Université Saâd DAHLAB de Blida

Faculté des Sciences

Département d'Informatique

Mémoire Présenté par :

GAROUI Adel

Promoteur :

Mme BENSETTITI Souad

En vue d'obtenir le diplôme de Master

Spécialité : Informatique

Sujet :

Courbes Elliptiques appliquées aux Cartes à Puce

Juillet, 2011

Résumé

L'usage des courbes elliptiques en cryptographie a été suggéré, par Neal Koblitz et Victor Miller en 1985, alors que RSA, un cryptosystème largement utilisé de nos jours, a été développé par Rivest, Shamir et Adleman en 1977. La taille des clés employées dans les cryptosystèmes basés sur les courbes elliptiques est nettement inférieure à celle des clés employées par RSA, ce qui rend les opérations de chiffrement, déchiffrement, signature et vérification de signature plus rapides.

Une carte à puce est une carte en plastique portant un circuit intégré qui peut stocker des données et exécuter des commandes. Aujourd'hui, les cartes à puce sont utilisées principalement pour le payement électronique, et pour stocker les informations de l'utilisateur. Elles sont également utilisées pour stocker des clés privées et pour exécuter des algorithmes cryptographiques qui emploient ces clés privées.

L'objet de cette thèse est de comparer les deux cryptosystèmes RSA et ECDSA afin de décider, lequel des deux algorithmes est le plus approprié pour être utilisé sur des cartes à puce. Les critères choisis pour cette comparaison sont, la sécurité, l'espace requis et le temps d'exécution.

Mots clés : courbe elliptique, cryptographie, RSA, cryptosystème, chiffrement, signature, carte à puce, ECDSA.

Abstract

In 1985 Neal Koblitz and V.S. Miller proposed elliptic curves to be used in cryptography, whereas RSA, a nowadays widely used cryptosystem, was developed by Rivest, Shamir, and Adleman in 1977. The elliptic curve cryptosystem benefits from smaller key sizes than RSA, which makes its cryptographic operations, encryption, decryption, signing, and signature verification faster than RSA's operations.

A smart card is a single-chip microcomputer. Today smart cards are used mainly for electronic identification and storing user information. Smart cards are also used to store private keys and to execute cryptographic operations which use private keys.

The purpose of this thesis is to compare the two cryptosystems RSA and ECDSA to decide, which of the two algorithms is most appropriate for use on smart cards. The criteria chosen for this comparison are security, space requirements and efficiency.

Keywords : elliptic curve, cryptography, RSA, cryptosystem, encryption, signature, smart card, ECDSA.

Remerciements

Je tiens à témoigner ma reconnaissance à ma promotrice Madame Souad BENSETTITI, Chargée de cours à l'université Saâd DAHLAB de Blida, pour tous ses conseils précieux, ses critiques constructives et ses encouragements. Qu'elle trouve ici l'expression de ma profonde gratitude.

Dédicaces

A mes très chers parents que personne ne peut compenser les sacrifices qu'ils ont consentis pour mon éducation et mon bien être.

A ma très chère Imene qui n'a jamais cessé de me soutenir moralement.

A mon frère et à mes trois sœurs.

Je dédie ce travail.

Table des matières

Liste des algorithmes

Liste des figures

Liste des tableaux

Liste des symboles

F_q	Corps fini de cardinal q.
F_q^*	Groupe multiplicatif de F_q.
F_p	Corps premier.
F_{2^m}	Corps binaire.
$\langle g \rangle$	Groupe cyclique généré par g.
$\oplus$	Ou-exclusif bit à bit.
$\&$	Et bit à bit.
$\gg i$	Décalage à droite par i positions.
$\ll i$	Décalage à gauche par i positions.
$a\|\|b$	Concaténation des chaines a et b.
$\#S$	Cardinalité d'un ensemble S.
E	Courbe elliptique.
∞	Le point à l'infini.
$E(L)$	Ensemble des points *L-rationnels* de E.
$\#E$	La cardinalité d'une courbe elliptique.

Liste des acronymes

ANSI	American National Standards Institute
APDU	Application Protocol Data Unit
API	Application Programming Interface
CPU	Central Processing Unit
DLP	Discrete Logarithm Problem
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
ECC	Elliptic Curve Cryptosystem
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
EEPROM	Electrical Erasable Programmable Read Only Memory
GF	Galois Field
GSM	Global System for Mobile communications
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IFP	Integer Factorization Problem
ISO	International Standards Organization
JCRE	Java Card Runtime Environment
JCVM	Java Card Virtual Machine
JCWDE	Java Card Workstation Development Environment
JVM	Java Virtual Machine
MIPS	Million Instructions Per Second
NFS	Number Field Sieve

NIST	National Institute of Standards
PKCS	Public Key Cryptography System
QS	Quadratic Sieve
RAM	Random Access Memory
ROM	Read Only Memory
RSA	Rivest-Shamir-Adleman public key encryption scheme

Introduction générale

De nos jours, les cartes à puces sont utilisées dans plusieurs domaines, tels que le payement électronique, la téléphonie mobile et le secteur de santé.

Les différents établissements qui utilisent cette technologie, doivent avoir un moyen d'identifier un client ou un utilisateur. Ceci est généralement accompli en utilisant la signature numérique à l'aide d'un algorithme de chiffrement à clé publique.

La signature numérique fournit un moyen très sûr pour vérifier l'identité d'une personne, cependant les algorithmes à clés publique nécessitent le calcul sur des grands nombres, ce qui rend leur implémentation difficile sur un système embarqué tel qu'une carte à puce.

Les algorithmes basés sur les courbes elliptiques fournissent le même niveau de sécurité que les autres algorithmes largement utilisés tels que RSA mais avec des clés de taille nettement inférieure. Ceci nous permet de minimiser l'espace requis pour le stockage des clés ainsi que l'accélération des opérations de chiffrement et de déchiffrement, d'où l'intérêt d'utiliser de tels algorithme sur des cartes à puce.

Dans notre cadre de travail, nous allons faire une comparaison entre RSA et ECDSA (*Elliptic Curve Digital Signature Algorithm*) qui est le standard du NIST (*National Institute of Standards and Technology*) pour la signature numérique à base de courbes elliptiques. Cette comparaison va nous permettre de décider, lequel des deux algorithmes est le plus approprié pour être employé sur des cartes à puce. Les critères choisis pour cette comparaison sont, la sécurité, l'espace requis et le temps d'exécution.

Cette thèse possède la structure suivante :

- Chapitre 1 : Nous introduisons les concepts de base de la cryptographie, et nous décrivons les deux types de base des algorithmes de chiffrement, les algorithmes de chiffrement à clé secrète et les algorithmes de chiffrement à clé publique, y compris la signature numérique.
- Chapitre 2 : Nous présentons les préliminaires mathématiques nécessaires pour réaliser les algorithmes RSA et ECDSA. Le chapitre commence par une généralisation sur les corps finis, après nous décrivons les algorithmes nécessaires pour effectuer l'arithmétique sur les corps premiers ainsi que les corps binaires. La théorie des courbes elliptiques ainsi que les opérations d'addition, doublement et multiplication de points sont détaillées dans le reste du chapitre.

- Chapitre 3 : Ce chapitre est composé de deux parties. La première partie est consacrée à la description de l'algorithme RSA. La deuxième partie est consacrée aux algorithmes basés sur les courbes elliptiques ainsi que la description de l'algorithme ECDSA.
- Chapitre 4 : Nous commençons par introduire les cartes à puce, après nous étudierons les cartes à puce Java Cards. Cette étude porte sur les caractéristiques matériels et logiciel de ces cartes ainsi que la création des applications Java Card.
- Chapitre 5 : Nous comparons les algorithmes RSA et ECDSA du point de vue, sécurité, espace requis et temps d'exécution.
- Conclusion et perspectives : Nous résumons les conclusions tirées de cette thèse et nous présentons quelques perspectives en rapport avec notre projet.

Chapitre 1

Chiffrement et Signature numérique

Supposons que deux parties veulent échanger des données confidentielles. Ces deux parties doivent s'assurer qu'aucune oreille indiscrète ne puisse s'informer des données échangées. D'autre part, une partie communicante doit être capable de vérifier l'intégrité ainsi que la source des données. Par exemple, une banque doit être capable de vérifier l'identité d'un client qui cherche à effectuer une transaction à l'aide d'une carte à puce. Le chiffrement des messages et la signature numérique sont deux techniques cryptographiques qui permettent de satisfaire ces besoins.

Nous commençons par présenter les concepts de base de la cryptographie. Apres nous décrivons les deux types fondamentaux de chiffrement, le chiffrement à clé secrète et le chiffrement à clé publique, ainsi que la notion de signature numérique.

1.1. Qu'est ce que la cryptographie ? [Fou03]

La cryptographie est une discipline ayant pour but de garantir la protection des informations transmises sur un canal non sécurisé contre différents types d'adversaires. La protection des informations se définit en termes de confidentialité, d'intégrité, d'authentification et de non-répudiation.

- La confidentialité garantit que les données transmises ne sont pas dévoilées à une tierce personne.
- L'intégrité assure que ces données n'ont pas été modifiées entre l'émission et la réception.
- L'authentification de message assure que les données proviennent bien de la bonne entité.
- La non-répudiation garantit qu'une entité ne peut pas nier d'avoir envoyer un message qu'elle a vraiment envoyé.

1.2. Texte en clair, Texte chiffré, Chiffrement et Déchiffrement [Sch94]

Un message source est appelé *texte en clair*. Le processus de transformation d'un message de telle manière à le rendre incompréhensible est appelé *chiffrement*. Le résultat de ce processus de chiffrement est appelé *texte chiffré* (ou *cryptogramme*).

Le processus de reconstitution du texte en clair à partir du texte chiffré est appelé *déchiffrement*. Ces différents processus sont illustrés par la figure ci-dessous.

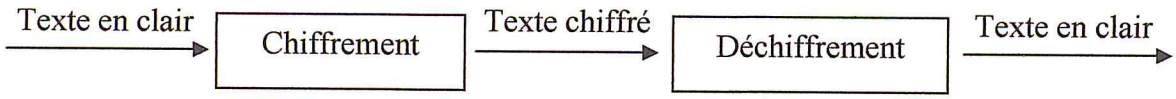


Figure 1.1 : Chiffrement et Déchiffrement

1.3. Algorithmes de chiffrement à clé secrète (Symétriques)

1.3.1. Définition [Sch94]

Un algorithme de chiffrement (cryptosystème) à clé secrète est un algorithme où la clé de chiffrement peut être calculée à partir de la clé de déchiffrement (Dans la plupart des cas les deux clés sont identiques).

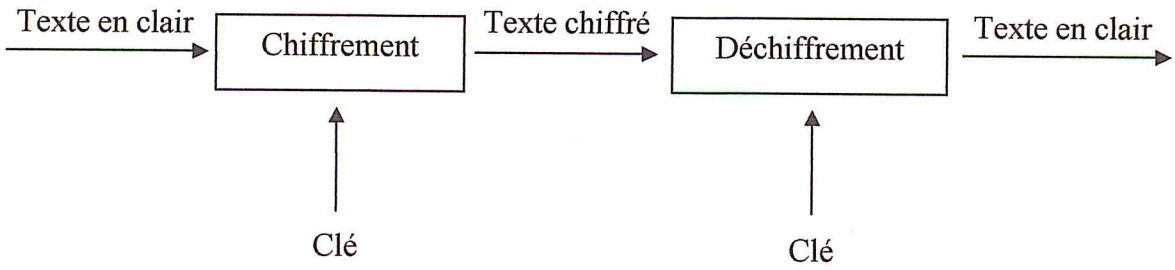


Figure 1.2 : Chiffrement et déchiffrement avec la même clé

Pour échanger des données à l'aide d'un tel algorithme, les deux parties doivent initialement s'entendre sur une clé secrète. La sécurité de l'algorithme repose sur cette clé : si celle-ci est dévoilée, alors n'importe qui peut chiffrer ou déchiffrer des messages dans ce cryptosystème.

1.3.2. Chiffre de César [Sch94]

Un exemple très simple de la cryptographie à clé secrète est le chiffre de César.

Dans le chiffre de césar, chaque caractère du texte en clair est remplacé par celui qui se trouve trois places plus loin dans l'alphabet. Observez la figure ci-dessous :

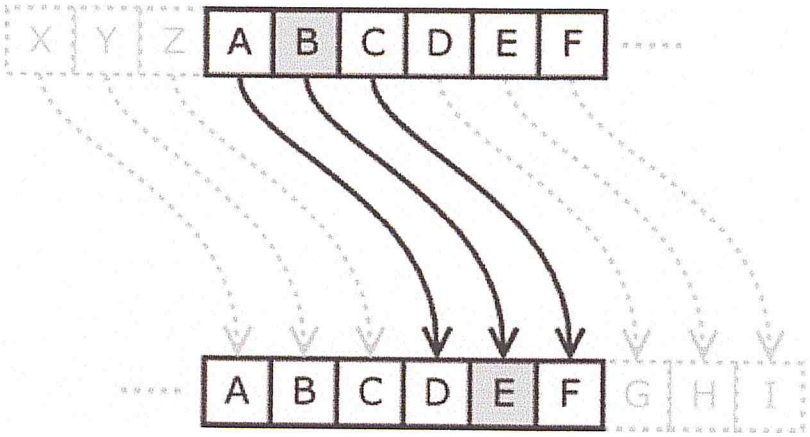


Figure 1.3 : Chiffre de César

Si on veut chiffrer le message "SECRET" on aura comme résultat "VHFUHW". Pour retrouver le message original, on applique le processus inverse, i.e. chaque caractère du texte chiffré est remplacé par celui qui se trouve trois places avant lui dans l'alphabet. Donc la clé secrète est le nombre de déplacement dans l'alphabet qui est égal à 3.

1.3.3. Avantages et inconvénients [Men93]

Le premier argument en faveur des cryptosystèmes symétriques est qu'ils sont très rapides, car ils utilisent directement les instructions câblées du processeur. Cependant, ces systèmes possèdent les inconvénients suivants qui rendent leur utilisation inappropriée dans certaines applications.

- **Le problème de distribution des clés** : Les deux parties doivent échanger une clé secrète sur un canal sécurisé avant de pouvoir échanger des messages. Dans certaines applications telles que le commerce ou la messagerie électronique, la présence d'un canal sécurisé n'est pas une chose évidente.
- **Le problème de gestion des clés** : Dans un réseau de n utilisateurs, chaque deux utilisateurs doivent partager une clé secrète. Ce qui donne un total de $n(n-1)/2$ clés à gérer. Si n est trop grand, alors le nombre de clés devient ingérable.
- **Pas de signature possible** : Une signature numérique est un analogue électronique d'une signature manuscrite. Elle permet au récepteur d'un message de convaincre une tierce partie que le message provient bien de l'expéditeur. Dans un cryptosystème à clé secrète, les deux parties *Alice* et *Bob* ont la même clé de chiffrement et de déchiffrement, et donc *Bob* ne peut pas convaincre une tierce partie que le message qu'il a reçu provient bien d'*Alice*.

1.4. Algorithmes de chiffrement à clé publique (Asymétrique)

1.4.1. Définition [Sch94]

En cryptographie à clé publique les clés de chiffrement et de déchiffrement sont différentes. L'une est appelée clé publique et est utilisée pour le chiffrement, l'autre est appelée clé privée et est utilisée pour le déchiffrement. Le calcul de la clé privée à partir de la clé publique est impossible à réaliser en un temps raisonnable. Ceci est illustré dans la figure ci-dessous.

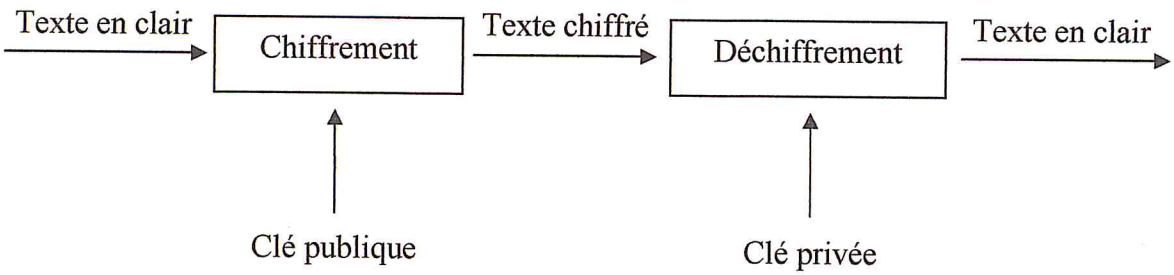


Figure 1.4 : Chiffrement et déchiffrement avec deux clés

Un utilisateur *Bob* souhaitant échanger des données à l'aide d'un tel algorithme crée une paire de clés, l'une privée qu'il garde secrète et l'autre publique qu'il diffuse en public. Si un utilisateur souhaite envoyer des messages à *Bob*, il utilise sa clé publique pour chiffrer les messages. *Bob* seul saura les déchiffrer en utilisant sa clé privée.

1.4.2. Les fonctions à sens unique [Fou03]

Les algorithmes asymétriques font appel à des fonctions dites *à sens unique*. De telles fonctions méritent leur nom du fait qu'il est quasiment impossible de les inverser.

Par exemple, connaissant deux nombre premier p et q, il est facile de les multiplier pour obtenir $(n = p.q)$. Par contre l'opération inverse de factorisation est impossible à réaliser en un temps raisonnable si les deux nombres p et q sont très grands.

Les fonctions à sens unique à trappe (à brèche secrète) sont des fonctions à sens unique telles que la connaissance d'une trappe (la clé privé), rend possible l'inversion efficace de la fonction. C'est grâce à ces fonctions qu'on a pu construire des systèmes de chiffrement à clé publique. Par exemple, dans l'algorithme RSA, la fonction à trappe repose sur le problème de factoriser des grands nombres (Voir 3.1).

1.4.3. Signature Numérique [MOV96]

La signature numérique permet de vérifier la source d'un message ainsi que son intégrité. C'est-à-dire, elle permet de vérifier l'identité de la personne à l'origine du message, et de s'assurer qu'il n'a pas été modifié avant ça réception. Ceci peut être très utile. Par exemple dans le monde des cartes à puces, cela permet à une banque de vérifier l'identité d'un client avant d'autoriser une transaction.

La signature numérique est réalisée à l'aide d'un algorithme de chiffrement à clé publique. La personne voulant prouver son identité, utilise sa clé privée pour signer le message. Cette signature est vérifiée en utilisant la clé publique associée.

Généralement, pour réaliser la signature numérique, il faut utiliser *une fonction de hachage*. Cette dernière permet de réduire la quantité des données à signer et donc, rendre l'opération de signature plus rapide (Voir 1.4.4).

Certains cryptosystèmes asymétriques, permettent de faire le chiffrement ainsi que la signature numérique (c'est le cas pour RSA). D'autres algorithmes sont destinés uniquement à faire de la signature numérique. Par exemple, l'algorithme ECDSA est un algorithme qui repose sur le problème du logarithme discret et qui permet seulement de faire la signature numérique (Voir 3.2).

1.4.4. Fonction de hachage [MOV96]

Une fonction de hachage est une fonction mathématique qui à partir d'une chaine de caractères quelconque produit une chaine de caractères de taille fixe, la chaine résultante est appelée *empreinte* ou *condensé*.

Afin d'être utilisée en cryptographie, la fonction de hachage doit être :

- *A sens unique* : c'est-à-dire étant donné une empreinte, il est très difficile de trouver un message qui produit cette empreinte.
- *Résistante aux collisions* : c'est-à-dire que deux messages distincts doivent avoir très peu de chances de produire la même empreinte.

Une fonction de hachage est généralement utilisée avec la signature numérique. En effet si le message à signer est trop grand, on lui applique une fonction de hachage afin de réduire la taille des données à signer. Cela nous permet de minimiser le temps d'exécution.

Les deux fonctions de hachage les plus utilisés sont *SHA (Secure Hash Algorithm)* qui produit une empreinte de 160 bits, et *MD5* qui produit une empreinte de 128 bits.

1.4.5. Avantages de la cryptographie asymétrique [Fou03]

- Elle permet de résoudre le problème d'échange de clé. En effet, un utilisateur souhaitant partager des informations confidentielles, doit juste créer une paire de clés (privée, publique). La première est gardée secrète et la dernière est diffusée en publique.
- Elle permet de réaliser la signature numérique.

1.4.6. Inconvénients de la cryptographie asymétrique [Fou03]

- Les performances des systèmes asymétriques sont beaucoup moins bonnes que celles des systèmes symétriques car ces systèmes nécessitent l'utilisation de l'arithmétique sur les grands nombres qui n'est pas implémentée dans les processeurs des ordinateurs usuels.
- La taille des clés est généralement plus grande pour ces systèmes que pour les systèmes à clé secrète.

Après avoir fini d'introduire les différentes notions de la cryptographie classique et moderne, on attaque le chapitre intitulé "Corps finis et Courbes elliptiques". Ce chapitre consiste à présenter les préliminaires mathématiques ainsi que les algorithmes nécessaires pour implémenter les cryptosytèmes RSA et ECDSA.

Chapitre 2

Corps finis et Courbes elliptiques

L'implémentation de l'arithmétique des corps finis est un pré-requis indispensable pour la mise en œuvre des systèmes de chiffrement à clé publique tels que RSA ou ECDSA. Par exemple, l'implémentation de l'algorithme RSA repose sur l'arithmétique des corps premiers. D'autre part, l'implémentation des systèmes de chiffrement basés sur les courbes elliptiques tels qu'ECDSA repose sur l'arithmétique de la courbe choisie. Une courbe elliptique est définie sur un corps fini de notre choix (corps premier, corps binaire, ...), donc l'implémentation de l'arithmétique des courbes elliptiques repose sur l'arithmétique du corps fini choisi.

Ce chapitre est divisé en deux parties. Dans la première partie, nous introduisons les corps finis et nous présentons des algorithmes pour réaliser les opérations arithmétiques dans les corps premiers et les corps binaires. La théorie des courbes elliptiques ainsi que les algorithmes nécessaires pour réaliser les opérations arithmétiques sur une courbe, sont présentés dans la deuxième partie.

2.1. Corps finis

2.1.1. Introduction aux corps finis [Ler97]

2.1.1.1. Loi de composition interne :

Soit F un ensemble. Une loi de composition interne sur F est une application T de $F \times F$ dans F.

2.1.1.2. Groupes Abéliens :

Soit F un ensemble, et $+$ une loi de composition interne sur F. On dit que $(F, +)$ est un groupe abélien si et seulement si :

- $+$ est commutative : $\forall x, y \in F : x + y = y + x$.
- $+$ est associative : $\forall x, y, z \in F : (x + y) + z = x + (y + z)$.
- $+$ admet un élément neutre : $\exists e \in F \; tel \; que \; \forall x \in F : x + e = x$.
- Tout élément de F admet un symétrique pour $+$:
  $\forall x \in F, \exists y \in F \; tel \; que \; x + y = e$ (e est l'élément neutre)

2.1.1.3. Groupes Cycliques :

Un groupe est appelé cyclique s'il peut être généré par un seul élément.

