Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Standardisation d'une librairie de cryptographie au format PKCS#11

Mamboug, Laurent

Award date:
2002

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 21. Mar. 2025

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'informatique

Standardisation d'une librairie de cryptographie au format PKCS #11

L. MAMBOURG

Sous la direction de Monsieur

Jean Ramaekers

Mémoire présenté par Laurent Mambourg
vue de l'obtention du grade de Maître en
informatique des Facultés Universitaires
Notre-Dame de la Paix à Namur

Année Académique 2001-2002

LBS 100 79957

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'informatique

Standardisation d'une librairie de cryptographie au format PKCS #11

Sous la direction de Monsieur

Jean Ramaekers

Mémoire présenté par Laurent Mambourg
vue de l'obtention du grade de Maître en
informatique des Facultés Universitaires
Notre-Dame de la Paix à Namur

Année Académique 2001-2002

ABSTRACT

Ce travail a pour objectif de présenter le standard de cryptographie *Public-Key Cryptography Standard* n°11 du laboratoire *RSA*. Celui-ci spécifie les fonctions à implémenter pour effectuer une cryptographie au moyen d'un "jeton". Nous étudions différents aspects de ce standard comme par exemple les problèmes de compatibilité mais également sa structure fonctionnelle. Un autre aspect de ce travail est l'explication d'une application qu'il nous a été demandé de réaliser : l'"interfaçage" d'une librairie de cryptographie développée par *BULL* appelée *Cryptographic Support Facility* (CSF) avec la librairie *PKCS #11*. Enfin un dernier point abordé consiste en une présentation et une classification des différents types de *Smart Cards* existants à l'heure actuelle

---

The goal of this thesis is to present the cryptography standard *Public-Key Cryptography Standard* n°11 of RSA laboratories. It specifies wich functions has to be implemented to be able to accomplish token cryptography. We study different aspects of this standard, for example the problems of compatibility but also his functionnal structure. Another side of this work is the explanation of the application we were asked to do: the interface between the cryptographic library developped by BULL called *Cryptographic Support Facility* (CSF) with *PKCS #11* library. Last but not least, we present and classify of the different types of Smart Card available at the moment.

Keywords : Token Cryptography, Cryptoki, PKCS #11, CSF, Smart Card

Cette page, bien que cela puisse paraître étrange, a déjà été écrite à de multiples reprises dans mes rêves, dans ses moments de doutes durant ces années d'études. Elle constitue pour moi le moyen le plus aisé de remercier toutes ces personnes, présentes à mes côtés, qui m'ont soutenues dans les moments de joies comme dans les moments de tristesses. Le nombre de ces personnes est malheureusement plus grand que la place qui m'est imparti, c'est pourquoi je prierai d'avance les personnes concernées non citées de m'en excuser... Je voudrai commencer par remercier Monsieur le professeur J.Ramaekers qui a bien voulu partager ses compétences et qui m'a guidé tout au long de ce travail. Il me faut bien évidemment remercier les membres du service SSO d'Evidian aux Clays-Sous-Bois (BULL) pour la gentillesse et l'attention qu'ils m'ont porté durant ces quelques mois passés avec eux. En particulier, je voudrai remercier Mr J.Lebastard, mon maître de stage, pour ses explications avisées et son aide durant mon stage. De même F.Viollet et J-J. Delgove qui ont bien voulu m'accorder un peu de leur temps pour répondre à mes petites, mais nombreuses, questions. Que soient remercié aussi Isabelle pour son soutient durant cette année de rédaction. Et pour terminer, ma famille pour son aide et sans qui je n'aurai sans doute pas eu la force d'entreprendre tout ce qui a été fait.

Merci à tous

Laurent

TABLE DES MATIÈRES

TABLE DES ILLUSTRATIONS

GLOSSAIRE

API	Application Programming Interface
AS	Authentification Server
ASCII	American Standard Code for Information Interchange
ASN.1	Abstract Syntax Notation One
ATR	Answer To Reset
BER	Basic Encoding Rules
CA	Certification Authority
CAD	Card Acceptance Device
CCITT	International Telegraph and Telephone Consultative Committee
CPU	Central Processor Unit
CTI	Cryptographic Token Interface
CSF	Cryptographic Support Facility
DER	Distinguished Encoding Rules
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
ECDSA	Elliptic Curve Digital Signature Algorithm
ECMA	European Computer Manufacturer's Association
EPROM	Electrically Programmable Read-Only Memory
EEPROM	Electrically Erasable Programmable Read-Only Memory
FIPS-PUB	Federal Information Processing Standard Publication
GSS-API	Generic Security Services Application Program Interface
HSH	Secure Hash Standard
IAB	Internet Architecture Board
IEC	International Electronical Commission
ISO	International Standards Oraganisation
IPRA	Internet PCA Registration Authority
ITU-T	Interational Telecommunication Union – Telecommunications Standardization Sector
KDS	Key Distribution Service
MAC	Message Authentification Code
NITS	National Institute for Standards and Technology
OID	Object Indentifier
OS	Operating System
OSI	Open System Interconnection
PAC	Privilege Attribute Certificate
PAS	Privilege Attribute Server
PCA	Policy Certification Authority
PEM	Privacy Enhanced Mail
PER	Packed Encoding Rules
PIN	Personal Identification Number
PKCS	Public Key Cryptography Standard
PKI	Public Key Infrastructure
PROM	Programmable Read Only Memory
PSRG	Privacy and Security Research Group
QOS	Quality Of Service

RAM	Random Access Memory
ROS	Reader Opertaing System
RFC	Request For Comment
ROM	Read Only Memory
RSA	Rivest-Shamir-Adleman
SESAME	Secure European System for Applications in a Multi-vendor Environment
SHA-1	Secure Hash Algorithm
SCOS	Smart Card Operating System
SO	Security Officier
SSO	Single Sign On
SMTP	Simple Message Transfer Protocol

INTRODUCTION

La généralisation de l'utilisation des moyens cryptographiques pour sécuriser les échanges de données en informatique à entraîné un accroissement des produits commerciaux touchant à la sécurité. Un nombre élevé de sociétés proposent de multiples solutions pour sécuriser les échanges. Le problème engendré par cette multitude de produits de sécurité est la faible ou la non compatibilité de ces solutions. Cette situation, bien connue en informatique, est aggravée dans le domaine de la sécurité pour de multiples raisons. Tout d'abord, la nature même du secteur entraîne qu'une société ne peut divulguer ses choix d'implémentations. Ensuite, par le fait que la sécurité est à la fois un domaine touchant aussi bien l'informatique logicielle que matérielle. Non content de bousculer les conceptions logicielles de ces dernières années, ce domaine, par son évolution, introduit également une nouveauté au niveau matériel. Cela aggrave les problèmes de compatibilités car une compatibilité est souhaitable non seulement entre les logiciels, mais également entre ceux-ci et les différents composants matériels présents sur le marché. Les solutions théoriques à ces problèmes sont entre les mains des différents organismes internationaux de normalisation. Elles sont théoriques car en plus des difficultés rencontrées au niveau économique pour trouver une entente, il existe aussi des facteurs aggravants comme des conflits géopolitiques entre les différents organismes de normalisation. Nous pouvons dire qu'à l'heure actuelle une standardisation des composants physiques est suivie (il ne viendrait à l'idée d'aucune banque de fabriquer des cartes d'une taille différente du format actuel) mais qu'il reste de nombreux problèmes de communication entre ces composants et les applications les utilisant.

Ce besoin de standardisation des procédés a été compris par le laboratoire *RSA* qui, au début des années 90, a commencé à développer ses propres standards. Ceux-ci, appelés *Public-Key Cryptography Standards* (PKCS), sont au nombre de 15 à l'heure actuelle. Rappelons que *RSA* est le nom d'un algorithme de cryptographie très répandu et portant le nom de ces trois inventeurs : R.Rivest, A.Shamir, L.Adleman.

Notre travail étudie un domaine bien précis de la sécurité qu'est la cryptographie au moyen d'un "jeton". Le terme "jeton" est utilisé pour référencer tous les moyens, connus et à venir, qui permettent d'effectuer des opérations cryptographiques. Les "jetons" ont l'inconvénient de ne pouvoir effectuer aucune opération complexe de manière autonome (bien que le futur laisse entrevoir le contraire) mais ont l'avantage, s'ils sont conçus de manière adéquate, de pouvoir fonctionner avec une multitude d'applications.

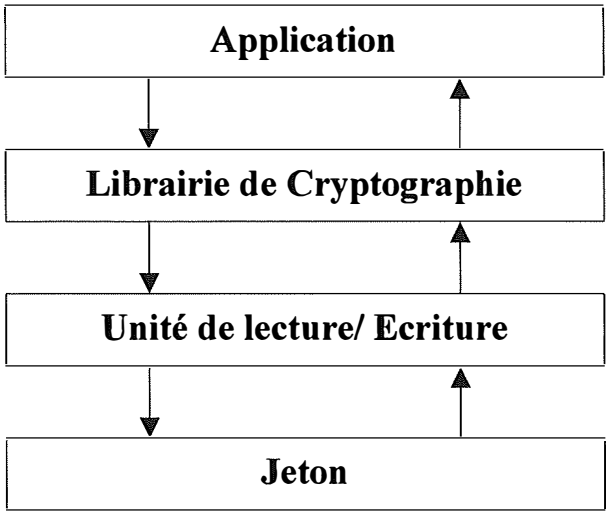

Figure I.1 Schéma des différentes couches d'une cryptographie par jeton

Lorsqu'une application veut effectuer une opération cryptographique quelconque sur un jeton, l'ordre doit d'abord transiter par la librairie qui s'occupe de ce type d'opération pour l'application. C'est la librairie qui se charge d'effectuer toutes les opérations cryptographiques nécessaire. Pour ce faire, elle en transmet ses ordres à l'unité de lecture / écriture qui s'occupe de faire le lien entre le jeton et la librairie. Le rôle du jeton consistant à interpréter les ordres et à fournir les données souhaitées. Remarquons que rien n'empêche que le jeton soit une solution logicielle .

L'ensemble du domaine de la cryptographie par jeton est normalisé par un des standards que nous avons évoqués auparavant, il s'agit du $11^{ième}$ de la série des *PKCS*. Etant le onzième d'une série de quinze, nous allons parcourir, de manière succincte, les différents standards développés par ce laboratoire. Comme un standard n'est d'aucune utilité s'il ne s'intègre pas avec les normes existantes, nous ne manquerons pas d'analyser le contexte dans lequel il intervient.

Pour mieux comprendre ce problème de compatibilité, nous complèterons notre exposé par une analyse détaillée de ce standard en présentant le problème qui m'a été demandé de résoudre durant mon stage de fin d'étude.

L'objectif de celui-ci était d'effectuer une mise à niveau de la librairie de cryptographie existante pour qu'elle puisse supporter le standard *PKCS #11*. En effet, pour de multiples raisons, la librairie existante, *Cryptographic Support Facility* (CSF), devait pouvoir supporter des "jetons" au format spécifié par le standard *PKCS #11*. Nous étudierons donc de manière approfondie le fonctionnement de la librairie *CSF*.

Ce travail serait incomplet si nous ne présentions pas l'élément essentiel pour de telles librairies : le "jeton". Pour ce faire, nous présenterons une de ses formes les plus utilisées à savoir : les *Smart Card*s, sous quelles formes il est possible de les rencontrer et quelles sont leurs principales applications.

CHAPITRE 1 : PUBLIC-KEY CRYPTOGRAPHY STANDARD

Au fur et à mesure du développement et de l'utilisation du chiffrement à clé publique, il est apparu évident qu'une standardisation des méthodes de chiffrement s'avérait nécessaire. Une compatibilité simple, au niveau de l'implémentation des techniques par exemple, entre partenaires ne suffisait pas étant donné leur utilisation croissante.

Dans ce but, les inventeurs d'un des mécanismes à clé publique le plus connu (R.Rivest, A.Shamir, L.Adleman) ont créé un laboratoire, *RSA corporation*, et ont soumis en 1991 un standard qu'il conviendrait d'utiliser avec leur mécanisme de chiffrement. Ce premier standard définit les méthodes pour signer et chiffrer des données avec l'algorithme *RSA* à clé publique. Par la suite, ce même laboratoire a défini d'autres spécifications qui sont devenues des normes à part entière vu leur pertinence et la réputation de ce laboratoire.

Section 1 : Origine et buts de ces standards[1]

Il est possible de scinder cette origine en deux phases : une standardisation des méthodes pour l'utilisation du mécanisme *RSA* à clé publique à l'origine. Ensuite un changement de politique de normalisation est intervenu, marqué par une volonté de produire des standards de sécurité ayant une application plus large. La première phase est la source des standards *PKCS #1*, *PKCS #2* et *PKCS #4*. D'ailleurs *PKCS #2* et *PKCS #4* ont été regroupés avec *PKCS #1*. Les standards suivants ont une portée beaucoup plus générale et ne se limitent plus au seul algorithme de cryptographie *RSA*. Pourquoi ce changement de politique ? Il est possible d'y répondre en analysant le contexte de standardisation même.

---

[1] Cette partie est largement inspirée par l'article de B.S.Jr. Kalisky et celui de S.T.Kent. Tous les éléments qui ne proviennent pas de ces auteurs seront nommés séparément dans les notes de bas de page.
B.S.Jr. KALISKY," An overview of the PKCS standards", *RSA Laboratories Technical Note*, 1993, pp. 1-19 et S.T.KENT, *Essay 17 Privacy Enhanced Mail*, http://www.acsac.org/secshelf/b.pdf, pp. 405-422.

En effet, le laboratoire *RSA* n'est pas une entité reconnue comme un organisme de standardisation à part entière comme l'est l'*International Standards Oraganisation* (ISO) ou l'*International Telecommunication Union – Telecommunications Standardisation Sector* (ITU-T).

Un standard produit par le laboratoire *RSA* correspond à une norme qui s'impose *de facto*[2] à l'inverse d'une norme *de jure* qui, elle, a été étudiée et publiée par un organisme officiel s'occupant de standardisation. La différence principale entre ces deux types de normes est la manière dont elles ont acquis ce statut.

Dans le cas d'une norme *de jure*, l'organisme publie cette norme qui est le résultat d'un compromis entre différents partenaires faisant partie de cette organisation. De la notoriété de celle-ci et de l'adéquation des recommandations dépendent la diffusion, la reconnaissance et l'utilisation de la norme.

Dans le cas d'une norme *de facto*, la publication d'un projet de standard n'est pas dû à un organisme internationalement reconnu comme éditeur de normes. Cela implique que le projet n'acquiert pas automatiquement, à sa publication, le statut de norme et que seule une reconnaissance par l'ensemble de la profession peut lui conférer le titre de standard à part entière. L'avantage de la deuxième situation est sans aucun doute que lorsque le projet acquiert le statut de norme son utilisation est déjà généralisée. Ce n'est pas du tout le cas des normes *de jure*, qui sont la solution d'un compromis, entre différents partenaires, qui arrange à la fois tout le monde et personne.

Il faut cependant noter que le laboratoire *RSA* suit une phase d'élaboration qui prend en compte l'avis de la communauté des programmeurs et utilisateurs pour que ces standards suivent le plus près possible leurs souhaits. De manière générale, une première esquisse est fournie, suivie par des modifications ou non, en fonction de la pertinence des demandes et enfin par la publication officielle du standard.

---

[2] Ceveil, *Préambule à la normalisation*, http://www.ceveil.qc.ca/Normes/premb.html , p. 1.

Mais une nouvelle norme ne doit pas simplement apporter une innovation dans un domaine précis, elle doit également veiller à s'intégrer au mieux avec l'environnement existant. On peut remarquer que bien que *PKCS* veuille se placer en tant que standard, il reste néanmoins tributaire de ceux développés avant lui et se trouvant sur une couche applicative plus basse. Si cette compatibilité n'est pas respectée, cela pourrait empêcher son adoption par la communauté informatique même s'il est conceptuellement innovateur. Selon Kalisky, il est possible de donner plusieurs significations au terme "compatible". Par exemple, un standard A peut être considéré compatible avec un standard B à partir du moment où A fournit un algorithme pouvant être utilisé par B.

Selon *Kalisky* un standard A sera compatible avec un standard B si la 1^{ier} fournit quelque chose d'utile au 2^{ième} . Il est à noter que, dans cette définition, l'information transmise peut avoir été transformée par un changement de présentation ou une omission d'information.

Le problème de compatibilité peut donc expliquer le changement de politique opéré dans la publication des *PKCS*. En effet, à partir du *PKCS #3* une ouverture du domaine de standardisation à des domaines plus larges que le seul algorithmes *RSA* est entamé. De plus, une compatibilité avec d'autres standards est recherchée. Par exemple, *PKCS #6* spécifie la forme que doit avoir un certificat étendu dans le contexte de l'*Open System Interconnection* (OSI) qui est une norme publié par l' *ISO* en collaboration avec l' *ITU-T*.

Les protocoles ayant retenu l'attention du laboratoire *RSA* sont les suivants :

1.1 Privacy-Enhanced Mail (PEM)

Développé à partir de 1985 par le *Privacy and Security Research Group* (PSRG) sous la supervision de l'*Internet Architecture Board* (IAB), ce projet consiste a fournir des moyens de sécurisation des échanges e-mails.[3] Le *PEM* fournit un ensemble de fonctions de sécurités à l'utilisateur, par exemple il fournit une confidentialité, une authentification de l'origine et une intégrité pour les échanges sans connections.

[3] Le PEM est défini dans les "request for comments" (RFC) de 1421 à 1424.

Il est possible d'implémenter *PEM* soit comme un filtre entre l'éditeur et l'agent du système e-mail, soit en l'intégrant à l'intérieur de l'agent. L'avantage de la deuxième solution étant que l'on peut fournir à l'utilisateur une nouvelle interface comprenant un ensemble de services de sécurité. Le graphique suivant montre un émetteur utilisant une implémentation du PEM, sous forme d'un filtre et un receveur ayant un service PEM intégré à son agent.

