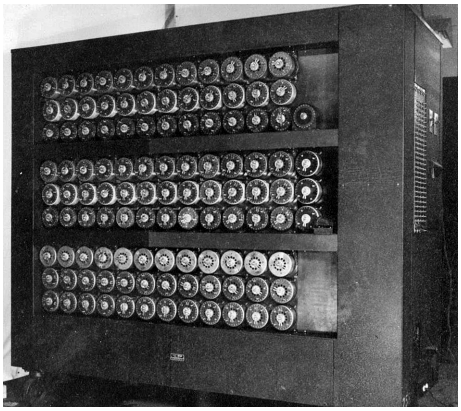
IUT Robert Schuman
Institut universitaire de technologie
Département **informatique**
Université de Strasbourg

M32 : Cryptographie

Travaux Dirigés et Projets



Sauf mention contraire explicite, tous les calculs demandés, sont à faire « à la main ».

1 Arithmétique

1.1 Rappels sur les entiers

Exercice 1 (Algorithme d'Euclide étendu)
1° À l'aide de l'algorithme d'Euclide, déterminer $\text{pgcd}(210, 385)$, puis $\text{pgcd}(89, 144)$.

2° Déterminer $\text{pgcd}(a, b)$, ainsi que les coefficients de Bézout pour les entiers

a) $a = 37$ et $b = 111$;

b) $a = 1876$ et $b = 365$.

Exercice 2 (Efficacité de l'algorithme d'Euclide)
Soit a et b deux entiers tels que $0 < b < a$.
Posons $r_{-1} = a$, $r_0 = b$, et soit r_1, r_2, …, $r_n = 0$, les restes des divisions euclidiennes successives, lorsqu'on applique l'algorithme d'Euclide à a et b (avec ces conventions, r_1 est donc le reste de la division de a par b et r_{n-1} est le pgcd de a et b), de sorte que $n = n(a, b)$ est le nombre de divisions euclidiennes nécessaires pour calculer le pgcd de a et b.
L'objet de l'exercice est de déterminer un majorant de n en fonction de (la taille de) l'entier a.

1° On rappelle que la suite de Fibonacci $(F_n)_{n\geq -1}$ est définie par les valeurs initiales $F_{-1} = 0$ et $F_0 = 1$, et, pour $n \geq 1$, par la relation de récurrence linéaire $F_n = F_{n-1} + F_{n-2}$.
Soit ϕ l'unique réel positif vérifiant $\phi^2 = \phi + 1$ (*aka.* le *nombre d'or*).
Montrer que, pour tout entier $m \geq 0$, $F_m \geq \phi^{m-1}$.

2° Montrer que pour $1 \leq k \leq n$, on a

$$r_{k-2} \geq r_{k-1} + r_k.$$

3° Montrer par récurrence sur l'entier k que, pour $-1 \leq k \leq n$, on a $r_{n-k-1} \geq F_k$.

4° En déduire la majoration

$$n \leq \frac{\ln a}{\ln \phi} + 1.$$

Donner par exemple un majorant du nombre de divisions euclidiennes nécessaires au calcul de $\text{pgcd}(a, b)$ lorsque a s'écrit avec 3 chiffres décimaux, puis avec 6 et 9 chiffres décimaux.

Remarque :
En utilisant les algorithmes vus à l'école primaire, l'addition (ou la soustraction) de deux entiers de k chiffres (binaires, disons, mais ça n'a pas d'importance) au plus se fait en $\mathcal{O}(k)$, et la multiplication ou la division euclidienne de tels nombres se fait en $\mathcal{O}(k^2)$. On en déduit que l'algorithme d'Euclide s'effectue (au pire) en $\mathcal{O}(k^3)$.

Exercice 3
1° Décomposer en facteurs premiers les entiers mentionnés dans l'énoncé de l'exercice 1, et en déduire les pgcd demandés.

2° Décomposer 3372 en facteurs premiers.

3° L'entier 2021 est-il premier ?

Exercice 4 (Crible d'Ératosthène)
Une méthode élémentaire permettant de construire la liste des nombres premiers inférieurs à un entier N donné est la suivante :

a) Écrire la liste des entiers de 2 à N, initialiser un entier $k = 2$.

b) Tant que $k^2 \leq N$, faire :

(a) supprimer de la liste tous les multiples de k ;

(b) le premier élément plus grand que k subsistant est la nouvelle valeur de k.

c) Les nombres restant dans la liste sont les nombres premiers inférieurs à N.

1° Justifier la correction de l'algorithme.

2° Déterminer au moyen de cet algorithme les nombres premiers inférieurs à 100, puis à 200. Combien exécute-t-on d'itérations de la boucle principale dans chaque cas ?

3° Montrer que dans l'étape de suppression, on peut directement se placer dans la liste au niveau de k^2.

1.2 Calculs modulaires

Les résultats (finaux) des calculs modulo n seront exprimés au moyen d'un représentant compris entre 0 et $n-1$ (mais on encourage à être plus souple dans les calculs intermédiaires).

Exercice 5

1° Écrire la table de multiplication de l'anneau $\mathbb{Z}/10\mathbb{Z}$.
Faire la liste des diviseurs de 0, et des éléments inversibles de $\mathbb{Z}/10\mathbb{Z}$, en précisant leurs inverses.

2° Mêmes questions pour l'anneau $\mathbb{Z}/13\mathbb{Z}$.

3° Calculer les puissances suivantes :

a) 2^{25} modulo 9 ;

b) 6^{37} modulo 13 ;

c) 12^{39} modulo 13 ;

d) 85^{85} modulo 19.

Exercice 6

Expliquer comment utiliser l'algorithme d'Euclide étendu pour calculer l'inverse modulaire d'un entier, puis déterminer :

a) l'inverse de 17 modulo 101 ;

b) l'inverse de 357 modulo 1234.

Exercice 7

À l'aide du théorème chinois des restes, résoudre dans $\mathbb{Z}$ les systèmes de congruence suivants

$$\begin{cases} x = 4 \mod 11 \\ x = 9 \mod 13 \end{cases} \qquad \begin{cases} x = 1 \mod 9 \\ x = 9 \mod 16 \end{cases}$$

Exercice 8 (Exponentiation rapide)

Dans le cadre de la cryptographie, on est amené à calculer de grandes puissances modulaires de nombres entiers, le module étant lui-même un entier très grand. Une approche trop naïve mènerait à des calculs trop lents et à des dépassements de capacité.

1° La première idée est de procéder par des élévations au carré successives (avec réduction du représentant après chaque multiplication, évidemment).
Effectuer par exemple $3^{16} \mod 100$ en 4 multiplications seulement ; en déduire les valeurs de $3^{17} \mod 100$, et $3^{33} \mod 100$.

2° La méthode se généralise en s'appuyant sur l'écriture binaire de l'exposant. Calculer par exemple $3^{42} \mod 100$ en effectuant 7 multiplications seulement.

3° Écrire l'algorithme d'exponentiation rapide suggéré à la question précédente (on pourra en donner une version récursive).

4° Essayez de majorer (le plus finement possible) en fonction de l'exposant n le nombre de multiplications nécessaires à l'évaluation de $a^n \mod m$ par exponentiation rapide.

1.3 Tests de primalité

Tous les systèmes cryptographiques à fondements arithmétiques font reposer leur sécurité sur l'usage de grands nombres premiers : concrètement la taille recommandée du module pour RSA, par exemple, est de l'ordre de 1000 à 4000 chiffres binaires, selon le niveau de sécurité requis.

La production de ces grands nombres premiers est par conséquent devenue un enjeu d'importance ; on connaît des algorithmes déterministes pour générer des nombres premiers arbitrairement grands (par exemple le crible d'Eratosthène), mais ils ne sont pas assez efficaces du point de vue algorithmique pour être utilisés en pratique.

L'idée est alors plutôt de tirer des entiers au sort et de tester leur primalité jusqu'à avoir un haut degré de certitude à ce sujet, avec un risque d'erreur, fixé à l'avance et contrôlé.

Exercice 9 (Test de Fermat)
On rappelle le « Petit Théorème de Fermat » :

Si n est premier, alors on a, pour tout $a \in \mathbb{Z}$, $(\text{pgcd}(a,n) = 1 \Rightarrow a^{n-1} = 1 \mod n)$.