Soit G un groupe, $g \in G$. L'*ordre* de g est le plus petit entier n strictement positif tel que $ng = 0$.

Si G est un groupe et $g \in G$, alors *le sous-groupe engendré par* g noté $\langle g \rangle$ est $\langle g \rangle = \{ ng \mid n \in Z \}$.

2.1.1.4. Corps :

Soit F un ensemble, et +, . Deux lois de composition interne sur F. On dit que $(F, +, .)$ est un corps si et seulement si :

- $(F, +)$ est un groupe abélien avec un élément neutre noté 0.
- $(F\backslash\{0\}, .)$ est un groupe abélien avec un élément neutre noté 1.
- $\forall a, b, c \in F : (a + b).c = a.c + b.c$

Si l'ensemble F est fini, on dit que $(F, +, .)$ est un corps fini.

2.1.1.5. Remarques :

- Dans tout ce qui suit on écrira F pour faire référence au corps $(F, +, .)$.
- Les deux lois de composition + et . sont respectivement appelées opération d'addition et opération de multiplication.

2.1.1.6. Soustraction et Division :

On définit deux nouvelles opérations, soustraction notée $(-)$ et division notée $(/)$ de la manière suivante :

- Soit $a, b \in F$
  - $a - b = a + (-b)$ où $(-b)$ est le symétrique de b par rapport a +.
    - $(-b)$ est appelé *l'opposé* de b.
- Soit $a, b \in F\backslash\{0\}$
  - $a / b = a . b^{-1}$ où b^{-1} est le symétrique de b par rapport a « . ».
    - b^{-1} est appelé *l'inverse* de b.

2.1.1.7. Existence et unicité :

- Le *cardinal* d'un corps fini est le nombre d'éléments dans le corps.

- Il existe un corps fini F de cardinal q si et seulement si q est un nombre primaire, i.e., $q = p^m$ où p est un nombre premier appelé la *caractéristique* de F, et m est un nombre naturel.
    - Si $m=1$ alors F est appelé *corps premier*.
    - Si $m \geq 2$ alors F est appelé *corps d'extension*.
- Deux corps finis quelconques de cardinal q sont structurellement identiques sauf que l'étiquetage utilisé pour représenter les éléments du corps peut être différent.
    - On dit que n'importe quels deux corps finis de cardinal q sont *isomorphes*, et on note un tel corps : F_q .

2.1.1.8. Corps premiers :

Soit p un nombre premier, les entiers modulos p formant l'ensemble $\{0,1,\ldots,p-1\}$ avec les opérations d'addition et multiplication réalisées modulo p forment un corps fini de cardinal p.

- Un tel corps est noté F_p et p est appelé le *modulo* de F_p.
- Soit a un entier, $a \bmod p$ représente l'entier résiduel r, $0 \leq r \leq p-1$ obtenu en divisant a par p.
    - Cette opération est appelée *réduction modulo p*.

2.1.1.9. Corps binaires :

Les corps finis de cardinal 2^m sont appelés des corps binaires.

- Une façon de construire F_{2^m} est d'utiliser une *représentation à base polynômiale*.
- Ici, les éléments de F_{2^m} sont les polynômes binaires (polynômes avec des coefficients dans $F_2 = \{0.1\}$) de degré inférieur strictement à m :
    - $F_{2^m} = \{ a_{m-1}z^{m-1} + \cdots + a_2z^2 + a_1z + a_0 : a_i \in \{0.1\}\}$.
- Un polynôme binaire irréductible $f(z)$ de degré m est choisi. L'irréductibilité de $f(z)$ signifie que $f(z)$ ne peut pas être factorisé en produit de polynômes binaires chacun de degré inférieur à m.
- L'addition des éléments du corps est l'addition usuelle des polynômes réalisée modulo 2.
- La multiplication des éléments est réalisée modulo le polynôme irréductible choisi $f(z)$.
    - Soit $a(z)$ un polynôme binaire, $a(z) \bmod f(z)$ représente le polynôme résiduel $r(z)$ de degré inférieur strictement à m, obtenus en divisant $a(z)$ par $f(z)$.
        - ➢ Cette opération est appelée *réduction modulo f(z)*.

2.1.1.10. Corps d'extension :

La représentation à base polynômiale des corps binaires peut être généralisée à tous les corps d'extension comme suit :

- Les éléments de F_{p^m} sont les polynômes avec des coefficients dans F_p de degré inférieur strictement à m :
  - $F_{p^m} = \{ a_{m-1}z^{m-1} + \cdots + a_2 z^2 + a_1 z + a_0 : a_i \in F_p \}$
- Un polynôme irréductible *f(z)* de degré *m* est choisi.
- L'addition des éléments du corps est l'addition usuelle des polynômes réalisée modulo *p*.
- La multiplication des éléments est réalisée modulo le polynôme irréductible choisi *f(z)*.

2.1.1.11. Sous-corps d'un corps fini :

Soit *(F, +, .)* un corps, et soit *f* un sous-ensemble de *F*.

- Si *(f, +, .)* est un corps, alors on dit que *f* est un sous-corps de *F*.
- *F* est appelé corps d'extension de *f*.

Les sous-corps d'un corps fini peuvent être facilement caractérisés.

- Un corps fini F_{p^m} possède un sous-corps de cardinal p^l pour chaque diviseur positif *l* de *m*.
- Les éléments du sous-corps sont les éléments $a \in F_{p^m}$ vérifiant : $a^{p^l} = a$.

2.1.1.12. Groupe multiplicatif d'un corps fini :

Les éléments différents de zéro d'un corps fini F_q noté F_q^*, forment un groupe cyclique par rapport à l'opération de multiplication.

Par conséquent ils existent des éléments $b \in F_q^*$ appelés générateurs tels que :

$$F_q^* = \{ b^i : 0 \leq i \leq q-2 \}$$

L'*ordre* d'un élément $a \in F_q^*$ est le plus petit entier t strictement positif tel que : $a^t = 1$

- Puisque F_q^* est un groupe cyclique, donc t est un diviseur de $q-1$.

2.1.2. Arithmétique des corps premiers

Cette section présente des algorithmes pour réaliser les opérations arithmétiques dans le corps fini F_p.

Les algorithmes présentés n'exigent aucune forme spéciale du modulo p , mais l'étape de réduction peut être accélérée considérablement lorsque le module p possède une forme spéciale. Les algorithmes efficaces de réduction pour les nombres premiers suggérés par le *NIST(National Institute of Standards and Technology)* tel que $p = 2^{192} - 2^{64} - 1$ sont considérés aussi.

Nous supposons que la plate-forme d'implémentation possède une architecture de W-bit où W est un multiple de 8. Les postes de travail possèdent généralement une architecture de 32, ou 64 bits. Les composants de faible puissance peuvent avoir un W plus petit, par exemple, les cartes à puce peuvent avoir W = 8.

Les éléments de F_p sont les nombre entiers de *0 à p - 1*. Soit $m = \lceil log_2 p \rceil$ le nombre de bits de p , et $t = m/W$ le nombre de mots de W-bit de p . La figure ci-dessous illustre le cas où la représentation d'un élément a du corps est stockée dans un tableau A de longueur t contenant des mots de W-bit.

A [0]	A [1]	…	…	A [t-1]

Figure 2.1 : Représentation d'un entier à l'aide d'un tableau

$$a = A[0] + 2^w A[1] + 2^{2w} A[2] + \cdots + 2^{(t-1)w} A[t-1]$$

2.1.2.1. Addition et soustraction

Les algorithmes d'addition et de soustraction dans le corps F_p sont donnés en termes des algorithmes correspondants pour les entiers multi-mots. La notation et la terminologie suivantes sont utilisées.

Une affectation de la forme "$(\varepsilon, z) \leftarrow w$" pour un entier w , est entendue par

$$z \leftarrow w \bmod 2^W, et$$

$$\varepsilon \leftarrow 0 \; si \; w \in [0, 2^W[\, , autrement \; \varepsilon \leftarrow 1$$

Si $w = x + y + \varepsilon'$ pour $x, y \in [0, 2^W[$ et $\varepsilon' \in \{0,1\}$, alors $w = \varepsilon 2^W + z$ et ε est appelé *la retenue* de l'addition de deux mots.

L'algorithme 2.1 effectue l'addition des entiers multi-mots.

Algorithme 2.1 : Addition multi-précision [Knu98]

Entrées : Les entiers $a, b \in [0, 2^{Wt}[$.
Sorties : (ε, c) où $c = a + b \ mod \ 2^{Wt}$ et ε est la retenue.

1. $(\varepsilon, C[0]) \leftarrow A[0] + B[0]$.
2. $pour\ i\ de\ 1\ à\ t-1\ faire$:
   2.1. $(\varepsilon, C[i\,]) \leftarrow A[i] + B[i] + \varepsilon$.
3. $Retourner\ (\varepsilon, c)$.

La soustraction des entiers multi-mots (Algorithme 2.2) est similaire à l'addition, avec la retenue appelé *"emprunt"* dans ce contexte.

Algorithme 2.2 : Soustraction multi-précision [Knu98]

Entrées : Les entiers $a, b \in [0, 2^{Wt}[$.
Sorties : (ε, c) où $c = a - b \ mod \ 2^{Wt}$ et ε est l'emprunt.

1. $(\varepsilon, C[0]) \leftarrow A[0] - B[0]$.
2. $pour\ i\ de\ 1\ à\ t-1\ faire$:
   2.1. $(\varepsilon, C[i\,]) \leftarrow A[i] - B[i] - \varepsilon$.
3. $Retourner\ (\varepsilon, c)$.

L'addition modulaire $((x + y)\ mod\ p)$ et la soustraction $((x - y)\ mod\ p)$ sont directement adaptées des algorithmes correspondants ci-dessus, avec une étape supplémentaire pour la réduction modulo p.

Algorithme 2.3 : Addition dans F_p [MOV96]

Entrées : Le modulo p , et les entiers $a, b \in [0, p-1[$.
Sorties : $c = a + b \ mod \ p$.

1. Utiliser l'algorithme 1 pour obtenir (ε, c) où $c = a + b \ mod \ 2^{Wt}$ et ε est la retenue.
2. Si $\varepsilon = 1$, alors soustraire p de $c = (C[0], C[0], \ldots, C[t-1])$;
   Sinon, si $c \geq p$ alors$c \leftarrow c - p$.
3. Retourner c .

Algorithme 2.4 : Soustraction dans F_p [MOV96]

Entrées : Le modulo p , et les entiers $a, b \in [0, p-1[$.
Sorties : $c = a - b \bmod p$.

1. Utiliser l'algorithme 1 pour obtenir (ε, c) où $c = a - b \bmod 2^{Wt}$ et ε est l'emprunt.
2. Si $\varepsilon = 1$, alors additionner p à $c = (C[0], C[0], \ldots, C[t-1])$.
3. Retourner c .

2.1.2.2. Multiplication des entiers

La multiplication de $a, b \in Fp$ peut être accomplie en multipliant d'abord les entiers a et b, puis réduire le résultat modulo p.

L'algorithme 2.5 effectue la multiplication des entiers multi-mots.

(uv) désigne un mot de (2W)-bit obtenu en concaténant u et v (mots de W-bit).

Algorithme 2.5 : Multiplication multi-précision [Knu98]

Entrées : $x \in [0, 2^{Wn}[$ *et* $y \in [0, 2^{Wt}[$.
Sorties : $w \in [0, 2^{W(n+t)}[$ *tel que* $w = x \times y$.

1. *Pour* i *de* 0 à $(n+t-1)$ *faire* : $w[i] \leftarrow 0$.
2. *Pour* i *de* 0 à $(t-1)$ *effectuer les opérations suivantes* :
   2.1. $c \leftarrow 0$.
   2.2. *Pour* j *de* 0 à $(n-1)$ *effectuer les opérations suivantes* :
      2.2.1. $uv \leftarrow w[i+j] + x[j].y[i] + c$.
      2.2.2. $w[i+j] \leftarrow v$.
      2.2.3. $c \leftarrow u$.
   2.3. $w[i+n] \leftarrow u$.
3. *Retourner* w.

2.1.2.3. Division

La division est la plus compliquée et la plus coûteuse des opérations de base.

L'algorithme 2.6 calcule le quotient q et le reste r en divisant x par y.

Algorithme 2.6 : Division multi-précision [Knu98]

Entrées : $x \in [0, 2^{Wn}[$ *et* $y \in [0, 2^{Wt}[$ avec $n \geq t \geq 2$ *et* $y_{t-1} \neq 0$.
Sorties : $q \in [0, 2^{W(n-t+1)}[$ *et* $r \in [0, 2^{Wt}[$.

1. *Pour* j *de* 0 à $(n-t)$ *faire* : $q_j \leftarrow 0$.
2. *Tant que* $\left(x \geq y2^{W(n-t)}\right)$ *faire* :
   2.1. $q_{n-t} \leftarrow q_{n-t} + 1$.
   2.2. $x \leftarrow x - y2^{W(n-t)}$.
3. *Pour* i *de* $(n-1)$ à t *faire* :
   3.1. *Si* $x_i = y_{t-1}$ *alors* $q_{i-t} \leftarrow 2^W - 1$; *Sinon* $q_{i-t} \leftarrow \lfloor (x_i 2^W + x_{i-1})/y_{t-1} \rfloor$.
   3.2. *Tant que* $(q_{i-t}(y_{t-1}2^W + y_{t-2}) > x_i 2^{2W} + x_{i-1}2^W + x_{i-2})$ *faire* :
   $q_{i-t} \leftarrow q_{i-t} - 1$.
   3.3. $x \leftarrow x - q_{i-t}y2^{W(i-t)}$.
   3.4. *Si* $x < 0$ *alors* $x \leftarrow x + y2^{W(i-t)}$ *et* $q_{i-t} \leftarrow q_{i-t} - 1$.
4. $r \leftarrow x$.
5. *retourner* (q, r).

Note :

La condition $n \geq t \geq 2$ peut être remplacée par $n \geq t \geq 1$ à condition de prendre $x_j = y_j = 0$ à chaque fois qu'un indice $j < 0$ est rencontrées dans l'algorithme.

2.1.2.4. Réduction

Pour un modulo p qui n'a pas de forme spéciale, la réduction $z \bmod p$ peut être coûteuse.

Les modulos possédant une forme spéciale tels que les nombres premiers suggérés par le NIST permettent une réduction rapide.

Dans cette section, nous présentons la méthode de réduction de Barrett ainsi que la méthode de réduction en utilisant les nombres premiers suggérés par le NIST.

2.1.2.4.1. La réduction de Barrett

La réduction de Barrett (Algorithme 2.7) calcule $r = x \bmod m$ étant donné x et m. L'algorithme exige le pré calcul de la quantité $\mu = \lfloor 2^{W(2k)}/m \rfloor$, il est avantageux lorsque plusieurs réductions sont réalisés avec le même modulo.

Algorithme 2.7 : Réduction de Barrett [Bar87]

Entrées : $x \in [0, 2^{W(2k)}[$ et $m \in [0, 2^{Wk}[$ $(avec\ m_{k-1} \neq 0)$ et $\mu = \lfloor 2^{W(2k)}/m \rfloor$.
Sorties : $r = x\ mod\ m$.

1. $q_1 \leftarrow \lfloor x/2^{W(k-1)} \rfloor, q_2 \leftarrow q_1.\mu, q_3 \leftarrow \lfloor q_2/2^{W(k+1)} \rfloor$.
2. $r_1 \leftarrow x\ mod\ 2^{W(k+1)}, r_2 \leftarrow q_3.m\ mod\ 2^{W(k+1)}, r \leftarrow r_1 - r_2$.
3. $Si\ r < 0\ alors\ r \leftarrow r + 2^{W(k+1)}$.
4. $Tant\ que\ (r \geq m)\ faire : r \leftarrow r - m$.
5. $retourner\ r$.

Les divisions effectuées dans l'algorithme sont de simples décalages à droite.

2.1.2.4.2. Les nombres premiers suggérés par le NIST [FIPS00]

Les nombres premiers suggérés par le NIST sont :

$p_{192} = 2^{192} - 2^{64} - 1$

$p_{224} = 2^{224} - 2^{96} + 1$

$p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$

$p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$

$p_{521} = 2^{521} - 1$

Ces nombres peuvent s'écrire comme la somme ou la différence d'un petit nombre de puissances de 2. En outre (sauf pour P_{521}), les puissances figurant dans ces expressions sont toutes multiples de 8. Ces propriétés rapportent des algorithmes de réduction qui sont particulièrement rapide sur des machines de W-bit où W est un multiple de 8.

Exemple :

Soit $p = p_{192} = 2^{192} - 2^{64} - 1$ et c $un\ entier\ tel\ que\ 0 \leq c < p^2$.
Soit $c = c_0 + c_1 2^{64} + c_2 2^{128} + c_3 2^{192} + c_4 2^{256} + c_5 2^{320}$ la représentation en base 64 de c avec $0 \leq c_i \leq 2^{64} - 1$.

On peut réduire les puissances supérieures de 2 en utilisant les congruences :

$2^{192} \equiv 2^{64} + 1\ mod\ p$

$2^{256} \equiv 2^{128} + 2^{64}\ mod\ p$

$2^{320} \equiv 2^{128} + 2^{64} + 1\ mod\ p$

On obtient :

$$c = c_0 + c_1 2^{64} + c_2 2^{128} + \\ c_3 + c_3 2^{64} + \\ c_4 2^{64} + c_4 2^{128} + \\ c_5 + c_5 2^{64} + c_5 2^{128}$$

Ainsi, $c \ mod \ p$ peut être obtenu en additionnant les quatre entiers de 192-bit dans F_{192}.

Algorithme 2.8 : Réduction rapide modulo p_{192}

Entrées : $c = (c_0, c_1, c_2, c_3, c_4, c_5)$ *en base* 64 *avec* $0 \leq c < p_{192}$
Sorties : $c \ mod \ p_{192}$.

1. $definir\ les\ quatre\ entiers\ de\ 192 - bit$:
   $s_1 = (c_0, c_1, c_2), s_2 = (c_3, c_3, 0), s_3 = (0, c_4, c_4), s_4 = (c_5, c_5, c_5)$.
2. $Retourner\ (s_1 + s_2 + s_3 + s_4 \ mod \ p_{192})$.

2.1.2.5. Inversion

Rappelons que l'inverse d'un élément non nul $a \in F_p$, dénotait a^{-1}, est l'unique élément $x \in F_p$ tel que $ax = 1 (ax \equiv 1 \ mod \ p)$. Les inverses peuvent être efficacement calculés en utilisant l'algorithme d'Euclide étendu.

Le plus grand commun diviseur (*pgcd*), de deux entiers naturels est le plus grand entier naturel qui divise simultanément ces deux entiers.

Les algorithmes calculant le pgcd, exploitent le résultat suivant : Soient a et b deux entiers naturel, non tous deux nuls, $pgcd(a, b) = pgcd(b - ca, a)$ pour tout entier c.

Dans l'algorithme d'Euclide, pour calculer le pgcd de deux entiers a et b tel que $b \geq a$, b est divisé par a pour obtenir un quotient q et un reste r satisfaisant $b = qa + r$ et $0 \leq r < a$. Par le résultat précédent $pgcd(a, b) = pgcd(r, a)$, Ainsi, le problème de détermination du $pgcd(a, b)$ est réduit à celui de déterminer le $pgcd(r, a)$ où les arguments (r, a) sont plus petits que les arguments d'origine (a, b). Ce processus est répété jusqu'à ce qu'un des arguments est 0, le résultat est alors immédiatement obtenu puisque $pgcd(0, d) = d$.

L'algorithme d'Euclide peut être étendue pour trouver les entiers x et y tels que $ax + by = d$ où $d = pgcd(a, b)$.

Algorithme 2.9 : L'algorithme d'Euclide étendu [MOV96]

Entrées : $Deux\ entiers\ positifs\ a\ et\ b\ avec\ a \leq b.$
Sorties : $d = pgcd(a,b) et\ les\ entiers\ x, y\ satisfaisant\ ax + by = d.$

1. $u \leftarrow a, v \leftarrow b.$
2. $x_1 \leftarrow 1, y_1 \leftarrow 0, x_2 \leftarrow 0, y_2 \leftarrow 1.$
3. $Tant\ que\ (u \neq 0)\ faire$:
   3.1. $q \leftarrow \lfloor v/u \rfloor, r \leftarrow v - qu, x \leftarrow x_2 - qx_1, y \leftarrow y_2 - qy_1.$
   3.2. $v \leftarrow u, u \leftarrow r, x_2 \leftarrow x_1, x_1 \leftarrow x, y_2 \leftarrow y_1, y_1 \leftarrow y.$
4. $d \leftarrow v, x \leftarrow x_2, y \leftarrow y_2.$
5. $Retourner\ (d, x, y).$

Supposons maintenant que p est premier et $a \in [1, p-1]$, donc $pgcd(a,p) = 1$.

Si l'algorithme 2.9 est exécuté avec les entrées (a,p), le dernière reste non nul r rencontrées à l'étape 3.1 est $r = 1$. Ainsi, les entiers u, x_1 et y_1 mis à jour à l'étape 3.2 vont satisfaire l'équation $ax_1 + py_1 = u\ avec\ u = 1$. Ainsi $ax_1 \equiv 1\ mod\ p$ et ainsi $a^{-1} = x_1$.

Notez que y_1 et y_2 ne sont pas nécessaires pour le calcul de x_1. Ces observations conduisent à l'algorithme 2.10 pour l'inversion dans F_p.

Algorithme 2.10 : Inversion dans F_p avec l'algorithme d'Euclide étendu [MOV96]

Entrées : $Un\ nombre\ premier\ p\ et\ un\ entier\ a \in [1, p-1].$
Sorties : $a^{-1}\ mod\ p.$

1. $u \leftarrow a, v \leftarrow p.$
2. $x_1 \leftarrow 1, x_2 \leftarrow 0.$
3. $Tant\ que\ (u \neq 1)\ faire$:
   3.1. $q \leftarrow \lfloor v/u \rfloor, r \leftarrow v - qu, x \leftarrow x_2 - qx_1.$
   3.2. $v \leftarrow u, u \leftarrow r, x_2 \leftarrow x_1, x_1 \leftarrow x.$
4. $Retourner\ (x_1\ mod\ p).$

Un algorithme de division $b/a = ba^{-1}\ mod\ p$ peut être obtenu directement à partir de l'algorithme d'inversion en modifiant l'initialisation $x_1 = 1\ par\ x_1 = b$. Le gain en temps d'exécution est remarquable parce que sans ce dernier algorithme, on aura fait une inversion, une multiplication, et une réduction.

2.1.2.6. Exponentiation modulaire [MOV96]

L'une des opérations arithmétiques les plus importantes pour le chiffrement à clé publique est l'exponentiation modulaire.

L'algorithme 2.11 calcule la valeur de $g^e \ mod \ m$, à partir des entiers g, e et m.

