card Operating System) : Le micro-code contenu dans la ROM de la carte. Il est utilisé pour la communication de la carte à puce, pour la gestion de la sécurité et la gestion des données dans les fichiers de la carte à puce.

Désélection : Action de rendre inactive une *applet* précédemment sélectionnée. À la réception d'une commande APDU de désélection d'une *applet* sur un canal logique, le JCRE marquera cette *applet* comme inactive après avoir invoqué sa méthode `deselect`.

EMV (Europay, Mastercard, Visa) : Standard pour les cartes à circuit intégré pour l'industrie bancaire, basé sur l'ISO 7816. Il a été créé en décembre 1993 par Europay International, Mastercard International et Visa International. Les applications suivant ces spécifications techniques sont déjà déployées dans nos cartes bancaires et devraient à terme remplacer l'application B0' que nous utilisons en France. Nos cartes seront ainsi compatibles

partout dans le monde puisqu'une majorité de pays intègrent également ces spécifications.

Glitch : brève variation positive ou négative d'un front de tension. Par exemple, l'envoi d'un octet sur la ligne d'entrée/sortie d'une carte à puce correspond à l'émission de plusieurs *glitches* et suit un des codages définis par la norme ISO 7816 [108].

GSM (Global System for Mobile communications) : (à l'origine Groupe Systèmes Mobile) le standard adopté par 18 pays Européens afin de développer une compatibilité dans les télécommunications mobiles numériques.

ISO 7816 : Document qui définit le standard des cartes à puce à contact.

JCRE (Java Card Runtime Environment) : Environnement d'exécution pour les *applets* Java Card.

JCVM (Java Card Virtual Machine) : Machine virtuelle Java Card. Elle se compose du convertisseur qui tourne hors carte et de l'interpréteur qui est embarqué sur la carte. Ces deux éléments implantent toutes les fonctions d'une machine virtuelle.

Multisélection : Action de rendre active plusieurs fois, et en même temps, une même *applet* sur différents canaux logiques. De telles *applets* doivent implanter l'interface `javacard.framework.MultiSelectable` pour accepter d'être sélectionnées sur plusieurs canaux logique en même temps. Ce mécanisme permet aussi à différentes *applets* du même paquetage d'être sélectionnées simultanément.

NVM (Non Volatile Memory) : Mémoire non volatile tels que l'EEPROM, la Flash, etc.

PIN (Personal Identification Number) : Code utilisé pour autoriser les transactions.

RMI (Remote Method Invocation) : Mécanisme permettant d'invoquer des méthodes sur un objet distant de façon transparente.

Sélection : Action de rendre une *applet* active. À la réception d'une commande APDU de sélection d'une *applet* sur un canal logique, le JCRE invoquera la méthode `select` de cette *applet* et la rendra active ou pas sur ce canal en fonction de la valeur retournée par `select`. Par la suite, il lui transférera toutes les commandes APDUs envoyées sur ce canal (excepté les nouvelles commandes APDUs de sélection ou de désélection).

SIM (Subscriber Identity Module) : Identifie l'utilisateur d'un téléphone GSM et fournit les clés de cryptage pour la transmission numérique de la voix.

Bibliographie

- [1] ACI GRID CGP2P. <http://www.lri.fr/~fci/CGP2P.html>.
- [2] Action Spécifique Sécurité Logicielle : modèles et vérification. http://www.univ-pau.fr/csysec/page_html/projets_-_as_securite/.
- [3] ActivCard Gold for CAC. http://www.activcard.com/en/products/4_2_2_gold_for_cac.php.
- [4] CCIMB. <http://www.commoncriteriaportal.org/>.
- [5] COFRAC. <http://www.cofrac.fr/>.
- [6] Cyberflex Access Home Page. <http://www.cyberflex.slb.com/>.
- [7] Décrypton. <http://www.decrypthon.com/>.
- [8] DCSSI. <http://www.ssi.gouv.fr/fr/dcssi/index.html>.
- [9] Design Issues for Foreign Function Interfaces. A survey of existing native interfaces for several languages and some suggestions. <http://xarch.tu-graz.ac.at/autocad/lisp/ffis.html>.
- [10] DIET Homepage. <http://graal.ens-lyon.fr/DIET/>.
- [11] DRBS. <http://drbs.sourceforge.net/>.
- [12] EMVCo web site. <http://www.emvco.com/>.
- [13] ETSI. <http://www.etsi.org/>.
- [14] Extended Static Checking for Java. <http://research.compaq.com/SRC/esc/>.
- [15] Formal interface specifications for the Java Card API 2.1.1 in JML and ESC/Java. http://www.cs.kun.nl/~erikpoll/publications/jc211_specs.html.
- [16] GemXpresso RAD III Kit. http://www.gemplus.com/products/gemxpresso_rad_v3_kit/.
- [17] Generic CCID IFD Handler home page. <http://pcsc-lite.alieth.debian.org/ccid.html>.
- [18] Genopole, la cité du gène et des biotechnologies (génétique, biotechnologies, génomique). <http://www.genopole.org/>.
- [19] GlobalPlatform. <http://www.globalplatform.org/>.
- [20] GNU Prolog Wrappers for PC/SC. <http://www.musclecard.com/middleware/files/gplpcsc.tgz>.
- [21] Gnutella. <http://www.gnutella.com/>.