On en déduit un test pour voir si un nombre n est composé : on choisit un entier a (le « témoin »), on calcule a^{n-1}, et si le résultat est différent de 1, on sait que n n'est pas premier.

1° Montrer que si n n'est pas premier, alors tout entier a, $0 < a < n$, tel que a n'est pas premier à n, est un « témoin » de la composition (non primalité) de n (ce sont les témoins *triviaux* de la composition de n).
Quels sont les témoins triviaux de la composition de 21 ?

2° Montrer que 2 est un « témoin » de la non primalité de 15 ou 21.

3° Déterminer la décomposition en facteurs premiers de 341. Montrer que 2 n'est pas un « témoin » de la non primalité de 341 (à la main, mais il y a un « truc » à remarquer).

Il y a des nombres non premiers (les *nombres de Carmichaël*) qui passent le test pour toutes les valeurs possibles de a, avec $\text{pgcd}(a,n) = 1$: ils sont rares, le plus petit est 561, mais on sait qu'il y en a une infinité (il y en a exactement 2163 qui sont plus petits que 25 milliards). Pour de tels nombres, les témoins de Fermat (pour la composition) sont très rares, et par conséquent le test de Fermat est pratiquement aussi inefficace que de chercher à factoriser l'entier n.

En pratique, on utilise plutôt un test raffinant le théorème de Fermat, et qui permet d'atteindre **efficacement** un degré de certitude arbitraire sur la primalité d'un entier : le test de Miller-Rabin.

Exercice 10 (Test de Miller-Rabin)
1° Montrer que, si p est premier, ± 1 sont les seules racines carrées de 1 dans $\mathbb{Z}/p\mathbb{Z}$ (autrement dit : les seules solutions de l'équation $X^2 = 1$ dans $\mathbb{Z}/p\mathbb{Z}$).

2° Soit p un nombre premier impair ; l'entier $p-1$ admet alors une unique écriture sous la forme $p - 1 = 2^s m$, où m est un nombre impair. Soit a un entier quelconque.
Déduire de la question précédente que
- soit $a^m = 1 \mod p$,
- soit il existe un entier ℓ, $0 \leq \ell \leq s-1$, tel que $a^{2^\ell m} = -1 \mod p$

3° Le principe du test de Miller-Rabin est de choisir un entier a au hasard (le « témoin de Rabin ») et de vérifier la validité de la propriété précédente pour l'entier p dont on veut tester la primalité : si elle n'est pas vraie on est sûr que p n'est pas premier.
En déduire que le premier nombre de Carmichaël est composé (utiliser un moyen électronique de calcul !).

4° Vérifier que 2 est témoin de Rabin de la non-primalité de 25, mais que 7 ne l'est pas.

5° On admettra la propriété suivante :
Pour tout nombre entier impair composé n, au moins trois quarts des entiers a, $0 < a < n$, sont des témoins de Rabin de la non primalité de n.
Autrement dit, contrairement aux témoins de Fermat, il y a, pour tout n composé, « beaucoup » de témoins de la composition de n.
Comment peut-on utiliser le test de Miller Rabin pour obtenir des entiers qui sont premiers avec une probabilité supérieure à $1 - \varepsilon$, pour une valeur $\varepsilon \in [0,1]$ fixée à l'avance ?

2 Cryptosystèmes à clef publique

2.1 Chiffrement RSA

Exercice 11
Déterminer les valeurs de la fonction indicatrice d'Euler pour tous les entiers inférieurs ou égaux à 20.

Exercice 12
Calculer $C = 11^{13} \mod 101$, puis déterminer un entier d tel que $C^d = 11 \mod 101$.

Exercice 13 (Cryptosystème RSA)
On rappelle le principe du système de chiffrement à clé publique dû à Rivest, Shamir, et Adleman (publié en 1977 ; c'est chronologiquement le premier système cryptographique asymétrique clé publique/clé privée), et connu sous le sigle R.S.A.
On part de deux nombres premiers (grands) p et q, dont les valeurs sont secrètes.

- **Clé publique :** deux entiers (n, e), tels que $n = pq$ et e est premier à $(p-1)(q-1)$
- **Clé privée :** un entier d, tel que $d = e^{-1} \mod (p-1)(q-1)$
- **Chiffrement :** pour un message clair m (entier inférieur à n), calculer $c = m^e \mod n$
- **Déchiffrement :** pour un message chiffré c, entier, calculer $m = c^d \mod n$

1° Un exemple : avec la clé publique $(35, 7)$, chiffrer le message 3.
Déterminer la clé privée et déchiffrer le message obtenu (à la main).
Quels sont les exposants de chiffrement admissibles modulo 35 ?

2° L'exemple historique (à la machine) : avec la clé publique $(2773, 17)$ chiffrer le message 920.
Déterminer la clé privée, et déchiffrer le message précédent.

3° Retour au cas général : démonstration de la validité du procédé de chiffrement/déchiffrement.

a) Déterminer la valeur de $\phi(n)$, en fonction de p et q.

b) En déduire que pour tout entier m premier à n, $(m^e)^d = m \mod n$.

c) Montrer que le procédé de déchiffrement est encore valide lorsque m n'est pas premier à n.

Exercice 14 (Signature RSA)
Le principe du chiffrement RSA est aussi utilisé pour « signer » des messages numériques : la signature vise à garantir au destinataire à la fois *l'intégrité* du message transmis, et *l'identité* de l'expéditeur.
On introduit la notion de *fonction de hachage cryptographique* :

a) il s'agit en premier lieu d'une fonction de *hachage* $h : \{0,1\}^* \longrightarrow \{0,1\}^N$, où N est une constante, autrement dit d'une fonction transformant un mot binaire quelconque en un mot binaire de longueur fixe ;

b) cette fonction a, de plus, la remarquable propriété qu'il est *impossible en pratique* de déterminer $x, x' \in \{0,1\}^*$ tel que $h(x) = h(x')$ (propriété de *résistance aux collisions*).

L'expéditeur A souhaite transmettre un message m à B. Pour mettre en œuvre l'algorithme de signature, il dispose d'une clé RSA (n_A, e_A, d_A), et d'une fonction de hachage cryptographique **publique**.
La signature du message m est obtenue en calculant $\sigma = h(m)^{d_A} \mod n_A$, et A expédie son message en y apposant sa signature : (m, σ).

1° Expliquer comment se fait la vérification de la signature et en quoi cela garantit l'authenticité du message.

2° Montrer qu'on peut se passer de la fonction de hachage pour signer le message. Quel est alors sont intérêt ?

3° En supposant que B dispose d'une clé RSA (n_B, e_B, d_B), comment peut-on, à l'intérieur de ce schéma, garantir en plus la *confidentialité* du message ?

2.2 Chiffrement de Rabin

Exercice 15 (Racines carrées modulo n)

1° Faire la liste des carrés modulo 7 et modulo 15.
Combien une classe modulo 7 a-t-elle de « racines carrées » ? Et une classe modulo 15 ?

2° On rappelle que si p est un nombre premier, la classe de 1 modulo p a exactement deux racines carrées (voir le début de l'exercice 10). Qu'en est-il lorsque le module n'est pas premier ?

3° Soit p un nombre premier. Combien une classe modulo p a-t-elle de racines carrées ?

4° Montrer que si p est un nombre premier congru à 3 modulo 4, et que a est un carré modulo p, alors $a^{\frac{p+1}{4}}$ est une racine carrée de a.

5° Soit $n = pq$, un entier produit de deux nombres premiers distincts. Montrer à l'aide du *théorème chinois des restes* que tout carré inversible modulo n admet exactement 4 racines carrées.
Qu'en est-il des carrés qui ne sont pas des éléments inversibles modulo n ?

6° Déterminer les racines carrées de 9 et de 25 modulo 77, puis modulo 91.

Exercice 16

On présente un système de chiffrement à clé publique, dû à Michael Rabin (1979).
Comme pour RSA, on part de deux nombres premiers (grands) p et q, dont les valeurs sont secrètes. On demande de plus que p et q soient congrus à 3 modulo 4 (ce n'est pas absolument nécessaire, mais ça rend le système plus efficace).