Algorithme 2.11 : Exponentiation modulaire

Entrées : $Trois\ entiers\ g, e\ et\ m.$
Sorties : $g^e \ mod \ m.$

1. $A \leftarrow 1, S \leftarrow$ g.
2. $Tant\ que\ (e \neq 0)\ faire$:
   2.1. $Si\ e\ est\ impaire\ alors : A \leftarrow A \cdot S \ mod \ m.$
   2.2. $e = \lfloor e/2 \rfloor$.
   2.3. $Si\ e \neq 0\ alors : S \leftarrow S \cdot S \ mod \ m.$
3. $Retourner\ (A).$

2.1.3. Arithmétique des corps binaires

Cette section présente des algorithmes pour réaliser les opérations arithmétiques dans les corps binaires.

Nous supposons que la plate-forme d'implémentation possède une architecture de W-bit où W est un multiple de 8.

La notation suivante est utilisée pour dénoter les opérations logiques sur deux mots U et V:

- U $\oplus$ V ou-exclusif bit à bit.
- U & V et bit à bit.
- U >> i décalage à droite de U par i positions.
- U << i décalage à gauche de U par i positions.

Soit *f(z)* un polynôme binaire irréductible de degré *m*. On écrit $f(z) = z^m + r(z)$.

Les éléments de F_{2^m} sont les polynômes binaires de degré inférieur strictement à *m*.

L'addition des éléments du corps est l'addition usuelle des polynômes réalisée modulo 2.

La multiplication des éléments est réalisée modulo le polynôme irréductible choisi *f(z)*.

Un élément du corps $a(z) = a_{m-1}z^{m-1} + \cdots + a_2z^2 + a_1z + a_0$ est associée au vecteur binaire $a = (a_{m-1}, \ldots, a_2, a_1, a_0)$ de longueur m.

Soit $t = \lceil m/W \rceil$, a peut être stocké dans un tableau A de longueur t contenant des mots de W-bits.

A[t-1]			A[1]	A[0]
$a_{m-1} \ldots a_{(t-1)W}$	…	…	$a_{2W-1} \ldots a_{W+1}a_W$	$a_{W-1} \ldots a_1a_0$

Figure 2.2 : Représentation d'un polynôme binaire à l'aide d'un tableau

2.1.3.1. Addition

L'addition des éléments du corps est réalisée bit à bit, nécessitant seulement t opérations sur les mots du tableau.

Algorithme 2.12 : Addition dans F_{2^m} [WSS03]

Entrées : *deux polynômes binaires* $a(z)$ *et* $b(z)$
de degré inférieur strictement à m.
Sorties : $c(z) = a(z) + b(z)$.

5. *Pour i de* 0 à $(t-1)$ *faire* :
   5.1. $C[i] \leftarrow A[i] \oplus B[i]$.
6. *Retourner* (c).

2.1.3.2. Multiplication

L'algorithme 2.13 pour la multiplication des polynômes est basé sur l'observation suivante : si $b(z).z^k$ a été calculé pour un certain $k \in [0, W[$, alors $b(z).z^{Wj+k}$ peut être facilement obtenue en décalant le vecteur représentant $b(z).z^k$ par j mots à droite.

La notation suivante est utilisée : si $C = (C[n], \ldots, C[2], C[1], C[0])$ est un tableau alors $C\{j\}$ représente le tableau tronqué $(C[n], \ldots, C[\mathrm{j}+1], C[\mathrm{j}])$.

Algorithme 2.13 : Multiplication dans F_{2^m} [LoDa00]

Entrées : *deux polynômes binaires* $a(z)$ *et* $b(z)$
de degré inférieur strictement à m.
Sorties : $c(z) = a(z).b(z)$.

1. $C \leftarrow 0$.
2. *Pour* k *de* 0 à $(W-1)$ *faire* :
   2.1. *Pour* j *de* 0 à $(t-1)$ *faire* :
   Si le k^{eme} *bit de* $A[j]$ *est* 1 *alors additionner* B à $C\{j\}$.
   2.2. *Si* $k \neq (W-1)$*alors* $B \leftarrow B \cdot z$.
3. *Retourner* (c).

2.1.3.3. Carré d'un polynôme

Puisque le calcul du carré d'un polynôme binaire est une opération linéaire, cette opération est beaucoup plus rapide que de multiplier deux polynômes arbitraires, c'est-à-dire :

Si $a(z) = a_{m-1}z^{m-1} + \cdots + a_2z^2 + a_1z + a_0$ alors :

$$a(z)^2 = a_{m-1}z^{2m-2} + \cdots + a_2z^4 + a_1z^2 + a_0.$$

La représentation binaire de $a(z)^2$ est obtenu en insérant le bit 0 entre les bits consécutifs de la représentation binaire de $a(z)$, l'algorithme 2.14 décrit cette procédure pour le paramètre W = 8.

Algorithme 2.14 : Calcul du carré d'un polynôme binaire [LoDa00]

Entrées : *un polynôme binaire* $a(z)$ *de degré inférieur strictement à* m.
Sorties : $c(z) = a(z)^2$.

1. *Pour chaque octet* $d = (d_7, \ldots, d0)$ *calculer les deux octets* :
   $T_1(d) = (0, d_3, 0, d_2, 0, d_1, 0, d_0)$ *et* $T_2(d) = (0, d_7, 0, d_6, 0, d_5, 0, d_4)$
2. *Pour* i *de* 0 à $(t-1)$ *faire* :
   $C[2i] \leftarrow T_1(A[i])$ *et* $C[2i+1] \leftarrow T_2(A[i])$
3. *Retourner* (c).

2.1.3.4. Réduction

Nous allons maintenant discuter des techniques permettant de réduire un polynôme binaire *c(z)* obtenu en multipliant deux polynômes binaires de degré inférieur strictement à *m*, ou en calculant le carré d'un polynôme binaire de degré inférieur strictement à *m*. De tels polynômes *c(z)* ont un degré inférieur strictement à *(2m-1)*.

2.1.3.4.1. Réduction avec des polynômes arbitraires

Rappelons que $f(z) = z^m + r(z)$, où *r (z)* est un polynôme binaires de degré inférieur strictement à *m*.

L'algorithme 2.15 réduit *c (z)* modulo *f (z)* un bit à la fois, en commençant par le bit de gauche. Il est basé sur l'observation suivante :

$$c(z) = c_{2m-2}z^{2m-2} + \cdots + c_m z^m + c_{m-1}z^{m-1} + \cdots + c_1 z + c_0$$
$$\equiv (c_{2m-2}z^{m-2} + \cdots + c_m)r(z) + c_{m-1}z^{m-1} + \cdots + c_0 \ (mod\ f(z)).$$

La réduction est accélérée en pré-calculant les polynômes $z^k r(z),\ 0 \leq k \leq W-1$.

Algorithme 2.15 : Réduction modulaire dans F_{2^m} [HMV04]

Entrées : *un polynôme binaire* $c(z)$ *de degré inférieur strictement* à $2m-1$.
Sorties : $c(z)$ mod $f(z)$.

1. *calculer* $u_k(z) = z^k r(z), 0 \leq k \leq W-1$.
2. *Pour i de* $(2m-2)$ à m *faire* :.
   2.1. *Si* $c_i = 1$ *alors*
   Soit $j = \lfloor (i-m)/W \rfloor$ *et* $k = (i-m) - Wj$.
   Additionner $u_k(z)$ à $C\{j\}$.
3. *Retourner* $(C = (C[t-1], \ldots, C[1], C[0])$.

2.1.3.4.2. Polynômes de réduction suggérés par le NIST [FIPS00]

Les polynômes de réduction suggérés par le NIST sont :

$f(z) = z^{163} + z^7 + z^6 + z^3 + 1$

$f(z) = z^{233} + z^{74} + 1$

$f(z) = z^{283} + z^{12} + z^7 + z^5 + 1$

$f(z) = z^{409} + z^{87} + 1$

$f(z) = z^{571} + z^{10} + z^5 + z^2 + 1$

L'algorithme 2.16 effectue une réduction rapide modulo $f(z) = z^{163} + \cdots$ pour le paramètre W = 8.

Algorithme 2.16 : Réduction rapide modulo $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$

Entrées : *un polynôme binaire* $c(z)$
de degré inférieur strictement à 325.
Sorties : $c(z)$ mod $f(z)$.

1. *Pour* i *de* 41 à 21 *faire* :
   1.1. $T \leftarrow C[i]$.
   1.2. $C[i-21] \leftarrow C[i-21] \oplus (T \ll 5)$.
   1.3. $C[i-20] \leftarrow C[i-20] \oplus (T \ll 4) \oplus (T \ll 3) \oplus T \oplus (T \gg 3)$.
   1.4. $C[i-19] \leftarrow C[i-19] \oplus (T \gg 4) \oplus (T \gg 5)$.
2. $T \leftarrow C[20] \gg 3$.
3. $C[0] \leftarrow C[0] \oplus (T \ll 7) \oplus (T \ll 6) \oplus (T \ll 3) \oplus T$.
4. $C[1] \leftarrow C[1] \oplus (T \gg 1) \oplus (T \gg 2)$.
5. $C[20] \leftarrow C[20]$ & $0x7$.
6. *Retourner* $(C = (C[20], \ldots, C[1], C[0])$.

2.1.3.5. Inversion et division

Dans cette section, nous simplifions la notation et dénotons les polynômes binaires *a(z)* par *a*.

Rappelons que l'inverse d'un élément non nul $a \in F_{2^m}$ est l'unique élément $g \in F_{2^m}$ tel que $ag = 1$ dans F_{2^m}, c'est-à-dire $ag \equiv 1(mod\ f)$. Cet élément inverse est noté a^{-1}.

L'algorithme d'inversion binaire (Algorithme 2.17) calcule l'inverse d'un élément non nul $a \in F_{2^m}$.

Algorithme 2.17 : L'algorithme d'inversion binaire dans F_{2^m} [Ste67]

Entrées : *un polynôme binaire* $c(z)$ *différent de* 0
de degré inférieur strictement à m.

Sorties : $a^{-1}\ mod\ f$.

1. $u \leftarrow a, v \leftarrow f$.
2. $g_1 \leftarrow 1, g_2 \leftarrow 0$.

3. *Tant que* $(u \neq 1$ *et* $v \neq 1)$ *faire* :
   3.1. *Tant que z divise u faire* :
      3.1.1. $u \leftarrow u/z$.
      3.1.2. *Si z divise* g_1 *alors* $g_1 \leftarrow g_1/z$;
      Sinon $g_1 \leftarrow (g_1 + f)/z$.
   3.2. *Tant que z divise v faire* :
      3.2.1. $v \leftarrow v/z$.
      3.2.2. *Si z divise* g_2 *alors* $g_2 \leftarrow g_2/z$;
      Sinon $g_2 \leftarrow (g_2 + f)/z$.
   3.3. *Si* $\deg(u) > \deg(v)$ *alors*: $u \leftarrow u + v, g_1 \leftarrow g_1 + g_2$;
   Sinon: $v \leftarrow v + u, g_2 \leftarrow g_2 + g_1$.
4. *Si* $u = 1$ *alors retourner* (g_1);
   Sinon retourner (g_2).

L'algorithme d'inversion binaire peut être facilement modifié pour effectuer la division $b/a = b.a^{-1}$.

Pour obtenir b/a, l'algorithme 2.17 est modifié à l'étape 2, en remplacement $g_1 \leftarrow 1$ par $g_1 \leftarrow b$.

2.2. Courbes elliptiques

2.2.1. Définition [Kob94]

Une courbe elliptique E sur un corps K est définie par l'équation :

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \qquad (2.1)$$

*O*ù $a_1, a_2, a_3, a_4, a_6 \in K$ *et* $\Delta \neq 0$, Δ est le *discriminant* de E défini de la manière suivante :

$$\left.\begin{aligned} \Delta &= -{d_2}^2 d_8 - 8{d_4}^3 - 27{d_6}^2 + 9d_2d_4d_6 \\ d_2 &= {a_1}^2 + 4a_2 \\ d_4 &= 2a_4 + a_1a_3 \\ d_6 &= {a_3}^2 + 4a_6 \\ d_8 &= {a_1}^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2{a_3}^2 - {a_4}^2 \end{aligned}\right\} \quad (2.2)$$

Si L est un corps d'extension de K, alors l'ensemble des *points L-rationnels* de E est:

$$E(L) = \{(x, y) \in L \times L : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6\} \cup \{\infty\}$$

où ∞ est *le point à l'infini.*

Remarques [Kob94]

- L'équation (2.1) est appelée une *équation de Weierstrass*. Pour un x donné, cette équation a au plus deux solutions y.
- La condition $\Delta \neq 0$ assure que la courbe est "lisse", c'est-à-dire, la courbe ne possède aucun point ayant deux ou plusieurs tangentes.
- Le point ∞ est le seul point de la courbe à l'infini qui satisfait la forme projective de l'équation de Weierstrass.

Exemple [Kob94]

Considérons les courbes elliptiques :

$E_1 : y^2 = x^3 - x$

$E_2 : y^2 = x^3 + \frac{1}{4}x + \frac{5}{4}$

définies sur le corps R des nombres réels. Les points $E_1(R)\backslash\{\infty\}$ et $E_2(R)\backslash\{\infty\}$ sont représentés graphiquement dans la figure ci-dessous.

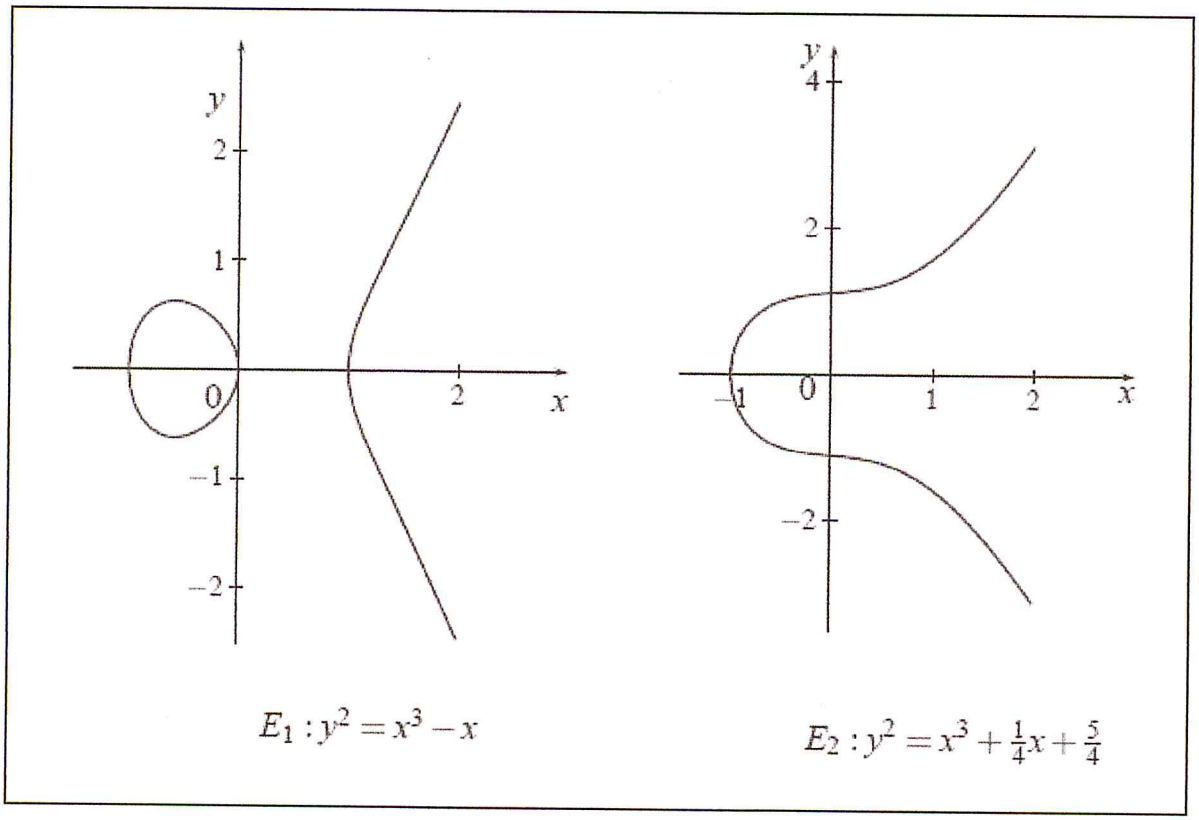


Figure 2.3 : Exemples de courbes elliptiques

2.2.2. Equations simplifiées de Weierstrass [HMV04]

Deux courbes elliptiques E_1 et E_2 définies sur K et données par les équations de Weierstrass :

$E_1 : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$

$E_2 : y^2 + \overline{a_1}xy + \overline{a_3}y = x^3 + \overline{a_2}x^2 + \overline{a_4}x + \overline{a_6}$

sont dites *isomorphes sur K* s'il existe $u, r, s, t \in K$ avec $u \neq 0$ tel que le changement de variables $(x, y) \rightarrow (u^2x + r, u^3y + u^2sx + t)$ transforme l'équation E_1 en l'équation E_2.

Cette transformation est appelée un *changement admissible de variables*.

Une équation de Weierstrass peut être considérablement simplifiée en lui appliquant des changements admissibles de variables.

Soit p la caractéristique de K, nous considérons séparément les trois cas :

- $p \neq 2 \; et \; p \neq 3$
- $p = 2$
- $p = 3$

Cas 1 $(p \neq 2 \; et \; p \neq 3)$

Le changement admissible de variables

$$(x, y) \rightarrow \left(\frac{x - 3{a_1}^2 - 12a_2}{36}, \frac{y - 3a_1x}{216} - \frac{{a_1}^3 + 4a_1a_2 - 12a_3}{24}\right)$$

Transforme E en :

$$y^2 = x^3 + ax + b \quad (2.3)$$

Le discriminant de cette courbe est $\Delta = -16(4a^3 + 27b^2)$.

Cas 2 $(p = 2)$

On considère 2 cas :

- $a_1 \neq 0$
- $a_1 = 0$

Cas 2.1 ($a_1 \neq 0$)

Le changement admissible de variables

$$(x, y) \rightarrow \left({a_1}^2 x + \frac{a_3}{a_1}, {a_1}^3 y + \frac{{a_1}^2 a_4 + {a_3}^2}{{a_1}^3} \right)$$

Transforme E en :

$$y^2 + xy = x^3 + ax^2 + b \quad (2.4)$$

Le discriminant de cette courbe est $\Delta = b$.

Une telle courbe est dite *non-supersingulière*.

Cas 2.2 ($a_1 = 0$)

Le changement admissible de variables

$$(x, y) \rightarrow (x + a_2, y)$$

Transforme E en :

$$y^2 + cy = x^3 + ax + b \quad (2.5)$$

Le discriminant de cette courbe est $\Delta = c^4$.

Une telle courbe est dite *supersingulière*.

Cas 3 ($p = 3$)

On considère 2 cas :

- ${a_1}^2 \neq -a_2$
- ${a_1}^2 = -a_2$

Cas 3.1 (${a_1}^2 \neq -a_2$)

Le changement admissible de variables

$$(x, y) \rightarrow \left(x + \frac{d_4}{d_2}, y + a_1 x + a_1 \frac{d_4}{d_2} + a_3 \right)$$
$$\text{où } d_2 = {a_1}^2 + a_2 \text{ et } d_4 = a_4 - a_1 a_3$$

Transforme E en :

$$y^2 = x^3 + ax^2 + b \quad (2.6)$$

Le discriminant de cette courbe est $\Delta = -a^3 b$.

Une telle courbe est dite non-supersingulière.

Cas 3.2 $(a_1{}^2 = -a_2)$

Le changement admissible de variables

$$(x, y) \rightarrow (x, y + a_1 x + a_3)$$

Transforme E en :

$$y^2 = x^3 + ax + b \quad (2.7)$$

Le discriminant de cette courbe est $\Delta = -a^3$.

Une telle courbe est dite supersingulière.

2.2.3. Loi de groupe [CoFr06]

Soit E une courbe elliptique définie sur K. Il existe une règle *corde-et-tangente* pour additionner deux points de $E(K)$ afin d'avoir un troisième point appartenant lui aussi à $E(K)$.

Avec cette addition l'ensemble de points $E(K)$ forme un groupe abélien avec ∞ servant d'élément neutre.

L'addition de deux points est mieux expliquée géométriquement. Soit $P = (x_1, y_1)$ et $Q = (x_2, y_2)$ deux points distincts d'une courbe elliptique E, la somme R de P et Q est définie de la manière suivante :

- On commence par dessiner une ligne passant par les points P et Q.
- Cette ligne intersecte la courbe elliptique en un troisième point.
- R est la réflexion de ce point par rapport à l'axe des x.

Le double R de P est défini de la manière suivante :

- On commence par dessiner la tangente au point P.
- Cette ligne intersecte alors la courbe en un deuxième point.
- R est la réflexion de ce point par rapport à l'axe des x.

Ceci est expliqué dans la figure ci-dessous :

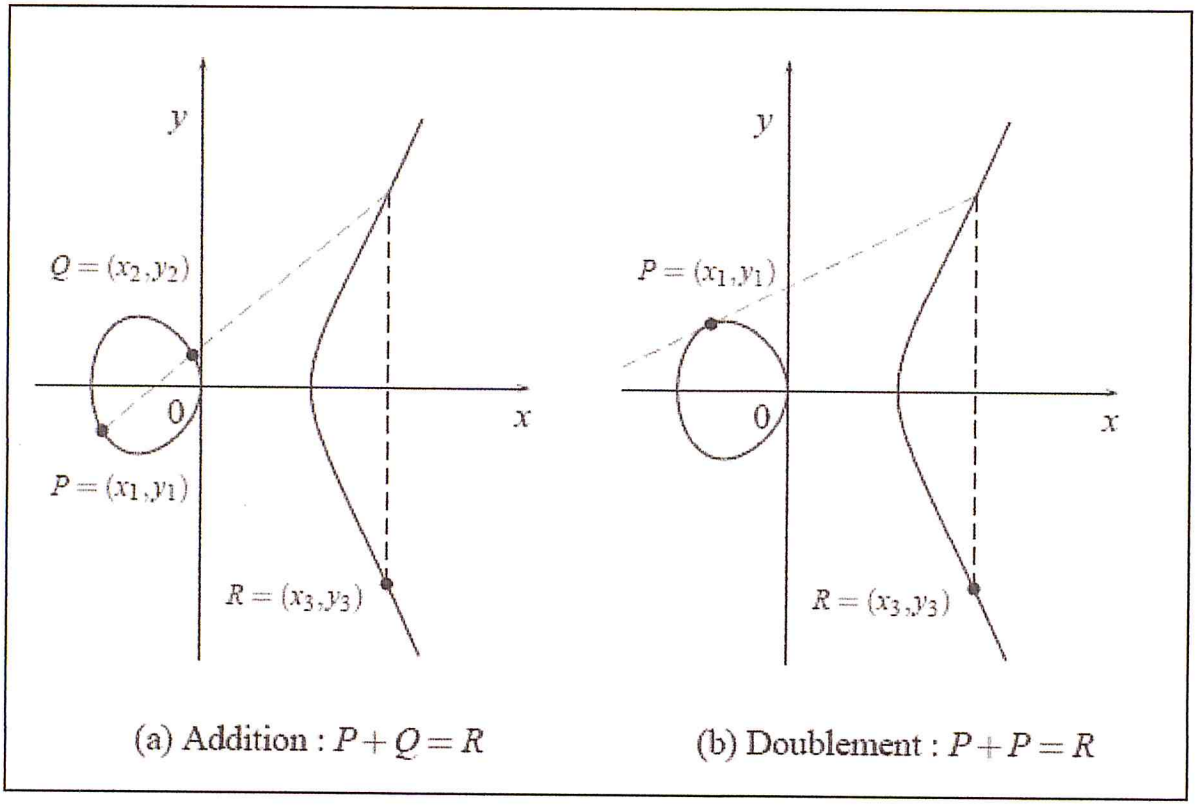


Figure 2.4 : Addition et doublement de points

Les formules algébriques peuvent être tirées de la description géométrique.