- [22] GSI Documentation. <http://www-unix.globus.org/toolkit/docs/3.2/gsi/index.html>.
- [23] IBM JCOP web site. <http://www.zurich.ibm.com/jcop/>.
- [24] International Common Criteria home page. <http://www.commoncriteriaportal.org/>.
- [25] ITSEC. http://www.ssi.gouv.fr/site_documents/ITSEC/ITSEC-fr.pdf.
- [26] JAR Files. <http://java.sun.com/docs/books/tutorial/jar/index.html>.
- [27] Java Card – Datasheet. <http://java.sun.com/products/javacard/datasheet.html>.
- [28] Java Card 2.2.1 Development Kit. http://java.sun.com/products/javacard/dev_kit.html.
- [29] Java Card Forum. <http://www.javacardforum.org/>.
- [30] Java-Powered Cryptographic iButton. <http://www.ibutton.com/ibuttons/java.html>.
- [31] Java Wrappers for PC/SC. <http://www.musclecard.com/middleware/files/jpcsc-0.8.0-src.zip>.
- [32] Jcave - javacard applet verification environment. <http://www.sics.se/fdt/vericode/jcave.html>.
- [33] Jive - java interactive verification environment. <http://www.sct.inf.ethz.ch/projects/#jive>.
- [34] Journée de la carte à puce à Bordeaux. http://labri-transfert.labri.fr/Archives_news/carte_a_puce.htm.
- [35] JXTA. <http://www.jxta.org/>.
- [36] Kazaa. <http://www.kazaa.com/>.
- [37] LaserCard web site. <http://www.lasercard.com/index.php>.
- [38] Legion : A Worldwide Virtual Computer. <http://legion.virginia.edu/>.
- [39] Les certificats. <http://www.commentcamarche.net/crypto/certificat.php3>.
- [40] Mobile Ad-hoc Networks (manet). <http://www.ietf.org/html.charters/manet-charter.html>.
- [41] MUSCLE home page. <http://www.musclecard.com/>.
- [42] MuscleCard home page. <http://www.musclecard.com/musclecard/index.html>.
- [43] Napster.com. <http://www.napster.com/>.
- [44] Netsolve. <http://icl.cs.utk.edu/netsolve/>.
- [45] Oasis. <http://www-sop.inria.fr/oasis/>.
- [46] OCF To PC/SC Shim. <http://www.musclecard.com/middleware/files/OCFPCSC1-0.0.1.tar.gz>.
- [47] OpenCard Framework. <http://www.opencard.org/>.
- [48] pcsc-lite home page. <http://alioth.debian.org/projects/pcsc-lite/>.
- [49] PC/SC Workgroup. <http://www.pcscworkgroup.com/>.