- **Clé publique :** l'entier $n = pq$
- **Clé privée :** les deux nombres premiers p et q, congrus à 3 modulo 4 ; il est aussi utile de pré-calculer les coefficients de Bézout (u_p, u_q) associés à p et q, et tels que : $pu_p + qu_q = 1$
- **Chiffrement :** pour un message clair m (entier inférieur à n), calculer $c = m^2 \mod n$
- **Déchiffrement :** pour un message chiffré c, entier, calculer

$$\begin{cases} m_p = c^{\frac{p+1}{4}} & \mod p \\ m_q = c^{\frac{q+1}{4}} & \mod q \end{cases}$$

Le message m est un élément de l'ensemble :
$\{m_p q u_q + m_q p u_p, m_p q u_q - m_q p u_p, -m_p q u_q + m_q p u_p, -m_p q u_q - m_q p u_p\}$.

1° Exemple : avec la clé publique 77, chiffrer le message 23, puis procéder au déchiffrement.

2° Vérifier la correction de l'algorithme, c'est-à-dire, ici, que le message m est bien l'un des (au plus) 4 entiers obtenus après déchiffrement.

3° Comparer le chiffrement de Rabin et RSA : similitudes, différences, avantages et inconvénients.

2.3 Problème du logarithme discret

Le groupe $(\mathbb{Z}/n\mathbb{Z})^\times$ des inversibles modulo n est dit *cyclique*, ou *monogène*, s'il existe $\alpha \in \mathbb{Z}/n\mathbb{Z}$ tel que, pour tout $\beta \in (\mathbb{Z}/n\mathbb{Z})^\times$, il existe un entier $k \in \mathbb{N}$ (dépendant de β) tel que $\beta = \alpha^k$.
Autrement dit les puissances de α décrivent tout l'ensemble $(\mathbb{Z}/n\mathbb{Z})^\times$; dans ce cas, on dit aussi que α est un *générateur*, ou un *élément primitif*, de $(\mathbb{Z}/n\mathbb{Z})^\times$.
On appelle *logarithme discret de β en base α* (modulo n) le plus petit entier k tel que $\beta = \alpha^k \mod n$.

Exercice 17 (Élément primitifs)

1° Déterminer les éléments primitifs de $(\mathbb{Z}/n\mathbb{Z})^\times$ pour n compris entre 2 et 11.

2° Montrer que le groupe $(\mathbb{Z}/15\mathbb{Z})^\times$ n'est pas monogène.

3° Quel est le plus petit entier $k > 0$ tel que $\alpha^k = 1$ lorsque α est un élément primitif de $(\mathbb{Z}/n\mathbb{Z})^\times$?
On admettra que lorsque le module est un nombre premier p, le groupe $(\mathbb{Z}/p\mathbb{Z})^\times$ est toujours cyclique. Autrement dit, il existe un élément primitif modulo tout p premier.
Quelle est la valeur de k dans ce cas ?

4° Déterminer la table des valeurs du logarithme de base 2 pour $\mathbb{Z}/13\mathbb{Z}$.

Exercice 18 (Protocole de Diffie-Hellmann)
La problème de déterminer un logarithme discret est un problème difficile en général, et c'est sur cette difficulté que repose la première proposition de mise en œuvre pratique de la cryptographie asymétrique, connue sous le nom de *protocole d'échange de clés de Diffie-Hellman* (*New directions in cryptography*, IEEE Transactions on Information Theory, vol. 22, n° 6, 1976).
Schématiquement le protocole est le suivant :

- A et B se mettent d'accord sur un entier premier p, et sur un élément primitif g modulo p (publiquement)
- A choisit secrètement, au hasard, un entier a et envoie $g^a \mod p$ à B
- B choisit secrètement, au hasard, un entier b et envoie $g^b \mod p$ à A
- le secret commun (clef de chiffrement par exemple) entre A et B est $(g^a)^b = (g^b)^a \mod p$

On fixe $p = 23$.

1° Déterminer le plus petit élément primitif g modulo 23.

2° La clef publique est donc (p, g), avec les valeurs déterminées précédemment.
Pendant le protocole d'échange on voit passer sur le réseau les valeurs 11 et 19. Quelle est la clef secrète ?

Exercice 19
On présente un système de chiffrement à clé publique, dû à Taher Elgamal (1985).
On part d'un nombre premier p (grand) et d'un élément primitif $g \in (\mathbb{Z}/p\mathbb{Z})^\times$. Soit A un autre élément de $(\mathbb{Z}/p\mathbb{Z})^\times$, alors, par définition de g, il existe un entier a tel que $A = g^a$ (c'est le logarithme en base α de β).

- **Clé publique :** le triplet (p, g, A)
- **Clé privée :** le couple (p, a)
- **Chiffrement :** pour un message clair m (entier inférieur à p) : choisir aléatoirement un entier $k \leq p - 1$ (secret), puis calculer la paire

$$\begin{cases} y_1 = g^k & \mod p \\ y_2 = mA^k & \mod p \end{cases}$$

  Le message chiffré est (y_1, y_2).
- **Déchiffrement :** pour un message chiffré $(y_1, y_2) \in (\mathbb{Z}/p\mathbb{Z})^\times \times (\mathbb{Z}/p\mathbb{Z})^\times$, calculer

$$m = y_2(y_1^a)^{-1} \mod p$$

1° Exemple : chiffrer 77 avec la clé publique $(89, 3, 11)$.

2° Déterminer la clé privée (on peut utiliser une machine), et procéder au déchiffrement.

3° Vérifier la correction de l'algorithme, c'est-à-dire, ici, que le message déchiffré est bien le message initial.

4° Quel est l'intérêt de faire intervenir l'entier aléatoire k ?

Exercice 20
Dans son article de 1985, Elgamal présente, outre le système de chiffrement asymétrique vu à l'exercice précédent, un schéma de signature numérique qui repose sur les mêmes idées. C'est une version légèrement modifiée (pour gagner en vitesse de calcul) de ce schéma qui est devenu le standard de signature électronique appelé DSA, puis DSS (*Digital Signature Standard*).
Voici le principe de la signature Elgamal :

- la clé secrète (détenue par le signataire uniquement) est le quadruplet (p, g, A, a) (avec les notations de l'exercice précédent) ;
- la clé publique est le triplet (p, g, A) ;
- k est un entier aléatoire premier à $p - 1$ (utilisé pour signer un message donné, et un seul) ;

- la signature du message (entier) m à signer est le couple (K, S) défini par :

$$\begin{cases} K = g^k \mod p; \\ S = (m - aK)k^{-1} \mod p - 1 \end{cases}$$

- la vérification de la signature consiste à comparer $g^m \mod p$ et $A^K K^S \mod p$, la signature étant valide en cas d'égalité

1° Pourquoi le procédé de chiffrement de Elgamal ne peut-il pas être utilisé pour signer des messages ?

2° Calculer et vérifier une signature du message 3 avec la clé $(19, 2, 16, 4)$.

3° Vérifier que le procédé est un bon mécanisme de signature, au sens où tout message authentique sera reconnu comme valide.

4° Quelle est l'utilité de l'entier k ? Y a-t-il unicité de la signature valide d'un message ?

3 Polynômes

3.1 L'algèbre $\mathbb{F}_2[X]$

La notion centrale pour cette section est celle de *polynômes (à une indéterminée)*, et plus précisément celle de polynômes à coefficients dans le corps fini $\mathbb{F}_2 = \mathbb{Z}/2\mathbb{Z}$ (se rappeler les règles de calcul dans ce corps).