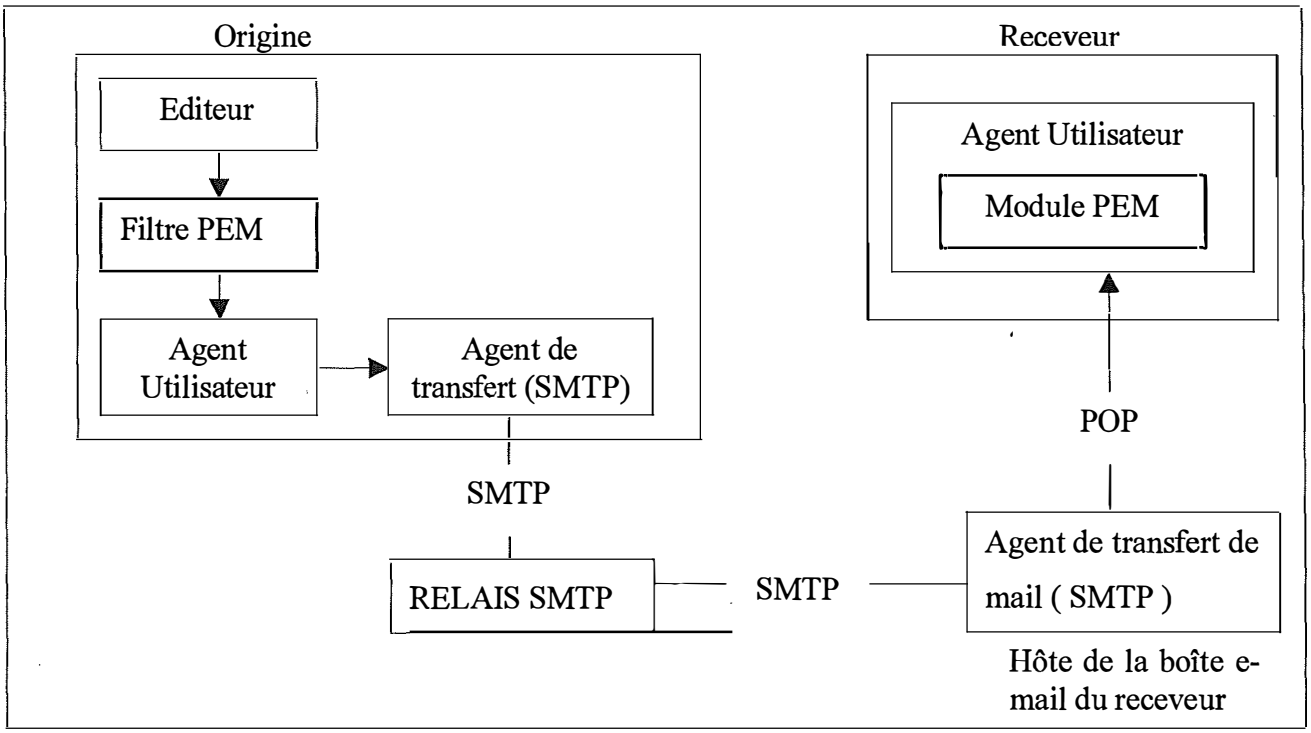

Figure 1.1 PEM environnement

PEM ne transforme pas le protocole de transport se trouvant en dessous de lui. Il conserve le format des messages avec l'en-tête définit dans le *Request For Comment 822 (RFC 822)* et n'effectue ses opérations de cryptage, signature... que sur le contenu même. Le traitement des messages se passe en trois phases :

- transformation en un format reconnu par le protocole *Simple Message Transfert Protocol* (SMTP), afin d'assurer une uniformité de la syntaxe pour un ensemble de systèmes informatiques hétérogènes ;
- calcul du code d'intégrité du message grâce à une fonction de *hashage* et chiffrement optionnel du message ;
- encodage sur 6-bit et limitation de la longueur des lignes pour la transmission via *SMTP* .

PEM utilise des certificats conformes aux recommandations du *International Telegraph and Telephone Consultative Comittee* (CCITT), rebaptisé ITU-T, sur les certificats plus connus sous le nom de *X.509*. Ces recommandations définissent un format pour les certificats et surtout introduisent le concept d'*Autorité de Certification* (CA) qui est une autorité reconnue pour la création et l'octroi de certificats.

Un certificat comporte un ensemble d'informations qui sont spécifiées par la norme *X.509* en utilisant la notation *Abstract Syntax Notation One* (ASN.1) :

```
Certificate ::= SIGNED SEQUENCE {
version [ 0 ]            version DEFAULT v1988,
serialNumber             CertificateSerialNumber,
signature                AlgorithmIdentifier,
issuer                   Name,
validity                 Validity
subject                  Name,
subjectPublicKeyInfo     SubjectPublicKeyInfo }
```

Figure 1.2 Format X.509 d'un certificat sous la forme ASN.1

Les différents champs utilisés sont :

- *version* utilisé pour différencier les versions successives des formats de certificats ;
- *serialNumber* identifie les certificats émis par les mêmes entités ;
- *signature* identifie l'algorithme utilisé pour signer le certificat ;
- *subject* contient le nom de la personne détenant le certificat ;
- *subjectPublicKeyInfo* contient la clé publique associé au certificat ;
- *issuer* permet d'identifier l'émetteur du certificat c'est-à-dire l'entité qui se porte garant pour le lien entre le nom contenu dans le champs *subject* et la clé publique associée qui est contenue dans le champ *subjectPublicKeyInfo*.
- *validity indique la durée de validité du certificat.*

Pour qu'une A*utorité de Certification* (CA) soit reconnue, il faut que l'entité ait été reconnue elle-même par un organisme appelé *Policy Certification Authority* (PCA) qui est autorisé à reconnaître les *CA*. A la racine de cet arbre, se trouve l'*Internet PCA Registration Authority* (IPRA), qui travaille sous la direction de l'*Internet Society*, qui se charge de reconnaître les *Policy Certification Authorities* (PCA).

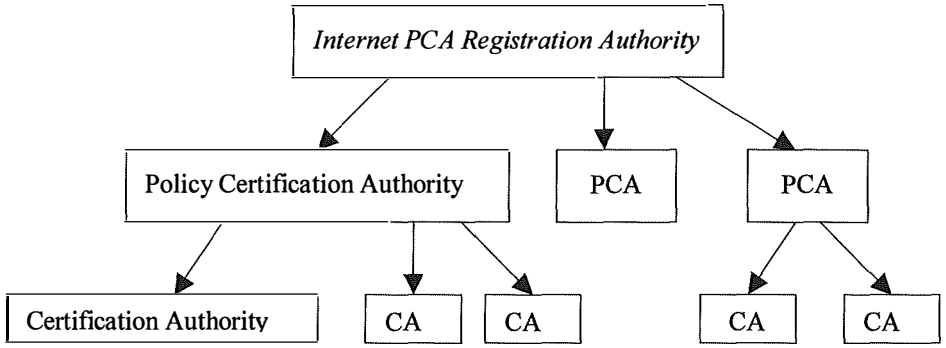

Figure 1.3 Hiérarchie des autorités de certification

Comme nous l'avons présenté, le protocole *PEM* s'intéresse essentiellement à la syntaxe des messages. Il s'attache à modifier les messages échangés de telles manière que ceux-ci soient sécurisés et soient toujours reconnus par les protocoles de transfert. C'est pourquoi, le PKCS qui attache le plus d'importance au protocole *PEM* est le 7^{ième}. Celui-ci, appelé *Cryptographic Message Syntax Standard,* traite de la syntaxe des messages sur lesquels peuvent être effectuées des opérations cryptographiques. Des messages générés selon le format *PEM* peuvent être utilisés par *PKCS #7* sans aucune modification ou opération cryptographique. Le message encapsulé dans le message *PEM* devient alors une donnée interne à *PKCS #7*.

1.2 Directory Services Authentification Framework

Ce standard, publié sous le nom de recommandation *X.509* du *International Telegraph and Telephone Consultative Comittee* (CCITT), définit le cadre général des certificats lors de l'interconnexion de systèmes ouverts[4]. Ce cadre spécifie les objets utilisés pour représenter les certificats et passe en revue l'ensemble des points critiques d'une structure du type *Public Key Infrastructure* (PKI). Il est possible de retrouver ce protocole dans le 6ième, 7ième et 10ième *PKCS*. En effet, *PKCS* #6, appelé *Extended Certificate Syntax Standard*, a comme point de départ cette norme et construit à partir de celle-ci des certificats étendus c'est à dire un certificat ordinaire agrémenté d'un ensemble d'attributs.

Comme nous l'avons vu *PKCS #7* spécifie la syntaxe des messages devant être utilisé pour effectuer des opérations cryptographiques. Enfin *PKCS #10*, appelé *Certification Request Syntax Standard* spécifie le format des messages devant être échangés lors d'une demande d'obtention de certificat. Il est donc, là aussi, tout à fait légitime de le retrouver comme référence.

Les certificats générés selon les spécifications de la norme *X.509* peuvent être convertis dans un format qui peut être utilisé dans les implémentations de *PKCS #6* et *PKCS #7* et inversement. De plus, les processus de signature sont les mêmes dans la norme *PKCS #6* et *X.509*. Il existe tout de même quelques points d'incompatibilité comme le chiffrement *RSA* qui n'est pas implémenté de la même façon dans les deux cas.

1.3 Message Handling Systems

Ce standard est défini dans la recommandation *X.400*[5] du *CCITT*. Il vient compléter l' *Open System Interconnection* (OSI) au niveau de la structure et de la manipulation des messages dans un tel système. Un message spécifié selon cette norme sera compatible avec *PKCS* dans le sens que *PKCS* pourra effectuer des opérations sur son contenu et qu'il pourra être manipulé sans difficulté par *PKCS*. Mais, par contre, *PKCS* ne pourra pas créer des messages répondant aux spécifications de cette norme.

---

[4] UIT-T, *Technologie de l'information - Interconnexion des systèmes ouverts – L'annuaire : Cadre général des certificats de clé publique et d'attribut* , Mars 2000, p. viii,

[5] UIT-T, *Non-Telephone Telecomunication Services - Telematic services - Message Handling Services – Message Handling system and services overview* , June 2000

1.4 Digital Signature Standard (DSS) et Secure Hash Standard (SHS)

Le *Digital Signature Standard* (DSS) est défini par le *National Institute for Standards and Technology* (NITS)[6] et est repris dans sa publication *Federal Information Processing Standard Publication* numéro 186-2 (FIPS-PUB 186-2). Le *Secure Hash Standard* (SHS)[7] est publié par le même organisme dans sa publication *FIPS-PUB 180-1*.

Le *Secure Hash Standard* spécifie un algorithme, connu sous le nom de *SHA-1*, qui permet de calculer des condensés de messages de manière sécurisée. Ce condensé est utilisé par des algorithmes de signature digitale.

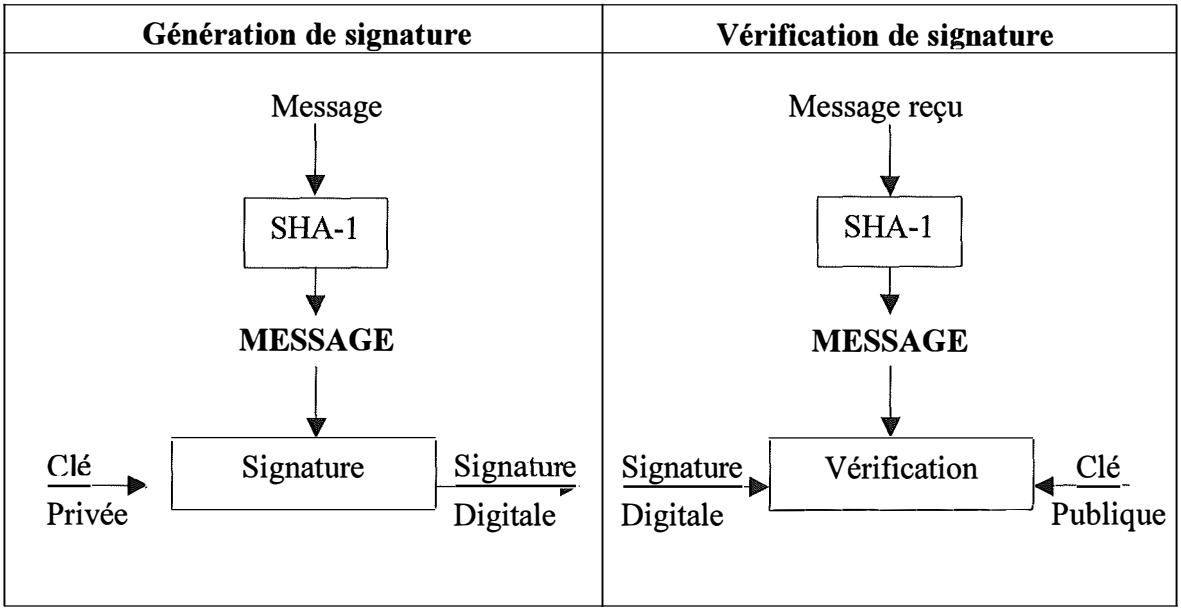

Figure 1.4 Opération de signature et de vérification de signature[8]

[6] R.G.KAMMER, "Federal Information Processing Standards Publication : Digital Signature Standard (FIPS-PUB 186-2) 27/01/2000 ", *National Institute of Standards and technology* , 72 pp .

[7] J.H.BURROW, "Federal Information Processing Standards Publication : Secure Hash Standard (FIPS-PUB 180-1) 17/04/1995 ", *National Institute of Standards and technology*, 24 pp.

[8] *Id.*, p.2

Ce standard a été développé suivant trois objectifs :

- Spécifier un algorithme de *hashage* sécurisé requis par le standard *DSS* sur les signatures digitales ;
- Spécifier un algorithme de *hashage* qui doit être utilisé lorsqu'un tel algorithme est requis dans une application de l'administration fédérale des Etats-Unis ;
- Encourager l'adoption et l'utilisation de tels algorithmes par les organisations commerciales privées .

Le standard sur les signatures digitales, *DSS*, recommande et analyse trois algorithmes qui peuvent être utilisés pour signer électroniquement des messages et pour vérifier ces signatures. Il s'agit du *Digital Signature Algorithm* (DSA), de l'algorithme *RSA* et de l'algorithme *Elliptic Curve Digital Signature Algorithm* (ECDSA).

La compatibilité avec *PKCS* n'est que partielle. Les certificats étendus, définis par *PKCS #6,* peuvent être signés avec le standard *DSS* mais pas ceux définis par *PKCS #7.* A l'inverse *PKCS* est compatible avec la norme *SHS* qui peut être utilisée dans *PKCS #7* comme un algorithme de *hashage.*

1.5 Digital Signature Scheme Giving Message Recovery

Le *Digital Signature Scheme Giving Message Recovery* est le premier standard relatif aux signatures digitales défini par la norme *ISO/IEC 9796* publié en 1991. Etant une norme au niveau des signatures numériques, il est normal de la retrouver dans les spécifications des PKCS #6 et PKCS #7. Ces standards, comme nous l'avons vu, traitent des signatures numériques et de la syntaxe des messages. Au niveau de la compatibilité, les données spécifiées par *PKCS #6* et *PKCS #7* peuvent être signées selon ce standard mais, vu la différence de format dans le chiffrement avec un algorithme *RSA*, il n'y a pas de compatibilité entre ce standard et *PKCS #1.*

1.6 ANSI X9.30 et X9.31

Les drafts *ANSI X9.30* et *X9.31* sont des standards définissant un protocole de chiffrement à clé publique avec des algorithmes réversibles et irréversibles. Les signatures définies dans le draft *X9.31-1* sont basées sur le standard *DSS* et celles définies dans le draft *X9.30-1* sont basées sur le standard *ISO/IEC 9796*. Or, nous avons vu auparavant que ces standards sont partiellement compatibles avec *PKCS*, cette compatibilité partielle reste toujours d'actualité.

Section 2 : Compatibilité "future"[9]

Les protocoles vus sont donc à la base du travail de standardisation effectués par *RSA*. *PKCS* essaie donc tant que possible d'intégrer les standards existants pour avoir une utilisation aussi large que possible. On peut remarquer qu'il existe parfois une compatibilité partielle ce qui est normal vu que certains de ces standards sont les premiers à avoir été publiés. Une évolution des technologies et des techniques imposent un changement dans la manière d'aborder la cryptographie. Nous avons vu qu'un autre but de *PKCS* était de rester suffisamment ouvert pour pouvoir être incorporé dans *l'Open Systems Interconnection* (OSI). Pour cela, *PKCS* va se baser sur deux standards propre à l'*OSI* qui sont l'*Abstract Notation One* (ASN.1) et le *Basic Encoding Rules* (BER).

2.1 Open System Inetrconnection (OSI)

Ce standard comporte un ensemble de normes basées sur le même modèle de départ. Celui-ci a été défini et implémenté en 1984 par deux organismes : *ISO* d'une part et l' *ITU-T* d'autre part. Ce programme de standardisation est né du besoin d'un standard international pour la communication réseau facilitant l'interopérabilité entre des équipements de vendeurs différents[10].

---

[9] Cette compatibilité était envisagée en 1991 pour le futur des *PKCS*. Actuellement les différents projets ayant aboutis, le terme futur n'est plus d'actualité.

[10] " Internetworking Technology Overview : Open System Protocols ", chapter 32, june 1999, 8 pp.

L'idée de base du standard *OSI* est que le processus de communication sur un réseau entre deux utilisateurs finaux peut être divisé en couches. Il en existe sept différentes et lors de l'envoi d'un message entre deux utilisateurs, le flux de données parcourt les sept couches de l'émetteur et du receveur une à une.

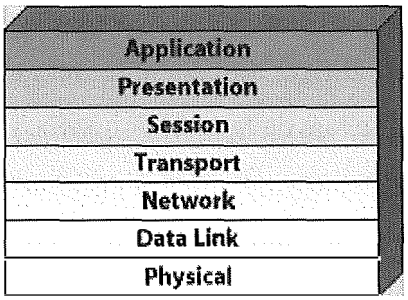

Figure 1.5 Les couches du modèle OSI

Les quatre couches supérieures (Application, Présentation, Session, Transport) ne traitent que les flux de données à destination d'un utilisateur final sur cet ordinateur. Un filtrage est effectué sur les trois couches inférieures (Network, Data Link, Physical) et si les données ne concerne pas un utilisateur présent sur cet ordinateur alors le flux est redirigé vers un autre hôte.

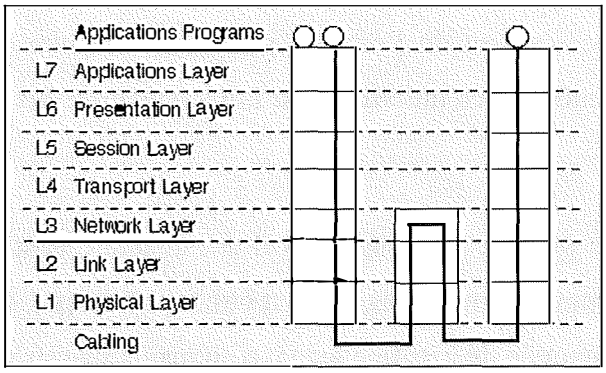

Figure 1.5 Communication entre deux utilisateurs utilisant un intermédiaire[11]

[11] "OSI reference model", http://www.erg.abdn.ac.uk/users/gorry/course/intro-pages/osi.html.