- [50] Perl Wrappers for PC/SC. <http://ludovic.rousseau.free.fr/software/pcsc-perl/>.
- [51] Projet XtremWeb. <http://www.xtremweb.net/>.
- [52] Pvs specification and verification system. <http://pvs.csl.sri.com/>.
- [53] Python Wrappers for PC/SC. <http://homepage.mac.com/jlgiraud/pycsc/Pycsc.html>.
- [54] RSA-155 is factored! <http://www.rsasecurity.com/rsalabs/node.asp?id=2098>.
- [55] RTP19 : Systèmes embarqués complexes ou contraintes. <http://www.cnrs.fr/STIC/Actions/Outils/RTP/PopupsRTP/RTP19.html>.
- [56] Ruby Wrappers for PC/SC. <http://raa.ruby-lang.org/project/pcsc-ruby/>.
- [57] SETI@home : Search for Extraterrestrial Intelligence at home. <http://setiathome.ssl.berkeley.edu/>.
- [58] Sharp Microelectronics Announces Development of Smart Card with 1MB Flash Memory. <http://www.sharpsma.com/sma/pressroom/press.htm?newsid=83>.
- [59] Sm@rtCafé Professional Toolkit 2.0. http://www.gi-de.com/portal/page?_pageid=44,76028&_dad=portal&_schema=PORTAL.
- [60] The Freenet Network Project. <http://freenet.sourceforge.net/>.
- [61] The Globus Alliance. <http://www.globus.org/>.
- [62] The Google File System. <http://www.cs.rochester.edu/sosp2003/papers/p125-ghemawat.pdf>.
- [63] The Java Modeling Language (JML) Home Page. <http://www.jmlspecs.org/>.
- [64] The Smart Sign project. <http://sourceforge.net/projects/smartsign/>.
- [65] UNICORE Forum e.V. <http://www.unicore.org/>.
- [66] USB Implementers Forum. <http://www.usb.org/info>.
- [67] X.509. <http://en.wikipedia.org/wiki/X.509>.
- [68] Peter Alfeld. The Mandelbrot Set. <http://www.math.utah.edu/~alfeld/math/mandelbrot/mandelbrot.html>.
- [69] K. Arnold, J. Gosling, and D. Holmes. *The Java programming language (3rd Edition)*. Addison-Wesley, 2000.
- [70] M. P. Atkinson and R. Morrison. Orthogonally persistent object systems. In *The VLDB Journal*, 1995.
- [71] Isabelle Attali, Denis Caromel, Carine Courbis, Ludovic Henrio, and Henrik Nilsson. Smart tools for Java Cards. In *Proceedings of the fourth working conference on smart card research and advanced applications on Smart card research and advanced applications*, pages 155–177. Kluwer Academic Publishers, 2001.
- [72] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The Sorcerer’s Apprentice Guide to Fault Attacks. In *Proceedings of Workshop on Fault Detection and Tolerance in Cryptography*, Italy, 2004.
- [73] G. Barthe, G. Dufay, M. Huisman, and S. Sousa. Jakarta : a toolset to reason about the JavaCard platform. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security (Proceedings of e-Smart’01)*, volume 2140 of Lecture Notes in Computer Science, pages 2–18. Springer Verlag, September 2001.

- [74] G. Barthe, G. Dufay, L. Jakubiec, S. Melo de Sousa, and B. Serpette. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of Lecture Notes in Computer Science, pages 302–319. Springer Verlag, 2001.
- [75] G. Barthe, G. Dufay, L. Jakubiec, and S. Melo de Sousa. A Formal Correspondence between Offensive and Defensive JavaCard Virtual Machines. In *Proceedings of VM-CAI'02*, volume 2294 of Lecture Notes in Computer Science, pages 32–45, Venice, Italy, 2002. Springer-Verlag.
- [76] J. van den Berg, B. Jacobs, and E. Poll. Formal Specification and Verification of JavaCard's Application Identifier Class. In I. Attali and T. Jensen, editors, *Java on Smart Cards : Programming and Security*, volume 2041 of Lecture Notes in Computer Science, pages 137–150. Springer-Verlag, 2001.
- [77] Klaus Bergner, Andreas Rausch, Marc Sihling, and Christoph Vilsmeier. CORBA and the Java Card - Connecting Small Devices to a Standard Event Service. In *1st Symposium on Reusable Architectures and Components for Developing Distributed Information Systems (RACDIS'99)*, 1999.
- [78] Gustavo Betarte, Eduardo Giménez, Boutheina Chetali, and Claire Loiseaux. FORMAVIE : Formal Modelling and Verification of Java Card 2.1.1 Security Architecture. In *Proceedings of E-Smart 2002*, pages 215–229, Nice, France, September 19-20 2002.
- [79] Pierre Bieber, J. Cazin, Pierre Girard, Jean Louis Lanet, Virginie Wiels, and Guy Zanon. Checking Secure Interactions of Smart Card : Applets Extended Version. *Computer Security*, 10(4) :369–398, December 2002. Special issue on ESORICS 2000.
- [80] Richard Brinkman and Jaap-Henk Hoepman. Secure Method Invocation in Jason.
- [81] Denis Caromel, Ludovic Henrio, and Bernard P. Serpette. Context Inference for Static Analysis of Java Card Object Sharing. In *Proceedings of the International Conference on Research in Smart Cards*, pages 43–57. Springer-Verlag, 2001.
- [82] CCIMB. *Common Methodology for Information Technology Security Evaluation*. january 2004.
- [83] Canadian System Security Center. The Canadian Trusted Computer Product Evaluation Criteria. <ftp://ftp.cse-cst.gc.ca/pub/criteria/CTCPEC/CTCPEC.ascii>, january 1993.
- [84] National Computer Security Center. Department of Defense Trusted Computer System Evaluation Criteria (TCSEC or Orange Book). <http://csrc.nist.gov/secpubs/rainbow/std001.txt>, 1983.
- [85] Alvin T.S. Chan, Florine Tse, Jiannong Cao, and Hong Va Leong. Enabling Distributed Corba Access to Smart Card Applications. In *IEEE Internet Computing*, pages 27–36, May/June 2002.
- [86] Serge Chaumette, Pascal Grange, Achraf Karray, Damien Sauveron, and Pierre Vignéras. Secure distributed computing on a Java Card Grid. Position paper. Technical report, LaBRI, 2004.
- [87] Serge Chaumette, Pascal Grange, Damien Sauveron, and Pierre Vignéras. Computing with Java Cards. In *Proceedings of CCCT'03 and 9th ISAS'03*, Orlando, FL, USA, July 31, August 1-2 2003.