Polynômes sur $\mathbb{F}_2$: structure d'espace vectoriel

- Un *polynôme à coefficients dans* $\mathbb{F}_2$ est une suite $(b_n)_{n\geq 0}$ d'éléments de $\mathbb{F}_2$ (de chiffres binaires) nulle à partir d'un certain rang. L'ensemble des polynômes à coefficients dans $\mathbb{F}_2$ est noté $\mathbb{F}_2[X]$.
- Le rang du dernier terme non nul de la suite est appelé *degré* du polynôme, avec la convention que le degré de la suite (constamment) nulle, appelée *polynôme nul*, et notée simplement 0, est égal à $-\infty$.
- L'*addition des polynômes* est définie par l'addition terme à terme (dans $\mathbb{F}_2$) des suites binaires. Par exemple :

$$(1, 0, 1, 0, 0, 0, \dots, 0, \dots) + (0, 0, 1, 1, 0, \dots, 0, \dots) = (1, 0, 0, 1, 0, \dots, 0, \dots)$$

- Soit $n \in \mathbb{N}$. L'usage est de noter X^n la suite dont tous les termes sont nuls sauf le terme b_n, appelée *monôme de degré n*, puis d'écrire tout polynôme comme *combinaison linéaire* de ces monômes. Le monôme X^0 est simplement noté 1.
  Par exemple, la somme précédente s'écrit :

$$(1 + X^2) + (X^2 + X^3) = 1 + X^3$$

  (les parenthèses sont inutiles, une fois qu'on a remarqué l'associativité de l'addition).
- L'ensemble $\mathbb{F}_2[X]$ muni de l'addition et de la multiplication (triviale) par un élément de $\mathbb{F}_2$ forme un $\mathbb{F}_2$-espace vectoriel (de dimension infinie) admettant pour base l'ensemble des monômes $\{X^n\}_{n\in\mathbb{N}}$.

Multiplication. Pour définir la *multiplication de deux polynômes*, on définit d'abord la multiplication par le monôme X : $XX^n = X^{n+1}$, puis on l'étend par **commutativité et associativité** au produit de deux monômes : $X^m X^n = X^{m+n}$. On obtient enfin une opération commutative et associative sur $\mathbb{F}_2[X]$, avec pour élément neutre le polynôme constant non nul, en étendant le produit par **distributivité** sur l'addition de $\mathbb{F}_2[X]$.

On a donc simultanément sur $\mathbb{F}_2[X]$ une structure d'espace vectoriel, et une structure d'anneau commutatif, qui font de $\mathbb{F}_2[X]$ une $\mathbb{F}_2$-algèbre.

Division euclidienne. L'anneau des polynômes à coefficients dans $\mathbb{F}_2$ partage beaucoup de propriétés avec l'anneau des entiers relatifs. La raison en est qu'il existe dans $\mathbb{F}_2[X]$ une *division euclidienne* tout à fait similaire à celle des entiers.

Soit A et B des polynômes de $\mathbb{F}_2[X]$, $B \neq 0$; il existe deux polynômes Q et R, uniquement déterminés, appelés respectivement *quotient* et *reste* de la division euclidienne de A par B, tels que :

$$\begin{cases} A = BQ + R \\ \deg(R) < \deg(B) \end{cases}$$

Généralisation. Toute la construction qui précède se généralise sans difficulté au cas d'un corps K quelconque : on obtient alors l'anneau *euclidien* (ce qui signifie « muni d'une division euclidienne ») $K[X]$ des polynômes à coefficients dans K.

On peut remarquer que, pour $K = \mathbb{R}$, on obtient un ensemble qui s'identifie aux fonctions polynômes à coefficients réels, mais cette identification n'est pas du tout pertinente dans le cas où le corps des constantes est fini, voir l'exercice 23.

Exercice 21

1° Établir la liste des polynômes de degré inférieur ou égal à 4 de $\mathbb{F}_2[X]$.

2° Soit $n \in \mathbb{N}$. Combien y a-t-il de polynômes de degré n exactement dans $\mathbb{F}_2[X]$? Et combien de degré inférieur ou égal à n ?
Mêmes questions dans $\mathbb{F}_p[X]$.

3° On considère les polynômes suivants de $\mathbb{F}_2[X]$:

$$A = X^5 + X^2 + X + 1$$
$$B = X^3 + X^2 + 1$$
$$C = X^3 + X^2 + X + 1$$

Effectuer les sommes $A + B$, $B + C$, puis $A + B + C$.

4° Reprendre la question précédente en considérant les polynômes A, B, et C, comme éléments de $\mathbb{F}_3[X]$.

5° Montrer l'existence et l'unicité de l'opposé d'un polynôme de $\mathbb{F}_2[X]$ (rappelez d'abord la définition d'un *opposé*).

Exercice 22

Pour cet exercice, tous les calculs se déroulent dans $\mathbb{F}_2[X]$.

1° Calculer les produits AB, AC, et ABC, où A, B, et C sont les polynômes définis dans l'exercice précédent.

2° Montrer que la multiplication admet un élément neutre dans $\mathbb{F}_2[X]$. Quels sont les polynômes inversibles de $\mathbb{F}_2[X]$?

3° Plus généralement, si K est un corps, quels sont les polynômes inversibles de $K[X]$?

4° Développer $(X + 1)^2$, puis $(X^2 + X + 1)^4$.

5° Calculer $(X + 1)^{32}$.

Exercice 23

Un polynôme $P \in K[X]$ est naturellement associé à une *fonction polynôme* $\tilde{P} : K \longrightarrow K$ définie en tout $x \in K$ comme le résultat de *l'évaluation* de P en x : on substitue x au symbole X et on effectue les calculs dans K.

Sur les ensembles de nombres usuels, on a l'habitude d'identifier le polynôme et la fonction associée, l'un déterminant totalement l'autre, mais il n'en va pas de même sur les corps finis.

Déterminer deux polynômes de degrés distincts de $\mathbb{F}_2[X]$ associés à la même fonction polynôme, puis deux polynômes distincts de même degré associés à la même fonction polynôme.

Exercice 24

Effectuer les divisions euclidiennes suivantes dans $\mathbb{F}_2[X]$:

1° $X^2 + 1$ par $X + 1$

2° $X^5 + 1$ par $X^2 + 1$

3° $X^8 + X^4 + X^2$ par $X^3 + X + 1$

Il est commode d'utiliser une présentation (ou, si l'on veut, un algorithme de division) analogue à celle utilisée pour la division « posée » des entiers.

Exercice 25

1° Estimer (majorer) le nombre d'opérations sur $\mathbb{F}_2$ (additions et multiplications) que nécessite l'addition, respectivement la multiplication, de deux polynômes de degrés respectifs p et q.

2° De même, majorer, en fonction de leurs degrés, le nombre d'opérations sur $\mathbb{F}_2$ nécessaires à la division euclidienne de deux polynômes.

3.2 Contrôle de Redondance Cyclique

Les codes de *contrôle de redondance cyclique*, ou codes CRC, sont des codes détecteurs d'erreurs universellement utilisés en informatique. On en présente le principe dans les deux exercices suivants.

Exercice 26

La représentation d'un polynôme de $\mathbb{F}_2[X]$ par un mot binaire est immédiate, avec la convention que le coefficient du monôme de degré 0 est représenté par le *bit* le plus à droite (*bit de poids faible*) du mot binaire.
Par exemple :

$$X^5 + X^3 + X^2 \longrightarrow 101100$$

de sorte qu'un polynôme de degré n s'écrit comme un mot binaire de longueur $n+1$ commençant par 1.

1° Quelle opération *bit à bit* correspond à l'addition de deux polynômes ?

2° Comment se traduit la multiplication par le monôme X^k dans la représentation des polynômes par des mots binaires ?

3° Effectuer les divisions euclidiennes de l'exercice 24 dans la représentation binaire des polynômes.

Exercice 27 (CRC)

On fixe un *polynôme de contrôle* G de degré r, connu et utilisé par l'émetteur et par le récepteur du message.
Le message à transmettre est un polynôme P de degré n, et on calcule le reste R de la division euclidienne de $X^r P$ par G. Le message transmis (le *code*) est alors le polynôme $X^r P + R$.

1° Exemple : prenons $P = X^4 + 1$ et $G = X^2 + X + 1$.
Quel est le message transmis, en notation polynomiale, et en notation binaire ?

2° Quel est le reste de la division euclidienne de $X^r P + R$ par G ? (En général, pas sur l'exemple.)

3° En déduire la procédure de décodage du message transmis, en notation polynomiale, et en notation binaire (en cas d'erreur détectée, on demande la retransmission du message).

4° Dans quel cas un message erroné n'est-il pas détecté ?