- *Physical Layer* : cette couche a pour fonction le transport du flux de bits au niveau physique. Elle maintient et active le réseau physique. (Ex : RJ45, NRZ,...)
- *Link Layer* : elle fournit un transport fiable des données en utilisant les adresses MAC et corrige si possible les erreurs de transmission. (Ex : Ethernet, FDDI, ...)
- *Network Layer* : cette couche définit l'adressage logique et identifie l'utilisateur final. Elle permet de diriger les données vers sa destination aussi bien lors de l'envoi de données que lors de la réception. (Ex : IP, IPX, ...)
- *Transport Layer* : cette couche fournit le contrôle end-to-end en s'assurant par exemple que tous les paquets sont bien arrivés, elle effectue également la fragmentation des données lors de l'envoi d'information et ré-assemble les données lors de la réception (Ex : TCP, UDP, ...)
- *Session Layer* : assure le départ, le contrôle et la fin des sessions de communication et leur synchronisation. (Ex : Telnet, FTP, ...)
- *Presentation Layer* : parfois appelée couche syntaxique, cette couche est généralement contenue dans le système d'exploitation . Elle a pour but de traduire le flux de données d'un format à l'autre. (Ex : HTML, ASCII, ...)
- *Application Layer* : cette couche est la plus proche de l'utilisateur et elle se différencie des autres par le fait qu'elle ne fournit aucun service à aucune autre couche. Elle s'assure de l'état du partenaire de communication et initialise les différentes procédures utilisées pour la récupération des erreurs et le contrôle des données.

PKCS décrit la syntaxe des messages d'une manière abstraite mais ne spécifie en aucun cas sous quel format les messages doivent être représentés. Or, la norme OSI fournit le standard *ASN.1* permettant de décrire une syntaxe de manière abstraite et le standard *BER* qui fournit les règles de base de représentations. A partir du moment où *PKCS* utilise ASN.1 pour décrire sa syntaxe, on peut supposer, bien qu'il n'y ait aucune obligation, que le choix d'une représentation similaire au format *BER* est recommandé.

2.2 Abstract Syntax Notation One (ASN.1)

Ce standard formalisme les types de données abstraits. Il utilise des types simples prédéfinis et permet la construction de nouveaux types. La raison principale du succès de cette notation est son utilisation dans d'autres standards tels que par exemple le *Basic Encoding Rules* (BER) ou le *Packed Encoding Rules* (PER). La standardisation *ASN1* provient de la recommandation *X.409*, publié en 1984, du *CCITT* qui traitait alors également du *BER*. Par la suite, *ISO* décida d'adopter cette notation mais partagea le document en deux parties *ASN.1* d'un côté et le *BER* de l'autre. En 1985, le *CCITT* décida de collaborer avec l'*ISO* sur ces deux documents.

Par rapport à d'autres syntaxes, *ASN.1* à l'avantage de fournir un support pour l'interconnexion entre nouveaux et anciens systèmes.

2.3 Basic Encoding Rules (BER)

Cette norme est définie dans la recommandation *X.690* du *CCITT*. La représentation *BER* fournit une ou plusieurs manières de représenter les valeurs *ASN.1* sous la forme d'un string de 8 bits. Il existe plusieurs manières de construire le string suivant que l'on connaît le type ou la longueur du type de la valeur à encoder. Il existe un sous ensemble de *BER* qui s'appelle le *Distinguished Encoding Rules* (DER) qui lui fournit une et une seule manière de représenter une valeur *ASN.1* sous forme de string.

Au début de cette section nous avions émis l'hypothèse qu'un recentrage de la politique de développement des standards PKCS avait été effectué. Ce recentrage avait pour but une standardisation plus large que la seule utilisation de l'algorithme RSA, en ne négligeant pas les standards déjà publiés et reconnus, ni ceux en cours de développement. Nous avons montré la justesse de cette hypothèse en analysant au cas par cas les différents standards utilisé par PKCS. De plus, cette politique est confirmée par le laboratoire *RSA* lui-même qui fournit 3 raisons officielles qui les ont guidés lors de la création de tels standards :

- Maintenir la compatibilité avec le protocole Internet *Privacy-Enhanced Mail* (PEM) de sécurisation des échanges e-mails. Celle-ci a pour but la possibilité d'échanger des certificats et de traduire d'un format à l'autre des messages chiffrés ou signés ;

- Etendre le protocole *PEM* pour qu'il puisse supporter un ensemble de fonctions supplémentaires : exemple, manipuler des données autres que celles formatées selon la norme *American Standard Code for Information Interchange* (ASCII), manipuler des certificats étendus.
- Spécifier un standard pouvant être incorporé dans l'*OSI.*

Mais que faut-il standardiser ? Pour répondre à cette question, il est important de noter les deux niveaux de travail possibles : le premier serait la syntaxe des messages tandis que le deuxième serait la spécificité des algorithmes et leur implémentation. Pour faciliter la standardisation et donc la compatibilité, il conviendrait de décrire une syntaxe de message indépendante de l'implémentation de l'algorithme. Cette première partie permettrait de faire abstraction du niveau inférieur et de ne standardiser que les échanges, l'utilisation, la construction... des messages proprement dit. Il conviendrait également d'implémenter un algorithme de manière à ce qu'il puisse travailler avec un ensemble de syntaxes de message différents. Ces deux manières de procéder, si elles sont appliquées ensembles, permettraient d'assurer une compatibilité parfaite.

Pour savoir ce qui doit être standardisé, Kalisky a utilisé quatre applications de la cryptographie à clé publique. Ces choix sont arbitraires et Kalisky assume l'existence d'autres fonctions nécessitant une standardisation. Ces quatre fonctions sont les suivantes : Signature numérique, Enveloppe numérique, Certification numérique et la processus d'Echange de clés. Dans le cas d'une signature numérique, nous pouvons constater qu'il est nécessaire d'obtenir trois choses :

- une syntaxe des messages indépendante des algorithmes ;
- des algorithmes spécifiques pour effectuer la génération et la vérification de signatures ;
- des algorithmes spécifiques servant à condenser un message.

En effectuant ce travail pour les différentes fonctions, on obtient un ensemble de procédures devant être standardisées et un autre ensemble ne devant pas l'être. C'est ce travail qui est effectué par la norme *PKCS.*

Pour résumer selon Kalisky, "Les normes *PKCS décrivent la syntaxe pour les messages d'une manière abstraite et donnent des détails complets à propos des algorithmes. Néanmoins, cette norme ne spécifie pas comment les messages doivent être représentés* ".

Section 3 : Présentation des différents standards PKCS

- **PKCS #1 : RSA Cryptography Standard**

  Ce premier standard traite du chiffrement à clé publique basé sur un algorithme *RSA*. Il fournit les recommandations pour tout utilisateur souhaitant implémenter un tel algorithme. Les points développés sont :

  - une présentation du fonctionnement général de l'algorithme avec des précisions sur la manière de l'implémenter correctement ;
  - la représentation des données entrantes et sortantes qui doivent respecter, par exemple, la notation ASN.1.

- **PKCS #2 : Incorporé dans PKCS #1**

- **PKCS #3 : Diffie-Hellman Key Agreement Standard**

  Ce troisième standard traite de la publication des clés secrètes entre des parties ne se connaissant pas. D'une manière générale ce standard s'occupe des protocoles d'établissement de communications sécurisées et de leurs compatibilités avec ceux proposés par l' OSI .

- **PKCS #4 Incorporé dans PKCS #1**

- **PKCS #5 : Password-Based Cryptography Standard**

  Ce cinquième standard a pour objet l'utilisation d'un mot de passe lors d'opérations de chiffrement. Pour effectuer ces opérations, certains algorithmes peuvent utiliser le mot de passe de l'utilisateur en le travaillant suivant une certaine technique pour fournir comme résultat une clé. Deux techniques sont possibles :

  - combiner le mot de passe avec du "sel" pour obtenir la clé ;
  - après un certains nombres d'itérations, obtenir la clé à partir d'une dérivation du mot de passe.

  La différence essentielle entre les deux techniques étant que, dans le premier cas, une déduction de la clé est possible. En effet, il est possible de construire un "dictionnaire d'attaque" sur un espace de recherche plus petit dans le deuxième cas si le nombre d'itérations effectués est suffisamment pertinent.

- **PKCS #6 : Extended-Certificate Syntax Standard**

  Un certificat ordinaire est basé sur les *recommandations X.509* qui traite des certificats et des clés publiques dans le contexte de OSI . Ce standard développe une syntaxe pour un certificat étendu. En effet, il pourrait être nécessaire pour un utilisateur de certifier des informations non essentielles, c'est à dire non reprises dans les certificats ordinaires, mais néanmoins intéressantes comme un adresse e-mail, un nom ...

- **PKCS #7 : Cryptographic Message Syntax Standard**

  Ce standard définit la syntaxe des données sur lesquelles peuvent être effectuées des opérations cryptographiques comme par exemple les signatures ou les enveloppes digitales. En particulier, il s'attache à fournir une syntaxe la plus générale possible pour ce type de données c'est à dire une syntaxe compatible avec le standard *PEM* .

- **PKCS #8 : Private-Key Information Syntax Standard**

  Ce standard précise la syntaxe qu'il serait préférable d'utiliser pour manipuler une clé privée. Généralement cette information est composée du type de l'algorithme, d'une clé privée et d'un ensemble d'attributs.

- **PKCS #9 : Selected Attribute Types**

  Ce standard va fournir un complément d'information quant aux attributs qu'il est possible d'utiliser avec *PKCS #6, PKCS #7, PKCS #8.*

- **PKCS #10 : Certification Request Syntax Standard**

  Pour obtenir un certificat, il faut effectuer une demande à une autorité de certification qui fournira un certificat de la forme *X.509.* Ce standard spécifie la syntaxe de la requête que l'utilisateur doit fournit à l'autorité de certification pour obtenir son certificat.

- **PKCS #11 : Cryptographic Token Interface Standard**

  Ce standard définit une "*Application Programming Interface*" (API) appelée *Cryptoki* qui effectue des opérations cryptographiques à partir d'un dispositif contenant les méthodes et les informations nécessaires. Cette interface est capable d'effectuer un ensemble d'opérations différentes à partir de dispositifs technologiquement hétérogènes.

- **PKCS #12 : Personal Information Exchange Syntax Standard**

  Ce standard spécifie la syntaxe qu'il convient d'utiliser pour effectuer des échanges d'informations personnelles. Ce standard doit pouvoir aussi bien être implémenté de manière logicielle que matérielle avec, par exemple, l'utilisation de smart-cards.

- **PKCS #13 : Elliptic Curve Cryptography Standard**

  Ce standard traite de tous les aspects de la cryptographie par courbe elliptique : la génération des clés, les paramètres....

- **PKCS #14 : Pseudorandom Number Generation Standard**

  Ce standard spécifie les différentes manières de générer des nombres pseudo-aléatoire.

- **PKCS #15 : Cryptographic Token Information Format Standard**

  Ce standard est la suite de *PKCS #11* et définit quel doit être le format du "jeton" de cryptage pour qu'il puisse être reconnu indépendamment de l'interface *Cryptoki* utilisée.

CHAPITRE 2 : PKCS #11 EN PARTICULIER

Nous allons maintenant analyser en détails un des standards composant la série *PKCS* car cette présentation sera importante pour la suite de ce travail. Il s'agit du 11ième qui définit une *Application Programming Interface* (API) pour les appareils supportant des informations ou des méthodes cryptographiques. Cette *API*, aussi appelée *Cryptographic Token Interface* ou *Cryptoki*, a pour but de fournir une vue logique des "appareils" (logiciel ou matériel) de cryptographie. Un "appareil" de cryptographie définit tout ce qui contient de l'information ou qui fournit des fonctions cryptographiques, ils sont appelés "jetons" de cryptographie.

Section 1 : Fonctionnement général de la librairie PKCS #11

Cette interface travaille avec des "jetons" en suivant deux buts majeurs[12], une indépendance au niveau matériel et une approche cohérente au niveau du partage des ressources. Cela permet de ne pas devoir modifier l'interface pour pouvoir supporter des appareils de constructeurs différents, d'être indépendant de l'environnement de travail et enfin d'être transparent, au niveau de l'implémentation, pour les applications.

Cette volonté forte se traduit par le fait qu'une application n'a pas besoin de posséder une interface propre vers l'appareil concerné. A la limite elle pourrait ne pas savoir sur quel appareil elle travaille. Comme la volonté poursuivie par cette interface est d'être complètement transparente, elle n'a pas été simplement construite en reprenant pour chaque appareil l'ensemble des commandes valides pour celui-ci. Cette manière de procéder aurait sans doute résolu le problème à court terme mais n'aurait été d'aucune utilité à long terme. *Cryptoki* va définir une vue abstraite de bas niveau qui ne prends pas en compte les détails de chaque appareil c'est à dire qui présente une vue logique identique pour chaque appareil.

---

[12] RSA Laboratories, *PKCS #11 v2.10 : Cryptographic Token Interface Standard*, décembre 1999, p. 12.

La vue générale du fonctionnement de *Cryptoki* est la suivante :

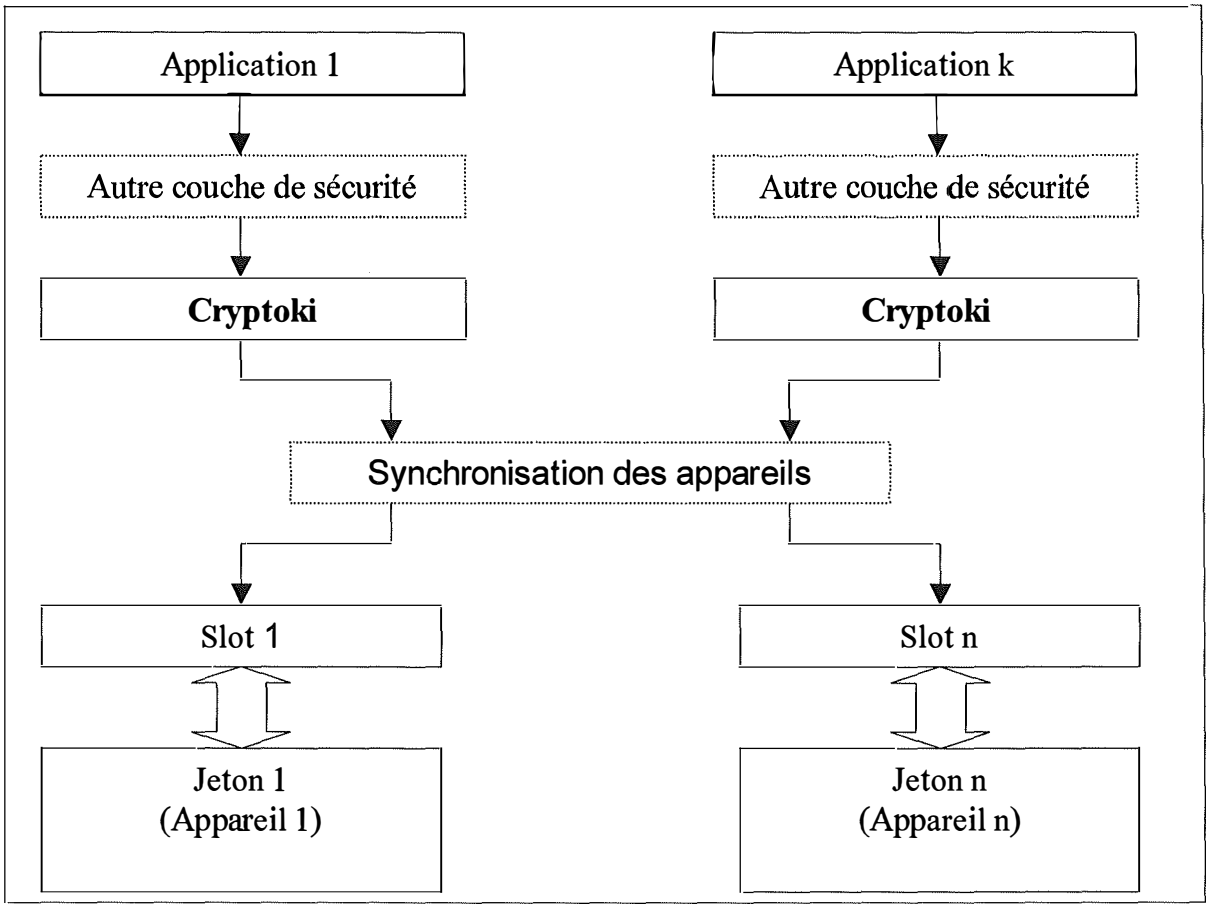

Figure 2.1 Modèle général[13]

Sur ce schéma nous pouvons voir apparaître clairement les deux buts poursuivit

- La transparence de l'appareil utilisé par rapport à l'application
- Le partage des ressources

Un nouveau concept important apparaît sur le schéma, celui de slot. Un slot est la porte par laquelle un "jeton" de cryptographie peut être actif dans le système. Un slot peut, par exemple, correspondre à un lecteur de *Smart Card.*

[13] *Id.*, p. 13.

Les deux états que peuvent prendre un slot sont libre ou occupé, ce qui correspond au fait que la carte est présente dans l'appareil ou non. Comme un slot n'est qu'une vue logique de quelque chose, nous pourrions imaginer toutes sortes de situations comme un ensemble de n slots qui dans une vue normale correspondraient à n lecteurs mais qui en fait se partagent le même lecteur.

Au niveau de l'implémentation même, deux manières de procéder peuvent être envisager :

- Sous forme d'un librairie directement liée à une application ;
- Sous forme d'une librairie partagée au moyen d'un lien dynamique.

Au niveau du fonctionnement de la librairie aucune différence n'apparaît mais on peut remarquer un avantage pour le lien dynamique car si de nouvelles librairies viennent à être installée, il sera plus aisé de les placer dans le système existant. Mais, si l'on peut facilement les installer, cela implique un risque plus important au niveau de la sécurité car il serait facile de remplacer cette librairie par une autre ayant un comportement dangereux. On pourrait envisager le remplacement de la librairie par une autre qui, par exemple, pourrait avoir comme but de divulguer le code des utilisateurs.