- [88] Serge Chaumette, Iban Hatchondo, and Damien Sauveron. JCAT : An environment for attack and test on Java Card. In *Proceedings of CCCT'03 and 9th ISAS'03*, volume 1, pages 270–275, Orlando, FL, USA, July 31, August 1-2 2003.
- [89] Serge Chaumette and Damien Sauveron. The Smart Cards Grid Project. <http://www.labri.fr/Person/~chaumett/recherche/cartesapuce/smartcardsgrid/documents/poster.pdf>. Poster présenté à Cartes 2003.
- [90] Serge Chaumette and Damien Sauveron. Modèles de mémoire en Java Card. Introduction du concept de pré-persistence. In *Proceedings of MAJECSTIC'04*, 13 – 15 octobre 2004.
- [91] Serge Chaumette and Damien Sauveron. New security problems raised by open multiapplication smart cards. Technical report, LaBRI, 2004.
- [92] Serge Chaumette and Pierre Vignéras. A framework for seamlessly making object oriented applications distributed. In *Parallel Computing 2003*, Dresden, Germany, September 2-5 2003.
- [93] Zhiquan Chen. *Java CardTM Technology for Smart Cards : Architecture and Programmer's Guide*. Addison-Wesley, 2000.
- [94] G. Chugunov, L. Fredlund, and D. Gurov. Model Checking of Multi-Applet Java-Card Applications. In *Fifth Smart Card Research and Advanced Application Conf. (CARDIS'2002)*, pages 87–96. USENIX, 2002.
- [95] R. M. Cohen. Defensive Java Virtual Machine Version 0.5 alpha., 1997.
- [96] OpenCard Consortium. OpenCard Framework (OCF) : Frequently Asked Questions : Positioning of OCF Versus PC/SC. <http://www.opencard.org/misc/OCF-FAQ.shtml#PCSC>.
- [97] David Corcoran. M.U.S.C.L.E : porting the PC/SC architecture to Linux. Gemplus Developer's Conference 99, june 1999.
- [98] Jean-Sébastien Coron, Paul Kocher, and David Naccache. Statistics and Secret Leakage. In *Proceedings of Financial Cryptography (FC2000)*, volume 1962 of Lecture Notes in Computer Science, pages 157–173. Springer-Verlag, 2001.
- [99] DCSSI. Rapport de certification PP/0303 JavaCard System Minimal Configuration Protection Profile version 1.0b. http://www.ssi.gouv.fr/site_documents/PP/PPCR0303.pdf, septembre 2003.
- [100] DCSSI. Rapport de certification PP/0304 JavaCard System Standard 2.1.1 Configuration Protection Profile version 1.0b. http://www.ssi.gouv.fr/site_documents/PP/PPCR0304.pdf, septembre 2003.
- [101] DCSSI. Rapport de certification PP/0305 JavaCard System Standard 2.2 Configuration Protection Profile version 1.0b. http://www.ssi.gouv.fr/site_documents/PP/PPCR0305.pdf, septembre 2003.
- [102] DCSSI. Rapport de certification PP/0306 JavaCard System Defensive Configuration Protection Profile version 1.0b. http://www.ssi.gouv.fr/site_documents/PP/PPCR0306.pdf, septembre 2003.
- [103] E. Denney. Correctness of Java Card Tokenisation. Technical report, 1999. Projet Lande, IRISA.