3.3 Quotients de $\mathbb{F}_p[X]$, corps finis

Exercice 28

On dit que $\alpha \in K$ est *une racine* du polynôme $P \in K[X]$ si l'évaluation de P en α (voir exercice 23) égale zéro : $\tilde{P}(\alpha) = 0$.

1° Montrer que si α est racine de P, alors $X - \alpha$ divise P.

2° En déduire qu'un polynôme de degré 2 ou 3 est irréductible si et seulement s'il n'a pas de racine.

3° Établir la liste des polynômes irréductibles de $\mathbb{F}_2[X]$ de degré inférieur ou égal à 4.

Exercice 29
1° Établir la table de multiplication de l'anneau $\mathbb{F}_2[X]/(X^3+1)$.
Quels sont les éléments inversibles et leurs inverses ?
Calculer les puissances successives de $\alpha = \overline{X}$.

2° Mêmes questions pour $\mathbb{F}_2[X]/(X^3+X+1)$

Exercice 30
Construire « explicitement » un corps à 16 éléments, puis un corps à 27 éléments.

Exercice 31
On travaille dans le corps de cardinal 2^8 utilisé dans Rijndael :

$$\mathbb{F}_2(\alpha) = \mathbb{F}_2[X]/(X^8+X^4+X^3+X+1),$$

les classes seront représentées par les restes de la division euclidienne par $X^8+X^4+X^3+X+1$, et la classe de X est notée α.
Tous les calculs peuvent être réalisés par divisions euclidiennes, mais on peut parfois aussi utiliser les « règles » de calcul qui découlent de $\alpha^8+\alpha^4+\alpha^3+\alpha+1=0$.

1° Que vaut α^8+1 ?

2° Calculez $(\alpha^3+\alpha+1)^3$.

3° Calculez l'inverse de α, et l'inverse de $\alpha+1$

4° Calculez $(\alpha^7+\alpha+1)^3$

4 Standards de chiffrement par blocs

Le premier standard de chiffrement informatique largement accepté a été le *Data Encription Standard*, ou DES ; il est considéré comme insuffisamment sûr au moins depuis la fin des années 90, et ne survit que sous la forme du *triple DES*. Le standard actuel s'appelle *Advanced Encryption Standard*, ou AES, et est la spécification d'un algorithme nommé *Rijndael*.
La description officielle (c'est-à-dire ayant reçu l'approbation des autorités des États-Unis) du DES, référencée ci-dessous par [FIPS46-3], se trouve ici :
`http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`
La description officielle d'AES, référencée ci-dessous par [FIPS197], se trouve ici :
`https://doi.org/10.6028/NIST.FIPS.197`

Exercice 32 (Principe du DES)
Le DES opère par blocs de taille fixe égale à 64 bits ; il s'appuie sur une clé secrète constituée de 56 bits d'information (la longueur de la clé est nominalement de 64 bits, dont 8 bits de contrôle, redondants).
Le chiffrement consiste en 16 *tours* ; au i^{e} tour le bloc de 64 bits en entrée est coupé en deux moitiés, gauche et droite, notées L_i et R_i respectivement, et le bloc en sortie est obtenue de la manière suivante :

$$\begin{cases} L_{i+1} = R_i \\ R_{i+1} = L_i \oplus f(R_i, K_i) \end{cases}$$

où :

- $\oplus$ désigne l'opérateur bit à bit « ou exclusif » ;
- f est une fonction des deux blocs R_i et K_i (ce dernier de taille 48 bits), détaillée dans [FIPS46-3], mais dont la définition exacte ne nous intéresse pas à ce stade ;
- K_i est obtenu à partir de l'unique clé secrète K par un procédé de *diversification de la clé*, détaillé dans [FIPS46-3].

Après le **dernier tour** de chiffrement, les parties gauche et droite L_{16} et R_{16} sont échangées.
De plus, une *permutation* σ opère sur les bits du bloc initial (texte clair), et son inverse opère sur le résultat des 16 tours de chiffrement.

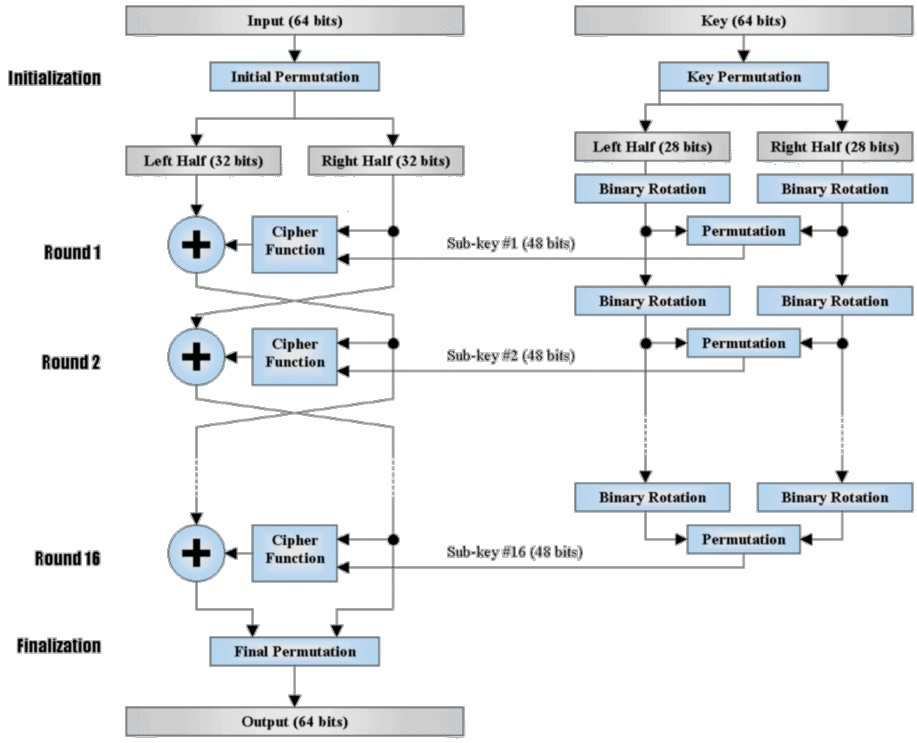


Figure 1 – Représentation schématique du DES (Source : Dave Rudolf, `http://homepage.usask.ca/~dtr467/400/`)

1° Faire un schéma synthétisant les opérations décrites ci-dessus.

2° Rappeler ce qu'est une permutation. Quelle est l'action de la permutation notée [3 2 1 4] sur les entiers de 1 à 4 ? Quelle est la permutation inverse de cette permutation ?
Combien y a-t-il de permutations de 4 éléments ? Faites-en la liste, classées en fonction de leur nombre de points fixes.

3° Comment peut-on retrouver L_0 et R_0 à partir de L_1 et R_1 en utilisant uniquement les opérations déjà utilisées pour le chiffrement ?

4° En déduire l'algorithme de déchiffrement du DES.

Une représentation schématique plus complète du DES est donnée à la figure 1.

Exercice 33

Dans l'algorithme Rijndael, les octets manipulés lors des opérations de chiffrement et déchiffrement représentent les éléments du corps fini $\mathbb{F}_{256}$, réalisé « concrètement » comme le quotient de $\mathbb{F}_2[X]$ par le polynôme irréductible $f = X^8+X^4+X^3+X+1$. On note α la classe de X dans ce quotient, l'anneau (le corps) quotient étant alors noté $\mathbb{F}_2(\alpha)$.

Comme d'habitude les classes des polynômes sont représentées par des polynômes de degré strictement inférieur à celui de f, et l'octet associé à la classe est la suite des coefficients du polynôme, le coefficient dominant étant stocké au bit de poids fort. Par commodité, on écrit souvent ces octets dans leur représentation hexadécimale.

1° Donnez la représentation binaire, puis hexadécimale de α, et de $\alpha^7+\alpha^4+\alpha^3+1$.
Réciproquement, décodez $\{2F\}$ et $\{9C\}$

2° Comment implémenter simplement la multiplication par α sur un octet ?