Il est à noter que la compatibilité de la librairie *Cryptoki* au niveau des algorithmes et des appareils de cryptage dépendra de chaque librairie et du niveau de son implémentation. *RSA* n'étant que le fournisseur de l'analyse du fonctionnement général et des en-têtes de chaque fonction, la liberté est laissée à chacun de supporter tel ou tel mécanisme, algorithme…

Section 2 : La structure des objets

On peut classifier les objets principaux utilisés par Cryptoki en trois catégories, les objets données qui sont définis et utilisés par une application, les objets contenant un certificat et les objets contenant une clé. Cette clé peut être de trois types : publique, privée et secrète.

Il est possible de représenter cette hiérarchie sous la forme suivante :

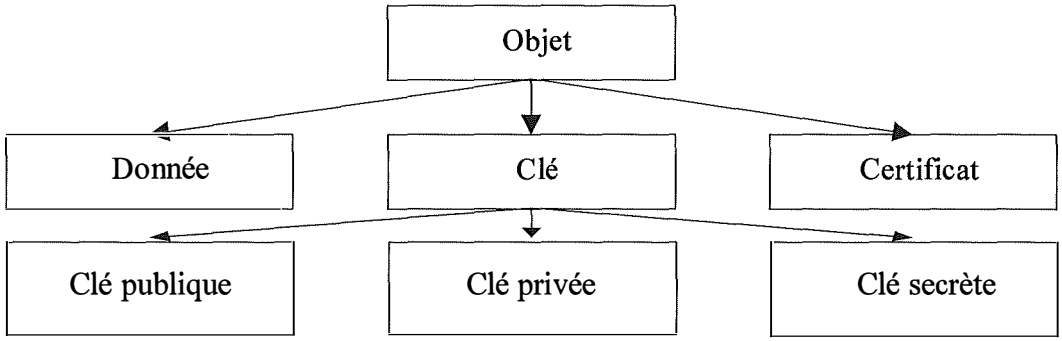

Figure 2.2 Hiérachie des objets[14]

En plus des trois catégories d'objets existants dans Cryptoki, il est encore possible de les classifier sous deux formes en prenant comme paramètres leur visibilité et leur durée de vie.

2.1 Les objets de jetons

Les objets de "jetons" ne sont visibles d'une application seulement si cette application est connectée à ce "jeton" et qu'elle possède les droits suffisants pour consulter ces objets. Ceux-ci restent sur le "jeton" même lorsque la session, entre l'application et le "jeton", est fermée et que le "jeton" est retiré du slot. En d'autres termes ce sont des objets ayant une longue durée de vie.

[14] *Id.*, p. 14.

2.2 Les objets de sessions

Pour qu'une application puisse utiliser les objets ou les fonctions sur un "jeton", celle-ci doit ouvrir une ou plusieurs sessions avec ce "jeton". La session est donc la vue logique d'une connexion entre un "jeton" et une application. Ce sont des objets ayant une durée de vie plus temporaire. Une fois qu'une session est fermée, tous les objets sessions crées par cette session sont automatiquement détruits. De plus, les objets sessions sont uniquement visibles par l'application qui les a créés.

Section 3 : Manipulation des objets

3.1 Les utilisateurs

Il existe deux types d'utilisateurs, le premier type est le *Secutity Officier* (SO) tandis que le deuxième est un utilisateur standard. Seul les utilisateurs normaux authentifiés peuvent avoir accès aux objets privés du "jeton". Le rôle du SO est d'initialiser les "jetons", d'initialiser les méthodes pour authentifier un utilisateur (de manière standard dans Cryptoki cela se passe grâce à un *Personal Identification Number* (PIN)) et parfois de manipuler des objets publics. Cryptoki ne gère pas la relation entre un SO et les utilisateurs standards, il se pourrait que le SO soit également un utilisateur simple.

3.2 Les sessions

Une session peut être ouverte, de deux manières différentes, en lecture seule tout d'abord ou en lecture et écriture. Il convient de noter que cette distinction concerne les objets de "jetons" et non les objets de session. Dans les deux cas, une application pourra créer, lire, écrire, détruire les objets de sessions tandis que seule une session ouverte en lecture lecture/écriture pourra effectuer toutes ces opérations sur des objets de "jetons". Autrement dit une session pourra manipuler comme elle l'entend les objets crées durant cette session mais elle devra être ouverte en lecture/écriture pour pouvoir manipuler des objets présents sur un "jeton".

De plus, il existe deux types d'objets, ceux publics qui peuvent être consultés par tout le monde et ceux privés qui ne peuvent être consultés que par des utilisateurs possédant les droits adéquats. C'est pourquoi une session, ouverte en lecture seulement, peut avoir deux états :

- Ouverte en lecture uniquement pour des objets publics. Lorsqu'une application ouvre une session avec un "jeton" sur lequel elle n'est pas été authentifiée, c'est à dire avec lequel elle n'a pas effectué de *login*, ou qu'aucune autre session authentifiée n'est ouverte sur le "jeton" par cette application, alors elle ne pourra consulter que des objets publics sans pouvoir les modifier.

- Ouverte en lecture seulement pour des objets privés. Lorsqu'une application ouvre une session authentifiée avec un "jeton", les objets contenus par ce "jeton" seront visibles mais non modifiables par cette application.

Lors de la fermeture d'une session, tous les objets de sessions crées durant celle-ci seront détruits même si d'autres sessions utilisent ces objets. *Cryptoki* supporte l'ouverture de plusieurs sessions sur plusieurs "jetons", en général un "jeton" peut avoir plusieurs sessions ouvertes avec, une ou plusieurs applications, mais remarquons qu'il est possible de limiter le nombre de sessions ouvertes sur un même "jeton".

L'illustration suivante représente les différents états lors de l'ouverture d'une session en lecture :

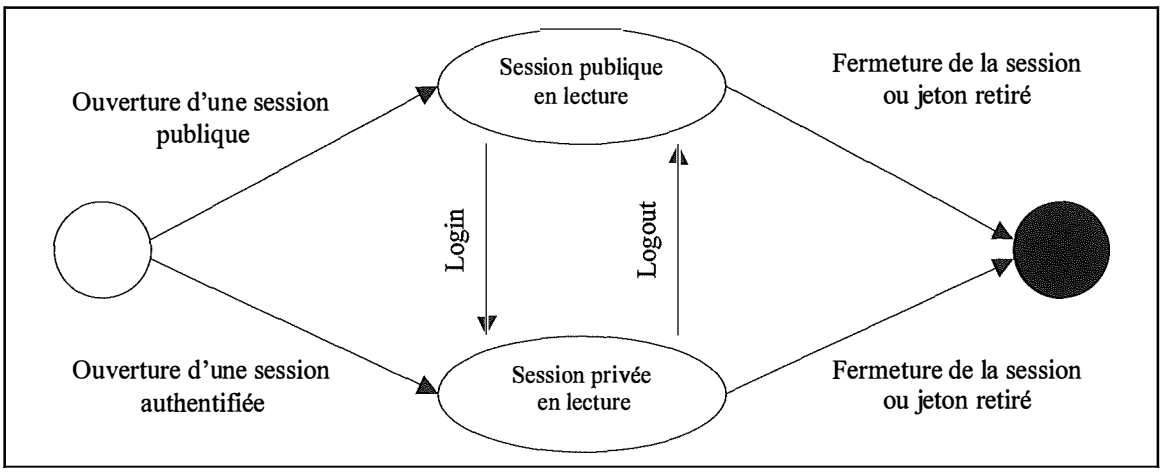

Figure 2.3 Etats d'une session ouverte en lecture seulement

De même, une session ouverte en lecture et écriture peut être :

- Ouverte en lecture et écriture pour des objets publics. Lorsqu'une application ouvre une session avec un "jeton" sur lequel elle n'est pas authentifiée (c'est à dire avec lequel elle n'a pas effectué de *login*) ou qu'aucune autre session authentifiée n'est ouverte sur le "jeton" par cette application, alors elle ne pourra consulter et modifier que des objets publics.

- Ouverte en lecture et écriture pour des objets privés. Lorsqu'une application authentifiée ouvre une session avec un "jeton" ou que la session est ouverte par le SO, les objets privés contenus sur le "jeton" seront visibles et modifiables par cette application.

Comme pour les sessions ouvertes en lecture seulement, il est possible de représenter les états que peuvent prendre les sessions ouvertes en lecture/écriture par l'illustration suivante :

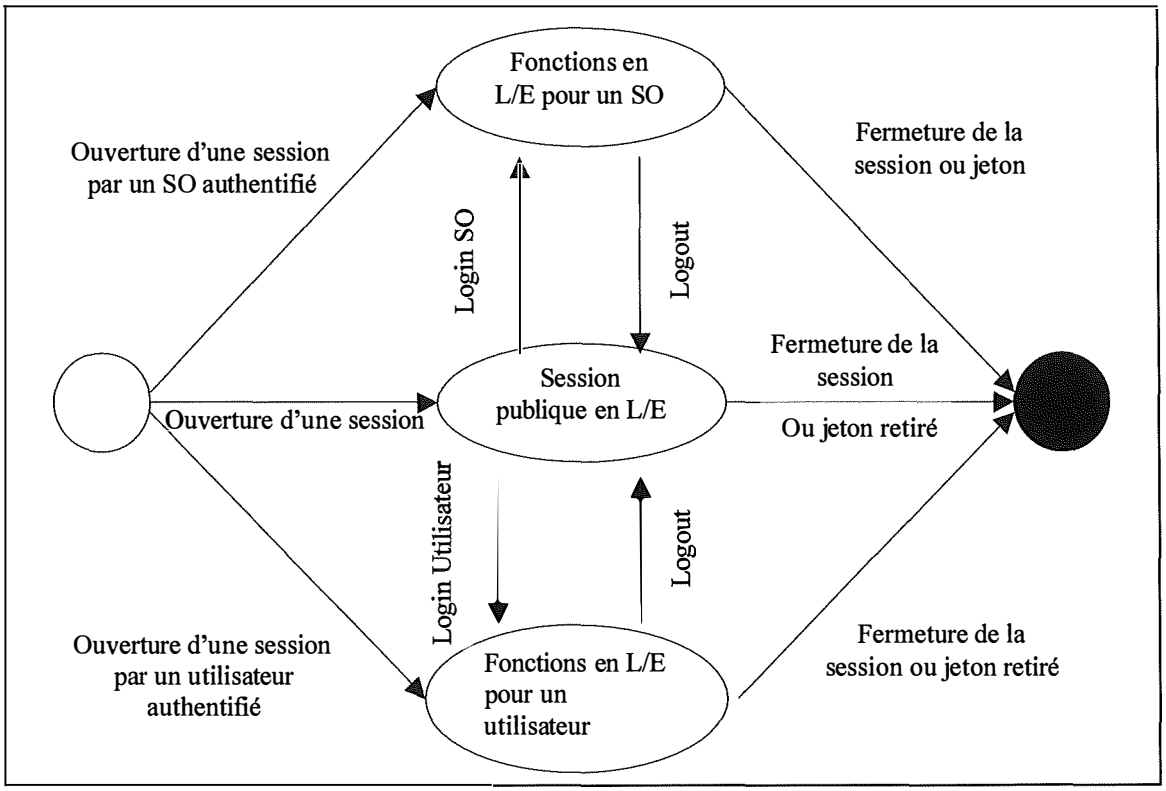

Figure 2.4 Etats d'une session ouverte en lecture et écriture

En conclusion, nous pouvons dire que les différents états que peut prendre une session apparaissent comme cohérents pour la protection des données privées contenues sur un "jeton". De plus, il est logique qu'il n'existe pas de session ouverte en lecture seulement pour un *Security Officier* (SO). Théoriquement, il n'aurait pas été faux de prévoir ce cas, mais cet état serait apparu inutile dans les faits.

Il est possible de résumer l'ensemble de ces graphes d'états par le tableau suivant :

Type d'objet	**Type de session**				
	Lecture seule publique	**Lecture/ Ecriture publique**	**Lecture seule utilisateur**	**Lecture/ Ecriture utilisateur**	**Lecture/ Ecriture SO**
Objet d'une session publique	Lecture/ Ecriture	Lecture/ Ecriture	Lecture/ Ecriture	Lecture/ Ecriture	Lecture/ Ecriture
Objet d'une session privée			Lecture/ Ecriture	Lecture/ Ecriture	
Objet d'un jeton public	Lecture seule	Lecture/ Ecriture	Lecture seule	Lecture Ecriture	Lecture/ Ecriture
Objet d'un jeton privé			Lecture seule	Lecture/ Ecriture	

Figure 2.5 Accès à différents types d'objets par différents types de sessions

3.3 Evénements d'une session

Il a été dit auparavant que lorsqu'un "jeton" était retiré, toutes les applications effectuaient automatiquement un *logout.* Toutes les sessions existantes entre cette application et le "jeton" étaient fermées avec une perte d'information possible. Il est clair que la librairie *Cryptoki* ne peut constamment vérifier la présence ou non du "jeton". Seule l'exécution d'une fonction sur ce "jeton" peut l'amener à se rendre compte du fait que le "jeton" n'est plus présent. Dans le même ordre d'idée, il se peut qu'un "jeton" ait été retiré puis réinséré sans que Cryptoki ne sache jamais qu'il était manquant. Dans cette version de Cryptoki, toutes les sessions ouvertes par une application doivent avoir le même statut, c'est-à-dire que toutes doivent être du type sessions publiques, sessions SO, etc...Dans la même idée, si une session existe en lecture seulement et que cette application ouvre une nouvelle session en écriture et lecture alors toutes les sessions deviennent des sessions ouvertes en écriture et lecture.

Section 4 : La structure fonctionnelle

Il est possible de diviser les fonctions proposées par cette librairie en 6 groupes distincts :

a) Le groupe des fonctions générales, qui met en place le module *Cryptoki* possède deux fonctions principales :

- **C_Initialize**
- **C_Finalize**

Ces deux fonctions sont utilisés pour démarrer et fermer le module *Cryptoki*. C'est au moyen de la fonction d'initialisation que l'on configure le fonctionnement de la librairie en *multi-threading*.

b) Le groupe des fonctions gérant les slots et les jetons est composé de six fonctions principales :

- **C_GetSlotList**
  Cette fonction renvoie la liste des slots disponibles dans le système.
- **C_GetSlotInfo**
  Cette fonction donne des informations sur un slot donné.
- **C_GetTokenInfo**
  Même comportement que la fonction précédente mais cette fois pour un "jeton".
- **C_GetMechanismList**
  Cette fonction renvoie la liste des mécanismes supportés par un "jeton".
- **C_GetMechanismInfo**
  Cette fonction donne des informations sur un mécanisme donné.
- **C_InitToken**
  Cette fonction initialise un "jeton".

c) Le groupe des fonctions gérant les sessions est composé quant à lui de cinq fonctions principales :

- **C_OpenSession**
  Cette fonction ouvre une session entre un "jeton" et une application.
- **C_CloseSession**
  Cette fonction ferme une session entre un "jeton" et une application.

- **C_GetSessionInfo**
  Cette fonction renvoie les informations sur un session particulière.
- **C_Login**
  Cette fonction est utilisée pour l'identification sur un "jeton"
- **C_Logout**
  Cette fonction est utilisée pour la fermeture d'une session authentifiée sur un "jeton".

d) Le groupe des fonctions gérant les objets est composé de cinq fonctions principales :

- **C_CreateObject**
  Cette fonction est utilisée pour créer un objet.
- **C_DestroyObject**
  Cette fonction est utilisée pour détruire un objet.
- **C_GetAttributeValue**
  Cette fonction renvoie les attributs liés à cette objet.
- **C_SetAttributeValue**
  Cette fonction modifie la valeur d'un attribut d'un objet.
- **C_FindObjectsInit, C_FindObjects, C_FindObjectsFinal**
  Ces trois fonctions sont utilisées pour effectuer une recherche d'objet.

e) Le groupe de protection des données est composé de l'ensemble des fonctions habituelles :

- **C_EncryptInit, C_Encrypt, C_EncryptUpdate, C_EncryptFinal**
  Ces fonctions permettent de chiffrer une donnée.
- **C_DecryptInit, C_Decrypt, C_DecryptUpdate, C_DecryptFinal**
  Ces fonctions permettent de déchiffrer une donnée.
- **C_SignInit, C_Sign, C_SignUpdate, C_SignFinal**
  Ces fonctions permettent de signer une donnée.
- **C_VerifyInit, C_Verify, C_VerifyUpdate, C_VerifyFinal**
  Ces fonctions permettent de vérifier une signature.

f) Le groupe des fonctions effectuant la gestion des clés est composé de trois fonctions principales :

- **C_GenerateKey**
  Cette fonction est utilisée pour générer une clé secrète.
- **C_GenerateKeyPair**
  Cette fonction est utilisée pour générer une paire de clé asymétrique.
- **C_DeriveKey**
  Cette fonction est utilisée pour dériver une clé à partir d'une clé de base.

Il existe un certain nombre d'autres fonctions auxiliaires que nous détaillerons pas ici et qui sont utilisées dans des cas bien particuliers. Par exemple, une fonction de génération de nombres pseudo-aléatoires ou certaines fonctions utilisées pour la gestion du comportement des fonctions.