- [104] E. Denney and T. Jensen. Correctness of Java Card method lookup via logical relations. *Theoretical Computer Science*, 283 :305–331, 2002.
- [105] Damien Deville and Gilles Grimaud. Building an “impossible” verifier on a Java Card. In *Proceedings of the 2nd USENIX Workshop on Industrial Experiences with Systems Software*, Boston, USA, 2002.
- [106] Damien Deville and Gilles Grimaud. Building an “impossible” verifier on a Java Card. In *2nd USENIX Workshop on Industrial Experiences with Systems Software*, Boston, USA, 2002.
- [107] Didier Donsez, Sebastien Jean, Sylvain Lecomte, and Olivier Thomas. (A)synchronous Use of Smart Cards Services Using SOAP and JMS.
- [108] International Organization for Standardization. *ISO7816*. ISO.
- [109] International Organization for Standardization. *Information Technology – Security Techniques — Evaluation Criteria for IT Security – Part 1 : Introduction and General Model, ISO-IEC 15408-1*. ISO, 1999.
- [110] International Organization for Standardization. *Information Technology – Security Techniques — Evaluation Criteria for IT Security – Part 2 : Security Functional Requirements, ISO-IEC 15408-2*. ISO, 1999.
- [111] International Organization for Standardization. *Information Technology – Security Techniques — Evaluation Criteria for IT Security – Part 3 : Security Assurance Requirements, ISO-IEC 15408-3*. ISO, 1999.
- [112] USB Implementers Forum. Universal Serial Bus Device Class Specification for USB Chip/Smart Card Interface Devices version 1.00. http://www.usb.org/developers/devclass_docs/ccid_classspec_1_00a.pdf, march 2001.
- [113] Ian Foster and Carl Kesselman. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.
- [114] Fujitsu. Comparison of FRAM® with other memory products. <http://www.fme.fujitsu.com/products/fram/technology.html>.
- [115] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley.
- [116] Karine Gandol, Christophe Mourtel, and Francis Olivier. ElectroMagnetic Analysis : Concrete Results. In *Proceedings of CHES’2001*, volume 2162 of Lecture Notes in Computer Science, pages 251–261. Springer-Verlag, 2001.
- [117] Pierre Girard. Which security policy for multiapplication smart cards ? In *Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard ’99)*, pages 21–28, Chicago, Illinois, USA, May 10-11 1999.
- [118] Pierre Girard and Jean-Louis Lanet. Java Card or How to Cope with the New Security Issues Raised by Open Cards ? In *Proceedings of Gemplus Developer Conference*, Paris, France, June 21-22 1999.
- [119] Pierre Girard and Jean-Louis Lanet. New Security Issues raised by Open Cards. In *Information Security Technical Report*, volume 4, pages 19–27, 1999.
- [120] C. Giraud and H. Thiebauld. A survey on fault attacks. In *Proceedings of CARDIS’04, Smart Card Research and Advanced Applications VI*, pages 159–176, Toulouse, France, August 24-26 2004. Kluwer academic publisher.

- [121] GlobalPlatform. Card Security Requirements Specifications v1.0. <http://www.globalplatform.org/specifications/card/card-sec-req-v10.zip>.
- [122] GlobalPlatform. Card Specification v2.1.1. http://www.globalplatform.org/specifications/card/Version2_1_1_cardtech.zip.
- [123] GlobalPlatform. Device Tools Supporting Documents. http://www.globalplatform.org/uploads/gpstip_2003_09_09_public.zip.
- [124] GlobalPlatform. GlobalPlatform Device API v2.0. <http://www.globalplatform.org/specifications/device/Dev-tech2.zip>.
- [125] GlobalPlatform. GlobalPlatform Guidelines for Using GPD 2.0 /STIP 2.1 API v1.0. http://www.globalplatform.org/specifications/device/Guidelines_For_Using_GDP_20_STIP_21_API_Ver1_0.zip.
- [126] GlobalPlatform. GlobalPlatform Messaging Specification v1.0. http://www.globalplatform.org/specifications/systems/GPMessagingSpecification_v1_0_031014.zip.
- [127] GlobalPlatform. GP Guide to Common Personalization v1.0. http://www.globalplatform.org/specifications/systems/LPIS_GCP_v1.0.zip.
- [128] GlobalPlatform. GP Load and Personalization Interface v1.0. http://www.globalplatform.org/specifications/systems/LPIS_GCP_v1.0.zip.
- [129] GlobalPlatform. Guidelines for Developing Java Card applications on GlobalPlatform Cards v1.0. http://www.globalplatform.org/documents/supporting/Guidelines_for_Dev_Java_apps_on_GP_Cards_1_0.zip.
- [130] GlobalPlatform. Key Management Requirements Systems Functional Requirements Specification. http://www.globalplatform.org/specifications/systems/KMS_Functional_Reqs_v1_0.pdf.
- [131] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (2nd Edition)*. Addison-Wesley, 2000.
- [132] Sudhakar Govindavajhala and Andrew Appel. Using Memory Errors to Attack a Virtual Machine. In *Proceedings of IEEE Symposium on Security and Privacy*, 2003.
- [133] William Grosso. *Java RMI*. O'Reilly & Associates, 2002.
- [134] Patrick Gueulle. *Cartes à puce*. Editions Techniques et Scientifiques Françaises, 1993.
- [135] Patrick Gueulle. *PC et cartes à puce*. Editions Techniques et Scientifiques Françaises, 1995.
- [136] Scott Guthery and Mary Cronin. *Mobile Application Development with SMS and the SIM Toolkit*. McGraw-Hill Professional, 2001.
- [137] P. Gutmann. Data Remanence in Semiconductor Devices. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., USA, August 13-17 2001.
- [138] Uwe Hansmann, Martin S. Nicklous, Thomas Schäck, and Frank Seliger. *Smart Card Application Development Using Java*. Springer, 2000.
- [139] Vesna Hassler, Martin Manninger, Mikhail Gordeev, and Christoph Müller. *Java Card for E-Payment Applications*. Artech House Publishers, 2001.