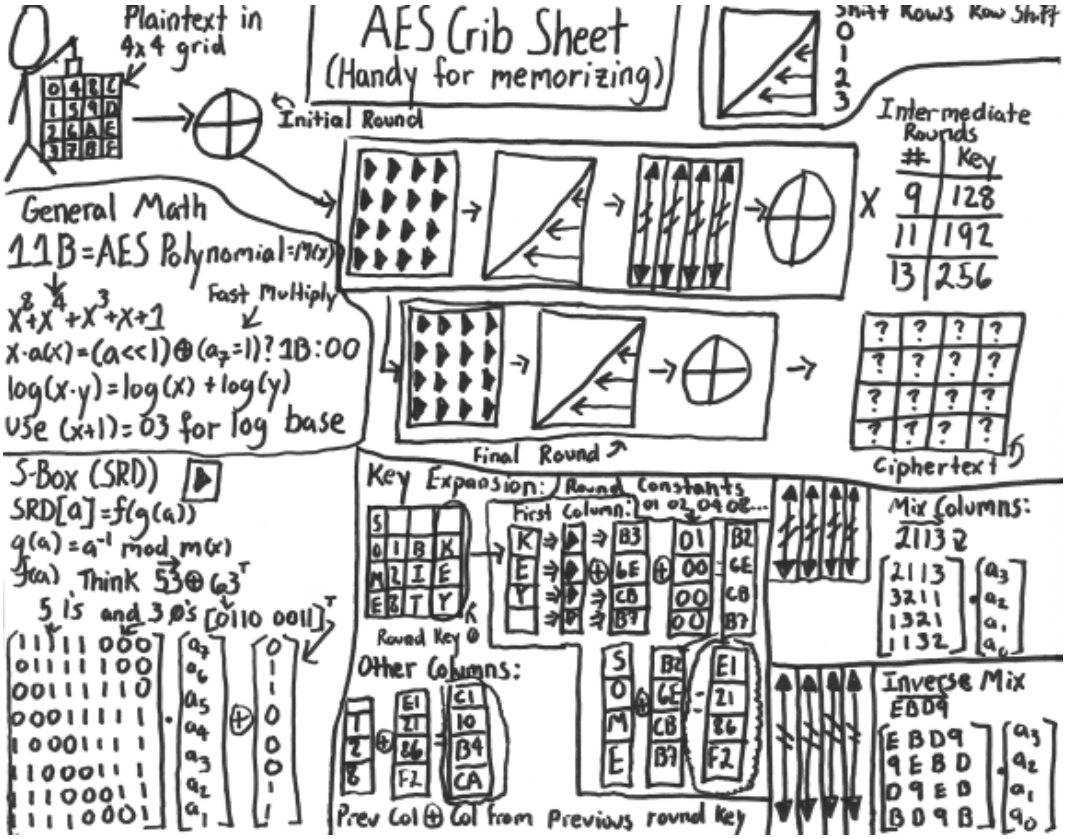


Figure 2 – Représentation schématique d'AES (Source : Jeff Moser, `http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html`)

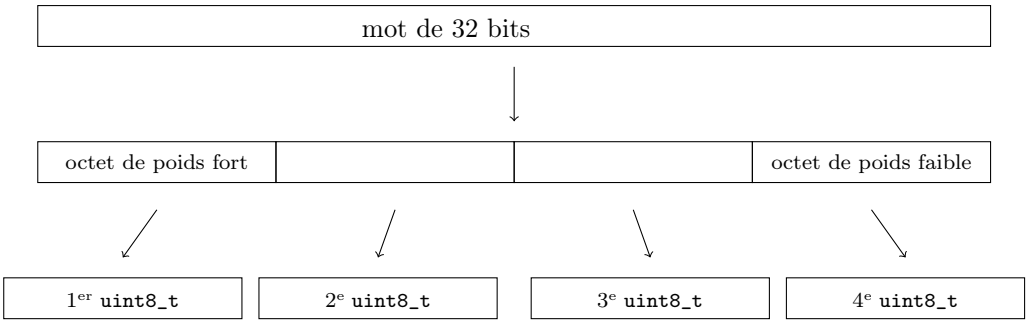
3° En déduire une implémentation du produit d'un octet par α^k, pour $0 \leq k \leq 7$, puis du produit de deux éléments quelconques de $\mathbb{F}_2(\alpha)$.

4° Dans l'algorithme Rijndael, une opération-clef dans le chiffrement est l'inversion d'un élément de $\mathbb{F}_2(\alpha)$ (c'est la seule opération non linéaire dans le chiffrement).
Comment réaliser concrètement cette opération ?
Remarque. Dans une implémentation de Rijndael, la vitesse de chiffrement/déchiffrement est primordiale, et on préfèrera utiliser des tables pour stocker en dur les fonctions appelées `SubBytes` et `InvSubBytes`.

5 Sujets des projets

Consignes générales :

- Le langage imposé est le langage C.
- Les fichiers à coder ou à chiffrer seront en général des fichiers « texte » ne contenant que des caractères ASCII 7 bits, codés sur 8 bits (donc avec un bit à « 0 » en tête de chaque octet).
- Le résultat final du codage ou chiffrage sera transmis sous la forme d'une suite de blocs de taille un octet (on pensera et, dans la suite, on fera référence au type `uint8_t`). Si, par exemple, la fonction de codage produit des blocs de taille 32 bits, ceux-ci seront scindés en 4 octets pour la transmission selon le schéma suivant



  Pour essayer d'être encore plus concret, si le mot initial de 32 bits contient le code hexadécimal `01FF10AA`, à l'issue du découpage on aura une suite de 4 `uint8_t` qui seront, dans l'ordre : `01`, `FF`, `10`, `AA` (on choisit, en quelque sorte, une représentation *grand-boutiste* ou *big-endian* du contenu des mots).
- Il faudra bien sûr prévoir une fonction réciproque, prenant une suite de `uint8_t`, et reconstituant un mot de code.
- Sauf mention contraire explicite, les bits d'un mot binaire sont ordonnés comme les chiffres d'un nombre binaire : le poids d'un bit est d'autant plus faible qu'il est plus à droite du mot considéré.
- Le résultat attendu est un exécutable en ligne de commande.
- En vue de la démo finale, prévoir des fonctions d'affichage permettant de contrôler le bon déroulement des fonctionnalités demandées sur la sortie standard.

5.1 CRC 8

Il s'agit de mettre en œuvre l'algorithme de contrôle de redondance cyclique pour un polynôme de contrôle G, qu'on pourra faire varier, mais de degré fixé égal à 8 : on renvoie à l'exercice 27 pour le principe de codage. Pour fixer les idées et, dans un premier temps, le polynôme, on pourra commencer par $G = X^8 + X^5 + X^4 + 1$.
Les polynômes à coefficients dans $\mathbb{F}_2$ seront encodés selon le principe décrit dans l'exercice 26, sur une longueur de 4 octets ; les bits de chaque bloc (composé de 4 octets) seront naturellement initialisés à zéro. On voit en particulier que tous les polynômes considérés seront de degré strictement inférieur à 32.

Les fonctionnalités attendues sont, par ordre de priorité :

a) Une fonction d'affichage des polynômes en notation polynomiale standard, par puissance décroissante, de gauche à droite.

b) Une fonction réalisant la division euclidienne de deux polynômes.

c) Une fonction d'encodage prenant une suite de `uint8_t` (ceux-ci représentant, typiquement, un texte en caractères `ASCII`), et renvoyant une suite de blocs de 4 octets : les 3 octets de poids fort (ceux de gauche) d'un bloc seront des copies des `uint8_t` en entrée (le premier

`uint8_t` étant le plus à gauche), le quatrième octet, de poids faible, étant défini comme le reste de la division euclidienne du polynôme correspondant aux trois premiers octets par G.

d) Une fonction de décodage restituant le message initial, après détection d'éventuelles erreurs. Si un bloc est corrompu, son décodage se fera en remplaçant chacun de ses octets d'information par le caractère dièse : `#`.

e) Une fonction de « bruitage » aléatoire dépendant d'un paramètre $p \in [0, 1]$: elle agit sur un bloc de code, modifiant chaque bit de ce bloc , indépendamment les uns des autres, avec la probabilité p.