L'illustration suivante présente la découpe des différents modules en suivant une approche fonctionnelle :

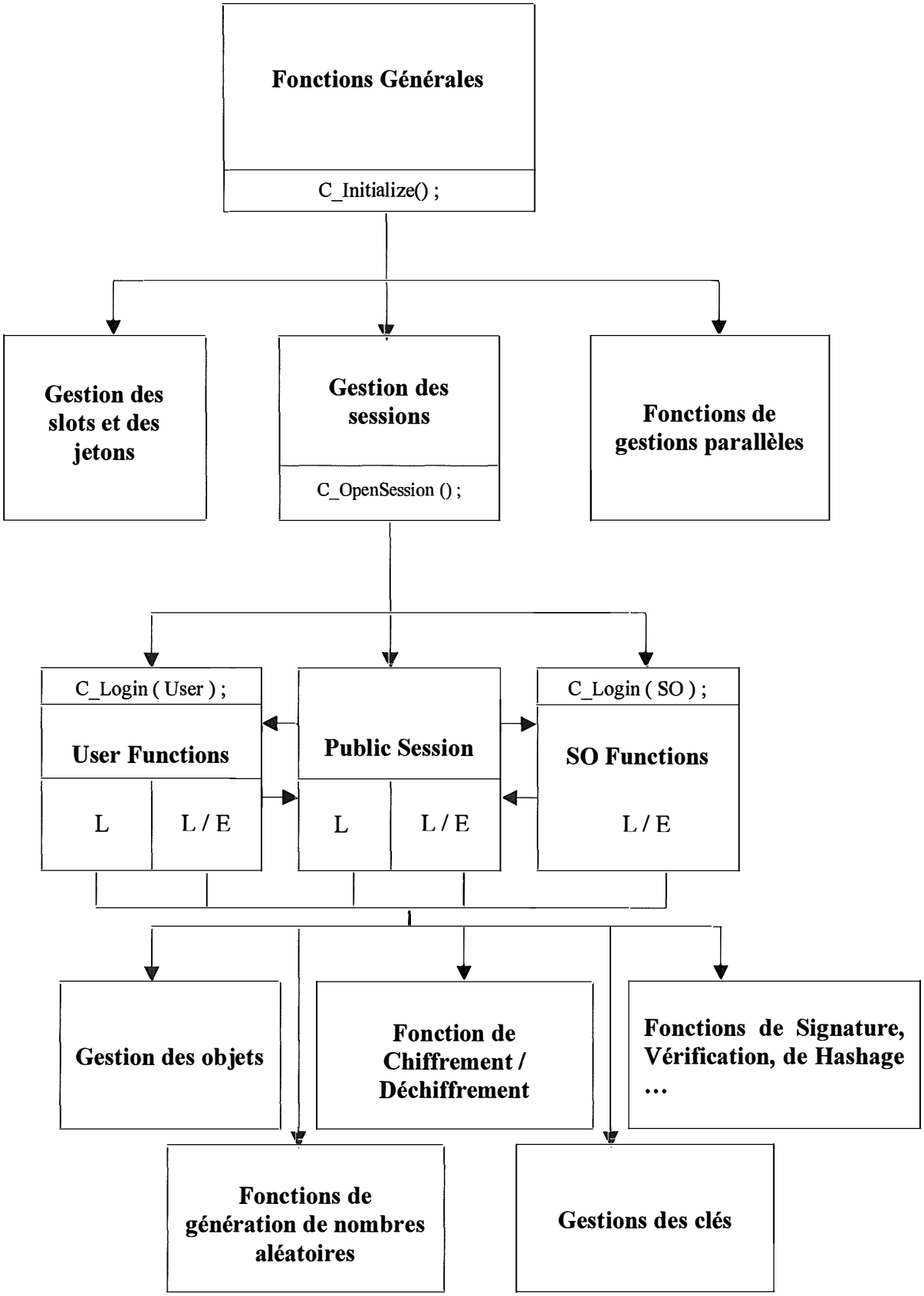

Figure 2.6 Découpe fonctionnelle de la librairie PKCS #11

Section 5 : Exemple d'utilisation de la librairie PKCS #11

Nous allons monter au travers d'un exemple, l'utilisation des différentes fonctions présentes dans *PKCS*. Imaginons que l'on veuille chiffrer un texte :

a) Tout d'abord, la librairie *PKCS #11* doit être initialisée. Cela s'effectue au moyen d'un appel à la fonction **C_Initialize** (pInitArgs) où *pInitArgs* sont les arguments pour initialiser des cas particuliers. Dans notre cas cette variable est toujours initialisée à un pointeur NULL.

b) Avant de pouvoir ouvrir une session, il faut initialiser le slot avec lequel nous allons travailler. Il faut effectuer l'appel à la fonction

C_InitToken (*slot_id ,pin ,strlen(pin), label)

slot_id : est la variable contenant l'identification du slot avec lequel nous allons travailler.

pin : est le code qui nous permet de nous identifier sur le "jeton"

strlen(pin) : définit la longueur du code pin

label : contient le label du "jeton", c'est une variable qui est utilisée pour des appels complexes. Dans notre cas, cette variable est toujours initialisée à NULL. Elle sert, dans le cas d'appels multiples, à définir quelle application a effectué l'appel et permettre aux fonctions de pouvoir faire du *callback*.

c) Ensuite, il convient d'ouvrir une session avec le "jeton" à partir d'un slot particulier. Un appel à **C_OpenSession** (*slot_id, flags, pApplication, notify, *sess) est effectué.

Slot_id : est la variable contenant l'identification du slot avec lequel nous allons travailler.

flags : cette variable permet de définir le type de session que l'on souhaite ouvrir (lecture seule, lecture / écriture, en tant qu'officier de sécurité). Dans notre cas cette variable est toujours positionnée comme une session ouverte en lecture / écriture.

pApplication : Cette variable ainsi que la suivante sont utilisées pour des appels complexes. Dans notre cas, celles-ci seront toujours initialisées à NULL. Elles servent, dans le cas d'appels multiples, à définir quelle application a effectué l'appel et pour permettre à ces fonctions de pouvoir faire un rappel vers la fonction appelante.

notify : voir *pApplication*

sess : est un pointeur vers l'identificateur de la session ainsi créé.

d) Une fois la session ouverte, on peut effectuer le chiffrement du texte proprement dit. Pour cela il convient d'effectuer un appel à la fonction **C_EncryptInit** (session, mech, key_handle) suivi d'un appel à **C_Encrypt** (session, clear_bloc, clear_data_len, encrypted_bloc, &encrypted_data_len). Où les variables représentent successivement :

session : est la variable contenant l'identificateur de la session.

mech : est la variable contenant le mécanisme utilisé pour le cryptage.

key_handle : est le pointeur vers la zone où est stocké la clé.

session : est la variable contenant l'identificateur de la session.

clear_bloc : est la variable contenant le texte à chiffrer.

clear_data_len : est la longueur du teste à chiffrer.

encrypted_bloc : est la variable qui contiendra le texte chiffré à la fin de l'exécution de la fonction .

encrypted_data_len : contiendra la longueur du texte chiffré.

Cet exemple termine la présentation de la librairie *PKCS #11*. Nous pouvons nous demander à ce stade-ci, s'il existerait une autre façon de concevoir une librairie de cryptographie. Dans un premier temps, pour nous permettre d'effectuer la comparaison, nous allons présenter une autre librairie de cryptographie appelée *Cryptographic Support Facility* (CSF) qui a été développé par l'entreprise *BULL*. Par la suite, cette comparaison servira de base pour présenter le travail qui a été effectué durant notre stage.

CHAPITRE 3 : CRYPTOGRAPHIC SUPPORT FACILITY (CSF)

La librairie Cryptographic Support Facility (CSF) est une librairie de cryptographie qui répond à la définition d'une librairie Cryptoki tel que définie auparavant. En effet, elle dirige l'ensemble des opérations de sécurité entre une application et un "appareil". La différence intervient dans le fait que cette librairie a été développée pour supporter l'ensemble des opérations cryptographiques d'une application de sécurité.

Section 1 : Origine du Cryptographic Support Facility (CSF)

L'origine du projet est à mettre au crédit de la Commission Européenne en concertation avec l'*European Computer Manufacturer's Association* (ECMA). Cette association avait, dans une série de recommandations, pointé un ensemble de domaines qui permettrait d'augmenter la sécurité dans un système ouvert. Cette sécurité concernant autant les applications entres elles que les relations entre utilisateurs et applications.

Le projet *Secure European System for Applications in a Multi-vendor Environment* (SESAME) n'est pas un projet commercial en tant que tel, mais a été développé par plusieurs partenaires européens qui sont *BULL SA*, *International Computers Ltd* (ICL), *Siemens Nixdorf Informations system* (SNI) et *Software System Engineering* (SSE). Le projet *SESAME*, au départ, devait reprendre les travaux préliminaires de l'*ECMA* et l'implémenter pour en prouver la validité. Depuis 1991, quatre versions furent publiées dont la dernière (V4) en décembre 1995.

Un point important dans l'implémentation de ce projet fut l'utilisation du *Generic Security Services Application Program Interface* (GSS-API) qui cache les mécanismes d'accès et d'authentifications. *SESAME* supporte un contrôle d'accès, une intégrité de communication et une confidentialité de communication et assure que l'accès aux services est contrôlé par un niveau de sécurité approprié[15].

---

[15] T.PARKER et D.PINKAS, "Sesame Technology Version 4 : Overview", Issue 1, december 1995, p. 7.

Après une authentification valide, généralement basée sur le principe de clé publique-privée à un *Serveur d'Authentification* (AS), l'initiateur reçoit un ticket. Est appelé initiateur un utilisateur ou une application agissant seule. Ce terme sera utilisé dans l'ensemble de la présentation de *SESAME* car ce système a pour vocation d'accroître la sécurité aussi bien au niveau des utilisateurs que des applications. En présentant ce ticket au *Privilege Attribute Server* (PAS), elle peut recevoir un *Privilege Atribute Certificate* (PAC) qui est la preuve de ses droits d'accès.

Pour effectuer la sécurisation de la communication entre l'initiateur et l'application cible, un échange d'informations sur les clés est nécessaire. Cette information peut être construite avec l'aide d'un *Key Distribution Service* (KDS) ou non mais comporte de toute façon deux parties. La première partie servira à protéger l'échange du *PAC* entre l'initiateur et l'application cible. L'autre partie sera utilisée pour assurer l'intégrité et la protection des données échangées entre l'initiateur et l'application cible.

Une fois l'initiateur en possession d'un *PAC* signé par le *PAS* et d'informations sur les clés[16], l'application cible peut effectuer un contrôle d'accès chaque fois qu'un appel à une ressource protégée est effectué.

Les concepts majeurs abordés par le projet *SESAME*[17] sont :

1.1 Sécurité sur un réseau non sécurisé

Par hypothèse, *SESAME* considère que le réseau entre l'application cible et l'initiateur est insécurisé. D'où l'utilisation de la cryptographie pour maintenir la confidentialité et l'intégrité des données. Il est clair qu'un chiffrement systématique de toutes les informations échangées serait disproportionné et entraînerait une surcharge de travail non acceptable. C'est pourquoi, seuls un certains nombres d'échanges sont chiffrés de manière automatique tandis que, dans les autres cas, une protection est effectué via un tamperproof sealing.

---

[16] *Id.*, p.8.

[17] *Id.*, p.12.

1.2 Authentification et login unique

Le premier moyen pour sécuriser un système informatique est évidemment de filtrer les accès à celui-ci. Le principe d'authentification de *SESAME* est basé sur deux techniques, soit une authentification par mot de passe soit via un système de clés publiques. Il existe deux aspects particuliers dans le cas de SESAME. Tout d'abord, à la différence d'autres systèmes, les points d'entrée ne sont pas nécessairement situés de manière contiguë aux points d'identification. Donc, si un initiateur s'authentifie à un point d'entrée éloigné d'un point d'identification alors le système doit effectuer une requête d'identification à travers le réseau. Ensuite, une politique de *Single Sign On* (SSO) ou login unique est d'application. Cette politique entraîne qu'un initiateur doit uniquement s'authentifier une et une seule fois à un point d'entrée. Le système s'assurant automatiquement que les actions effectuées par après par l' initiateur sont compatibles avec les droits de celui-ci. Ces deux particularités et l'hypothèse qu'il existe un réseau non-sécurisé entre les différents systèmes entraînent qu'il est nécessaire d'assurer un minimum de sécurité lors d'une requête d'identification par exemple.

1.3 Gestion des privilèges

Comme nous l'avons vu, la politique du login unique entraîne une particularité au niveau de la gestion des privilèges d'un initiateur. Ces privilèges contiennent les paramètres utilisés pour savoir ce qu'un initiateur a le droit d'effectuer comme opération, les endroits du système qui lui sont accessibles,... Ces informations importantes doivent être facilement accessible. Ce n'est évidemment pas le cas pour SESAME dont la vocation est la sécurisation de systèmes ouverts. C'est pourquoi une politique cohérente et efficace de gestion des privilèges doit être effectuée.

Deux techniques sont utilisées pour obtenir les privilèges d'un initiateur, la première, le "push", est activée par l'initiateur lui même. Celui-ci fournit ses privilèges à la cible après les avoir obtenues de la source sécurisée. La deuxième, appelée "pull", est initiée par la cible, qui après avoir reçu l'identité de l'initiateur, effectue une requête auprès du gestionnaire de privilèges.

SESAME pour plusieurs raisons ne supporte que la technique "push"[18] :

- au niveau de la performance, les décisions peuvent être prise directement par la cible sans coût de communication supplémentaire ;
- au niveau des coûts de communication, statistiquement les accès sur le système où sont stocké les attributs seront moins fréquents ;
- au niveau des privilèges, l'application cible ne devrait voir et obtenir que ceux nécessaires pour effectuer ses décisions de contrôle sans plus. C'est le principe du "need to know"

1.4 Serveur de sécurité en ligne versus hors ligne

L'accroissement de l'utilisation de certificats signés par des autorités de confiance hors-ligne permet aux applications cibles de vérifier directement les privilèges sans devoir recourir à un service en-ligne d'authentification ou de vérification de privilèges. Cependant, il existe plusieurs désavantages :

- Les informations de sécurité sont sauvegardées sous forme de certificats signés par une autorité de confiance et la clé publique de cette autorité. Généralement il va exister un nombre de copies de ces certificats ou clés sur un ensemble de systèmes éloignés, et il va devenir très difficile de gérer l'ensemble de cette information, par exemple, si on veut effectuer une révocation.
- Il n'y a pas de gestion centralisée des logins, il donc est impossible de savoir à un moment donné quel utilisateur est en ligne et quels sont ses privilèges d'accès.
- Tous les composants de l'architecture doivent être capables de supporter les opérations de cryptage à clé publique.

SESAME utilise des serveurs de sécurité en ligne car il faut privilégier une gestion des contrôles immédiate. Ceci est particulièrement vrai pour la gestion des initiateurs qui est la population la plus instable des utilisateurs et la principale source des problèmes de sécurité.

---

[18] *Id.*, p.14.

1.5 Attributs de privilèges hétérogènes

Nous avons déjà présenté les problèmes et les réponses apportés par SESAME au niveau de la gestion des privilèges mais nous n'avons pas encore solutionné le problème de l'hétérogénéité du réseau. En effet, dans un système ouvert, peuvent cohabiter des réseaux de types différents. Or un initiateur, lors d'une même session, peut souhaiter naviguer de l'un à l'autre. C'est pourquoi, il faut effectuer une gestion des privilèges adaptée à cette situation. Une solution serait de gérer les privilèges de manière autonome chaque réseau gérant ses propres privilèges. Cette solution, bien que techniquement correcte, est peu efficiente car de manière générale, bien que le format soit différent, les valeurs contenues dans ces différents types de privilèges sont similaires. Comme *SESAME* est le fruit d'une association, ce projet n'est pas attaché à un protocole de communication spécifique. L'*ECMA* a défini un standard de référence pour les attributs des privilèges qui ne soit pas spécifique à un système particulier. De plus, ce standard a été défini de telle manière que la traduction vers un système spécifique est aisée. C'est pourquoi, lors de l'obtention de privilèges par un système extérieur, les valeurs fournies peuvent être récupérées par le système cible sans devoir être transformées. Ce sont des valeurs standards et non des valeurs locales.

1.6 Utilisations d'identités

Indépendamment des attributs attachés à une identité, plusieurs utilisations peuvent en être faites :

- une utilisation pour revendiquer son identité : *login*.
- une utilisation pour prouver la responsabilité des actions : l'audit.
- une utilisation pour comptabiliser l'utilisation du système : le *charging*.
- une manière d'obtenir accès à des informations : l'accès.
- une manière d'identifier l'origine d'une information à une tierce personne : la non-répudiation.
- une manière de localiser et modifier ses attributs de privilège, ceci étant un cas particulier de l'accès : accès à ses attributs de privilège.

Toutes ses utilisations pourraient être regroupées en un attribut unique utilisé pour tous les cas mais il en résulterait un ensemble de politiques de sécurité difficiles voire impossible à implémenter. Mais quel est le nombre approprié de type d'attributs qu'il faudrait attribuer à une identité sur une architecture de sécurité distribuée ? Pour *SESAME*, potentiellement tous les types pourraient apparaître comme attribut mais seuls certains sont essentiels.

- **L'identité authentifiée**, l'identité contenue sur le ticket reçu lors de l'authentification, celle-ci sera utilisée pour l'accès aux attributs de privilège.
- **L'identité d'accès**, champ du *Privilege Atribute Certificate* (PAC) fourni par le *Privilege Attribute Server* (PAS) et qui est utilisé pour l'accès.
- **L'identité d'audit**, champ séparé dans le *PAC* donné par le *PAS* et qui utilisée pour l'audit.
- Les identités pour le *charging* et la non-répudiation ne sont pas supporté par *SESAME* mais pourraient être rajouté au *PAC*.
- Le **nom de login** est utilisé pour le *login* mais on le distingue des autres formes d'identité car il n'est pas sauvegardé dans le *PAC*

1.7 Rôles

SESAME supporte un contrôle d'accès qui est basé sur le concept du rôle organisationnel de l'utilisateur. Pour pouvoir accéder à un groupe d'applications cibles, un utilisateur devra faire partie d'un groupe particulier.

En pratique l'administrateur défini un ensemble de privilèges pour un rôle particulier. A l'ouverture d'une session, un utilisateur demande pour obtenir un rôle particulier ou un rôle par défaut et si l'administrateur l'accepte, l'utilisateur obtient l'ensemble des privilèges liés à ce rôle.

Les avantages de cette manière de procéder sont nombreux :

- l'utilisateur ne doit pas être conscient des privilèges qu'il possède ;
- le passage d'un rôle à un autre est grandement facilité ;
- la gestion des privilèges est fortement facilité dans le cas où un nombre important d'utilisateurs prennent le même rôle, l'administrateur ne doit définir qu'une seule fois l'ensemble des privilèges ;
- le contrôle d'accès peut se faire sur l'appartenance à un rôle et non plus sur des identités individuelles.

1.8 Chemins d'accès et délégation

Une fois les problèmes d'authentification et de privilèges résolus, il nous reste à analyser la distribution de services. Comme nous nous trouvons dans un système distribué, il est fréquent que les différents services soient partagés entre plusieurs systèmes. Or un initiateur, accédant à un service, ne peut savoir à l'avance de manière certaine, quel serveur va répondre à sa requête. De plus, si le serveur, pour une raison quelconque, ne peut répondre à sa requête, l'initiateur est en droit d'espérer que celle-ci sera dirigée vers un serveur susceptible d'y répondre. Cette re-direction s'appelle la délégation. Elle induit également un risque au niveau de la sécurité car entre la première requête et la délégation, un utilisateur peut ne plus avoir les privilèges suffisants pour exécuter la requête. Par exemple, si l'accès au réseau sur lequel est effectuée la délégation lui est interdit.