- [140] Iban Hatchondo and Damien Sauveron. The JCatools website. <http://sourceforge.net/projects/jcatools/>.
- [141] Mike Hendry. *Smart Card Security and Applications, Second Edition*. Artech House Publishers, 2001.
- [142] Hive-Minded. Smartcard.NET. <http://www.hiveminded.com/>.
- [143] E. Hubbers and E. Poll. Reasoning about Card Tears and Transactions in Java Card. In *Fundamental Approaches to Software Engineering (FASE'2004)*, volume 2984 of Lecture Notes in Computer Science, pages 114–128, Barcelona, Spain, 2004. Springer-Verlag.
- [144] .NET Brings Web Services to Smart cards. *SmartCards Trends*, 1 :12, April-May 2004.
- [145] Simon Moore Jacques J.A. Fournier, Huiyun Li, Robert Mullins, and George Taylor. Security Evaluation of Asynchronous Circuits. In *Proceedings of CHES'2003*, volume 2779 of Lecture Notes in Computer Science, pages 137–151, 2003.
- [146] Sebastien Jean, Didier Donsez, and Sylvain Lecomte. Smart Card Integration in Distributed Information Systems : the Interactive Execution Model.
- [147] Thimothy M. Jurgensen and Scott B. Guthery. *Smart Cards : The Developer's Toolkit*. Prentice Hall, 2002.
- [148] Roger Kehr, Michael Rohs, and Harald Vogt. Issues in Smarcard Middleware.
- [149] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM Press, 1973.
- [150] Oliver Kömmerling and Markus G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '99)*, pages 9–20, Chicago, Illinois, USA, May 10-11 1999.
- [151] Paul Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 104–113. Springer-Verlag, 1996.
- [152] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 388–397. Springer-Verlag, 1999.
- [153] RSA Laboratories. *PKCS #11 v2.20 : Cryptographic Token Interface Standard*. RSA Laboratories, june 2004.
- [154] Xavier Leroy. Java bytecode verification : an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of 13th International Conference on Computer-Aided Verification*, volume 2102 of Lecture Notes in Computer Science, pages 265–285. Springer-Verlag, 2001.
- [155] Xavier Leroy. On-Card Bytecode Verification for Java Card. In *Proceedings of the International Conference on Research in Smart Cards, E-Smart 2001*, pages 150–164. Springer-Verlag, 2001.
- [156] Xavier Leroy. Bytecode verification on Java smart cards. *Software-Practice & Experience*, 32 :319–340, 2002.