Nous présentons ci-dessous les formules correspondantes à la courbe non-supersingulière, définie sur le corps F_{2^m}. Cette courbe est définie par l'équation suivante : $y^2 + xy = x^3 + ax^2 + b \quad (car(K) = 2, a_1 \neq 0)$

Loi de groupe pour $E\backslash F_{2^m}$: $y^2 + xy = x^3 + ax^2 + b$

---

1. Identité. $\forall\, P \in E(F_{2^m}) : P + \infty = P.$
2. Négation. $Si\ P = (x, y) \in E(F_{2^m}), alors\ (x, y) + (x, x + y) = \infty.$
   Le point $(x, x + y)$ est noté $-P$ et est appelé *l'opposé* de P.
   Il faut noter que $-P \in E(K)$.
3. Addition. $Soit\ P = (x_1, y_1), Q = (x_2, y_2) \in E(F_{2^m})$ où $P \neq \pm Q.$
   $P + Q = (x_3, y_3),$ où :
   $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a\ et\ y_3 = \lambda(x_1 + x_3) + x_3 + y_1$
   $avec : \lambda = (y_1 + y_2)/(x_1 + x_2).$
4. Doublement. $Soit\ P = (x_1, y_1) \in E(F_{2^m})$ où $P \neq -P, 2P = (x_3, y_3)$
   $$où : x_3 = \lambda^2 + \lambda + a = {x_1}^2 + \frac{b}{{x_1}^2}\ et\ y_3 = {x_1}^2 + \lambda x_3 + x_3$$
   $$avec : \lambda = x_1 + y_1/x_1$$

2.2.4. Le cardinal du groupe

Soit E une courbe elliptique définie sur F_q. Le nombre de points de $E(F_q)$, notée $\#E(F_q)$, est appelé *le cardinal de E sur* F_q. [HMV04]

Le Théorème de Hasse fournit des bornes pour $\#E(F_q)$.

Théorème de Hasse [Sil94]

Soit E une courbe elliptique définie sur F_q, alors :

$$q + 1 - 2\sqrt{q} \leq \#E(F_q) \leq q + 1 + 2\sqrt{q}.$$

Une autre formulation du théorème de Hasse est la suivante :

Soit E une courbe elliptique définie sur F_q, alors :

$$\#E(F_q) = q + 1 - t \quad \text{où} \quad |t| \leq 2\sqrt{q}\,.$$

- t est appelé *la trace de E sur* F_q.

Définition [HMV04]

Soit p la caractéristique de F_q. Une courbe elliptique E définie sur $F_q$est *supersingulière* si p divise t, où t est la trace. Si p ne divise pas t, E est *non-supersingulière*.

Théorème de Weil [Wei49]

Soit E une courbe elliptique définie sur F_q, et $\#E(F_q) = q + 1 - t$. Alors :

$\#E(F_{q^n}) = q^n + 1 - V_n \quad \forall\, n \geq 2$, où V_n est la séquence définie récursivement par : $V_0 = 2, V_1 = 1\ et\ \forall\, n \geq 2, V_n = V_1 V_{n-1} - qV_{n-2}$.

2.2.5. Addition et doublement de points [HMV04]

Nous présentons dans cette section les algorithmes d'addition et doublement pour la courbe non-supersingulière, définie sur le corps F_{2^m}. Cette courbe est définie par l'équation suivante :

$$y^2 + xy = x^3 + ax^2 + b \quad (car(K) = 2, a_1 \neq 0)$$

Algorithme 2.18 : Addition de deux points

Entrées : $P = (x_1, y_1), Q = (x_2, y_2) \in E(F_{2^m})$ où $P \neq \pm Q$.
Sorties : $P + Q = (x_3, y_3)$.

1. $\lambda = \dfrac{(y_1 + y_2)}{(x_1 + x_2)}$.
2. $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$.
3. $y_3 = \lambda(x_1 + x_3) + x_3 + y_1$.
4. $Retourner\ (x_3, y_3)$.

Algorithme 2.19 : Doublement d'un point

Entrées : $P = (x_1, y_1) \in E(F_{2^m})$ où $P \neq -P$.
Sorties : $2P = (x_3, y_3)$.

1. $\lambda = x_1 + y_1/x_1$.
2. $x_3 = \lambda^2 + \lambda + a$.
3. $y_3 = {x_1}^2 + \lambda x_3 + x_3$.
4. $Retourner\ (x_3, y_3)$.

2.2.6. Multiplication de points [CoFr06]

La multiplication de points est une opération qui consiste à additionner un point à lui-même répétitivement.

Soient E une courbe elliptique, P un point de E, et k un entier positif. La multiplication de k par P, est définie comme suit :

- Si $k = 0$ alors $kP = \infty$.
- Si $k > 0$ alors $kP = \underbrace{P + P + \cdots + P}_{k\ fois}$.

Nous allons présenter une méthode permettant de calculer kP, cette méthode est appelée *la méthode binaire pour la multiplication de points* et est présentée dans l'algorithme (2.20).

Algorithme 2.20 : Méthode binaire pour la multiplication de points

Entrées : $k = (k_{t-1}, \dots, k_1, k_0)_2, P \in E(F_q)$.

Sorties : kP.

4. $Q \leftarrow \infty$.
5. $Pour\ i\ de\ 0$ à $(t-1)\ faire$:
   5.1. $Si\ k_i = 1\ alors\ Q \leftarrow Q + P$.
   5.2. $P \leftarrow 2P$.
6. $Retourner\ (Q)$.

Le présent chapitre avait pour objet d'étudier la théorie ainsi que l'arithmétique des corps finis et des courbes elliptiques. Les algorithmes présentés dans ce but vont permettre de réaliser les cryptosystèmes RSA et ECDSA qui sont présentés dans le chapitre suivant.

Chapitre 3

Description des algorithmes RSA et ECDSA

Dans ce chapitre, nous allons décrire les algorithmes RSA et ECDSA. La première partie est consacrée à RSA, le problème de factorisation, la génération de clés ainsi que les différentes opérations telles que le chiffrement et la signature sont discutées dans cette partie. La deuxième partie concerne l'algorithme ECDSA, le problème du logarithme discret, la génération des paramètres de domaines et des clés ainsi que les opérations de signature et de vérification sont discutés tout au long de cette partie.

3.1. Cryptosystème RSA

En 1977 le premier algorithme à clé publique apparut. Il fonctionne pour le chiffrement aussi bien que pour la signature numérique. Baptisé d'après les noms de ses inventeurs, Rivest, Shamir, et Adleman. La sécurité de RSA repose sur la difficulté de factoriser des grands nombres. [Sch94]

Dans cette section nous allons décrire *le problème de factorisation*, sur lequel repose la sécurité de l'algorithme RSA. Nous présentons ensuite les algorithmes relatifs à la génération des clés, le chiffrement et la signature numérique avec RSA. La réalisation de RSA nécessite de générer des grands nombres premiers, les algorithmes utilisés dans ce but son discutés à la fin de cette section.

3.1.1. Le problème de factorisation des entiers

Etant donné un entier N, écrire N sous forme d'un produit de nombres premiers. Par exemple si $N = 15$ alors la factorisation de N consiste à trouver les deux nombres premiers 3 et 5.

Si N est très grand ($N > 2^{1024}$), alors il devient impossible de factoriser N en un temps raisonnable. [PKCS02]

3.1.2. Génération des clés

Une paire de clés RSA peut être générée en utilisant l'algorithme 3.1. La clé privée est le couple (n, d) et la clé publique est le couple (n, e).

Pour retrouver la clé privée (n, d) depuis la clé publique (n, e), il faut factoriser le nombre n. La taille de n est égale au niveau de sécurité l.

Algorithme 3.1 : Génération d'une paire de clés RSA [RSA78]

Entrées : Le niveau de sécurité l.
Sorties : La clé publique (n, e) et la clé privée (n, d).

7. Choisissez deux nombres premiers, p et q de longueur $(l/2)$ $bits$.
8. Calculez $n = pq$ et $\varphi = (p-1)(q-1)$.
9. Choisissez un entier arbitraire e tel que $1 < e < \varphi$ et $pgcd(e, \varphi) = 1$.
10. Calculer l'entier d satisfaisant $1 < d < \varphi$ et $ed \equiv 1\ (mod\ \varphi)$.
11. $Retourner\ (n, e, d)$.

3.1.3. Chiffrement

Pour chiffrer un message M à l'aide de RSA, il faut transformez le message en un entier (ou plusieurs entiers) $m \in [0, n-1]$. m est par la suite chiffré en utilisant la formule : $C \equiv M^e\ mod\ n$. Pour retrouver le texte en clair utilisez la formule suivante : $M \equiv C^d\ mod\ n$.

Les algorithmes 3.2 et 3.3 permettent de réaliser le chiffrement et le déchiffrement en utilisant l'algorithme RSA.

Algorithme 3.2 : Chiffrement RSA [RSA78]

Entrées : La clé publique (n, e) et le message $m \in [0, n-1]$.
Sorties : Le texte chiffré c.

1. Calculez $c = m^e\ mod\ n$.
2. $Retourner\ (c)$.

Afin d'accélérer l'opération de chiffrement, on choisi e de taille petite (Les choix les plus courants sont 3, 17 et 65537). Aucun problème de sécurité n'interdit de prendre l'une de ces trois valeurs pour e.

Algorithme 3.3 : Déchiffrement RSA [RSA78]

Entrées : La clé privée (n, d) et le texte chiffré $c \in [0, n-1]$.
Sorties : Le message en clair m.

1. Calculez $m = c^d \ mod \ n$.
2. $Retourner\ (m)$.

3.1.4. Signature

Pour signer un message m à l'aide de RSA, il faut calculer son empreinte $h = H(m)$ à l'aide d'une fonction de hachage h. La signature s est par la suite calculée en utilisant la formule : $s \equiv h^d \ mod \ n$. Pour vérifier la signature, utilisez la formule suivante : $h' \equiv s^e \ mod \ n$. Si $h = h'$ alors la signature est juste ; sinon elle est fausse.

Les algorithmes 3.4 et 3.5 permettent de réaliser signature numérique et la vérification de signature en utilisant l'algorithme RSA.

Algorithme 3.4 : Signature RSA [RSA78]

Entrées : La clé privée (n, d) et le message $m \in [0, n-1]$.
Sorties : La signature s.

1. Calculez $h = H(m)$ où H est une fonction de hachage.
2. Calculez $s = h^d \ mod \ n$.
3. $Retourner\ (s)$.

Algorithme 3.5 : Vérification de signature RSA [RSA78]

Entrées : La clé publique (n, e), le message m et la signature s.
Sorties : Accepter ou rejeter la signature.

1. Calculez $h = H(m)$.
2. Calculez $h' = s^e \ mod \ n$.
3. Si $h = h'$ alors retourner ("$Signature\ acceptée$") ;
   Sinon retourner ("$Signature\ rejetée$").

La vérification de signature peut être accélérée considérablement si la taille de e est petite.

3.1.5. Génération de nombres premiers

Afin de générer des nombres premiers, on commence par générer des nombres aléatoires, on leur applique par la suite un test de primalité pour savoir si ils sont premiers ou pas.

a. Générateur de nombres pseudo-aléatoires [Gen03]

Nous avons choisi d'utiliser un *générateur congruentiel linéaire* défini récursivement comme suit :

$$X_{n+1} = (a \cdot X_n + c)\ mod\ m.$$

- m est appelé *le modulo*.
- a est appelé *le multiplicateur*.
- c est appelé *l'incrément*.
- X_0 est appelé *la graine*.

La période maximale d'un générateur congruentiel linéaire est égale à m.

Le générateur aura une période égale à m quelque soit la valeur de la graine si et seulement si :

- c et m sont premiers entre eux.
- Pour chaque nombre premier p divisant m, $(a-1)$ est un multiple de p.
- $(a-1)$ est un multiple de 4 si m en est un.

b. Test de primalité de Fermat [CLRS01]

Le test de primalité de Fermat est un test probabiliste pour déterminer si un nombre est probablement premier.

Ce teste repose sur *le petit théorème de Fermat* défini comme suit :

- Si p est un nombre premier et $1 \leq a < p$ alors : $a^{p-1} \equiv 1\ mod\ p$.

Si nous voulons tester si p est premier, alors on choisi a au hasard dans l'intervalle $[0, p-1[$ et on vérifie l'égalité précédente.

Si l'égalité n'est pas vérifiée, alors p est composite.

Si l'égalité est vérifiée, alors nous pouvons dire que p est probablement premier.

- Si l'égalité est vérifiée pour 100 valeurs de a choisies au hasard, alors p est certainement premier.

3.2. Cryptosystème ECDSA

Dans cette section nous discutons *le problème du logarithme discret*, sur lequel repose la sécurité de tous les cryptosystèmes basés sur les courbes elliptiques. Nous présentons ensuite les algorithmes relatifs à la génération des *paramètres de domaine* et des *paires de clés* qui seront utilisés dans des protocoles basés sur les courbes elliptiques. Le reste de la section est réservé au système de signature numérique *ECDSA(Elliptic Curve Digital Signature Algorithm)*.

3.2.1. Le problème du logarithme discret (DLP) [Sil06]

Soit G un groupe, $g \in G$. Le problème du logarithme discret pour G est : Soit $h \in \langle g \rangle$, trouver l'entier m tel que $h = g^m$.

- Le problème du logarithme discret est utilisé dans de nombreux systèmes cryptographiques, tel que l'échange de clés, chiffrement, signatures numériques, et les fonctions de hachage.

Difficulté du problème

Pour certains groupes, le DLP est très facile :

- Z/mZ sous l'addition.
- R^* et C^* sous la multiplication.

Pour certains groupes, le DLP est difficile. L'exemple classique est:

- ${F_p}^*$sous la multiplication.

Le meilleur algorithme connu pour résoudre le DLP dans ${F_p}^*$ prend un temps sous-exponentielle.

Pour des besoins cryptographiques, il serait préférable d'utiliser un groupe G dont la résolution du DLP prend un temps exponentiel.

3.2.1.1. Le problème du logarithme discret sur les courbes elliptiques (ECDLP)

Soit E une courbe elliptique définie sur F_q, un point $P \in E(F_q)$ d'ordre n, et un point $Q \in \langle P \rangle$.

Trouver l'entier $l \in [0, n-1]$ tel que $Q = lP$.

L'entier l est appelé *le logarithme discret* de Q à la base P, et il est noté $l = log_P Q$.

- La meilleure attaque connu contre le ECDLP est une combinaison de l'algorithme Pohlig-Hellman et l'algorithme de Pollard, cette attaque s'exécute en un temps entièrement exponentielle égal à $O(\sqrt{p})$ où p est le plus grand nombre premier qui divise n.
- Pour résister à cette attaque, les paramètres de la courbe elliptique doit être choisis de telle sorte que n soit divisible par un nombre premier p suffisamment grand pour que $\sqrt{p}$ devient un nombre très grand et ainsi exécuter $\sqrt{p}$ étapes soit impossible ($exp : p > 2^{160}$).

3.2.2. Paramètres de domaines

Les paramètres de domaine pour un système basé sur les courbes elliptiques décrivent :

- Une courbe elliptique E définie sur un corps fini F_q.
- Un point de base $P \in E(F_q)$, et son ordre n.

Les paramètres doivent être choisis de sorte que le ECDLP résiste à toutes les attaques connues.

3.2.2.1 Définition [HMV04]

Les paramètres de domaine $D = (q, FR, S, a, b, P, n, h)$ sont composés de :

1. Le cardinal du corps q.
2. Une indication FR (*représentation du corps*) sur la représentation utilisée pour les éléments de F_q.
3. Une graine S si la courbe elliptique a été générée aléatoirement.
4. Deux coefficients $a, b \in F_q$ qui définissent l'équation de la courbe elliptique E sur F_q (i.e., $y^2 = x^3 + ax + b$ dans le cas d'un corps premier et $y^2 + xy = x^3 + ax^2 + b$ dans le cas d'un corps binaire).
5. Deux éléments du corps x_P et y_P dans F_q qui définissent un point fini $P = (x_P, y_P) \in E(F_q)$ en coordonnées affines. P a un ordre premier et est appelé *le point de base*.
6. L'ordre n de P.
7. Le cofacteur $h = \#E(F_q)/n$.

Contraintes de sécurité

Afin de résister aux attaques connues contre le ECDLP les conditions suivantes doivent être satisfaites :

- $\#E(F_q)$ est divisible par un nombre premier n suffisamment grand. Au minimum, on devrait avoir $n > 2^{160}$.
- $\#E(F_q)$ est *presque premier*, c'est-à-dire, $\#E(F_q) = hn$ où n est un nombre premier et h est petit (par exemple, $h = 1, 2, 3\ ou\ 4$).
- $\#E(F_q) \neq q$.
- n ne divise pas $q^k - 1$ pour tout $1 \leq k \leq C$, où C est assez grand pour que le DLP dans $F_{q^C}^*$ soit insoluble (si $n > 2^{160}$, alors $C = 20$ suffit).
- Si $F_q = F_{2^m}$ alors m doit être premier.

3.2.2.2. Génération des paramètres de domaine

L'algorithme 3.6 est un moyen de générer des paramètres de domaine cryptographiquement sûr, toutes les contraintes de sécurité mentionnées ci-dessus sont remplies.

Algorithme 3.6 : Génération des paramètres de domaine [HMV04]

Entrées : Le cardinal du corps q, une représentation du corps FR, Le niveau de sécurité L satisfaisant $160 \leq L \leq \lfloor log_2 q \rfloor$ et $2^L \geq 4\sqrt{q}$.

Sorties : Les paramètres de domaine $D = (q, FR, S, a, b, P, n, h)$.

5. Sélectionnez $a, b \in F_q$ *vérifiables au hasard* en utilisant les algorithmes 3.7, 3.8 si F_q est un corps premier ou un corps binaire, respectivement.
6. Calculez $N = \#E(F_q)$.
7. Vérifiez que $\#E(F_q)$ est divisible par un grand nombre premier n satisfaisant $n > 2^L$. Sinon allez à l'étape 1.
8. Vérifiez que n ne divise pas $q^k - 1$ pour tout $1 \leq k \leq 20$. Sinon allez à l'étape 1.
9. Vérifiez que $n \neq q$. Sinon allez à l'étape 1.
10. $h \leftarrow N/n$.
11. Sélectionnez un point arbitraire $P' \in E(F_q)$ et fixer $P = hP'$. Répétez jusqu'à ce que $P \neq \infty$.
12. Retourner $(q, FR, S, a, b, P, n, h)$.

3.2.2.3. Génération des courbes elliptiques vérifiables au hasard

Les algorithmes 3.7 et 3.8 sont des spécifications pour générer des courbes elliptiques vérifiables au hasard sur les corps premiers et les corps binaires, respectivement. Les algorithmes sont tirés du standard $X9.62$ de la *ANSI(American National Standards Institute)*.

Algorithme 3.7 : Génération d'une courbe elliptique aléatoire sur un corps premier F_p [ANSI99]

Entrées : Un nombre premier $p > 3$, une fonction de hachage H de l $bits$.
Sorties : Une graine S, et $a, b \in F_p$ définissant la courbe elliptique
$E : y^2 = x^3 + ax + b$.

1. Fixez $t \leftarrow \lceil log_2 p \rceil$, $s \leftarrow \lfloor (t-1)/l \rfloor$, $v \leftarrow t - sl$.
2. Sélectionnez une chaîne de bits arbitraire S de longueur $g \geq l$ $bits$.
3. Calculez $h = H(S)$, et soit r_0 la chaîne de bits de longueur v obtenue en prenant les v bits à droite de h.
4. Soit R_0 la chaîne de bits obtenue en mettant le bit le plus à gauche de r_0 à 0.
5. Soit z l'entier dont la représentation binaire est S.
6. *Pour i de 1 à s faire :*
   6.1. Soit s_i la représentation binaire de l'entier $(z + i) \ mod \ 2^g$.
   6.2. Calculez $R_i = H(s_i)$.
7. Soit $R = R_0 \ || \ R_1 \ || \ \dots \ || \ R_s$.
8. Soit r l'entier dont la représentation binaire est R.
9. Si $r = 0$ ou si $4r + 27 \equiv 0 \ (mod \ p)$, alors allez à l'étape 2.
10. Sélectionnez arbitrairement $, b \in F_p$, pas tous les deux nuls, tel que $r \cdot b^2 \equiv a^3 \ (mod \ p)$.
11. Retourner (S, a, b).

Algorithme 3.8 : Génération d'une courbe elliptique aléatoire sur un corps binaire F_{2^m} [ANSI99]

Entrées : Un entier positif m, une fonction de hachage H de l $bits$.
Sorties : Une graine S, et $a, b \in F_{2^m}$ définissant la courbe elliptique
$E : y^2 + xy = x^3 + ax^2 + b$.

1. Fixez $s \leftarrow \lfloor (m-1)/l \rfloor$, $v \leftarrow m - sl$.
2. Sélectionnez une chaîne de bits arbitraire S de longueur $g \geq l$ $bits$.
3. Calculez $h = H(S)$, et soit b_0 la chaîne de bits de longueur v obtenue en prenant les v bits à droite de h.
4. Soit z l'entier dont la représentation binaire est S.
5. *Pour i de 1 à s faire :*
   5.1. Soit s_i la représentation binaire de l'entier $(z + i) \ mod \ 2^g$.
   5.2. Calculez $b_i = H(s_i)$.
6. Soit $b = b_0 \ || \ b_1 \ || \ \dots \ || \ b_s$.