Cette délégation peut prendre deux formes selon les souhaits de la cible et de l'initiateur, la première vérifiera simplement les privilèges de l'initiateur sans garder de trace de la route effective de la requête. L'autre forme de délégation conserve de la trace complète de l'action effectuée.

Une restriction peut être effectuée par l'initiateur au niveau de l'aire d'action de son *Privilege Atribute Certificate* (PAC). Il peut, s'il le souhaite, restreindre son *PAC* à un domaine d'application et la délégation ne pourra se faire qu'à l'intérieur de cette zone. *SESAME* supporte la délégation simple, c'est à dire la délégation sans traçage.

1.9 Cryptographie et gestion des clés

Tous les points de sécurité que nous avons vu n'auraient pas de sens sans le cryptage de données étant donné que nous nous trouvons sur un système distribué. Cette cryptographie a pour but de préserver la confidentialité et l'intégrité des données durant la transmission et la sauvegarde de ces données.

L'intégrité empêche les données d'être modifiées d'une manière indétectable tandis que la confidentialité empêche qu'une personne non autorisée puisse consulter ces données. *SESAME* utilise trois types d'algorithme :

- les **algorithmes symétriques** où une même clé, la clé secrète, est utilisée pour chiffrer, déchiffrer, signer et vérifier la signature. Ce type d'algorithme est utilisé pour protéger les communications aussi bien au niveau de l'intégrité que de la confidentialité.
- les **algorithmes asymétriques** où deux clés sont utilisées, la première pour crypter ou signer et la deuxième pour décrypter ou vérifier la signature. Ces deux clés sont : la clé privée qui est seulement connue de son détenteur et la clé publique associée qui est connue de tout le monde. Dans *SESAME* les algorithmes asymétriques ont plusieurs cas d'utilisations :
    - pour authentifier l'origine des données et l'intégrité des *PACs* ;
    - pour la distribution des clés entre des domaines de sécurité différents ;
    - pour signer les certificats.
- Les **algorithmes à sens unique** avec lesquels il est possible de chiffrer les données mais avec lesquels il est impossible de reconstruire les données originales. Ceux-ci sont utilisés pour contrôler l'utilisation des *PACs*, pour protéger l'intégrité des données de l'utilisateur et pour calculer les clés utilisées pour protéger l'échange de données. *SESAME* a été conçu de telle façon que les algorithmes peuvent être facilement remplacés et les tailles de clés ajustées, pour permettre de respecter une législation particulière ou pour utiliser un algorithme plus performant.

Pour mettre en place ces différentes politiques de sécurité, il a été décider de créer une entité séparée qui s'occupe de tous les appels de cryptographie nécessaire au bon fonctionnement du système. Comme nous l'avons déjà dit, au départ la librairie *CSF* n'avait que pour seul but de supporter l'ensemble des appels émanant du système *SESAME*. Or dans un souci de standardisation et d'innovation, cette librairie a été développée de tel manière qu'elle puisse supporter un ensemble d'"appareils" différents. Selon les concepteurs du projet, la librairie *CSF* a été développée de telle manière qu'elle devienne le standard de référence dans la cryptographie par "jeton". Or, la librairie *PKCS #11* est devenu le standard de référence. Pour expliquer cette préférence, il convient de souligner la méfiance des partenaires Américains pour des produits de sécurité d'origine Européens.

Section 2 : Cryptographic Support Facility (CSF)

Comme nous l'avons vu précédemment, la librairie *CSF* contient l'ensemble des fonctions de cryptage nécessaires au fonctionnement de *SESAME*. Le but de cette librairie est de cacher tous les détails concernant les différents algorithmes supportés et leurs implémentations. Elle peut être utilisée sans que l'utilisateur sache si les opérations qu'il effectue ont une implémentation matérielle, logicielle ou les deux à la fois. *CSF* fournit l'ensemble des fonctions habituelles comme[19] : le cryptage réversible symétrique ou asymétrique, le cryptage irréversible, la génération et vérification de signature... Chaque fonction étant indépendante de :

- l'algorithme
- l'implémentation de cet algorithme (logicielle, matérielle,...)
- la manière dont est stockée la clé (logicielle, matérielle, ...)

Un autre objectif poursuivi est la non différentiation, au niveau opérationnel, de l'utilisation d'une cryptographie à clé publique ou clé secrète. Pour ce faire, il sera nécessaire d'établir ce qui est appelé, dans *CSF*, un contexte.

Le contexte contient l'ensemble des caractéristiques qui seront utilisées pour effectuer les opérations demandées et qui seront fournies par l'utilisateur.

---

[19] Bull, ICL, SSE et SNI, "Sesame Technology Version 4 : Cryptographic Application Developer guide", Issue 1, december 1995, p. 5.

2.1 La structure des données

a) Les clés

Les clés peuvent prendre deux formes différentes dans *CSF* : soit

- une chaîne de caractères qui représente la clé elle-même ; cela se produit lorsque la clé est disponible pour son détenteur ou en d'autres mots lorsqu'elle n'est pas sauvegardée sur un appareil physique ;
- une référence vers la clé ; cela se produit lorsque la clé est stockée sur un appareil physique.

b) Les données des clés

Une clé ne peut être utilisée dans *CSF* sans les données qui lui sont attachées. Il existe cinq sortes de données pouvant être rattachées à une clé :

- l'**usage**, qui est le seul attribut obligatoire, peut prendre plusieurs valeurs. La clé peut être une clé de chiffrement, de déchiffrement, d'intégrité (pour la génération et la vérification de signature) ou une clé permanente. Une clé, par défaut, est temporaire. Pour la sauvegarder, il faut explicitement le préciser dans cet attribut ;
- le **temps de validité** de la clé qui définit combien de temps la clé est valable dans le système ;
- le **vecteur initial** qui est utilisé avec cette clé ;
- le **numéro de version** de la clé ;
- la **longueur** de la clé ;

c) Les identificateurs d'algorithmes

Pour identifier un algorithme, *CSF* utilise un *Object Identifier* (OID) propre qui est importé du standard *X509*. Les *OID* de ce standard sont des numéros uniques identifiant chaque mécanisme de cryptographie. *CSF* va prendre ces *OID* et, à partir d'un tableau reprenant les algorithmes supportés, va attribué un identificateur propre à *CSF*.

d) La qualité de service (QOS)

Même s'il n'est plus d'application actuellement, il nous semble intéressant d'expliquer le concept de qualité de service. Ce *QOS* permettait à l'administrateur de restreindre l'utilisation d'algorithme à l'intérieur d'un domaine. Ce QOS était défini par

- un service : intégrité ou confidentialité ;
- une force de cryptage : faible, moyenne, forte ;
- une classe d'algorithme : symétrique ou asymétrique.

A chaque qualité de service était attachée une liste d'algorithmes permis. Ainsi, un utilisateur, une fois sa qualité de service définie, ne devait plus s'inquiéter du choix de l'algorithme. En fonction des paramètres de sa qualité de service, l'utilisateur s'il le souhaitait, pouvait laisser le système choisir automatiquement l'algorithme à utiliser. C'est à dire un algorithme appartenant à la liste des algorithmes permis, avec la qualité de service de cet utilisateur.

e) Les "handles"

Les "handles" sont des nombres aléatoires de 32 bits attachés à une entité et unique parmi l'ensemble de ces entités, il en existe deux sortes :

- le **"handle" de clé** qui est une référence opaque vers une clé et les données de celle-ci. Le terme référence opaque est utilisé pour désigner le fait qu'il n'est pas possible de retrouver la zone de stockage et le contenu de celle-ci facilement. Il aurait été inadéquat de stocker les données sensibles de la même manière que les données courantes. La durée de vie du "handle" dépend de la durée de vie de la clé qui est définie dans les données de la clé, comme vu précédemment.
- Le **"handle" de contexte** qui est une référence opaque vers un handle de clé, un algorithme ou une paire d'algorithmes et une qualité de service. Ce "handle" est valide tant que le processus qui l'a créé l'est également et qu'un appel vers une fonction de libération de "handle" n'est pas effectué.

2.2 La structure fonctionnelle

Il existe six groupes d'*Application Programming Interface* (API) distincts :

a) l'API d'initialisation, qui met en place le module *CSF*, possède deux fonctions :

- **csf_begin**
- **csf_end**

Ces deux fonctions sont utilisés pour démarrer et fermer le module *CSF* pour un algorithme donné, pour initialiser les données pour un algorithme logiciel et pour configurer les appareils physiques.

b) l'API de génération des clés est composé de trois fonctions :

- **csf_gen_sym_key**
  Cette fonction va générer, à partir d'un algorithme, d'une longueur de clé et d'attributs passés en paramètre, une clé symétrique ;
- **csf_gen_asym_key_pair**
  Cette fonction va générer, à partir d'un algorithme, d'une longueur de clé et d'attributs passés en paramètre, une paire de clés asymétrique ;
- **csf_deriv_key**
  A partir d'une autre clé ou d'une chaîne de caractères, cette fonction va en dériver une clé symétrique.

c) l'API de gestion des clés est composé de quatre fonctions :

- **csf_init_key**
  Cette fonction initialise une clé en retournant un *handle* opaque sur cette clé. Elle reçoit en paramètre une indication sur la manière dont la clé est sauvegardée, l'utilisation de celle-ci et enfin la clé elle-même ou la référence vers cette clé.
- **csf_release_key**
  Cette fonction libère un *handle* de clé.
- **csf_read_key_info**
  Cette fonction permet de retrouver une clé ou la référence d'une clé à partir de son *handle*.

- **csf_get_key_data**

  Cette fonction permet de retirer d'un *handle* de clé, les informations attachées à cette clé.

d) l'API de gestion de contextes cryptographiques comporte six fonctions :

- **csf_init_context**

  A partir d'une paire d'algorithmes, d'un *handle* de clé et d'une longueur de clé, cette fonction va initialiser un contexte et retourner un *handle* opaque vers ce contexte.

- **csf_create_owf_context**

  Cette fonction a le même comportement que la précédente sauf que le contexte créé n'est valable que pour un chiffrement irréversible.

- **csf_release_context**

  Cette fonction libère un *handle* de contexte.

- **csf_duplicate_context**

  Après un appel à cette fonction, un nouveau contexte, copie exacte d'un autre, et son *handle* existeront dans le système.

- **csf_retrieve_key_from_context**

  A partir d'un contexte, cette fonction renvoie le *handle* de clé attaché à celui-ci.

- **csf_query_context**

  Cette fonction est similaire à la précédente sauf que le résultat renvoyé est la paire d'algorithmes attachée à ce contexte.

e) l'API de protection de données est composée de cinq fonctions :

- **csf_encrypt**

  Cette fonction chiffre une donnée en recevant comme paramètre cette donnée et un contexte.

- **csf_decrypt**

  Cette fonction va générer une donnée en clair à partir de la donnée chiffrée et du contexte utilisé pour chiffrer cette donnée.

- **csf_generate_check_value**

  Cette fonction va signer une donnée à partir d'un contexte.

- **csf_verify_check_value**

  Cette fonction va vérifier la validité d'une signature à partir d'un contexte.

- **csf_owf**

  Cette fonction va chiffrer de manière irréversible une donnée reçue à partir d'un contexte.

f) l'API d'importation et d'exportation comporte quatre fonctions :

- **csf_extract_key**

  Cette fonction va empaqueter une clé ainsi que toutes les données relatives à celle-ci dans un format exportable.

- **csf_extract_context**

  Cette fonction a un comportement similaire à la précédente mais pour l'exportation d'un contexte cette fois.

- **csf_restore_key**

  Cette fonction va créer un *handle* de clé à partir d'un clé reçue par une machine distante sous une forme exportable.

- **csf_restore_context**

  Fonction similaire à la précédente mais dans le cas d'un contexte.

Il existe un certains nombres d'autres fonctions auxiliaires que nous ne détaillerons pas ici, utilisées dans des cas bien particuliers, comme par exemple, la génération de nombres pseudo-aléatoires ou de libération de mémoire. *CSF* sépare le contexte de cryptographie et la clé associée dans le but d'utiliser une même clé avec plus d'un contexte.

A partir de cette analyse, nous pouvons obtenir une découpe fonctionnelle qui aura la forme suivante :

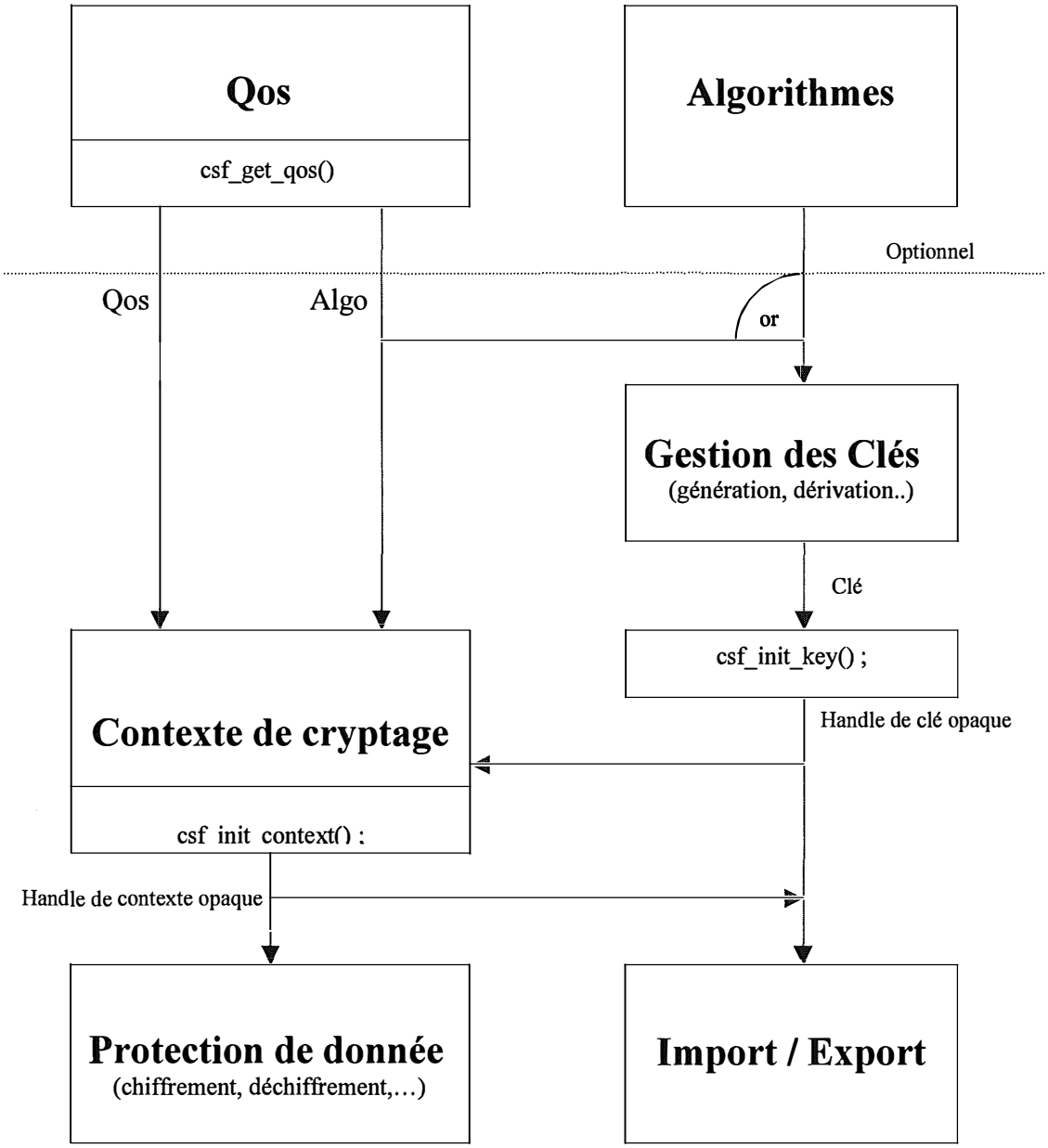

Figure 3.1 Découpe fonctionnelle de la librairie CSF

2.3 Exemple d'utilisation de la librairie CSF

Comme dans le cas de l'analyse de la librairie *PKCS#11*, nous allons montrer au travers d'un exemple, l'utilisation des différentes fonctions présentes dans *CSF*[20]. Imaginons que l'on veuille chiffrer une donnée, la succession des étapes est la suivante :

- **initialisation la librairie** : cela s'effectue au moyen d'un appel à la fonction **csf_begin** avec, comme paramètre, l'identifiant de l'algorithme que l'on va utiliser. Cette appel a pour but d'initialiser la librairie et l'algorithme.
- **obtention des clés de travail :** une paire de clés est générée mais il aurait été possible de la récupérer ou de la dériver (si tel était le cas, il faut initialiser ces clés au moyen de la fonction **csf_init_key()**). Pour ce faire, à partir du moment où l'on n'utilise plus de qualité de service, il convient tout d'abord de définir les différents attributs de cette paire de clé. Ces attributs sont contenus dans une structure **key_data** définie comme ceci :

```
typedef struct {
        OM_uint32       id_count;
        csf_data_t      *ids;
              } key_data_t;
typedef struct {
        data_type_t     type;
        data_value_t    value;
              } csf_data_t;
typedef enum {
        validity,
              iv,
        version,
      algorithm,
         usage,
        length
              } data_type_t;
typedef union {
        utc_time_t              validity;
        bit_string_t            iv;
        OM_uint32               version;
        algorithm_identifier_t  key_algo;
        OM_uint32               usage;
        OM_uint32               length;
              } data_value_t;
```

[20] Nous adopterons le fonctionnement actuel c'est à dire un fonctionnement de la librairie sans le module de gestion de la qualité de service

Une fois l'initialisation effectuée, il convient de générer la paire de clés au moyen de l'appel à la fonction :

csf_gen_asym_key_pair (algorithm, key_data, secret_key, pub_key, err_status).

Cet appel retourne à l'intérieur des deux variables **secret_key** et **pub_key**, les pointeurs vers les zones mémoires où ont été stockées les deux clés générées.

Il est possible, si l'utilisateur le souhaite, de les transformer dans un format exportable pour une autre application.

- **création du contexte : s**i l'on veut effectuer une autre opération, ce qui est notre cas dans cet exemple, il convient alors de créer un contexte dans lequel va s'effectuer l'opération. Ce contexte rassemble les pointeurs vers les clés, l'identifiant de l'algorithme et auparavant la qualité de service. Il convient de créer un contexte pour la clé secrète et un autre pour la clé publique. Ces contextes seront initialisés au moyen des deux appels suivants :

  csf_init_context (*secret_key, algorithm, qos, ctx_priv, err_status);

  et **csf_init_context** (*pub_key, algorithm, qos, ctx_pub, err_status).