- [157] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.
- [158] Trusted Logic. Formal methods, smart card, security. <http://www.trustedlogic.com/>.
- [159] Sergio Loureiro. *Mobile Code Protection*. PhD thesis, École Nationale Supérieure des Télécommunications, Sophia Antipolis, France, Janvier 2001.
- [160] Sergio Loureiro and Refik Molva. Mobile Code Protection with Smarcards.
- [161] Marc Éluard, Thomas P. Jensen, and Ewen Denney. An Operational Semantics of the Java Card Firewall. In *Proceedings of E-smart 2001*, pages 95–110. Springer-Verlag, 2001.
- [162] Hugues Martin and Lydie du Bousquet. Automatic Test Generation for Java-Card Applets. In *Revised Papers from the First International Workshop on Java on Smart Cards : Programming and Security*, pages 121–136. Springer-Verlag, 2001.
- [163] Hans Meijer and Erik Poll. Towards a full formal specification of the Java Card. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, number 2140 of Lecture Notes in Computer Science, pages 165–178. Springer-Verlag, September 2001.
- [164] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Investigation of Power Analysis Attacks on Smartcards. In *Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '99)*, Chicago, Illinois, USA, May 10-11 1999.
- [165] Microsoft. CryptoAPI. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/seccrypto/security/cryptography_essentials.asp.
- [166] Microsoft. SCard API. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthn/security/smart_card_resource_manager_api.asp.
- [167] Sun microsystems. Java Commerce/Smart Cards. <http://java.sun.com/products/commerce/javax.smartcard/>, 2001.
- [168] Sun microsystems. Java Card System Protection Profile Collection version 1.0b. <http://java.sun.com/products/javacard/JCSPPC-1.0b.pdf>, august 2003.
- [169] Sun microsystems. *Java CardTM 2.2.1 Application Programming Interface*. Sun microsystems, 2003.
- [170] Sun microsystems. *Java CardTM 2.2.1 Runtime Environment (JCRE) Specification*. Sun microsystems, 2003.
- [171] Sun microsystems. *Java CardTM 2.2.1 Virtual Machine Specification*. Sun microsystems, 2003.
- [172] Sun microsystems. Overview of the Java Card System Protection Profile Collection version 1.0. <http://java.sun.com/products/javacard/JCSPPC-Overview-1.0.pdf>, august 2003.
- [173] Michael Montgomery and Ksheerabdh Krishna. Secure Object Sharing in Java Card. In *Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '99)*, Chicago, Illinois, USA, May 10-11 1999.
- [174] Simon Moore, Ross Anderson, Paul Cunningham, Robert Mullins, and George Taylor. Improving Smart Card Security using Self-timed Circuits. In *Proceedings of ASYNC'02*, pages 211–218, 2002.

- [175] James Alexandre Muir. Techniques of Side Channel Cryptanalysis. Master's thesis, University of Waterloo, Ontario, Canada, 2001. Master of Mathematics in Combinatorics and Optimization.
- [176] MULTOS. The MULTOSTM Specification. <http://www.multos.com/>.
- [177] George C. Necula and P. Lee. Proof-Carrying Code. In *Conference Record of POPL '97 : The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.
- [178] Marcus Oestreicher. Transactions in Java Card. In *Proceedings of 15th Annual Computer Security Applications Conference (ACSAC)*, Phoenix, Arizona, USA, 1999.
- [179] Marcus Oestreicher and Ksheerabdh Krishna. Object lifetimes in Java Card. In *Proceedings of Usenix Workshop on Smartcard Technology (Smartcard'99)*, Chicago, Illinois, USA, May 10-11 1999.
- [180] D. Perovich. Secure Object Sharing Development Kit for JavaCard. Presented at the Verificard project workshop, January 2002.
- [181] Michael Philippsen and Matthias Zenger. JavaParty : Transparent remote objects in Java. *Concurrency : Practice and Experience*, November 1997.
- [182] E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000)*, pages 135–154. Kluwer Acad. Publ., 2000.
- [183] E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the Java Card API in JML : the APDU class. *Computer Networks*, 36(4) :407–421, 2001.
- [184] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA) : Measures and Countermeasures for Smart Cards. In *Proceedings of E-smart 2001*, volume 2140 of Lecture Notes in Computer Science, pages 200–210. Springer-Verlag, 2001.
- [185] Jean-Jacques Quisquater and David Samyde. Automatic Code Recognition for smart cards using a Kohonen neural network. In *Proceedings of the 5th Smart Card Research and Advanced Application Conference (CARDIS'02)*, 2002.
- [186] Andrew Roberts. Smart Meiji Contactless Roadmap. http://www.eurosmart.com/meiji/download/March2003/TK_0303_CtlWs_Roberts.pdf, march 2003.
- [187] E. Rose and K. Rose. Lightweight bytecode verification. In *In Workshop on Fundamental Underpinnings of Java, OOPSLA '98 Workshop.*, Vancouver, Canada, October 1998.
- [188] Damien Sauveron. La technologie Java Card. Présentation de la carte à puce. La Java Card. Rapport de recherche, LaBRI, mai 2001.
- [189] Damien Sauveron. Sécurité et vérification d'applications embarquées en environnement Java Card. Mémoire de dea, LaBRI, juin 2001.
- [190] Gerhard Schellhorn, Wolfgang Reif, Axel Schairer, Paul Karger, Vernon Austel, and David Toll. Verification of a Formal Security Model for Multiapplicative Smart Cards. In *Proceedings of ESORICS 2000*, volume 1895 of Lecture Notes in Computer Science, pages 17–36, 2000.