5.2 Cryptosystème RSA

On attend une mise en œuvre du système de chiffrement RSA présenté à l'exercice 13. Les clés et les mots de code seront représentés par des blocs de 32 bits, mais attention aux problèmes de dépassement de capacité lors des opérations de chiffrement/déchiffrement ! Il faudra aussi veiller à ce que la taille du module soit adaptée aux données à chiffrer : on supposera en particulier que le bit le plus à gauche de chaque bloc de message clair est à 0.

Les fonctionnalités attendues sont, par ordre de priorité :

a) Une fonction de chiffrement RSA, la clé publique étant donnée : on chiffrera une suite de blocs de 32 bits, le résultat du chiffrement étant au même format.

b) Une fonction de déchiffrement, étant donné la clé privée ; même format d'entrée et de sortie que pour le chiffrement, évidemment.

c) Une fonction d'encodage prenant une suite de `uint8_t` (typiquement un texte en caractères `ASCII` 7 bits) et les concaténant par 4 pour produire des mots de 32 bits (des *blocs*), le premier `uint8_t` se trouvant à gauche du bloc. Les blocs seront complétés si nécessaire par des zéros à droite.

d) Une fonction de décodage réalisant l'opération inverse de la fonction précédente.

e) Une fonction générant des nombres (pseudo-)premiers à l'aide de l'algorithme de Miller-Rabin (voir l'exercice 10) : la fonction dépendra d'un entier k, inférieur ou égal à 32, fixant le nombre de bits du nombre premier généré, et d'un réel ε contrôlant la probabilité que le nombre obtenu ne soit pas premier.

f) Une fonction de génération d'un système clé publique/clé privée pour RSA, pour un module (clé publique) de 32 bits au maximum : les nombres de bits des entiers p et q sont des paramètres de la fonction, de même qu'un réel ε contrôlant la probabilité de primalité de p et q.

g) Une fonction calculant une « signature » RSA à partir du message à chiffrer : on calculera le « ou exclusif bit à bit » de tous les blocs construits à partir du message en clair (une sorte de fonction de hachage), puis on élèvera le résultat à la puissance l'exposant privé de l'expéditeur pour obtenir une signature sur 32 bits du message.

h) Des fonctions de chiffrement et déchiffrement des messages **signés** : il s'agit simplement d'apposer la signature décrite ci-dessus à la fin des messages chiffrés, d'une part, et de « vérifier » cette signature au moment du déchiffrement, d'autre part.

5.3 Signature numérique RSA

L'objectif est d'implémenter un schéma de signature numérique reposant sur le cryptosystème RSA et la fonction de hachage cryptographique SHA-256, schéma dont le principe est décrit dans l'exercice 14. Le document de référence pour SHA-256 est la [FIPS180-4] du NIST
`https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf`
On ne demande pas d'implémenter la gestion des systèmes de clés : pas de génération de nombres premiers, pas de calcul de clé privée, etc. Des clés de chiffrement/déchiffrement RSA adaptées seront fournies au besoin.

Les fonctionnalités attendues sont, par ordre de priorité :

a) Une fonction réalisant le calcul de l'empreinte SHA-256 d'une suite de `uint8_t` : l'empreinte sera représentée sous la forme d'une suite de 8 mots de 32 bits. Afin de vérifier la correction de la fonction, on recommande d'utiliser les exemples *pas à pas* fourni par le NIST : `https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/SHA256.pdf`

b) Une fonction de chiffrement RSA, la clé publique étant donnée, pour un module de chiffrement codé sur 32 bits : on chiffrera une suite de blocs de 32 bits, le résultat du chiffrement étant au même format.

c) Une fonction de déchiffrement, étant donné la clé privée, toujours pour un module codé sur 32 bits ; même format d'entrée et de sortie que pour le chiffrement, évidemment.

d) Afin de chiffrer l'empreinte d'un message avec un module RSA de 32 bits, il faut modifier le format de l'empreinte (pourquoi ?) ; écrire une fonction prenant l'empreinte SHA-256 sous forme de 8 mots de 32 bits, et la transformant en 9 mots de 32 bits : les deux bits de poids forts du k^e mot sont recopiés en position $2k-1$ et $2k-2$ du 9^e mot.

e) Une fonction de signature RSA d'un message : en entrée, une suite de `uint8_t` (typiquement des caractères `ASCII` 7 bits), ainsi que la clé RSA de chiffrement ; en sortie, le message d'entrée suivi de sa signature (empreinte de 9 mots de 32 bits chiffrée) sous forme de 36 `uint8_t`.

f) Une fonction de vérification de la signature RSA telle que nous l'avons implémentée, renvoyant *valide* ou *invalide*, selon le résultat de la vérification.

5.4 Cryptosystème d'Elgamal

On demande une implémentation du système de chiffrement d'Elgamal présenté à l'exercice 19. Les clés et les mots de code seront représentés par des blocs de 32 bits, mais attention aux problèmes de dépassement de capacité lors des opérations de chiffrement/déchiffrement !

Il faudra aussi veiller à ce que la taille du module soit adaptée aux données à chiffrer : on supposera en particulier que le bit le plus à gauche de chaque bloc de message clair est à 0 (ce qui est le cas pour le chiffrement d'une suite de caractères `ASCII` 7 bits).

Les fonctionnalités attendues sont, par ordre de priorité :

a) Une fonction de chiffrement, étant donné une clé publique pour le crypto-système d'Elgamal : on chiffrera une suite de blocs de 32 bits, le résultat du chiffrement étant une suite de blocs de 64 bits.

b) Une fonction de déchiffrement, étant donné la clé privée, échangeant formats d'entrée et de sortie de la fonction de chiffrement, évidemment.

c) Une fonction d'encodage prenant une suite de `uint8_t` (typiquement un texte en caractères `ASCII` 7 bits) et les concaténant par 4 pour produire des mots de 32 bits, le premier `uint8_t` se trouvant à gauche du bloc. Les blocs seront complétés si nécessaire par des zéros à droite.

d) Une fonction de décodage réalisant l'opération inverse de la fonction précédente.

e) Une fonction calculant, pour p premier, le plus petit élément primitif de $\mathbb{Z}/p\mathbb{Z}$.

f) Une fonction de génération d'un système clé publique/clé privée pour le système d'Elgamal. On génèrera en particulier un nombre (pseudo-)premier à l'aide de l'algorithme de Miller-Rabin (voir l'exercice 10) : la fonction dépend d'un entier k, inférieur ou égal à 32, fixant le nombre de bits du nombre premier généré, et d'un réel ε contrôlant la probabilité que le nombre obtenu ne soit pas premier.

5.5 Signature numérique Elgamal

L'objectif est d'implémenter un schéma de signature numérique reposant sur le procédé de signature numérique d'Elgamal, décrit dans l'exercice 20, et sur la fonction de hachage cryptographique SHA-256. Le document de référence pour SHA-256 est la [FIPS180-4] du NIST

`https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf`
On ne demande pas d'implémenter la génération des clés : pas de recherche de nombres premiers, pas de calcul d'éléments primitifs, etc. Des clés privées pour la signature Elgamal seront fournies, ou obtenues via une source externe au besoin.
Les modules (nombres premiers p des clés) seront codés sur 32 bits au maximum.
Les fonctionnalités attendues sont, par ordre de priorité :

a) Une fonction calculant la signature d'un message représenté par un bloc de 32 bits par le procédé d'Elgamal, étant donné la clé privée, et la signature étant codée sur deux blocs de 32 bits (*aka* `uint32_t`).

b) Une fonction de vérification de la signature Elgamal d'un message constitué d'un bloc de 32 bits (en entrée : le message, la signature, et une clé publique).

c) Une fonction réalisant le calcul de l'empreinte SHA-256 d'une suite de `uint8_t` : l'empreinte sera représentée sous la forme d'une suite de 8 mots de 32 bits. Afin de vérifier la correction de la fonction, on recommande d'utiliser les exemples *pas à pas* fourni par le NIST : `https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/SHA256.pdf`

d) Une fonction calculant la signature Elgamal de l'empreinte SHA-256 d'un message : en entrée 8 *mots* de 32 bits, en sortie 16 blocs de 32 bits, chacun des mots de l'empreinte étant signé par 2 blocs consécutifs

e) Une fonction de signature Elgamal d'un message : en entrée, une suite de `uint8_t` (typiquement des caractères `ASCII` 7 bits), ainsi qu'une clé privée pour la signature Elgamal ; en sortie, le message d'entrée suivi de sa signature sous forme de 64 `uint8_t`.
Il convient de noter que si la clé privée de signature est évidemment la même pour les 8 mots de l'empreinte, la valeur de l'entier aléatoire k doit être différente pour chaque bloc.

f) Une fonction de vérification de la signature Elgamal d'un message signé, telle que nous l'avons implémentée, renvoyant *valide* ou *invalide*, selon le résultat de la vérification.