7. Si $= 0$, alors allez à l'étape 2.
8. Sélectionnez arbitrairement $\in F_{2^m}$.
9. Retourner (S, a, b).

3.2.2.4. Déterminer le nombre de points sur une courbe elliptique

Comme indiqué dans la section 3.2.2.1 (contraintes de sécurité), le cardinal $\#E(F_q)$ d'une courbe elliptique E utilisée dans un protocole cryptographique doit satisfaire certaines contraintes imposées par des considérations de sécurité. Ainsi, la détermination du nombre de points sur une courbe elliptique est un ingrédient important de la génération des paramètres de domaine.

Un algorithme naïf de comptage de points consiste à trouver, pour chaque $x \in F_q$, le nombre de solutions $y \in F_q$ satisfaisant l'équation de Weierstrass correspondante à E. Cette méthode est clairement impossible pour les corps d'intérêt cryptographique.

En pratique, l'une des trois techniques suivantes est utilisée pour déterminer le nombre de points sur une courbe elliptique :

Méthode 1 [CoFr06]

Soit $q = p^{ld}$, où $d > 1$.

On sélectionne une courbe elliptique E définie sur F_{p^l}, on compte le nombre de points de $E(F_{p^l})$ en utilisant une méthode naïve, puis on détermine facilement $\#E(F_q)$ en utilisant le théorème de Weil décrit dans la section 3.1.3.

Le groupe utilisé pour l'application cryptographique est $E(F_q)$.

Méthode 2 [AtMo93]

Dans cette méthode, on sélectionne d'abord un cardinal N qui répond aux contraintes de sécurité nécessaires, puis on construit une courbe elliptique avec ce cardinal N.

Afin de construire la courbe elliptique on utilise une méthode appelée *la méthode de multiplication complexe (CM).*

Méthode 3 [Sat00], [Sch85]

Cette méthode consiste à compter les points de la courbe elliptique.

En 1985, *Schoof* a présenté le premier algorithme polynomial pour le calcul de $\#E(F_q)$ pour une courbe elliptique arbitraire E.

L'algorithme calcule $\#E(F_q)\ mod\ l$ pour des petits nombres premiers l, puis détermine $\#E(F_q)$ en utilisant le théorème des restes chinois.

Il est inefficace pour des valeurs de q d'intérêt pratique, mais a ensuite été amélioré par plusieurs personnes, y compris *Atkin* et *Elkies* résultant *l'algorithme de Schoof-Elkies-Atkin (SEA)*.

L'algorithme SEA, qui est le meilleur algorithme connu pour le comptage des points sur des courbes elliptiques arbitraires sur des corps premiers, prend quelques minutes pour des valeurs de q d'intérêt pratique.

En 1999, *Satoh* a proposé une méthode radicalement nouvelle pour compter le nombre de points sur un corps fini de caractéristique petite.

Des variantes de la méthode de Satoh, y compris l'algorithme *Satoh-Skjernaa-Taguchi (SST)*, sont extrêmement rapides pour le cas des corps binaires et peuvent compter le nombre de points d'une courbes elliptique sur $F_{2^{163}}$ en quelques secondes.

3.2.3. Génération des clés

Une paire de clés de la courbe elliptique est associé à un ensemble particulier de paramètres de domaine $D = (q, FR, S, a, b, P, n, h)$.

La clé publique est un point choisi au hasard dans le groupe $\langle P \rangle$ générés par P.

La clé privée correspondante est $d = log_p Q$.

Algorithme 3.9 : Génération de clés ECC [ANSI99]

Entrées : Les paramètres de domaine $D = (q, FR, S, a, b, P, n, h)$.

Sorties : La clé publique Q, et la clé privée d.

1. Sélectionnez $d \in [1, n-1]$.
2. Calculez $Q = dP$.
3. Retourner (Q, d).

Observez que le problème de retrouver la clé privée d depuis la clé publique Q est précisément le problème du logarithme discret sur les courbes elliptiques.

3.2.4. Elliptic Curve Digital Signature Algorithm (ECDSA)

L'*ECDSA* est l'analogue de l'algorithme de signature numérique *DSA* apparu dans le standard *FIPS 186*. C'est le schéma de signature basé sur les courbes elliptiques le plus largement standardisées, figurant dans les standards *ANSI X9.62*, *FIPS 186-2*, *IEEE 1363-2000* et *ISO / IEC 15946-2*, ainsi que plusieurs brouillons de standards.

Dans ce qui suit, H désigne une fonction de hachage de $l\,bits$ tel que $l \leq n$ (si cette condition n'est pas remplie, alors les sorties de H peuvent être tronquées).

Algorithme 3.10 : ECDSA Génération de signature [ANSI99]

Entrées : Les paramètres de domaine $D = (q, FR, S, a, b, P, n, h)$, La clé privée d, et le message m.
Sorties : La signature (r, s).

1. Sélectionnez $k \in [1, n-1]$.
2. Calculez $kP = (x_1, y_1)$ et transformer x_1 en un entier $\overline{x_1}$.
3. Calculez $r = \overline{x_1} \bmod n$. Si $r = 0$ alors : allez à l'étape 1.
4. Calculez $e = H(m)$.
5. Calculez $s = k^{-1}(e + dr) \bmod n$. Si $s = 0$ alors : allez à l'étape 1.
6. Retourner (r, s).

Algorithme 3.11 : ECDSA Vérification de signature [ANSI99]

Entrées : Les paramètres de domaine $D = (q, FR, S, a, b, P, n, h)$, La clé publique Q, le message m, et La signature (r, s).
Sorties : Accepter ou rejeter la signature.

1. Vérifier que r et s sont des entiers dans l'intervalle $[1, n-1]$. Si ce n'est pas le cas, alors Retourner ("*Signature rejetée*").
2. Calculez $e = H(m)$.
3. Calculez $w = s^{-1} \bmod n$.
4. Calculez $u_1 = ew \bmod n$ et $u_2 = rw \bmod n$.
5. Calculez $X = u_1 P + u_2 Q$.
6. Si $X = \infty$ alors Retourner ("*Signature rejetée*").
7. Transformer la coordonnée-x de X, x_1 en un entier $\overline{x_1}$; Calculez $v = \overline{x_1} \bmod n$.
8. Si $v = r$, alors Retourner ("*Signature acceptée*") ;
   Sinon, Retourner ("*Signature rejetée*").

Apres la description des algorithmes RSA et ECDSA, nous entamons le chapitre "Cartes à puce et Java Cards". Ce chapitre consiste à introduire les cartes à puces et à présenter les cartes à puces Java Cards.

Chapitre 4

Cartes à puce et Java Cards

Ce chapitre est divisé en deux parties. Dans la première, nous introduisons les différents concepts du monde des cartes à puce. La deuxième partie est consacrée aux cartes à puce Java Cards. Nous commençons par présenter les caractéristiques matériels et logiciels de ces cartes, ensuite nous montrons comment créer des applications à l'aide d'un simulateur de cartes à puce Java Card.

4.1. Introduction aux cartes à puce

Aujourd'hui, les cartes à puce sont utilisées dans de nombreux domaines dans la vie quotidienne. Une carte à puce peut être une carte de téléphone, une carte portant des renseignements sur l'assurance ou la santé d'une personne, un portefeuille électronique, … etc.[HNSF00]

Le succès des cartes à puce a commencé en Europe dans les années 80, lorsque Carte Bancaire (Groupe français de cartes bancaires), a mis sur le marché sa première carte à puce. En collaboration avec Bull, Philips et Schlumberger, Carte Bancaire a lancé des expérimentations dans les villes françaises de Blois, Caen et Lyon. Les essais ont été un grand succès, et suite à ces essais, les banques françaises ont lancé l'utilisation de cartes à puce pour les services bancaires.[HNSF00]

En Algérie, le moyen le plus populaire d'utiliser des cartes à puce est dans la téléphonie mobile (Les cartes SIM). *Algérie Poste* utilise aussi des cartes à puces pour permettre à ses clients de faire des retraits d'espèces, ou des déclarations d'avoir, sans avoir recours au vieux chèque. Il faut aussi citer la carte *Chifa*, une carte de sécurité sociale qui permet d'identifier un assuré. Cette carte sert en premier lieu à rembourser sans avoir à formuler une demande ni à remplir et présenter une feuille de soins.

4.1.1 Qu'est-ce qu'une carte à puce ? [Ran07]

Une carte à puce est une carte en plastique portant un circuit intégré qui peut stocker des données et exécuter des commandes.

4.1.2 Types des cartes à puce ? [Ran07]

Il existe deux types de base de cartes à puce:

- *Une carte à puce intelligente* qui contient un microprocesseur et une mémoire. Cette carte permet de lire, écrire et faire des calculs, comme un micro ordinateur.
- *Une carte mémoire*, qui ne dispose pas d'un microprocesseur et est destiné uniquement pour le stockage de l'information.

Toutes les cartes à puce intelligentes contiennent trois types de mémoire :

- *ROM (Read Only Memory)*
  - Les informations stockées dans ce type de mémoire sont écrites lors de la production et ne peuvent jamais être modifiées ou effacées. Elle contient le système d'exploitation de la carte et peut également contenir certaines applications.
- *EEPROM (Electrical Erasable Programmable Read Only Memory)*
  - Utilisée pour le stockage permanent des données. Même si la carte à puce n'est pas alimentée, l'EEPROM conserve les données. Contrairement à la ROM, les données contenues dans cette mémoire peuvent être effacées ou modifiées.
- *RAM (Random Access Memory)*
  - La RAM est la mémoire temporaire de la carte, elle conserve les données aussi longtemps que la carte est alimentée en courant.

Le schéma suivant illustre les caractéristiques physiques générales d'une carte à puce, définis dans la norme *ISO 7816, partie 1*.

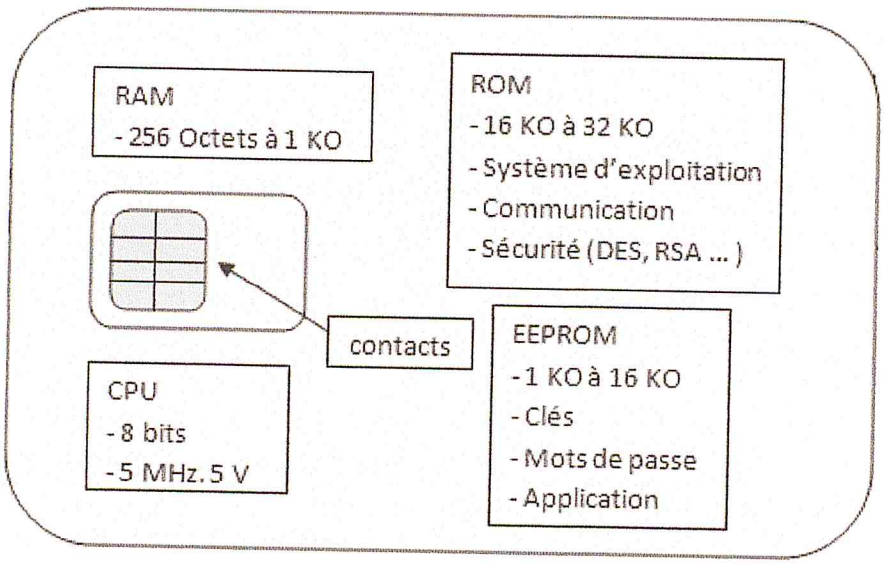


Figure 4.1 : Exemple d'une carte à puce

4.1.3 Contacts d'une cartes à puce [RaEf97]

Généralement, une carte à puce ne contient pas une source de courant, un écran ou un clavier. Elle interagit avec le monde extérieur en utilisant une interface de communication par l'intermédiaire de ses huit points de contact. La figure suivante montre les contacts sur une carte à puce.

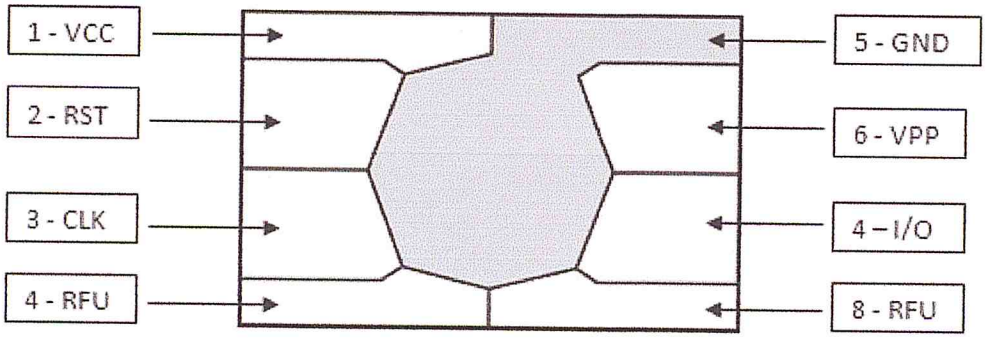


Figure 4.2 : Les 8 contacts d'une carte à puce

1. *VCC* : Correspond à la tension d'alimentation de la carte, et est généralement de 5 volts.
2. *RST* : C'est l'abréviation de *reset*, ce contact est utilisé pour réinitialiser le microprocesseur de la carte à puce.
3. *CLK* : C'est l'abréviation de *clock*, ce contact est utilisé pour fournir un signal d'horloge à la carte.
4. *RFU* : C'est l'abréviation de *reserved for future use*, c'est-à-dire réservé pour une utilisation future.
5. *GND* : C'est l'abréviation de *ground*, ce contact correspond à la terre électrique de la carte.
6. *VPP* : Correspond à une tension de programmation de la carte, et est généralement de 21 volts.
7. *I/O* : C'est l'abréviation de *input/output*, ce contact correspond donc aux entrées et sorties de données en provenance ou à destination de la carte.
8. *RFU* : Réservé pour une utilisation future.

4.1.4. Card Acceptance Device [RaEf97]

Une carte à puce est insérée dans un *dispositif d'acceptation de carte,* qui peut se connecter à un ordinateur. Ce dispositif est appelé en anglais *card acceptance device (CAD)*. D'autres termes sont utilisés pour le dispositif d'acceptation de carte comme *terminal*, *lecteur*, ... etc. Ils offrent tous les mêmes fonctions de base, alimenter la carte en courant et établir une connexion pour transporter des données.

4.1.5. Les APDUs [RaEf97]

Lorsque deux ordinateurs veulent communiquer entre eux, ils échangent des paquets de données, suivant un certain protocole. De même, les cartes à puce parlent au monde extérieur en utilisant leurs propres paquets de données, appelés *APDU(Application Protocol Data Units)*. Un APDU contient une commande ou un message de réponse.

Dans le monde des cartes à puce, le modèle client-serveur est utilisé. Une carte à puce joue toujours le rôle du serveur. En d'autres termes, une carte à puce attend toujours un APDU de commande à partir d'un terminal. Elle exécute l'action spécifiée dans l'APDU et répond au terminal en envoyant un APDU de réponse.

Les tableaux suivants illustrent le format des APDU de commande et de réponse, respectivement. La structure des APDU est décrite dans la norme *ISO 7816, partie 4*.

En-tête obligatoire				Corps additionnel		
CLA	INS	P1	P2	LC	Données	LE

Tableau 4.1 : APDU de Commande

L'*en-tête obligatoire* permet de coder la commande sélectionnée. Il se compose de quatre champs:

- *CLA* : C'est l'abréviation de *class*. Ce champ est codé sur un octet et est utilisé pour identifier une application.
- *INS* : C'est l'abréviation d'*instruction*. Ce champ est codé sur un octet et est utilisé pour indique le code de l'instruction.
- *P1, P2* : C'est l'abréviation de *paramètre 1*, *paramètre 2*. Chacun de ces paramètres est codé sur un octet. Ils fournissent une qualification supplémentaire à l'APDU de commande.

Le *corps additionnel* est une suite d'octets non obligatoire qui correspond aux données envoyées avec la commande :

- *LC* : C'est l'abréviation de *Length of Command Data(Longueur de données de la commande)*. Ce champ peut avoir jusqu'à 3 octets, il désigne le nombre d'octets dans le champ *Données* de l'APDU de commande.
- *Données* : Contient les données à envoyer avec la commande.
- *LE* : C'est l'abréviation de *Length of Response Data(Longueur de données de réponse)*. Ce champ peut avoir jusqu'à 3 octets, il désigne le nombre maximum d'octets attendus dans le champ *Données* de l'APDU de réponse suivant.

En-tête obligatoire		Corps Additionnel
SW1	SW2	Données

Tableau 4.2 : APDU de Réponse

L'*en-tête obligatoire* se compose de deux champs, *SW*1 et *SW*2. *SW* est l'abréviation de *Status Word(octet d'état)*. Ces deux octets nous renseignent sur l'état d'exécution de l'APDU de commande dans la carte à puce.

Le *corps additionnel* est une suite d'octets non obligatoire qui contient les données à envoyer avec la réponse.

4.1.6. Sécurité des cartes à puce [Ran07]

Les données stockées dans une carte à puce doivent être protégées contre l'accès non autorisé et le fouillage grâce a des techniques cryptographiques.

Par exemple un code secret d'une carte bancaire ne doit jamais être révélé. La banque elle-même ne doit pas connaitre les codes secrets de ses clients.

Pour satisfaire ces conditions, l'un des approches utilisées est la *Signature Numérique* (Voir 1.4.3).

La banque attribue à chacun de ses clients une paire de clés, l'une est secrète et détenue par le client (Dans la carte à puce), l'autre est publique et détenue par la banque.

Lorsque un client souhaite s'identifier près de la banque, la banque envoie un message aléatoire au client, la carte à puce s'occupe de signer ce message en utilisant la clé secrète du client et envoie le résultat à la banque, la banque à son tour vérifie la signature en utilisant la clé publique du client, si la signature est juste alors la banque peut être sure qu'il ne s'agit pas d'un fraudeur.

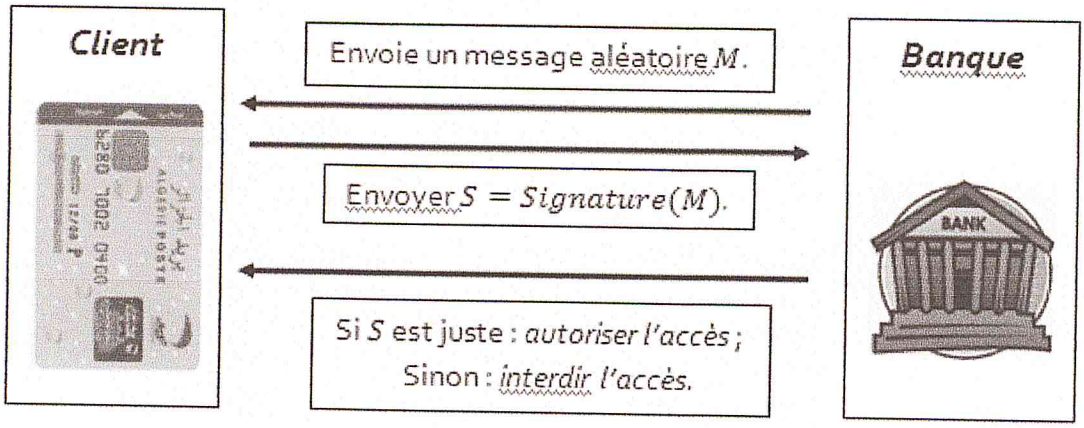


Figure 4.3 : Schéma de vérification d'un code secret

4.2. Les Java Cards

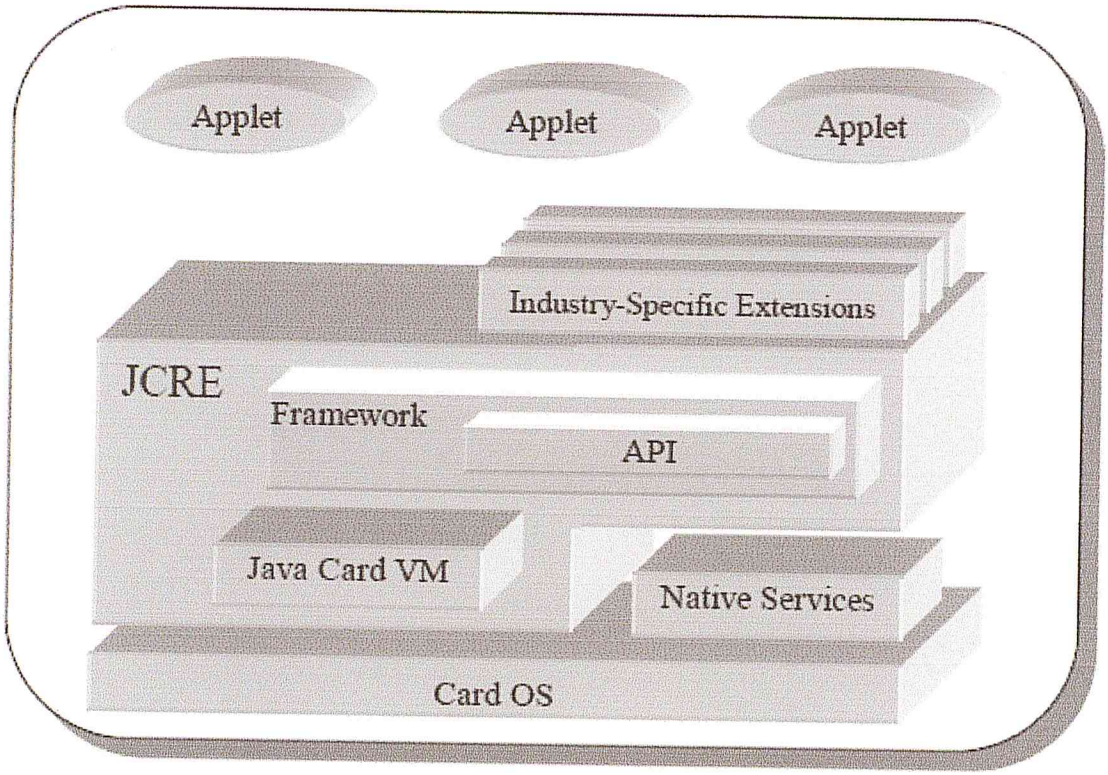


Figure 4.4 : Architecture de la technologie Java Card

4.2.1 Qu'est-ce qu'une Java Card ? [Che00]

Une Java Card est une carte à puce qui est capable d'exécuter des programmes Java.

Une Java Card doit contenir au minimum :

- Un CPU de 8-bits.
- 16 kilo-octets de mémoire morte (ROM).
- 8 kilo-octets d'EEPROM.
- 256 octets de mémoire vive (RAM).

En addition, une Java Card contient plusieurs composants logiciels, tel que décrit dans la figure 4.4 ci-dessus.