- [191] A. Schultz. Verification of Java Card applets. Master's thesis, Universität Kaiserslautern, 2003.
- [192] Frank Seliger. The OpenCard Framework and PC/SC – Two New Industry Initiatives for Smart Cards. <http://www.opencard.org/docs/ocfpcsc.pdf>, 1999.
- [193] Igor Siveroni and Chris Hankin. A Proposal for the JCVMLe Operational Semantics. Technical report, SecSafe, 2001.
- [194] Igor Siveroni, Thomas Jensen, and Marc Eluard. A Formal Specification of the Java Card Firewall. Technical report, SecSafe, 2001.
- [195] Sergei Skorobogatov and Ross Anderson. Optical Fault Induction Attacks. In *Proceedings of Workshop on Cryptographic Hardware and Embedded Systmes (CHES 2002)*, San Francisco Bay (Redwood City), USA, August 13-15 2002.
- [196] Alicja Beata Szczurowska. MRAM – Preliminary analysis for file system design. Technical report, UNIVERSITY of CALIFORNIA, 2002.
- [197] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1994.
- [198] K. Trentelman and M. Huisman. Extending JML specifications with temporal logic. In *Algebraic Methodology And Software Technology (AMAST '02)*, volume 2422 of Lecture Notes in Computer Science, pages 334–348. Springer-Verlag, 2002.
- [199] Joachim van den Berg and Bart Jacobs. The LOOP Compiler for Java and JML. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–312. Springer-Verlag, 2001.
- [200] Pierre Vigneras and Pascal Grange. The DJFractal project. <http://djfractal.sf.net/>.
- [201] Pierre Vigneras and Pascal Grange. The Mandala website. <http://mandala.sourceforge.net/>.
- [202] Visa. Open Platform. <http://international.visa.com/fb/vendors/openplatform/techdownloads.jsp>.
- [203] David A. Watt and Deryck F. Brown. *Programming Language Processors in Java : Compilers and Interpreters*. Prentice Hall, 2000.
- [204] PC/SC Workgroup. PC/SC Workgroup Specifications 1.0. <http://www.pcscworkgroup.com/specifications/specdownloadV1.php>.
- [205] PC/SC Workgroup. Introducing the PC/SC Specifications 2.0. http://www.pcscworkgroup.com/library/files/PCSC2_0WhitePaper.pdf, 1999.
- [206] PC/SC Workgroup. PC/SC Workgroup Specifications 2.0 draft for public review. <http://www.pcscworkgroup.com/specifications/specdownload.php>, juin 2004.
- [207] ZeitControl. BasicCard. <http://www.basiccard.com/>.
- [208] Sha Zhu, Amame Inoue, Seigen Otani, and Eiichi Nagai. A Non-Volatile Ferroelectric Memory. In *Proceedings of ICDA2000 - International Conference on Chip Design Automation*, 2000.

RÉSUMÉ :

En s'appuyant sur une collaboration menée entre le LaBRI et SERMA Technologies sur la sécurité des Java Cards, cette thèse a pour objectif d'étudier les nouvelles problématiques de sécurité liées à l'apparition de la technologie Java Card. Elle a donné lieu au développement d'un émulateur Java Card capable de simuler des attaques au niveau matériel et logiciel. Elle a également mis en évidence un certain nombre de problèmes dans les spécifications Java Card et des vulnérabilités potentielles dans le cadre de la multi-application sur cartes à puce ouvertes. Par ailleurs, les travaux de cette thèse ont permis de proposer des solutions visant à protéger des codes mobiles pour faire du calcul distribué grâce à l'utilisation de Java Cards. Une application directe de ces travaux permettra de sécuriser le calcul sur la grille.

MOTS-CLÉS : Java Card, modèle mémoire, attaques, plate-forme de confiance, outils logiciels, smart card, plate-forme distribuée.

ABSTRACT :

Based on a partnership between LaBRI and SERMA Technologies on the security of Java Cards, the aim of this thesis is to study the new security problems raised by the appearance of the Java Card technology. We have developed a Java Card emulator that can simulate attacks at both hardware and software levels. We also highlighted some problems in the Java Card specifications and potential vulnerabilities within open multiapplication smart cards. This thesis also made it possible to propose solutions to protect mobile code used to achieve distributed computations based on Java Cards. A direct application of this work will allow to securise the grid computing paradigm.

KEYWORDS : Java Card, memory model, attacks, trusted platform, software tools, smart card, distributed platform.