  Ces appels fournissent en retour deux pointeurs vers les zones où sont stockés ces contextes.

- **chiffrement proprement dit :** celui-ci s'effectue au moyen de l'appel suivant : **csf_encrypt** (*ctx_pub, clear_text,encrypted_text,err_status). Remarquons que le texte clair est contenu dans une structure du type *ASN1* à deux champs : un champ qui contient le texte lui-même et un autre qui contient la longueur de ce texte. Il n'est pas nécessaire de manipuler le texte pour lui donner un format particulier.

Comme nous pouvons le remarquer, le processus de chiffrement d'une donnée s'effectue, au niveau fonctionnel, quelque peu de la même manière avec la librairie *CSF* qu'avec la librairie *PKCS #1*. Il faut effectuer trois phases, tout d'abord initialiser la librairie, ensuite créer le contexte (la session dans le cas de *PKCS #11*) avec lequelle l'utilisateur va travailler et enfin chiffrer la donnée. La seule difficulté rencontrée réside dans le fait qu'il n'y a pas d'implémentation d'une librairie *PKCS #11* standard par *RSA*. Les programmeurs souhaitant utiliser une telle librairie sont donc tributaires de la manière dont a été implémentée la librairie.

CHAPITRE 4 : CRÉATION DE L'INTERFACE CSF – PKCS #11

L'ensemble des analyses effectuées précédemment trouvent une application concrète que nous présenterons ci-après. Celle-ci a été effectuée durant notre stage qui déroulé au sein de l'entreprise *Evidian (G*roupe *Bull)*. Cette entité est spécialisée dans les programmes de gestion et de configuration de réseaux à distance. Elle propose principalement deux produits qui sont *OpenMaster* et *AccesMaster*.

- *OpenMaster* est un programme d'administration de réseaux qui administre et sécurise, à partir d'une seule et même plate-forme, un ensemble de réseaux hétérogènes ;
- *AccessMaster* est à l'origine le module de sécurité d'*OpenMaster*. Maintenant, il est développé comme une application à part entière. C'est à l'intérieur d'*AccesMaster* que se trouve le module qui nous intéresse plus particulièrment et que nous avons déjà présenté : la librairie *Cryptographic Support Facility* (CSF). Le stage s'est déroulé au sein du département *AccessMaster* et plus particulièrement dans le service *Single Sign On* (SSO) et *Public Key Interface* (PKI).

Section 1 : Travaux préliminaires

L'objectif du stage était d'effectuer des transformations au sein du programme *AccessMaster* de manière à utiliser la librairie de cryptographie *PKCS #11* et non plus la librairie *CSF*. Les motifs donnés par notre maître de stage sont les suivantes : au départ, la librairie *CSF* avait comme objectif de se positionner en tant que référence sur le marché de la cryptographie. Or après quelques années, force est de constater qu'un autre standard, en l'occurrence *PKCS#11*, s'est imposé plus largement en tant que standard de cryptographie par "jeton". Pour repositionner leur logiciel et satisfaire le souhait émis par quelques uns de leurs clients, il a été décidé de rendre compatible *CSF* avec ce standard.

1.1 Choix de la librairie PKCS #11

Le choix de la librairie *PKCS #11* apparaît logique. Etant le standard de référence dans son domaine, il aurait été inapproprié d'en choisir un autre pour effectuer la transition. Précisons tout de même que lors de notre arrivée, ce choix avait déjà été effectué et il ne nous a pas été demandé d'effectuer une analyse pour confirmer ou infirmer ce choix. Au niveau de l'implémentation de la librairie même, il nous a fallu trouver une version implémentée en libre distribution. Nous avons donc utilisé une implémentation existante développée par la fondation *GNU*[21] qui apparaissait comme étant suffisante pour le travail demandé.

1.2 Lecture préparatoire

Le choix de la deuxième librairie effectué, la première partie de notre travail a consisté à apprendre les bases de la sécurité. Nous avons effectué durant le premier mois une lecture approfondie des manuels de la librairie *PKCS #11* et *CSF.* Cette lecture fut complétée par le livre "*Handbook of applied Cryptography*"[22].

1.3 Analyse fonctionnelle

Après cette mise à niveau, nous avons analysé au niveau l'ensemble des fonctions et des types appartenant aux deux librairies. Cette analyse a permis de clarifier certains points et de servir de base à la comparaison. Nous avons constaté que les ensembles des fonctions étaient sensiblement identiques et qu'au niveau du typage, les deux ensembles se basaient sur la notation *ASN.1,* un avantage important si l'on veut passer de l'un à l'autre.

A partir de cette analyse, nous avons également commencé à imaginer le mode d'intégration de *PKCS #11* comme librairie de cryptographie d'*AccessMaster*.

Deux approches sont possibles :

- La première consiste à remplacer chaque appel à une fonction *CSF*, apparaissant dans *AccessMaster,* par un appel (ou une suite d'appels) à une fonction *PKCS #11* effectuant la même opération.

---

[21] Cette fondation a pour but la promotion du logiciel "libre" en favorisant l'échange de celui-ci.
[22] A. MENEZES, P. VAN OORSCHOT et S. VANSTONE, *Handbook of Applied Cryptography*, CRC Press, 1996, 754 pp.

Le problème de cette approche est le nombre de ces appels étant donné la taille du module *AccessMaster* et l'impact sur le typage qu'introduirait une telle solution. Il faudrait modifier en profondeur *AccessMaster* avec les risques que cela comporte.

- La deuxième approche consiste en un "interfaçage" entre *PKCS #11* et *AccessMaster* via le module *CSF*. Une modification en profondeur de la librairie existante est envisageable et a l'avantage de ne pas changer radicalement le logiciel. Pour effectuer un chiffrement au moyen de *CSF* ou de *PKCS #11*, il suffit de mettre en place la bonne librairie. Dans le 1^{ier} cas la librairie *CSF* seule, dans le 2^{ième} la nouvelle librairie *CSF* accompagné de *PKCS #11*.

Au niveau de l'implémentation, une modification interviendrait au niveau du corps de chaque fonction *CSF* sans toucher à leurs en-têtes. Cette modification serait, de manière générale, le remplacement du corps de la fonction par un ou plusieurs appels aux fonctions *PKCS #11* similaires. L'utilisateur aurait alors l'impression de toujours effectuer un appel au module *CSF* alors qu'en réalité le travail cryptographique aurait été effectué par des fonctions de la librairie *PKCS #11*.

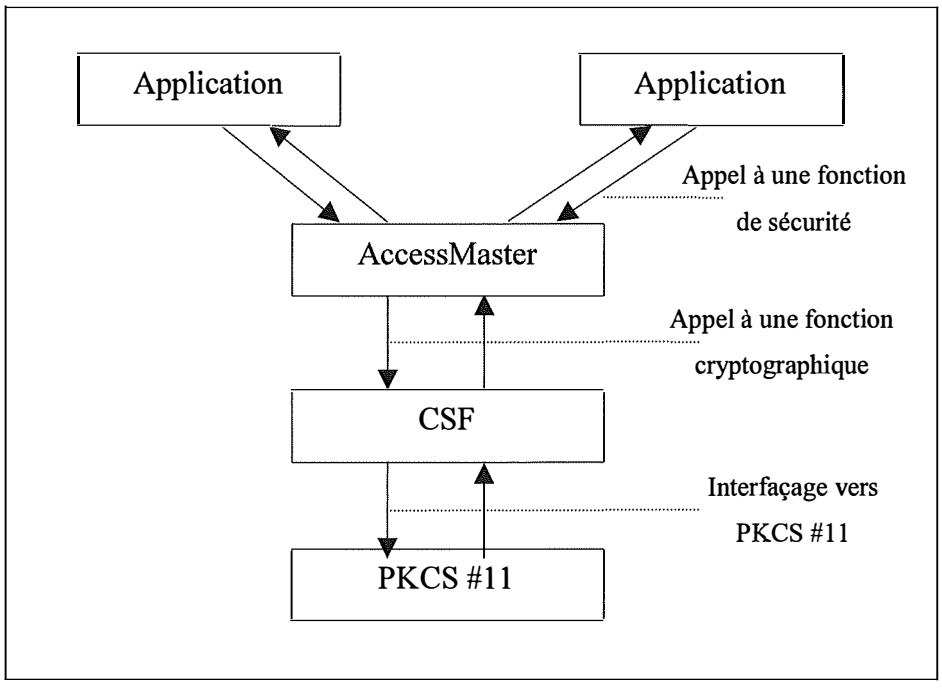

Figure 4.1 CSF devient une interface entre AccessMaster et PKCS #11

Section 2 : Implémentation

Cette phase, la plus longue, a duré 4 mois. Au moyen d'un appel de fonction qui servira d'exemple,nous allons exposé quel a été notre démarche.

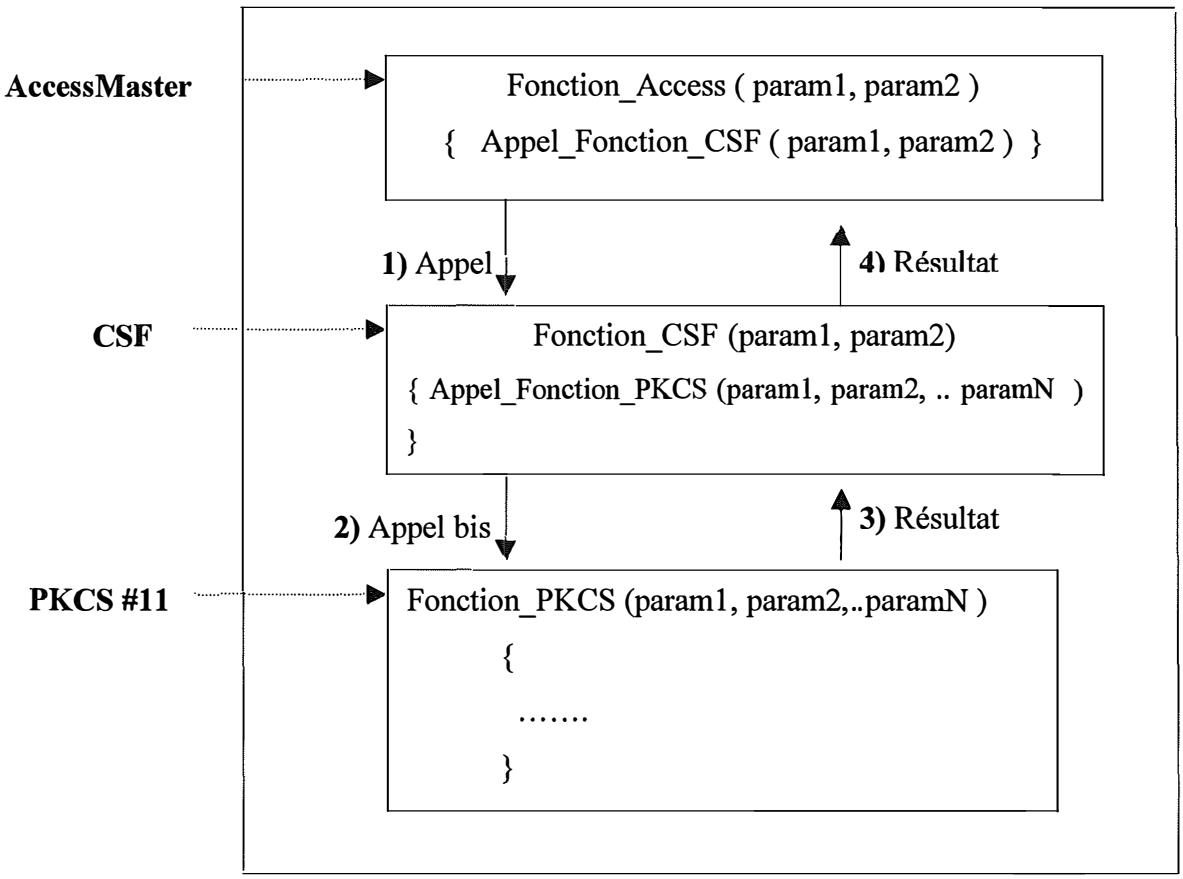

Figure 4.2 Interfaçage d'une fonction CSF par une fonction PKCS #11

Lors d'un appel à une fonction *CSF* **(1)** émanant d'*AccessMaster*, celle-ci ne calcule plus le résultat attendu via son implémentation propre. Elle récupère le ou les paramètres et effectue un appel à la librairie *PKCS #11* avec ces paramètres et peut être d'autres si besoin est **(2)**. A la fin de l'exécution de la fonction *PKCS #11*, les résultats obtenus sont transmis à la fonction *CSF* appelante **(3)** qui se charge de les renvoyer au module *AccessMaster* appelant **(4)**.

Plusieurs problèmes se posent alors :

- D'une part, il convient de trouver une fonction similaire ou une suite d'appels de fonctions similaires dans *PKCS #11,* dont le résultat final sera équivalent à l'exécution de la fonction *CSF*. Comme nous l'avons déjà évoqué, il est apparu des similitudes évidentes et à de rares exceptions il a toujours été possible de trouver une fonction équivalente dans *PKCS #11* à une fonction *CSF* donnée.

- D'autre part le typage des objets pose problème car il est utopique de s'imaginer que l'on puisse récupérer un paramètre lors du premier appel **(1)** et le fournir dans le même format à la librairie *PKCS #11* **(2)**. Une transformation des objets dans un type compatible avec *PKCS #11* est nécessaire. Le problème est équivalent lors du retour de l'appel où une transformation des objets vers un format compatible avec *CSF* est nécessaire.Si la construction fonctionnelle de *CSF* et *PKCS #11* comporte des similitudes, il n'en est pas de même pour les types utilisés. Cette difficulté est simplifiée, comme nous l'avons déjà vu également, par le fait que les deux librairies se basent, à leur niveau le plus bas, sur la notation *ASN.1*. Donc, seule une modification de la structure des objets est nécessaire. Nous avons procédé au moyen d'une structure de sauvegarde pour traduire plus facilement une variable d'un type vers un autre. Cette structure contient un ensemble d'informations présentés dans les deux formats. Par exemple, après avoir ouvert une session, la librairie *CSF* reçoit un identificateur de session qui peut être assimilé à un identificateur de contexte pour *PKCS #11*. La sauvegarde est donc effectuée dans la structure sous deux formats différents, bien que fondamentalement les valeurs soient identiques. Ceci permet de ne pas effectuer de traduction de variable chaque fois mais cela grossit la structure inutilement. Autour de cette identificateur, on effectue la sauvegarde de toutes les informations liées à cette session (clés, mécanismes, ...). A partir de l'identificateur de session ou de contexte, il est possible de retrouver l'information voulue directement suivant le format adéquat.

- Un autre problème, induit par la légère différence de structure entre les deux librairies, est l'absence d'une information requise à un moment donné dans le processus. Il arrive qu' une fonction *PKCS #11* ait besoin d'une information qui n'est pas encore disponible dans *CSF*.

  Cette situation entraîne une postposition de l'exécution de la fonction, jusqu'au moment où l'information voulue est obtenue. C'est le cas, par exemple, lors de l'initialisation d'une clé ou une paire de clé. Lorsque l'on veut utiliser une clé existante, il est nécessaire, dans *PKCS #11*, de générer un objet clé et de le remplir avec les informations en notre possession comme la valeur de cette clé par exemple. Le problème est qu'une initialisation de clé dans *CSF* s'effectue bien avant la création du contexte. Comme seul l'attribut de clé USAGE[23] est obligatoire, on ne connaît pas automatiquement le mécanisme qui sera utilisé avec cette clé. Or cette information est demandée par *PKCS #11* pour générer l'objet qui contiendra la clé. La seule solution est de générer un handle de clé aléatoire, de le sauver dans la structure en indiquant qu'il est incomplet, et lors de la création du contexte, de vérifier, et si nécessaire, terminer l'initialisation de la clé.

- L'utilisation de l'implémentation de la librairie *PKCS #11* posa également problème. Normalement, vu les spécifications de cette librairie, les seuls problèmes que nous aurions dû rencontrer étaient ceux déjà évoqués dans ce travail. Mais pour diverses raisons, la librairie n'était pas implémentée complètement : tous les mécanismes de chiffrement n'étant pas implémentés avec cette version (n'oublions pas que cette version était une version gratuite disponible sur Internet). Par exemple, dans le chiffrement avec le mécanisme *RC2_CBC*, il convient tout d'abord de vérifier si ce mécanisme est supporté par la librairie *PKCS #11*. Nous pouvons constater[24] qu'en effet il est supporté au niveau des spécifications de *RSA* et qu'il est possible de le définir, dans *PKCS #11*, de deux manières différentes. Soit comme un *CKM_RC2_CBC* ou un *CKM_RC2_CBC_PAD*.

---

[23] La structure du type clé est montrée à la page 55

[24] RSA Laboratories, *PKCS #11 v2.10 : Cryptographic Token Interface Standard*, december 1999 p. 212

La seule différence réside dans le fait que dans le deuxième cas, le mécanisme est implémenté avec une méthode de *padding*[25] intégrée. Malheureusement pour nous, la librairie récupérée ne supportait que le premier type de mécanisme, c'est à dire celui sans padding. Cette faiblesse d'implémentation a impliqué que nous devions prévoir nous même le mécanisme de padding pour effectuer le chiffrement au moyen du mécanisme *RC2*.

[25] Rappelons que le padding est une méthode qui consiste à ajouter une série de caractères quelconques à une donnée devant être chiffrée pour qu'elle puisse atteindre la taille requise par le mécanisme de chiffrement.

CHAPITRE 5 : SMART CARD

Quand une librairie *PKCS #11* sert d'interface entre des applications et tous les "appareils" de cryptographie existants, ceux-ci peuvent revêtir différentes formes. Une première distinction peut être faite sur leur matérialité ou leur immatérialité. Ils peuvent être implémentés de manière purement logicielle. C'est avec ce type de "jeton", pour reprendre la terminologie de *PKCS #11* , que nous avons effectué notre stage. Il se présente sous forme purement immatérielle et toutes les opérations cryptographiques sont effectuées par des fonctions. Il est possible de modifier quelque peu le comportement de cette forme de "jeton" pour en améliorer son fonctionnement avec *PKCS #11* ce qui est un avantage ; par contre, la manière dont ils ont été développés est un désavantage. Dans notre cas, nous avons eu des difficultés à mettre en place le "jeton" avec par exemple, des problèmes de portabilité, la configuration des options (en allant activer des paramètres spécifiques dans le code du programme),...