- La première couche (*Card OS*) correspond au système d'exploitation de la carte.
- *Native Services* Effectue les opérations d'entrée/sortie et les services d'allocation de mémoire de la carte.
- *Java Card VM* correspond à la machine virtuelle de Java Card.
- *API* est l'abréviation d'*Application Programming Interface*, qui se traduit *interface de programmation*. Elle définit un ensemble de fonctions, routines et méthodes de base nécessaires pour programmer des applications.
- *Framework* définit un ensemble de classes qui implémentent l'API. Ces classes aident le programmeur à développer des applications Java Card.
- *JCRE* est l'abréviation de *Java Card Runtime Environment*, qui se traduit *Environnement d'Exécution Java Card*. Il inclut la Machine Virtuelle Java Card, le Framework, et l'API. Cet environnement permet de cacher les propriétés de la technologie liée au fabriquant de la carte et fournit une interface de programmation commune.
- *Industry extensions* définit un ensemble de classes supplémentaire qu'une entreprise particulière peut ajouter.
- Les *Applets* sont les applications Java Card. Plusieurs Applets peuvent coexister sur une même carte. Chaque applet est identifié par son *AID* (*identifiant d'application*).

4.2.2. Limitations de La machine virtuelle Java Card [Che00]

En raison de ressources mémoire limitées, pas toutes les fonctionnalités du langage Java sont pris en charge sur Java Card. Plus précisément, Java Card ne supporte pas:

- Le chargement dynamique de classes.
- Les Threads et la synchronisation.
- La finalisation.
- Les grands types primitifs de données (float, double, long, et char).
- Les tableaux à plusieurs dimensions.

4.2.3. Créer des Applets

La meilleure façon de montrer comment créer une Applet Java Card, est de travailler sur un exemple. L'exemple suivant est une application *carte étudiant* qui stocke des informations sur un certain étudiant (nom, prénom, adresse), et permet de les extraire plus tard.

4.2.3.1. Les outils utilisés

Afin de pouvoir développer une Applet Javacard et une application cliente, nous allons installer un environnement de développement. Les outils utilisés dans ce but sont :

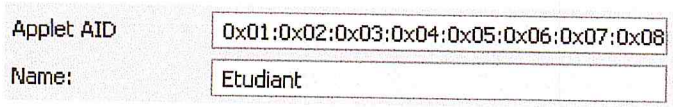
- L'environnement de développement Eclipse version 3.6.
  - *Eclipse est un environnement de développement intégré libre permettant de créer des projets de développement mettant en œuvre n'importe quel langage de programmation.*
    - *[http://www.eclipse.org/org/]*
  - *Dans notre cas, c'est le langage Java qui sera utilisé.*
- Le Kit de Développement Java Card (JCDK) version 2.2.2.
  - *Le Kit de Développement Java Card fournit un environnement de développement autonome dans lequel les Applets Java Card peuvent être développées et testées.*
    - *[http://java.sun.com/javacard/devkit/]*
- Le plugin d'intégration Eclipse Java Card Development Environment (Eclipse-JCDE) version 0.1.
  - *Eclipse-JCDE fournit un environnement de développement visuel qui permet d'automatiser de nombreuses étapes nécessaires pour développer une application Java Card.*
    - *[http://sourceforge.net/projects/eclipse-jcde/]*
  - *Il contient un simulateur appelé Java Card Workstation Development Environment (JCWDE), qui nous permet de tester nos applications Java Card. Ce simulateur va remplacer une Java Card et un programmateur de cartes réels.*

4.2.3.2. Coder une Applet sous eclipse [Sun98], [Sun00]

La première étape consiste à créer un nouveau projet Java Card.

Dan notre projet nous créons une nouvelle Applet.

Nous appelons notre Applet *Etudiant* et on lui affect un *AID*.



Nous allons maintenant programmer notre Applet.

On va donner le code source en essayant d'expliquer chaque détail dans ce code.

Afin d'alléger notre code on ne va traiter que l'adresse de l'étudiant, la procédure est la même pour le nom et le prénom.

```
package etudiant;
```

Java Card supporte les *packages* comme dans la technologie Java standard.

```
import javacard.framework.*;

public class Etudiant extends Applet {
```

Une Applet est une *instance* d'une *classe* qui étend la classe : `javacard.framework.Applet`.

```
final static byte cla = (byte) 0xb0;
```

CLA identifie l'application.

```
final static byte lire = 0x00;
final static byte ecrire = 0x01;
```

Lire et Ecrire correspondent aux instructions supportées par notre Applet.

```
byte[] adresse = null;
```

L'adresse de l'étudiant est initialisée à `null`.

```
private Etudiant() {

    }
```

Le constructeur de notre Applet est *privé*. Ainsi une Applet ne peut être *instanciée* que par sa *méthode* `install`.

```
    public static void install(byte bArray[], short bOffset, byte
bLength)
                throws ISOException {
            new Etudiant().register();
    }
```

La méthode `install` est invoquée par la machine virtuelle Java Card. Cette méthode crée une instance de notre Applet, ensuite la méthode `register` de la classe Applet est appelée pour inscrire l'Applet dans la machine virtuelle Java Card.

```
public void process(APDU apdu) throws ISOException {
```

Apres avoir installer l'applet avec succès, la machine virtuelle Java Card envoie les APDUs entrant à cette méthode.
L'objet *APDU* est maintenu par la machine virtuelle Java Card. Il encapsule les détails du protocole de transmission en fournissant une interface commune. Cet objet permet d'envoyer des informations dans les deux sens entre la carte et le CAD.

```
byte[] buffer = apdu.getBuffer();
```

L'objet APDU contient un tableau (buffer), qui permet de transférer les APDUs entrants et sortants entre la carte et le CAD.

```
if (this.selectingApplet())
    return;
```

La méthode `selectingApplet` permet de préciser si l'Applet est en cours de sélection.

```
if (buffer[ISO7816.OFFSET_CLA] != cla) {
                ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
    }
```

Une *exception* de type *' application non supporté '* est générée si le champ CLA de l'APDU de commande ne correspond pas au CLA de notre Applet.

```
if (buffer[ISO7816.OFFSET_INS] == ecrire) {
```

On vérifie la valeur du cham INS pour savoir quel est l'instruction à exécuter. Dans ce premier cas, c'est l'instruction *écrire* donc on va recevoir des données.

```
short length = (short) (buffer[ISO7816.OFFSET_LC] & 0xff);
```

On extrait la taille des données à recevoir depuis le champ LC de notre APDU.

```
adresse = new byte[length];
short apduDataOffset = 0;

short bytesRead = apdu.setIncomingAndReceive();
```

L'applet doit diriger l'objet APDU à recevoir les données entrantes en invoquant la méthode `setIncomingAndReceive`.

```
while (bytesRead > 0) {

                        Util.arrayCopyNonAtomic(buffer,
ISO7816.OFFSET_CDATA, adresse,apduDataOffset, bytesRead);

apduDataOffset += bytesRead;

bytesRead = apdu.receiveBytes(ISO7816.OFFSET_CDATA);

      }
```

La méthode `receiveBytes` est appelée répétitivement tant qu'il reste toujours des données à recevoir.
La méthode `arrayCopyNonAtomic` est utilisée pour copier ces données dans notre paramètre *adresse*.

```
            return;
      }

if (buffer[ISO7816.OFFSET_INS] == lire) {
```

On vérifie la valeur du cham INS pour savoir quel est l'instruction à exécuter. Dans ce cas, c'est l'instruction *lire* donc on va émettre des données.

```
apdu.setOutgoing();
```

L'applet doit d'abord appeler la méthode `setOutgoing` pour diriger le sens de transfert de données à l'extérieur.

```
apdu.setOutgoingLength((short) adresse.length);
```

Elle appelle ensuite la méthode `setOutgoingLength` pour informer le CAD de la taille des données de réponse.

```
apdu.sendBytesLong(adresse,(short)0, (short)adresse.length;
```

La méthode `sendBytesLong` est invoquée pour envoyer les données de réponse.

```
      return;
      }
```

```
        else {
                ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
```

Une *exception* de type ' *instruction non supporté* ' est générée si le champ INS de l'APDU de commande ne correspond pas au INSs de notre Applet.

```
            }
        }

}
```

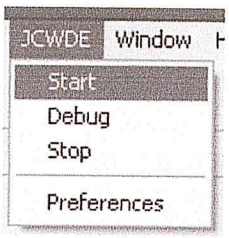
4.2.4. Créer une application cliente [Sun06]

On va utiliser les mêmes outils utilisés pour créer une Applet Java Card.

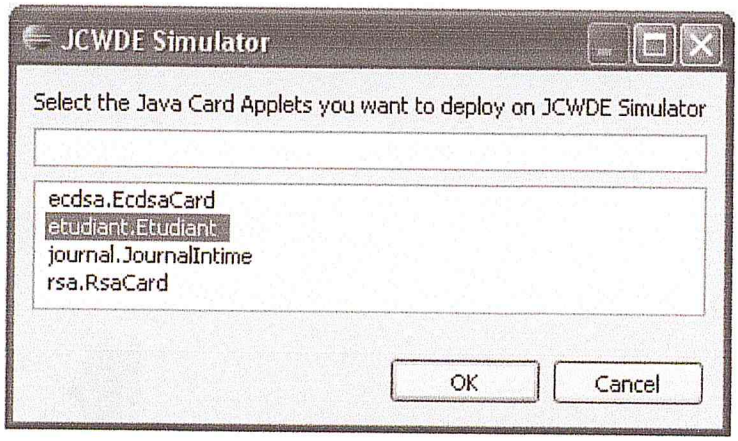
4.2.4.1. Le simulateur JCWDE

Afin de pouvoir échanger des données entre l'Applet que nous venons de créer et une application cliente. Nous allons utiliser le simulateur JCWDE.

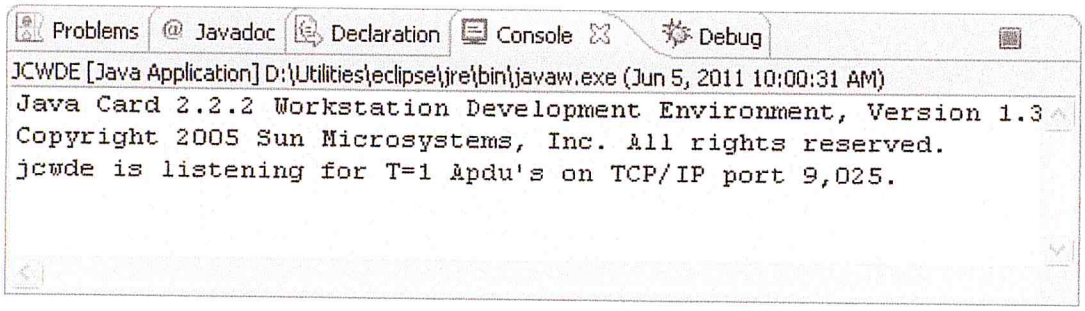
Nous commençons par lancer notre simulateur.



Le simulateur nous interroge pour savoir quel Applet va être déployée. Nous sélectionnons notre Applet *Etudiant*.



Le simulateur est lancé, nous obtenons l'écran suivant.



Comme vous pouvez voir. Notre simulateur écoute les APDUs sur le port 9025 suivant le protocole $T = 1$.

Le protocole $T = 1$ est un protocole normalisé pour l'échange de données défini dans la norme ISO 7816, partie 3. La carte à puce et le terminal échangent des données à l'aide de ce protocole.

4.2.4.2. Coder l'application cliente

Comme dans la partie (*4.2.3. Créer des Applets*), on va donner le code source en essayant d'expliquer les détails importants dans ce code.

```
CadT1Client cad = null;
```

L'objet *cad* de type CadT1Client est utilisé pour communiquer avec le simulateur via le protocole $T = 1$.

```
Socket s=new Socket("localhost",9025);

            BufferedInputStream input = new
BufferedInputStream(s.getInputStream());
            BufferedOutputStream output = new
BufferedOutputStream(s.getOutputStream());

cad = new CadT1Client(input, output);
```

Nous utiliserons une *socket* cliente pour se connecter au simulateur via le port 9025. Notre objet cad va être construit à l'aide des *flux d'entrée et de sortie* de cette socket.

```
cad.powerUp();
```

La méthode `powerUp` nous permet de mettre la carte sous tension.

```
/********************** Créer l'applet ************************/

Apdu apdu = new Apdu();

apdu.command[Apdu.CLA]=(byte) 0x80;
apdu.command[Apdu.INS]=(byte) 0xB8;
apdu.command[Apdu.P1] = 0x00;
apdu.command[Apdu.P2] = 0x00;

            byte[] create={0xb, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
0x07, 0x08, 0x09, 0x00, 0x00, 0x001};

apdu.setDataIn(create);

cad.exchangeApdu(apdu);
```

Nous créons un objet *apdu* pour envoyer les commandes vers la carte.
La méthode command permet de remplir l'en-tête de l'APDU. Pour créer une Applet, le champ *CLA* va avoir la valeur 80 et le champ *INS* va avoir la valeur B8 en héxadécimal.
Le tableau `create` va permettre de spécifier l'AID de notre Applet.
La méthode `setDataIn` va copier le contenu du tableau `create` dans la partie données de l'APDU.
La méthode `exchangeApdu` permet d'échanger les APDUs entre carte et terminal.

```
/******************** Selectionner l'applet ********************/

apdu.command[Apdu.CLA]=0x00;
apdu.command[Apdu.INS]=(byte) 0xa4;
apdu.command[Apdu.P1] = 0x04;
apdu.command[Apdu.P2] = 0x00;

      byte[] select={0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
0x09, 0x00, 0x00};

apdu.setDataIn(select);

cad.exchangeApdu(apdu);
```

Pour sélectionner une Applet, le champ *CLA* va avoir la valeur 00, le champ *INS* va avoir la valeur A4, et le champ *P1* va avoir la valeur 04.
Le tableau select va permettre de spécifier l'AID de l'Applet qu'on veut sélectionner.

```
/********************* Remplir une adresse ********************/

final static byte cla =(byte) 0xb0;
final static byte ecrire = 0x01;
```

CLA identifie l'application, il doit correspondre au *CLA* de l'Applet sélectionnée.
Ecrire est une instruction supportée par notre Applet, qui permet de remplir une adresse.

```
apdu.command[Apdu.CLA]=cla;
apdu.command[Apdu.INS]=ecrire;
apdu.command[Apdu.P1] = 0x00;
apdu.command[Apdu.P2] = 0x00;

      byte[] adresse={'C','i','t','é',' ','M','e','f','t','a','h','
','B','l','i','d','a'};

apdu.setDataIn(adresse);

cad.exchangeApdu(apdu);
```

Pour remplir une adresse, les champs *CLA, et INS* vont avoir les valeurs correspondantes spécifiées au préalable.
Le tableau adresse va contenir l'adresse à remplir.
La méthode `setDataIn` va copier l'adresse dans la partie données de l'APDU.

```
/********************* Lire une adresse ********************/

final static byte lire = 0x02;
```

Lire est une instruction supportée par notre Applet, qui permet de lire l'adresse de l'étudiant contenue dans la carte.

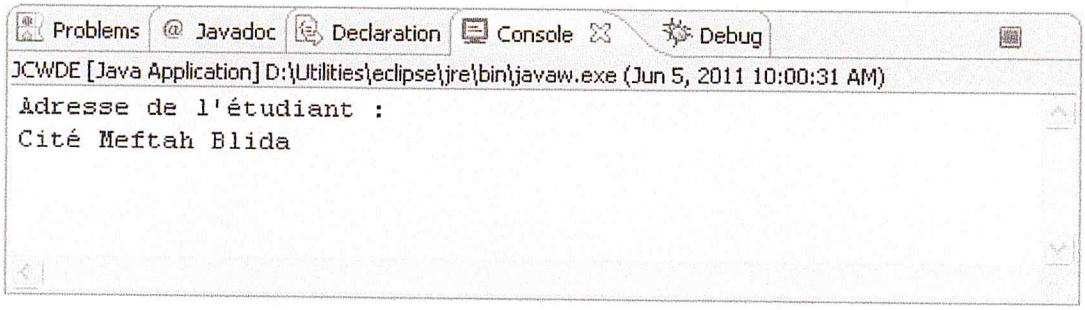
```
apdu.command[Apdu.CLA]=cla;
apdu.command[Apdu.INS]=lire;
apdu.command[Apdu.P1] = 0x00;
apdu.command[Apdu.P2] = 0x00;

cad.exchangeApdu(apdu);
```

Pour lire une adresse, *INS* va avoir la valeur *lire* égale à 02.
On remarque l'absence de la partie données dans notre APDU de commande, ce qui est logique puisque on veut extraire des données de la carte, donc aucune donnée à envoyer.

```
String str=new String(apdu.dataOut);
System.out.println("Adresse de l'étudiant :");
System.out.println(str);
```

La carte répond avec un APDU de réponse et nous livre l'adresse de l'étudiant dans un tableau `dataOut`.
L'adresse de l'étudiant est la même que celle spécifiée lors de l'écriture.




```
cad.powerDown();
```

La méthode `powerDown` permet de retirer la carte du lecteur.

Apres avoir introduit les cartes à puce et présenter les Java Cards. Nous entamons le prochain chapitre qui a pour titre "Comparaison des algorithmes RSA et ECDSA". Les algorithmes RSA et ECDSA sont comparés dans ce chapitre selon les critères, sécurité, espace requis et temps d'exécution.

Chapitre 5

Comparaison des algorithmes RSA et ECDSA

Dans ce chapitre nous allons comparer les algorithmes RSA et ECDSA. Afin de faire cette comparaison, nous avons considéré trois facteurs distincts :

- La sécurité : Sur quoi est fondée la sécurité du cryptosystème, depuis combien de temps est il utilisé et combien sa sécurité a été étudiée.
- L'espace requis : Quel est l'espace nécessaire pour stocker les paires de clés et les paramètres du cryptosystème associé.
- Temps d'exécution : Quel est le temps nécessaire pour effectuer les opérations de génération de clés, signature et vérification de signature.

Le chapitre est divisé en trois parties. Dans les deux premières parties, nous comparons RSA et les cryptosystèmes basés sur les courbes elliptique du point de vue sécurité et espace requis. Dans la dernière partie, nous implémentons les algorithmes RSA et ECDSA sur des cartes à puce Java Card afin de comparer le temps nécessaire pour leurs exécutions.

5.1. Sécurité

5.1.1. Les attaques contre RSA

Rappelons que la sécurité de RSA repose sur le problème de factoriser un grand nombre n. Les attaques contre RSA visent à résoudre ce problème.

Il existe deux types d'algorithmes qui permettent de casser RSA : *les algorithmes à usage spécial* et *les algorithmes à usage général*. Le premier type nécessite de connaitre quelque détails sur les paramètres du système utilisé, le deuxième type ne nécessite de connaitre aucune information sur les paramètres utilisés (mis à part la clé publique bien sure). [Nic99]

Nous ne discutons que les algorithmes à usage général dans le reste de cette partie.

Avant l'apparition de RSA, le meilleur algorithme de factorisation à usage général était *l'algorithme de fraction continue*, cet algorithme est capable de factoriser des nombres allant jusqu'a 133 $bits$. Cet algorithme est fondé sur l'idée d'utiliser *une base factorielle de nombres premiers* et de générer un ensemble d'équations linéaires associé à cette base. La solution de cet ensemble d'équations permet de factoriser le nombre en question.

C'est le même principe qui est utilisé dans les meilleurs algorithmes de factorisation à usage général, connus de nos jours : *le crible quadratique* (abrégé *QS* de l'anglais *quadratic sieve*) et le crible sur le corps des nombres (abrégé *NFS* de l'anglais *number field sieve*). Ces deux algorithmes peuvent être exécutés en parallèle sur des réseaux distribués d'ordinateurs, afin d'accélérer la factorisation. [Nic99]

En 1984, *Carl Pomerance* développa le crible quadratique. Au départ, il a été utilisé pour factoriser des nombres allant jusqu'a 223 $bits$. En 1994, il a été utilisé par un groupe de chercheurs dirigé par *Arjen Lenstra* pour factoriser un nombre de 429 $bits$. Ce nombre était un défi posé par *Martin Gardner* en 1977. La factorisation a été réalisé en *huit mois* par près de *1600 ordinateurs* à travers le monde. La durée totale pour la factorisation a été estimée à 5000 $MIPS - Ans$ (*Le nombre d'instructions exécutées pendant un an à une vitesse d'un million d'instructions par seconde*). [Nic99]

Le crible sur le corps des nombres (NFS) a été inventé en 1989 par *John Pollard* et est l'algorithme le plus efficace pour factoriser des nombres ayant plus de 400 $bits$. Le 22 août 1999, une équipe de scientifiques provenant de six pays différents, dirigée par *Herman te Riele* du *CWI*, est arrivé à factoriser un nombre de 512 $bits$ en utilisant le crible sur le corps des nombres. Ce nombre a été pris de la liste des défis RSA (RSA Challenge List). La factorisation a été réalisé en *sept mois* et la durée totale pour la factorisation a été estimée à 8000 $MIPS - Ans$. [Nic99]

Une estimation de la durée nécessaire pour factoriser un nombre en utilisant le NFS a été donnée par *Randall K. Nichols* [Nic99]. Cette estimation est décrite dans la table ci-dessous.

Longueur du nombre (En bits)	MIPS-Ans
512	3×10^4
768	2×10^8
1024	3×10^{11}
1280	1×10^{14}
1536	3×10^{16}
2048	3×10^{20}

Tableau 5.1 : Durée nécessaire pour factoriser avec NFS

Il est généralement admis que 10^{12} MIPS-Ans représente une garantie raisonnable de sécurité de nos jours. Donc on peut déduire qu'un *modulo* n de 512 $bits$ fournit seulement un niveau de sécurité marginal dans le cas de RSA. Pour une sécurité à long terme, un *modulo* de 1024 $bits$ ou plus doit être utilisé. [Nic99]

5.1.2. Les attaques contre les ECC

Les attaques contre les ECC sont basées sur la résolution du problème du logarithme discret sur les courbes elliptiques (ECDLP).

Les deux attaques les plus efficaces connues de nos jours contre les ECC sont :

- L'algorithme de *Pohlig-Hellman* : Cet algorithme est basé sur la factorisation de n (l'ordre du point de base P de la courbe elliptique). L'algorithme réduit le problème de récupération de la clé secrète d au problème de trouver les facteurs premiers de n, d est ensuite calculé en utilisant *le théorème des restes chinois*. Pour résister à cette attaque il faut choisir une courbe elliptique dont *le cardinal* est divisible par un grand nombre premier n ($n > 2^{160}$). [PoHe78]
- L'algorithme de *Pollard's Rho* : c'est l'algorithme le plus efficace pour résoudre le ECDLP, avec un temps d'exécution égal à $\sqrt{\pi\, n/2}$ étapes. Une étape correspond à une addition ou un doublement sur la courbe elliptique et n est l'ordre du point de base P de la courbe. Comme pour l'algorithme *Pohlig-Hellman*, afin de résister à cette attaque il faut choisir une courbe elliptique dont *le cardinal* est divisible par un grand nombre premier n ($n > 2^{160}$). [Pol78]

Le Tableau ci-dessous donne une estimation de la puissance de calcul nécessaire pour résoudre le ECDLP avec la méthode de *Pollard's Rho*. Cette estimation figure aussi dans [Nic99].