5.6 Cryptosystème de Rabin

On demande une implémentation du système de chiffrement de Rabin présenté à l'exercice 16. Les clés et les mots de code seront représentés par des blocs de 32 bits, mais attention aux problèmes de dépassement de capacité lors des opérations de chiffrement/déchiffrement !
Il faudra aussi veiller à ce que la taille du module soit adaptée aux données à chiffrer : on supposera en particulier que le bit le plus à gauche de chaque bloc de message clair est à 0.

Les fonctionnalités attendues sont, par ordre de priorité :

a) Une fonction déterminant les racines carrées d'une classe modulo $n = pq$, où p et q sont congrus à 3 modulo 4.

b) Une fonction de chiffrement, étant donné une clé publique pour le crypto-système de Rabin : on chiffrera une suite de blocs de 32 bits, le résultat du chiffrement étant également une suite de blocs de 32 bits.

c) Une fonction de déchiffrement, étant donné la clé privée : mêmes formats d'entrée et de sortie que la fonction de chiffrement, évidemment.
L'opération de chiffrement n'est pas injective en raison de l'existence de plusieurs racines carrées modulo n. L'idée pour choisir entre les différentes racines obtenues est de tester la condition de répétition de l'octet de poids fort (voir la description de l'encodage ci-dessous). Si une seule des racines obtenues vérifie cette condition, il n'y a pas d'ambiguïté dans le déchiffrement ; s'il y en a plusieurs, on signalera l'ambiguïté en déchiffrant par le bloc correspondant à la chaîne `###` .

d) Une fonction d'encodage prenant une suite de `uint8_t` et produisant des mots de 32 bits : chaque bloc de longueur 32 est obtenu par concaténation de 3 `uint8_t`, le premier de ces 3 `uint8_t` étant répété à gauche du bloc ; par exemple, la suite de 3 octets 72 5A 3B sera encodée par 72725A3B. Les blocs seront complétés si nécessaire par des zéros à droite.

e) Une fonction de décodage réalisant l'opération inverse de la fonction précédente.

f) Une fonction générant des nombres (pseudo-)premiers congrus à 3 modulo 4, à l'aide de l'algorithme de Miller-Rabin (voir l'exercice 10) : la fonction dépend d'un entier k, inférieur ou égal à 32, fixant le nombre de bits du nombre premier généré, et d'un réel ε contrôlant la probabilité que le nombre obtenu ne soit pas premier.

g) Une fonction de génération d'un système clé publique/clé privée pour le cryptosystème de Rabin, pour un module (clé publique) s'écrivant sur 32 bits au maximum (le nombre de bits des nombres p et q seront aussi paramétrables).

5.7 DES

On demande une implémentation de l'algorithme de chiffrement symétrique *Data Encryption Standard* (D.E.S.) dont le principe est présenté à l'exercice 32. Pour le détail des opérations, on conseille de se référer à [FIPS46-3].
Vous trouverez sur la plateforme Moodle, des fichiers texte contenant les tableaux et matrices constants auxquels il est fait référence ci-dessous : permutations initiale et finale, matrice d'expansion, *S-boxes*, permutation utilisée dans f, choix permutés.

La programmation du DES ne présente pas de difficulté conceptuelle particulière, mais il faut être **très** rigoureux, et valider soigneusement les différentes étapes du chiffrement. Vous êtes donc invités à respecter strictement le découpage qui est donné ci-dessous : vous trouverez ainsi des exemples de test pour chaque étape sur Moodle, qui vous permettront de valider votre code.

Les fonctionnalités attendues sont, par ordre de priorité :

a) Une fonction de chiffrement d'un bloc de 64 bits, réalisant les 16 tours décrits dans l'exercice 32, sans les permutations initiale et finale. Dans un premier temps, on propose que la fonction f ne dépende que du numéro i du tour de chiffrement, et soit définie par $f(i) = 37453123 \times i$. On demande également, à ce stade, une fonction de déchiffrement.

b) Les fonctions de chiffrement et de déchiffrement intégrant les permutations initiale et finale, IP et IP^{-1}.

c) Une fonction d'expansion appliquant la matrice d'expansion E à un bloc de 32 bits, afin de l'étendre à 48 bits (les bits du bloc d'origine sont mélangés et 16 d'entre eux sont dupliqués et répartis dans les 48 bits résultants, *cf.* [FIPS46-3]).

d) Une fonction de substitution appliquant les 8 fonctions de sélection S_i (S-box) (mises à disposition sur le site Moodle du cours) à un bloc de longueur 48 (découpé en 8 sous-blocs de longueur 6), et renvoyant un bloc de longueur 32.

e) Une fonction f appliquant l'expansion, le mélange (ou exclusif) avec une clé constante $K = 123456ABCDEF$ (à ce stade on utilise la même clé à chaque tour, pas de diversification de la clé), la fonction de substitution puis la permutation finale P.

f) Une fonction de génération des 16 clés K_i (une clé par tour) à partir de la clé unique K (diversification de la clé) ; les tables donnant les « choix permutés » utilisés sont sur Moodle.

g) Une fonction de chiffrement (et de déchiffrement) implémentant le D.E.S. complet sur un bloc.

h) Finalement, une fonction découpant une suite de caractères `ASCII` en blocs de 64 bits (on complètera au besoin les blocs par *padding* de zéros à droites), et chiffrant le code obtenu, ainsi que la fonction symétrique de déchiffrement et de décodage ; on renvoie à ce sujet aux consignes générales pour les projets, en début de section.

5.8 AES

Une description détaillée de l'algorithme, avec des indications pour l'implémentation, est donnée dans [FIPS197] qui est la meilleure référence : vous y trouverez en particulier, en appendice, des exemples de déroulement pas à pas, et tour par tour d'un chiffrement.

La version à implémenter est la plus proche de l'algorithme Rijndael, avec une longueur de clé égale à 128 bits (version dite AES-128).

Il est évidemment plus que conseillé de faire au préalable l'exercice 33 qui permet de se familiariser avec le corps $\mathbb{F}_2(\alpha) \simeq \mathbb{F}_{256}$.

Les fonctionnalités attendues sont, par ordre de priorité :

a) Une fonction réalisant la multiplication par α dans $\mathbb{F}_2(\alpha)$, puis une fonction réalisant la multiplication de deux octets dans $\mathbb{F}_2(\alpha)$.

b) Les fonctions `SubBytes`, `ShiftRows`, `MixColumns`, ainsi que leurs inverses agissant sur les *états* (des matrices 4×4 à coefficient dans $\mathbb{F}_2(\alpha)$). (Vous trouverez facilement des versions tabulées de la fonction `SubBytes` et de son inverse sur le *web*.)

c) Les fonctions de manipulation des clefs `AddRoundKey` et `KeyExpansion`.

d) Les fonctions `Cypher` et `InvCypher` de chiffrement et déchiffrement d'un bloc de 128 bits (Il faudra évidemment aussi prévoir des fonctions d'affichage des blocs et des états pour les tests et le débogage).

e) Des fonctions de chiffrement et déchiffrement d'un fichier, pour une clef donnée : en entrée, et en sortie, il s'agit simplement d'une suite d'octets (qu'on peut interpréter comme du « texte » côté entrée, mais certainement pas côté sortie). On complètera au besoin les blocs par *padding* de zéros à droites.

f) On conseille aussi de prévoir des fonctions auxiliaires d'affichage des *états*, des *mots*, et des *blocs* afin de faciliter le déboggage.

Table des matières