L'analyse de la forme hardware que peut prendre le "jeton" de cryptographie est l'objet de ce chapitre. Cette technologie assez ancienne est à l'heure actuelle en train de bouleverser les concepts de sécurité et son utilisation se généralise dans des domaines très variés. L'ensemble de ces applications sont regroupées sous le nom de *Smart Card.* Une *Smart Card* peut aussi bien servir de carte de téléphone, de carte de banque, de clé électronique pour passer une porte,... Il serait normal de penser que l'ensemble de ces utilisations ne sont pas basées sur la même technologie. Et pourtant, c'est le cas. Il est clair que la configuration matérielle sera différente en fonction de l'utilisation que l'on veut faire de la carte. Mais il existera une configuration minimale qui est identique pour toutes ces applications différentes. Ce sont ces différentes configurations que nous allons étudier dans cette partie du travail.

Section 1 : Historique des Smart Cards

Le concept de *Smart Card* fut déposé par le Dr. Kunitaka Arimura dans les années 70 au Japon. Notons, néanmoins, que le concept théorique d'associer une carte de plastique avec une puce électronique reviendrait à deux chercheurs allemands J. Dethloff et H. Grötrupp[26] et daterait de 1968.

Le brevet Européen et Américain quant à lui, fut déposé par Roland Moreno sous le nom de *Smart Card* en 1974. A ce moment de l'histoire, les *Smart Card*s étaient un produit encore en développement et il a fallu attendre les années 80 pour que trois constructeurs *Bull CP8, SGS Thomson* et *Schlumberger* fabriquent la première version commerciale des *Smart Card*s. Et ce fut seulement au cours des années 90 que l'usage de ce type de cartes s'est généralisé dans des domaines différents.

Section 2 : Classification des Smart Cards

Avant de s'attacher à classifier les différentes cartes existantes, il convient de rappeler les éléments communs présents sur de telles cartes. Une carte intelligente se présente sous un format équivalent à une carte bancaire, avec généralement une puce électronique qui fournit un ensemble de fonctions à l'utilisateur. Pour pouvoir utiliser ces cartes, il faut posséder une unité de lecture /écriture reliée à l'application qui utilise ces cartes. Cette unité de lecture / écriture est appelée *Card Acceptance Device* (CAD).

Suivant le type de *Smart Card* avec lesquelles on travaille, il est possible de retrouver les composants suivants : une capacité de calcul (optionnel), une capacité de sauvegarde de données, un moyen pour échanger de l'informations et une source d'énergie[27].

a) *Le microprocesseur :*

Le microprocesseur est la partie "intelligente" de la carte, celui-ci peut manipuler et transformer les données. De plus, l'ensemble du processus est géré par celui-ci.

---

[26] K. M. SHELFER et J. D. PROCACCINO, "Smart Card Evolution", *Communication of the ACM,* vol. 45, No. 7, July 2002, p.83.

[27] "Smart Card Overview", http://www.javacard.org/others/smart_card.htm.

b) La mémoire :

Il existe deux types de mémoires pouvant se trouver sur une carte intelligente. Celle-ci peut être non-volatile ou volatile. Dans ce dernier cas, la carte doit être accompagnée d'une batterie pour pouvoir alimenter cette mémoire. Ces deux formes de mémoire peuvent être mise en place par trois types de mémoire "physique" différentes : la mémoire *Random Access Memory* (RAM), la mémoire *Read Only Memory* (ROM) et la *Programmable Read Only Memory* (PROM). La mémoire *ROM* est non volatile et son contenu est défini lors de la fabrication. La mémoire *RAM* est, quant à elle, volatile et est utilisée comme espace de stockage temporaire. Au niveau de la mémoire *PROM*, il en existe deux types : la première est appelée *Electrically Programmable Read-Only Memory* (EPROM) tandis que la seconde est appelée *Electrically Erasable Programmable Read-Only Memory* (EEPROM). La mémoire *PROM* est programmable une seule fois tandis que les mémoires *EPROM* et *EEPROM* peuvent reprogrammées plusieurs fois. La différence entre les deux dernières étant que dans le premier cas, l'initialisation de la mémoire se fait via une fenêtre, placée sur la puce, qui doit être exposée sous un rayonnement ultra-violet. Dans le cas de l'*EEPROM*, un courant électrique effectue quant à lui cette initialisation. Notons que ce type de mémoire est réinscriptible un millier de fois.

c) L'échange de données :

Il est possible de classifier les cartes intelligentes en trois catégories, suivant le mode d'échange de données :

- celles ayant un contact qui peut être lu par un lecteur de cartes
- celles permettant d'échanger des donnés sans requérir de contact physique (le placement de la carte sur ou à côté de l'unité de lecture/écriture suffit)
- celles possédant un clavier et écran qui ne doivent pas échanger de données[28].

[28] Il est bien entendu possible d'intégrer un circuit de transmission (avec contact ou non) sur de telles cartes.

d) La source d'énergie :

Comme pour les autres points, il existe ici aussi plusieurs manières de travailler.

- La façon la plus classique est de prévoir deux contacts sur la puce qui fourniront l'énergie nécessaire à la carte pendant ses opérations. Cette solution ne peut évidemment s'envisager que dans le cas d'une carte à contact.
- La deuxième solution est de transmettre en même temps que le flux d'information, un flux d'énergie. Cette solution est utilisée avec les cartes sans contact mais qui doivent être placées près de l'unité de lecture/écriture.
- Enfin, la dernière solution consiste à intégrer une batterie dans la carte.

Cette solution est peu rencontrée car la taille et la matière de la batterie va à l'encontre de la volonté des constructeurs d'avoir des cartes de plus en plus petites et de plus en plus flexibles.

A partir de ces caractéristiques, il est possible de classer les *Smart Card*s en plusieurs catégories

2.1 Smart Cards avec contact

Ces cartes sont composées d'une puce placée à gauche qui sert de lien entre le lecteur de carte et la carte. La localisation de la puce est régi par un standard officiel sur les cartes à puces. Le schéma suivant présente la structure du contact. On peut retrouver les différents éléments déjà présentés tel que l'alimentation électrique, l'horloge utilisée pour la synchronisation et les broches servant aux opérations d'Entrées / Sorties.

Voltage	Terre
Reset	non connecté
Horloge	Entrée / Sortie

Figure 5.1 Différents éléments se trouvant sur un contact[29]

[29] R. FOURNIER et KAMIONEK, "The Cryptographic Smart Card : An Integrated Security Solution", 25/04/2001, http://www.rsasecurity.com/even.pdf.

2.2 Smart Cards sans contact

Ce type de carte remplace les échanges de données et d'énergie au travers d'un contact par un système un peu plus sophistiqué. La carte doit être placée très près de l'unité de transfert pour pouvoir recevoir l'énergie nécessaire à son fonctionnement ainsi que les données.

Il existe deux systèmes de transfert.

- Le premier transfert en même temps les données et l'électricité. A l'envoi des données, l'unité module l'électricité en fonction des données qu'elle doit envoyer.

  A la réception, la carte, dans le même temps démodule, l'électricité pour récupérer les données envoyées et utilise l'énergie reçue pour fonctionner.
- Le deuxième système quant à lui se passe en deux phases, tout d'abord l'envoi d'énergie active le transmetteur placé sur la carte qui peut alors recevoir les données.

2.3 CombiCards

Ces cartes sont l'addition des deux types de cartes vues précédemment. Elles permettent d'être utilisées aussi bien avec un lecteur à contact qu'avec un lecteur à induction.

2.4 Super Smart Cards

Les trois types de cartes présentées précédemment sont considérées comme passives car elles ont toutes besoin d'un lecteur pour la transmission de données et d'énergie. Cette nouvelle génération de carte est couplée avec un écran et un clavier ce qui permet de travailler directement sur la carte sans devoir passer par une unité de transfert. Il n'existe pas encore de standard pour des cartes de ce type car elles sont toujours en développement et on peut supposer que leur commercialisation sera très limitée vu leur coût élevé.

Ces classifications sont effectuées seulement au niveau de la technologie utilisée à l'intérieur de ces cartes. Il est également possible d'effectuer une classification suivant une approche plus fonctionnelle. Dans ce cas, il est possible de diviser l'ensemble des *Smart Cards* en deux groupes[30] :

2.5 Smart Cards Intelligentes

Les *Smart Cards* dites intelligentes possèdent une unité de mémoire ainsi qu'un *Central Processor Unit* (CPU). Une fois la carte mise sous tension et liée au lecteur, elle devient un véritable ordinateur. Elle fonctionne sous la direction d'un *Operating System* (OS) qui est appelé *Smart Card Operating System* (SCOS) ou *Reader Operating System* (ROS).

Le protocole de communication utilisé par ce type de carte est un protocole asynchrone, c'est pour quoi on les appelle aussi cartes asynchrones.

Lorsque la carte est activée par un lecteur, c'est à dire qu'elle reçoit l'énergie suffisante pour fonctionner, celle-ci envoie un paquet de données. Ce message, appelé *Answer To Reset* (ATR), contient les informations nécessaires au lecteur pour reconnaître la carte. A partir de ce message, le lecteur reconnaît le type de carte inséré et peut commencer la communication avec celle-ci.

2.6 Smart Cards de mémoire

Les *Smart Cards* de mémoire quant à elles ne possèdent ni processeur ni *OS*. Elles se contentent de fournir un espace de stockage. Elles sont contrôlées de l'extérieur par le l'unité de lecture / écriture. Le protocole utilisé avec ce type de carte est un protocole synchrone car la carte ne peut renvoyer de message *ATR*.

Un problème de compatibilité se pose ici car l'unité de lecture / écriture ne peut savoir à l'avance avec quel type de carte il est en communication. Ce problème est aggravé par le fait qu'il existe un nombre élevé de constructeurs de cartes différents ayant chacun leurs standards. Pour résoudre ce problème, l'unité de lecture / écriture va effectuer plusieurs requêtes sous un format différent jusqu'à ce qu'il reçoive une réponse. A ce moment, il aura identifié d'où provient la carte et comment dont il doit communiquer avec celle-ci.

---

[30] "Smart Card Overview", *op.cit.*

CONCLUSIONS

Pour ce mémoire, nous avons analysé la standardisation d'une librairie de cryptographie au format *PKCS #11*. Pour ce faire, nous avons passé en revue différents points de la cryptographie par "jeton". Dans un premier chapitre, nous avons exposé les standards *PKCS*. Lors du deuxième chapitre, nous avons analysé un standard plus particulier : le onzième. Ensuite, dans un troisième chapitre, nous avons présenté une librairie similaire développée par l'entreprise *BULL*. Grâce à un stage effectué dans cette entreprise, nous avons également pu mettre en pratique l'"interfaçage" entre leur librairie et *PKCS #11*. Le quatrième chapitre traite de cette mise en pratique. Enfin, dans un cinquième et dernier chapitre, nous avons classifié les différents types de *Smart Cards* qu'il est possible d'utiliser avec les librairies.

En ce qui concerne les standards *PKCS*, nous pouvons conclure que leur force est basée sur deux points :

1) le fait que ce standard s'est imposé *de facto* dans le monde informatique. Cette qualité nous permet d'affirmer que les spécifications contenues dans cette norme sont pertinentes et cohérentes ;
2) le fait que cette norme ait été développée de manière à s'intégrer au mieux avec les normes déjà présentes sur le marché. Lors de la présentation de celles-ci nous avons pu constater que *PKCS* respectait au mieux la compatibilité avec les normes existantes à l'époque et qu'un effort de compatibilité avait été effectué avec celles en cours de développement. Cette volonté d'ouverture à permis aux différents *PKCS* de ne pas être technologiquement dépassé après leur publication.

Lors du deuxième chapitre, nous avons présenté le concept majeur introduit par ce standard : celui de "jeton" de cryptographie. Cette vue logique permet de regrouper, sous le même terme, l'ensemble des outils permettant d'effectuer des fonctions cryptographiques.

Cette vue a l'avantage, lors de la spécification d'une librairie de cryptographie, de ne pas s'intéresser à la manière dont est construit le "jeton" de chiffrement. Quelque soit la forme de celui-ci, aucune différence n'est remarquée au niveau du fonctionnement de la librairie de cryptographie. Cette approche permet de développer une même librairie pour un ensemble de "jetons" hétérogènes, ce qui réduit considérablement la complexité de l'implémentation de la librairie.

En comparaison avec la librairie de cryptographie *CSF* développée par *BULL,* nous n'avons pas constaté de différence majeure au niveau de l'agencement des fonctions. Les fonctions proposées de part et d'autres sont similaires et seul un concept de qualité de service est introduit dans le cas de *CSF.* Ce qui prouve que la librairie développée par *BULL* avait été développée avec beaucoup d'ingéniosité vu sa ressemblance avec le standard *PKCS #11* qui est la référence actuelle.

Nous avons par la suite présenté la manière suivant laquelle nous avons travailler pour effectuer l'"interfaçage" entre les deux librairies. Cette tâche fut simplifiée par le fait que les deux librairies se basaient au niveau du typage par la notation *ASN.1* et par le fait qu'il n'y avait pas de différence fondamentale au niveau fonctionnel.

Pour terminer la présentation du fonctionnement d'une cryptographie par "jeton", nous avons classifié la forme la plus courante des "jetons" : les smart card. Nous avons classifié ces cartes en suivant deux approches, une première suivant les composants pouvant se trouver sur ce type de cartes, une seconde suivant les fonctionnalités pouvant être fournies par de telles cartes. A ce sujet, nous pouvons conclure qu'après une période de développement d'une dizaine d'années, le nombre d'applications utilisant cette technologie ne cesse de croître. Pour ne citer qu'un exemple, prenons l'introduction future en Belgique des nouvelles cartes d'identités digitales. Cette carte, suivant les spécifications données par le Ministère de l'Intérieur, contiendra entre autre une paire de clés asymétriques utilisée pour la signature électronique. L'ensemble du processus de cryptographie devra être compatible avec le standard *PKCS #11.* Nous aurions voulu approfondir le sujet mais malheureusement personne au Ministère de l'Intérieur ne désirait nous rencontrer car le sujet est, selon leurs dires, trop sensible.

Les quelques éléments mis en avant lors de ce mémoire ne sont pas suffisants pour couvrir un sujet aussi vaste et complexe que celui-ci. Il en résulte que nous avons dû concentré nos efforts sur un seul domaine qui reflète au mieux le travail effectué durant le stage de fin d'études. Néanmoins, nous espérons que ce travail aura fourni une entrée en matière et permettra d'inspirer d'autres travaux et plus particulièrement dans le domaine des Smart Cards qui est encore en pleine évolution.

BIBLIOGRAPHIE

Publications

Bull, ICL, SSE et SNI, "Sesame Technology Version 4 : Cryptographic Application Developer guide", December 1995.

BURROW J.H., "Federal Information Processing Standards Publication (FIPS PUB 180-1) Secure Hash Standard", *National Institute of Standards and Technology,* March 1995.

" Internetworking Technology Overview : Open System Protocols ", chapter 32, June 1999.

ITU-T Recommendation X.690, " Information technology – ASN.1 encoding rules: pecification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", December 1997.

ITU-T Recommendation X.680, " Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation", December1997.

ITU-T Recommendation F.400/X.400, " Message handling system and service overview", June 1999.

ITU-T Recommendation X.509, " Technologies de l'information . Interconnexion des systèmes ouverts . L'annuaire: Cadre général des certificats de clé publique et d'attribut", Mars 2000.

ITU-T Recommendation X.200, "Information Processing Systems - OSI Reference Model - The Basic Model ", 1994

ITU-T, "Non-Telephone Telecomunication Services - Telematic services - Message Handling Services – Message Handling system and services overview ", June 2000.

KALISKY Jr., B.S., " An overview of the PKCS standards", *RSA Laboratories Technical Note*, November 1993.

KALISKY Jr., B.S.," A Layman's guide to a subset of ASN.1, BER, and DER", *RSA Laboratories Technical Note*, November 1993.

KAMMER R.G., "Federal Information Processing Standards Publication (FIPS PUB 186-2) Digital Signature Standard", *National Institute of Standards and Technology, January 2002.*

KENT S.," Request for Comments 1422 : Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures", February 1993.

LINN J.," Request for Comments 1421 : Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management", February 1993.

MENEZES A., VAN OORSCHOT P. et VANSTONE S., *Handbook of Applied Cryptography*, CRC Press, 1996.

PARKER T. et PINKAS D.,"Sesame Technology Version 4 : Overview", December 1995.

RSA Laboratories,"PKCS #11 v2.10 : Cryptographic Token Interface Standard", December 1999.

SHELFER K. M.et PROCACCINO J. D., "Smart Card Evolution", *Communication of the ACM,* Vol. 45, No. 7, July 2002.

VANDENWAUVER M., GOVAERTS R., VANDEWALLE J., " Overview of Authentification Protocols", Katholieke Universiteit Leuven.

Liens Internet

Ceveil, "Préambule à la normalisation", http://www.ceveil.qc.ca/Normes/premb.html , date of access 20/03/2002.

CHAN S.," An overview of Smart Card Security", http://home.hkstar.com/~alanchan/papers/smartCardSecurity/, date of access 27/03/2002.

Cicso, "Open System Interconnection Routing Protocol", http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/osi_prot.htm, date of access 27/02/2002.

DESCAMPS O. "Standardisation : les constructeurs compliquent le jeu", http://www.weblmi.com/ENQUETES/2001/896_6_normesstandardis00/htm, date of access 20/03/2002.

FOURNIER R. et KAMIONEK T., "The Cryptographic Smart Card : An Integrated Security Solution", http://www.rsasecurity.com/even.pdf, 25/04/2001, date of access 27/03/2002.

"Introduction to ASN.1",http://asn1.elibel.tm.fr/introduction/, 18/02/2002, date of access 27/02/2002.

KENT S. T., Essay 17 Privacy Enhanced Mail, http://www.acsac.org/secshelf/b.pdf, date of access 24/02/2002.

"Public-Key Cryptography Standards", http://www.pkcs.org/, date of access 26/02/2002.

"Sesame History", http://www.esat.kuleuven.ac.be/cosic/sesame/html/sesame_history.html, date of access 26/03/2002.

"Smart Card Basics",http://www.smartcardbasics.com/, date of access 27/03/2002.

"Smart Card Overview", http://www.javacard.org/others/smart_card.htm , date of access 27/03/2002.

http://www.meehan.cs.wwu.edu/nw3courses/cs417f/common/pkcs/overview.htm, date of access 26/02/2002.