Longueur de n (en bits)	$\sqrt{\pi\, n/2}$	MIPS-Ans
160	2^{80}	9.6×10^{11}
186	2^{93}	7.9×10^{15}
234	2^{117}	1.6×10^{23}
354	2^{177}	1.5×10^{41}
426	2^{213}	1.0×10^{52}

Tableau 5.2 : Durée nécessaire pour résoudre le ECDLP avec L'algorithme de Pollard's Rho

5.1.3. Comparaison

Menezes et *Jurisic* ont comparé le temps nécessaire pour casser RSA avec le temps nécessaire pour casser les ECC. Ils ont utilisé le meilleur algorithme connu en employant différents tailles de modulos.

Les résultats de cette comparaison sont donnés dans le tableau ci-dessous, ce tableau est issu de [Nic99].

MIPS-Ans	Taille de clé RSA (en bits)	Taille de clé ECC (en bits)	Rapport des clés RSA / ECC
10^4	512	106	5 : 1
10^8	768	132	6 : 1
10^{11}	1024	160	7 : 1
10^{20}	2048	210	10 : 1
10^{78}	21000	600	35 : 1

Tableau 5.3 : Taille des clés : comparaison selon le niveau de sécurité

Il est généralement admis que 10^{12} MIPS-Ans représente une garantie raisonnable de sécurité de nos jours. D'après ces résultat, RSA a besoin d'employer des clés de 1024 $bits$, tandis qu'une clé de 160 $bits$ est suffisante pour les ECC. On remarque aussi que l'écart de sécurité entre les deux cryptosystèmes augmente à chaque fois que la taille de la clé est augmentée. [Nic99]

En Septembre 1999, près de 200 personnes utilisant 740 ordinateurs ont réussi à casser une clé ECC de 97 $bits$. Le processus a pris environ 16000 $MIPS - Ans$, ce qui est à peu prêt le double du temps utilisé par le groupe dirigé par *te Riele* pour casser la clé RSA de 512 $bits$. [Sch94]

5.2. Espace requis

Les cryptosystèmes basés sur les courbes elliptiques ont le potentiel de pouvoir fournir une sécurité équivalente à celle des autres systèmes à clé publique, mais avec des clés de tailles plus courtes. Une clé de taille courte est un facteur qui peut être crucial dans certaines applications, par exemple, la conception des systèmes de sécurité pour les cartes à puce.

Nous commençons par comparer l'espace nécessaire pour stocker les paires de clés ainsi que les éventuels paramètres du système. Apres nous comparons la taille des messages résultants d'une opération de chiffrement ou de signature numérique. Les tailles des clés choisies sont, 1024 *bits* pour RSA et 163 *bits* pour les ECC.

5.2.1. Paires de clés et paramètres de système

L'espace nécessaire pour le stockage des clés et des paramètres de système est comparé dans le tableau suivant. [Nic99]

	Paramètres de système (bits)	Clé publique (bits)	Clé privée (bits)
RSA 1024 *bits*	-	1088	2048
ECC 160 *bits*	481	161	160

Tableau 5.4 : espace nécessaire pour le stockage des clés et des paramètres du système.

- Les paramètres de système pour les ECC sont :
  - n le cardinal de la courbe elliptique.
  - f le polynôme irréductible.
  - x et y, les coordonnées du point de base P.
  - a et b, les paramètres qui définissent l'équation de la courbe.
- RSA ne possède aucun éventuel paramètre de système.
- La clé publique pour RSA est le couple (n, e), et la clé privée est le couple (n, d).
- La clé publique pour les ECC est le point Q, et la clé privée est l'entier d.

5.2.2. Taille des messages résultants

L'espace nécessaire pour stocker les messages chiffrés ou signés est comparé dans le tableau suivant. [Nic99]

	Message chiffré (bits)	Message signé (bits)
RSA 1024 *bits*	1024	1024
ECC 160 *bits*	320	320

Tableau 5.5 : espace nécessaire pour le stockage des messages résultants.

5.3. Temps d'exécution

Nous avons implémenté les cryptosystèmes *RSA* et *ECDSA*, Afin de comparer leurs temps d'exécution.

L'environnement utilisé pour cette implémentation est similaire à celui du chapitre 4 (*4.2.3 Créer des Applets*), donc les outils utilisés sont :

- L'environnement de développement Eclipse version 3.6.
- Le Kit de Développement Java Card (JCDK) version 2.2.2.
- Le plugin d'intégration Eclipse Java Card Development Environment (Eclipse-JCDE) version 0.1.

Le langage de programmation utilisé est Java.

L'implémentation de RSA nécessite la manipulation de grands nombres. Les algorithmes nécessaires à cette implémentation tels que, la génération de grands nombres premiers, l'exponentiation modulaire, sont entièrement basés sur l'arithmétique des corps premiers.

L'implémentation d'ECDSA repose sur l'arithmétique des corps premiers ainsi que l'arithmétique de la courbe elliptique choisie. Rappelons qu'une courbe elliptique E est défini sur un corps K, et ainsi les opérations arithmétiques sur une courbe elliptique telles que l'addition et le doublement de points sont entièrement basés sur l'arithmétique du corps choisi, dans notre cas nous avons choisi les corps binaires.

La hiérarchie des modules utilisés est illustrée dans la figure suivante.

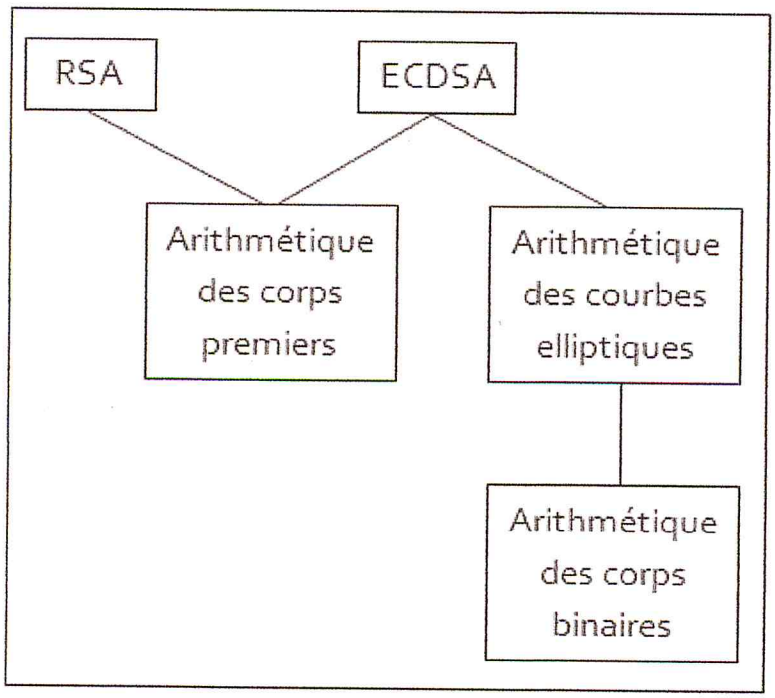


Figure 5.1 : Modules utilisés pour implémenter RSA et ECDSA

Cette partie possède la structure suivante :

- 5.3.1 : Nous décrivons les différents modules utilisés pour l'implémentation.
- 5.3.2 : Nous décrivons l'implémentation des algorithmes RSA et ECDSA.
- 5.3.3 : Nous présentons et nous analysons les résultats de l'implémentation.

5.3.1. Description des modules utilisés

5.3.1.1. Arithmétique des corps premiers

L'arithmétique des corps premiers est réalisée dans la classe `PrimeFieldArithmetic`. Cette classe contient des méthodes statiques qui permettent de réaliser les différentes opérations arithmétiques sur les corps premiers telles que l'addition, la multiplication, la réduction et l'inversement. Ces méthodes correspondent en grande partie aux algorithmes discutés dans le chapitre 2 (*2.1.2. Arithmétique des corps premiers*).

Dans tous ce qui suit p dénotera le modulo du corps et a, b dénoterons des éléments de F_p. Chacun de ces trois entiers est représenté sous forme de tableau de longueur t contenant des mots de 8-bits (Voir 2.1.2). Ce choix est justifié par le fait qu'une carte à puce possède généralement un processeur de 8-bits (Voir chapitre 4).

Le type primitif `byte` est le type approprié pour stocker des mots de 8-bits.

Afin de réaliser ces méthodes, nous définissons deux nouvelles classes :

- La classe `IntegerAssignement` qui va représenter une affectation de la forme "$(\varepsilon, z) \leftarrow w$" pour un entier w (Voir 2.1.2.1). Cette classe contient deux attributs :
  - `byte[] number` qui va représenter l'entier z.
  - `short carry` qui va représenter la retenue ε.
- La classe `DivisionResult` qui va représenter le quotient q et le reste r obtenus en divisant un entier x par un entier y (Voir 2.1.2.3). Cette classe contient deux attributs :
  - `byte[] quotient` qui va représenter l'entier q.
  - `byte[] remainder` qui va représenter l'entier r.

Les différentes méthodes de la classe `PrimeFieldArithmetic` sont décrites dans ce qui suit.

```
public static IntegerAssignement add(byte[]a,byte[]b)
```

Cette méthode effectue l'addition des entiers multi-mots (Algorithme 2.1). Elle prend en entrée les entiers a et b, et calcule leur somme. La sortie est de type `IntegerAssignement`, l'attribut `number` est un tableau de taille t qui correspond au résultat de l'addition et l'attribut `carry` nous indique s'il y a eu un débordement.

```
public static IntegerAssignement sub(byte[]a,byte[]b)
```

Cette méthode effectue la soustraction des entiers multi-mots (Algorithme 2.2). Elle prend en entrée les entiers a et b, et calcule leur différence. La sortie est de type `IntegerAssignement`, l'attribut `number` est un tableau de taille t qui correspond au résultat de la soustraction et la l'attribut `carry` nous indique s'il y a eu un débordement.

```
public static byte[] FpAddition(byte[] a,byte[] b,byte[] p)
```

Cette méthode effectue l'addition modulaire (Algorithme 2.3). Elle prend en entrée les trois entiers a, b et p. Elle renvoie en résultat un tableau de taille t qui correspond à la somme de a et b, modulo p $((a + b)\ mod\ p)$.

```
public static byte[] FpSubtraction(byte[] a,byte[] b,byte[] p)
```

Cette méthode effectue la soustraction modulaire (Algorithme 2.4). Elle prend en entrée les trois entiers a, b et p. Elle renvoie en résultat un tableau de taille t qui correspond à la différence de a et b, modulo p $((a - b)\ mod\ p)$.

```
public static byte[] multiply(byte[] a,byte[] b)
```

Cette méthode effectue la multiplication des entiers multi-mots (Algorithme 2.5). Elle prend en entrée les entiers a et b. Elle renvoie en résultat un tableau de taille $(2t)$ qui correspond au résultat de la multiplication de a par b.

```
public static DivisionResult divide(byte[] z,byte[] p)
```

Cette méthode effectue la division des entiers multi-mots (Algorithme 2.6). Elle prend en entrée un entier z contenu dans un tableau de taille $(n \geq t)$ et le modulo p. Elle renvoie un résultat de type `DivisionResult`, l'attribut `quotient` est un tableau de taille $(n - t + 1)$ qui correspond au résultat de la division de z par p et l'attribut `remainder` est un tableau de taille t qui correspond au reste de la division de z par p.

```
public static byte[] barrettReduction(byte[] z,byte[] p,byte[] u)
```

Cette méthode effectue la réduction de Barrett (Algorithme 2.7). Elle prend en entrée le modulo p, un entier z contenu dans un tableau de taille $(n \geq t)$ et un entier u qui est une valeur pré-calculée (Voir 2.1.2.4.1). Elle renvoie en résultat un tableau de taille t qui correspond au reste de la division de z par p.

```
public static byte[] gcd(byte[] a,byte[] b)
```

Cette méthode permet de calculer *le plus grand commun diviseur (pgcd)* de deux nombres. Elle prend en entrée les entiers a et b et renvoie en résultat un tableau de taille t qui correspond au *pgcd* de a et b.

```
public static byte[] inverse(byte[] a,byte[] p)
```

Cette méthode permet d'effectuer l'inversion dans F_p (Algorithme 2.10). Elle prend en entrée l'entier a et le modulo p. Elle renvoie en résultat un tableau de taille t qui correspond à l'inverse de a par rapport à p $(a^{-1} \ mod \ p)$.

```
public static byte[] FpDivision(byte[] a,byte[] b,byte[] p)
```

Cette méthode permet d'effectuer la division dans F_p. Elle prend en entrée les entiers a, b et le modulo p. Elle renvoie en résultat un tableau de taille t qui correspond au résultat de la division de b par a dans F_p $(b.a^{-1} \ mod \ p)$.

```
public static byte[] randomNumber(short graine, short t)
```

Cette méthode permet de générer un grand nombre aléatoire, elle utilise un *générateur congruentiel linéaire* (Voir 3.1.5-a). Elle prend en entrée deux attributs de type `short` (`graine`, `t`). `graine` correspond à la graine de notre générateur et `t` correspond à la taille du tableau qui contiendra le nombre à générer. Elle renvoie en résultat un tableau de taille t qui correspond au nombre aléatoire généré.

```
public static byte[] prime(short graine, short t)
```

Cette méthode permet de générer un grand nombre premier, on commence par générer un grand nombre aléatoire en utilisant la méthode `randomNumber`, on lui applique par la suite le test de primalité de Fermat (Voir 3.1.5-b) pour vérifier s'il est premier ou non. Elle prend en entrée deux attributs de type `short` (`graine`, `t`). `graine` correspond à la graine de notre générateur de nombres aléatoires et `t` correspond à la taille du tableau qui contiendra le nombre premier à générer. Elle renvoie en résultat un tableau de taille t qui correspond au nombre premier généré.

```
public static byte[] primeWith(byte[] a)
```

Cette méthode permet de générer un nombre premier avec un autre nombre choisi. Elle prend en entrée l'entier a. Elle renvoie en résultat un tableau de taille t qui correspond à un nombre premier avec a.

```
public static byte[] pow(byte[]a,byte[]b,byte[] p,byte[] u)
```

Cette méthode effectue l'exponentiation modulaire (Algorithme 2.11), elle emploie la méthode de Barrett pour faire les réductions (Algorithme 2.7). Elle prend en entrée le modulo p, les deux entier a, b et un entier u qui est une valeur pré-calculée (Voir 2.1.2.4.1). Elle renvoie en résultat un tableau de taille t qui correspond à ($a^b \ mod \ p$).

5.3.1.2. Arithmétique des corps binaires

L'arithmétique des corps binaires est réalisée dans la classe `BinaryFieldArithmetic`. Cette classe contient des méthodes statiques qui permettent de réaliser les différentes opérations arithmétiques sur les corps binaires telles que l'addition, la multiplication, la réduction et l'inversement. Ces méthodes correspondent en grande partie aux algorithmes discutés dans le chapitre 2 (*2.1.3. Arithmétique des corps binaires*).

Dans tous ce qui suit f dénotera un polynôme binaire irréductible de degré m et a, b dénoterons des polynômes binaires de degré inférieur strictement à m. Chacun de ces trois polynôme est représenté sous forme de tableau de longueur t contenant des mots de 8-bits (Voir 2.1.3).

Les différentes méthodes de la classe `BinaryFieldArithmetic` sont décrites dans ce qui suit.

```
public static byte[] add(byte[]a,byte[]b)
```

Cette méthode effectue l'addition des polynômes binaires (Algorithme 2.12). Elle prend en entrée les polynômes a et b. Elle retourne la somme de a et b qui est un polynôme binaire de degré inférieur strictement à m représenté sous forme de tableau de longueur t.

```
public static byte[] mult(byte[] a,byte[] b)
```

Cette méthode effectue la multiplication des polynômes binaires (Algorithme 2.13). Elle prend en entrée les polynômes a et b. Elle retourne le résultat de la multiplication de a par b qui est un polynôme binaire de degré inférieur strictement à $(2m - 1)$ représenté sous forme de tableau de longueur $(2t)$.

```
public static byte[] square(byte[] a)
```

Cette méthode calcule le carré d'un polynôme binaire (Algorithme 2.14). Elle prend en entrée le polynôme a. Elle retourne en résultat le carré du polynôme a qui est un polynôme binaire de degré inférieur strictement à $(2m - 1)$ représenté sous forme de tableau de longueur $(2t)$.

```
public static byte[] reduct(byte[] c,byte[] f,short m)
```

Cette méthode effectue la réduction modulaire des polynômes binaires (Algorithme 2.15). Elle prend en entrée le polynôme f, son degré m et un polynôme binaire c de degré inférieur strictement à $(2m-1)$ représenté sous forme de tableau de longueur $(2t)$. Elle retourne en résultat un polynôme binaire de degré inférieur strictement à m représenté sous forme de tableau de longueur t. Ce dernier polynôme correspond au résultat de la réduction de c modulo f.

```
public static byte[] reductZ163(byte[] c)
```

Cette méthode effectue la réduction modulaire dans F_{2^m} en utilisant le polynôme de degré 163 suggéré par le *NIST* (Algorithme 2.16). Elle prend en entrée un polynôme binaire c de degré inférieur strictement à $(2m-1)$ représenté sous forme de tableau de longueur $(2t)$. Elle retourne en résultat un polynôme binaire de degré inférieur strictement à m représenté sous forme de tableau de longueur t. Ce dernier polynôme correspond au résultat de la réduction de c modulo $z163$ où $z163$ est le polynôme de degré 163 suggéré par le *NIST* (Voir 2.1.3.4.2).

```
public static byte[] inverse(byte[] a,byte[] f)
```

Cette méthode effectue l'inversion dans F_{2^m} (Algorithme 2.17). Elle prend en entrée les polynômes a et f. Elle retourne l'inverse de a par rapport à f $(a^{-1} \ mod \ f)$ qui est un polynôme binaire de degré inférieur strictement à m représenté sous forme de tableau de longueur t.

```
public static byte[] divide(byte[] a,byte[]b,byte[] f)
```

Cette méthode effectue la division dans F_{2^m}. Elle prend en entrée les polynômes a, b et f. Elle retourne le résultat de la division de b par a dans F_{2^m} $(b.a^{-1} \ mod \ f)$ qui est un polynôme binaire de degré inférieur strictement à m représenté sous forme de tableau de longueur t.

5.3.1.3. Arithmétique des courbes elliptiques

Nous avons décidé d'utiliser une courbe elliptique *non-supersingulière* définie sur un corps binaire (Voir 2.2.2). Ainsi l'équation qui définie notre courbe sera la suivante :

$$y^2 + xy = x^3 + ax^2 + b$$

Les paramètres de notre courbe elliptique (y compris le polynôme irréductible du corps binaire f (Voir 2.1.3)) sont les paramètres suggérés par le *NIST* pour la signature numérique avec *ECDSA* en utilisant une clé de *163 bits* (Voir [FIPS00]). Nous les listerons ci-dessous :

```
a = 1.
b = 00000002 0A601907 B8C953CA 1481EB10 512F7874 4A3205FD.
```

$$f(z) = z^{163} + z^7 + z^6 + z^3 + 1.$$

Les valeurs de a et b sont en hexadécimal.

L'arithmétique de notre courbe elliptique est réalisée dans la classe `EllipticCurveArithmetic`. Cette classe contient des méthodes statiques qui permettent de réaliser les différentes opérations arithmétiques sur notre courbe elliptique telles que l'addition et le doublement de points. Ces méthodes correspondent en grande partie aux algorithmes discutés dans le chapitre 2 *(2.2.5 et 2.2.6)*.

Nous définissons une nouvelle classe appelée `Point` qui va permettre de représenter les points de notre courbe elliptique. Cette classe contient deux attributs :

- `byte[] x` qui va représenter la coordonné x de notre point.
- `byte[] y` qui va représenter la coordonné y de notre point.

Nous avons choisi le point (0,0) qui n'appartient pas à notre courbe pour représenter *le point à l'infini*.

Dans ce qui suit p et q dénoterons deux point de notre courbe elliptique, ∞ dénotera le point à l'infini, et k dénotera un entier positif.

Les différentes méthodes de la classe `EllipticCurveArithmetic` sont décrites dans ce qui suit.

```
public static boolean equals(Point p, Point q)
```

Cette méthode prend en entrée les deux points p et q. Elle retourne $vrai$ si ces deux point sont égaux, elle retourne $faux$ sinon.

```
static boolean negative(Point p, Point q)
```

Cette méthode prend en entrée les deux point p et q. Elle retourne $vrai$ si $(p = -q)$, elle retourne $faux$ sinon.

```
public static Point add(Point p,Point q)
```

Cette méthode effectue l'addition de points sur la courbe elliptique (le doublement si $p = q$), il s'agit d'une combinaison des algorithmes (2.18) et (2.19). Elle prend en entrée les deux point p et q et renvoie un point qui correspond à leur somme. On distingue les cas suivants :

- $p = \infty$: Dans ce cas q est renvoyé comme résultat.
- $q = \infty$: Dans ce cas p est renvoyé comme résultat.
- $p = -q$: Dans ce cas ∞ est renvoyé comme résultat.
- $p = q$: Il s'agit de renvoyer le résultat du doublement de p.
- $p \neq q$: Il s'agit de renvoyer le résultat de l'addition de p et q.

```
public static Point mult(Point p, byte[] k)
```

Cette méthode permet de réaliser *la méthode binaire pour la multiplication de points* (Algorithme 2.20). Elle prend en entrée le point p et l'entier k et renvoie en résultat un point égal à kP.

5.3.2. Implémentation de RSA et ECDSA

L'implémentation d'un cryptosystème à clé publique nécessite de générer une paire de clés, l'une privé et l'autre publique. Apres, notre cryptosystème est implémenté en deux partie, une partie réalisant le chiffrement (ou la vérification de la signature) qui utilise la clé publique, et une partie réalisant le déchiffrement (ou la signature numérique) qui utilise la clé privée.

Dans notre cadre de travaille nous allons implémenter deux cryptosytèmes (*RSA* et *ECDSA*) sur des carte à puces Java Card afin de comparer leurs performances. Ces deux algorithmes vont être utilisés pour réaliser la signature numérique. La partie qui détiendra la clé privée et réalisera la signature numérique sera la carte à puce, tandis que le terminal détiendra la clé publique et réalisera la vérification de la signature.

5.3.2.1. Implémentation de RSA

5.3.2.1.1. Génération de clés

La taille des clés choisis est de 1024 *bits*. Cette taille est recommandée actuellement par les laboratoires RSA afin que le problème de factorisation soit impossible à résoudre [PKCS02]. La taille d'une clé RSA se réfère généralement à la taille du modulo n. Ainsi si la taille de n est de 1024 *bits*, alors la taille des deux nombres premiers, p et q, qui composent le modulo n, est égale à 512 *bits* (Voir 3.1.2).

La classe `KeyGeneration` permet de générer une paire de clés (n, e) et (n, d) grace à la méthode statique `generate`. Cette méthode s'appuie sur la classe `PrimeFieldArithmetic` pour réaliser les différentes opérations arithmétiques.

Nous définissons une nouvelle classe appelée `KeyPair` qui va permettre de représenter les paires de clés. Cette classe contient trois attributs :

- `byte[] n` qui va représenter le modulo n.
- `byte[] e` qui va représenter la clé publique e.
- `byte[] d` qui va représenter la clé privée d.

La méthode `generate` est décrite ci-dessous :

```
public static KeyPair generate(short graine, short t)
```

Cette méthode permet de générer une paire de clés, (n, e) et (n, d), en utilisant l'algorithme 3.1. Elle prend en entrée deux attributs de type `short` (`graine`, `t`). `graine` correspond à la graine de notre générateur de nombres premiers et `t` correspond à la taille des tableaux qui contiendront les deux nombres premiers p et q. La sortie est de type `KeyPair`, les attributs `n, e, d` sont des tableaux de taille $(2t)$ qui correspondent aux modulo n, la clé publique e et la clé privée d respectivement.

Si on veut générer des clés de 1024 *bits*, l'attribut t aura la valeur 64, parceque le type `byte` permet de représenter 8 *bits* et ainsi un tableau de 64 cases permettra de représenter 512 *bits*, la taille des deux nombres premiers p et q. Les attributs `n, e, d` seront donc des tableaux de 128 cases.

5.3.2.1.2. Signature

La signature numérique est réalisée au niveau de la carte à puce à l'aide des trois classes `Parameters`, `SignatureGeneration`, `RsaCard`. Nous décrivons chacune de ces classes dans ce qui suit.

La classe "Parameters"

Cette classe contient deux tableaux statiques de type `byte`, `n` et `d`. Ces deux attributs correspondent à la clé privée (n, d) crée dans la phase génération de clés (5.3.2.1.1).

```
public class Parameters {

                    static byte[] n={77,1,113,106,…};

                    static byte[] d={-79,10,-69,37,…};

}
```

La classe "SignatureGeneration"

Cette classe permet de signer un message à l'aide de la méthode `sign`. Cette méthode s'appuie sur la classe `PrimeFieldArithmetic` pour réaliser les différentes opérations arithmétiques et emploie les clés contenues dans la classe `Parameters`. La méthode `sign` est décrite ci-dessous.

```
public static byte[] sign(byte[] m, byte[] n,byte[] d)
```

Cette méthode permet de signer un message à l'aide de l'algorithme 3.4. Elle prend en entrée le message à signer `m` et la clé privée (`n`, `d`) sous formes de tableaux de taille t. Elle renvoie en résultat un tableau de taille t qui correspond à la signature du message source.

La classe "RsaCard"

Cette classe est une Applet Java Card (Voir 4.2.1) qui est créée de la même manière que L'Applet *Etudiant* (Voir 4.2.3). Son rôle consiste à recevoir un message à signer depuis le terminal, elle signe ce message à l'aide de la classe `sign` et renvoie le résultat au terminal.

5.3.2.1.3. Vérification de signature

La vérification de la signature est réalisée au niveau du terminal à l'aide des trois classes `Parameters`, `SignatureVerification`, `RsaClient`. Nous décrivons chacune de ces classes dans ce qui suit.

La classe "Parameters"

Cette classe contient deux tableaux statiques de type `byte`, `n` et `e`. Ces deux attributs correspondent à la clé publique (n, e) crée dans la phase génération de clés (5.3.2.1.1).

```
public class Parameters {

                    static byte[] n={77,1,113,106,…};

                    static byte[] e={17};

}
```

La classe "SignatureVerification"

Cette classe permet de vérifier une signature à l'aide de la méthode `verify`. Cette méthode s'appuie sur la classe `PrimeFieldArithmetic` pour réaliser les différentes opérations arithmétiques et emploie les clés contenus dans la classe `Parameters`. La méthode `verify` est décrite ci-dessous.

```
public static boolean verify(byte[]m, byte[]s, byte[]n, byte[]e)
```

Cette méthode permet de vérifier une signature à l'aide de l'algorithme 3.5. Elle prend en entrée le message `m`, la signature à vérifié `s` et la clé publique (`n`, `e`) sous formes de tableaux de taille t. Elle renvoie *vrai* si la signature est juste, elle renvoie *faux* sinon.

La classe "RsaClient"

Cette classe est une *application cliente* de l'Applet `RsaCard` discuté dans *(5.3.2.1.2)*. Elle est créée de la même manière que l'application cliente de l'Applet *Etudiant* (Voir 4.2.4).

Son rôle consiste à envoyer un message au carte à puce, attendre une réponse de la carte qui correspond à la signature du message envoyé, vérifier la signature à l'aide de la méthode `verify` de la classe `SignatureVerification`.

5.3.2.2. Implémentation d'ECDSA

Rappelons que l'implémentation d'un cryptosystème basé sur les courbes elliptiques tel qu'*ECDSA* nécessite de choisir des paramètres de domaine pour ce système (Voir 3.2.2).

Nous avons choisi d'utiliser les paramètres de domaine suggérés par le *NIST* pour une courbe elliptique définie sur le corps binaire $F_{2^{163}}$ [FIPS00]. Les paramètres dont on a besoin sont listés ci-dessous :

$$f(z) = z^{163} + z^7 + z^6 + z^3 + 1.$$

```
a = 1.

b = 00000002 0A601907 B8C953CA 1481EB10 512F7874 4A3205FD.

n = 00000004 00000000 00000000 000292FE 77E70C12 A4234C33.

x = 00000003 F0EBA162 86A2D57E A0991168 D4994637 E8343E36.

y = 00000000 D51FBC6C 71A0094F A2CDD545 B11C5C0C 797324F1.
```

$f(z)$ est le polynôme irréductible du corps, a et b sont les paramètres qui définissent l'équation de notre courbe elliptique, x et y sont les coordonnés du point de base P, et n est l'ordre de P (Voir 3.2.2.1).

Les valeurs de a, b, n, x et y sont en hexadécimal.

5.3.2.2.1. Génération de clés

La paire de clés, privée et publique, est générée depuis les paramètres de domaine. La clé privée est un entier $d \in [1, n-1]$ où n est l'ordre de P. La clé publique associée est le point $Q = dP$.

La taille de la clé privée est 163 $bits$. Cette taille est recommandée actuellement par le *NIST* afin que *le problème du logarithme discret sur les courbes elliptiques* (Voir 3.2.1) soit impossible à résoudre [FIPS00]. Rappelons que cette taille de clé offre le même niveau de sécurité qu'une clé RSA de 1024 $bits$.

La classe `KeyGeneration` permet de générer une paire de clés d et Q grace à la méthode statique `generate`. Cette méthode s'appuie sur la classe `PrimeFieldArithmetic` pour générer l'entier d, ainsi que la classe `EllipticCurveArithmetic` pour générer le point Q.

Nous définissons une nouvelle classe appelée `KeyPair` qui va permettre de représenter les paires de clés. Cette classe contient deux attributs :

- `byte[] d` qui va représenter la clé privée d.
- `Point q` qui va représenter la clé publique Q.

La méthode `generate` est décrite ci-dessous :

```
public static KeyPair generate(short graine)
```

Cette méthode permet de générer une paire de clés, d et Q, en utilisant l'algorithme 3.9. Elle prend en entrée un attribut de type `short (graine)` qui correspond à la graine utilisée pour générer l'entier d. La sortie est de type `KeyPair`, l'attribut `d` est un tableau de taille t qui correspond à la clé privée d, et l'attribut `q` est de type `Point` et correspond à la clé publique Q.

5.3.2.2.2. Signature

Une signature numérique consiste en un couple d'entiers (r, s). Nous définissons une classe `Signature` qui va représenter ce couple.

```
public class Signature {

                    public byte[] r;

                    public byte[] s;

}
```

La signature numérique est réalisée au niveau de la carte à puce à l'aide des trois classes `Parameters`, `SignatureGeneration`, `EcdsaCard`. Nous décrivons chacune de ces classes dans ce qui suit.

La classe "Parameters"

Cette classe contient un tableau statique de type `byte`, `d` qui correspond à la clé privée d crée dans la phase génération de clés (5.3.2.2.1).

```
public class Parameters {

                    static byte[] d={-27,122,99,…};
}
```

La classe "SignatureGeneration"

Cette classe permet de signer un message à l'aide de la méthode `sign`. Cette méthode s'appuie sur les classes `PrimeFieldArithmetic` et `EllipticCurveArithmetic` pour réaliser les différentes opérations arithmétiques. Elle emploie les clés contenues dans la classe `Parameters`. La méthode `sign` est décrite ci-dessous.

```
public static Signature sign(byte[] d,byte[] e)
```

Cette méthode permet de signer un message, en utilisant l'algorithme 3.10. Elle prend en entrée le message à signer e et la clé privée d sous forme de tableau de taille t. Elle renvoie un résultat de type `Signature` qui correspond à la signature du message source.

La classe "EcdsaCard"

Cette classe est une Applet Java Card (Voir 4.2.1) qui est créée de la même manière que L'Applet *Etudiant* (Voir 4.2.3). Son rôle consiste à recevoir un message à signer depuis le terminal, elle signe ce message à l'aide de la classe `sign` et renvoie le résultat au terminal.

5.3.2.2.3. Vérification de signature

La vérification de la signature est réalisée au niveau du terminal à l'aide des trois classes `Parameters`, `SignatureVerification`, `EcdsaClient`. Nous décrivons chacune de ces classes dans ce qui suit.

La classe "Parameters"

Cette classe contient deux tableaux statiques de type `byte`, `x` et `y`. Ces deux attributs correspondent aux coordonnées (x, y) de la clé publique Q crée dans la phase génération de clés (5.3.2.2.1).

```
public class Parameters {

                    public static byte[] x={-98,34,7,…};

                    public static byte[] y={-4,-2,-93,…};

}
```

La classe "SignatureVerification"

Cette classe permet de vérifier une signature à l'aide de la méthode `verify`. Cette méthode s'appuie sur les classes `PrimeFieldArithmetic` et `EllipticCurveArithmetic` pour réaliser les différentes opérations arithmétiques. Elle emploie les clés contenues dans la classe `Parameters`. La méthode `verify` est décrite ci-dessous.

```
public static boolean verifiy(Point q,byte[] e,Signature si)
```

Cette méthode permet de vérifier une signature, en utilisant l'algorithme 3.11. Elle prend en entrée la clé publique `q` (le point Q), la signature à vérifier `si`, et le message e sous forme de tableau de taille t. Elle renvoie *vrai* si la signature est juste, elle renvoie *faux* sinon.

La classe "EcdsaClient"

Cette classe est une *application cliente* de l'Applet `EcdsaCard` discuté dans *(5.3.2.2.2)*. Elle est créée de la même manière que l'application cliente de l'Applet *Etudiant* (Voir 4.2.4). Son rôle consiste à envoyer un message au carte à puce, attendre une réponse de la carte qui correspond à la signature du message envoyé, vérifier la signature à l'aide de la méthode `verify` de la classe `SignatureVerification`.

5.3.3. Tests et résultats

Cette partie contient les résultats de l'implémentation de RSA et ECDSA, ainsi qu'une analyse et une comparaison de ces résultats.

Nous avons effectué les tests sur 10 échantillons de clés (10 pour RSA et 10 pour ECDSA).

La machine utilisée pour les tests possède la configuration suivante :

- Processeur : Intel Pentium 4, 2.53 GHz.
- RAM : 512 MB.
- Mémoire virtuelle : 768 MB.

5.3.3.1. Résultats de RSA

Le temps d'exécution en millisecondes pour les 10 échantillons est décrit dans le tableau ci-dessous. On s'intéresse aux opérations de génération de clés, signature et vérification de la signature.

Echantillon	Génération de clés (ms)	Signature (ms)	Vérification de la signature (ms)
1	60531	1422	47
2	61890	1438	47
3	119531	1437	32
4	89641	1422	47
5	17953	1453	31
6	113578	1421	31
7	259110	1375	47
8	216500	1453	47
9	81688	1437	31
10	174062	1469	47

Tableau 5.6 : Temps nécessaire pour effectuer les opérations de RSA (selon 10 échantillons de clés).

La moyenne de ces résultats est résumée dans le tableau ci-dessous.

Génération de clés (ms)	Signature (ms)	Vérification de la signature (ms)
119448.4	1432.7	40.7

Tableau 5.7 : Temps moyen nécessaire pour effectuer les opérations de RSA.

5.3.3.2. Résultats d'ECDSA

Le temps d'exécution en millisecondes pour les 10 échantillons est décrit dans le tableau ci-dessous. On s'intéresse aux opérations de génération de clés, signature et vérification de la signature.

Echantillon	Génération de clés (ms)	Signature (ms)	Vérification de la signature (ms)
1	812	703	1390
2	719	688	1406
3	719	688	1359
4	719	687	1375
5	719	672	1375
6	719	672	1313
7	703	672	1391
8	703	672	1344
9	719	656	1484
10	703	687	1360

Tableau 5.8 : Temps nécessaire pour effectuer les opérations d'ECDSA (selon 10 échantillons de clés).

La moyenne de ces résultats est résumée dans le tableau ci-dessous.

Génération de clés (ms)	Signature (ms)	Vérification de la signature (ms)
723.5	679.7	1379.7

Tableau 5.9 : Temps moyen nécessaire pour effectuer les opérations d'ECDSA.

5.3.3.3. Analyse et comparaison de résultats

Rappelons que le but de notre travail était de comparer les deux cryptosystèmes RSA et ECDSA afin de décider, lequel des deux algorithmes est le plus approprié pour être utilisé sur des cartes à puce.

Les trois critères choisis pour cette comparaison sont la sécurité, l'espace requis et le temps d'exécution. La sécurité et l'espace requis ont été discutés dans les parties 5.1 et 5.2.

Nous comparons le temps d'exécution grâce aux résultats obtenus plus haut dans (*5.3.3.1* et *5.3.3.2*).

Génération de clés

La génération des clés pour ECDSA est 165 fois plus rapide que la génération de clés pour RSA (165,09 exactement).

Dans le cadre de notre travail, cette opération s'exécute au niveau d'un ordinateur, après la clé privée est chargée dans la carte, et la clé publique est gardé au niveau de l'ordinateur relié au terminal (Voir 4.1.4).

Puisque cette opération n'est pas réalisée au niveau de la carte mais sur un ordinateur puissant, alors on ne peut pas juger ECDSA meilleur que RSA en s'appuyant sur le temps d'exécution de cette opération. Mais ceci donne un point pour ECDSA.

Signature

La signature avec ECDSA est 2 fois plus rapide que la signature avec RSA (2,10 exactement).

Cette opération est la plus importante car elle se réalise au niveau de la carte à puce. Ce qui nous laisse déduire qu'ECDSA est meilleure que RSA du point de vue, temps d'exécution.

Vérification de la signature

La vérification de signature avec ECDSA est 33 fois plus lente que la vérification de signature avec RSA (33,89 exactement). Ceci est expliqué par le fait que l'exposant *e* de RSA est choisi petit pour optimiser RSA. Si nous avions choisi *e* de manière aléatoire, le temps d'exécution pour la vérification de signature aurai été le même pour RSA et ECDSA.

La rapidité de vérification de signature n'a aucune importance, car cette opération n'est pas réalisée au niveau de la carte à puce mais plutôt au niveau de l'ordinateur relié au terminal.

Conclusion et perspectives

L'objet de cette thèse était de comparer les deux cryptosystèmes RSA et ECDSA afin de décider, lequel des deux algorithmes est le plus approprié pour être utilisé sur des cartes à puce. Afin de comparer ces deux cryptosystèmes, les critères sécurité, espace requis et temps d'exécution ont été pris en compte.

Du point de vue sécurité, nous avons vu que les ECC peuvent fournir le même niveau de sécurité que RSA avec des clés de taille nettement inférieur. Cependant, il faut toujours garder à l'esprit que l'implémentation des ECC est beaucoup plus complexe et nécessite une connaissance mathématique plus profonde, ce qui la rend plus susceptible aux erreurs et diminue donc sa sécurité.

L'espace requis dans une carte à puce est un critère très important, car l'espace réservé pour le stockage des clés est limité. Les ECC peuvent fournir un niveau de sécurité équivalent à celui de RSA avec des clés de taille nettement inférieur, ceci implique que les ECC ont besoin de moins d'espace pour le stockage des clés. Bien que les ECC nécessitent le stockage de paramètres supplémentaires (ce qui n'est pas le cas pour RSA), Ces paramètres n'ont pas besoin d'être stockés dans la carte à puce, mais dans le dispositif qui sera connecté à la carte.

Notre implémentation de RSA et ECDSA montre qu'ECDSA est plus rapides que RSA pour la génération des clés et pour la signature numérique, cependant RSA est plus rapide pour la vérification de signatures. La seule opération qui sera exécuté au niveau de la carte à puce est la signature numérique. La génération de clés et la vérification de signature s'exécuteront au niveau du dispositif connecté à la carte. Donc, du point de vue temps d'exécution, ECDSA est plus rapide que RSA.

Conformément aux paragraphes précédents, nous pouvons déduire qu'ECDSA est le plus approprié pour être utilisé sur des cartes à puce.

Perspectives

Les perspectives en rapport avec cette thèse se résument en ce qui suit :

- Dans notre cadre de travail, nous avant choisi d'utiliser des cartes à puce Java Card afin de réaliser les différents tests. Ce choix est justifié par le fait que, la documentation ainsi que les outils de simulation sont disponibles pour ce type de cartes. Un travail plus approfondi consiste à effectuer les tests sur une carte à puce arbitraire en utilisant un langage d'assemblage. Ceci a pour conséquence de rendre les opérations de RSA et ECDSA encore plus rapides.

- L'implémentation de l'arithmétique des courbes elliptiques a été réalisée en utilisant des coordonnées affines pour la représentation des points. Par conséquence, l'addition de deux points de la courbe nécessite de faire une inversion dans le corps correspondant, qui est une opération très couteuse. L'utilisation des coordonnées projectives permet d'additionner deux points sans avoir à inverser les éléments du corps, ce qui peut améliorer le temps d'exécution. Il faut noter que l'utilisation de ces coordonnées a pour effet de rendre l'implémentation de notre système beaucoup plus complexe.
- Notre implémentation d'ECDSA a été réalisée à l'aide d'une courbe elliptique définie sur un corps binaire, suggérée par le NIST. Une étude plus approfondie consiste à employer plusieurs types de courbes tels que les courbes définies sur les corps premiers ou les courbes définies sur les corps d'extension. Ceci nous permettra d'avoir une idée plus claire sur les performances des systèmes basés sur les courbes elliptiques.

Bibliographie

[ANSI99] ANSI X9.62, *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, American National Standards Institute, 1999.

[AtMo93] A. Atkin, F. Morain, *Elliptic curves and primality proving*, Mathematics of Computation-61(29–68), 1993.

[Bar87] Paul Barrett, *Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor*, LNCS-263 (311-323), 1987.

[Che00] Zhiqun Chen, *Java Card™ Technology for Smart Cards: Architecture and Programmer's Guide*, Prentice Hall, 2000.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms(2nd edition)*, MIT Press, 2001.

[CoFr06] Henri Cohen, Gerhard Frey, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, CRC Press, 2006.

[FIPS00] FIPS 186-2, *Digital Signature Standard*, Federal Information Processing Standards Publication / National Institute of Standards and Technology, 2000.

[Fou03] Pierre-Alain Fouque, *Cryptographie appliquée*, DCSSI, 2003.

[Gen03] Gentle, James E, *Random Number Generation and Monte Carlo Methods (3d edition)*, Springer, 2003.

[HNSF00] Uwe Hansmann, Martin S.Nicklous, Thomas Schack, and Frank Seliger, *Smart Card Application Development Using Java*, Springer, 2000.

[HMV04] Darrel Hankerson, Alfred Menezes, and Scott Vanstone, *Guide to Elliptic Curve Cryptography*, Springer, 2004.

[Knu98] Donald Knuth, *The Art of Computer Programming(3d edition)*, Addison Wesley, 1998.

[Kob94] Neal Koblitz, *A Course in Number Theory and Cryptography*, Springer, 1994.

[Ler97] Reynald Lercier, *Algorithmique Des Courbes Elliptiques Dans Les Corps Finis*, 1997.

[LoDa00] Julio Lopez, Ricardo Dahab, *High-speed software multiplication in* F_{2^m}, LNCS-1977(203-212), 2000.

[Men93] Alfred Menezes, *Elliptic Curve Public Key Cryptosystems*, Kluwer Academic Publishers, 1993.

[MOV96] Alfred Menezes, Paul van Oorschot, Scott Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

[Nic99] Randall K. Nichols, *ICSA Guide to Cryptography*, Computing McGraw-Hill, 1999.

[PKCS02] RSA Laboratories, *PKCS #1 v2.1: RSA Cryptography Standard*, RSA Laboratories, 2002.

[PoHe78] S. Pohlig, M. Hellman, *An improved algorithm for computing logarithms over GF(p) and its cryptographic significance*, IEEE Transactions on Information Theory-24(106-110), 1978.

[Pol78] J. Pollard, *Monte carlo methods for index computation mod p*, Mathematics of Computation-32(918-924), 1978.

[RaEf97] W.Rankl and W.Effing, *Smart Card Handbook*, John Wiley & Sons, 1997.

[Ran07] Wolfgang Rankl, *Smart Card Applications*, Wiley, 2007.

[RSA78] Ronald Rivest, Adi Shamir, and Leonard Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM-21 (120–126), 1978.

[Sat00] T. Satoh, *The canonical lift of an ordinary elliptic curve over a prime field and its point counting*, Journal of the Ramanujan Mathematical Society-15(247–270), 2000.

[Sch85] R. Schoof, *Elliptic curves over finite fields and the computation of square roots mod p*, Mathematics of Computation-44(483–494), 1985.

[Sch94] Bruce Schneier, *Applied Cryptography*, John Wiley & Sons, 1994.

[Sch99] Bruce Schneier, *Crypto-Gram Newsletter November 1999*, 1999.

[Sil94] Joseph H. Silverman, *Advanced Topics in the Arithmetic of Elliptic Curves*, Springer, 1994.

[Sil06] Joseph H. Silverman, *An Introduction to the Theory of Elliptic Curves*, Summer School on Computational Number Theory and Applications to Cryptography (University of Wyoming), 2006.

[Ste67] J. Stein, *Computational problems associated with Racah algebra*, Journal of Computational Physics-1(397–405), 1967.

[Sun98] Sun Microsystems, *Java Card Applet Developer's Guide*, Sun Microsystems, 1998.

[Sun00] Sun Microsystems, *Java Card™ 2.2 Application Programming Interface*, Sun Microsystems, 2000.

[Sun06] Sun Microsystems, *Application Programming Notes, Java Card Platform, Version 2.2.2*, Sun Microsystems, 2006.

[Wei49] André Weil, *Numbers Of Solutions Of Equations In Finite Fields*, American Mathematical Society-55(497,508), 1949.

[WSS03] André Weimerskirch, Douglas Stebila, and Sheueling Chang Shantz, *Generic GF(2^m) arithmetic in software and its application to ECC*, LNCS-2727(79-92), 2003.