Problème 1. Cryptographie

L'objet de ce problème est l'étude deux systèmes cryptographiques modernes, en particulier des algorithmes de chiffrement et déchiffrement associés.

Les deux parties de ce problème sont indépendantes.

Vocabulaire Un système de chiffrement permet de transformer un *message clair* (c'est-à-dire compréhensible) en *message chiffré* (incompréhensible). La fonction qui permet de passer du message clair au message chiffré est appelée *fonction de chiffrement*, et la fonction qui permet de faire l'opération inverse est appelée *fonction de déchiffrement*. Ces deux fonctions utilisent chacune une *clé* (qu'on peut voir comme des suites de bits, ou des entiers).

Les systèmes cryptographiques modernes se divisent en deux catégories : symétriques (à clé secrète) et asymétriques (à clé publique). Dans un système symétrique la clé de chiffrement est utilisée à la fois dans l'algorithme de chiffrement et dans celui de déchiffrement : la sécurité du système repose sur la complexité de la clé et sa non-divulgation. Dans un système asymétrique il existe deux clés : la clé de chiffrement, publique, notée `pk`, et la clé de déchiffrement, privée, notée `sk`. La sécurité du système repose sur le fait que la connaissance de `pk` ne permette pas en temps raisonnable de déterminer `sk`.

1.1 Réseau de substitution-permutation : schéma de Feistel

Un réseau de substitution-permutation (SPN [1]) est un système de chiffrement à clé secrète très rapide car consistant en une succession de substitutions (remplacer des `0` par des `1` et des `1` par des `0` si le message est exprimé en binaire) et des permutations (qui peuvent se voir comme une ré-indexation d'une liste).

Question 1.1. Permutations Soit $t = (t_0, t_1, \ldots, t_{n-1})$ un n-uplet et π une permutation de $\{0, \ldots, n-1\}$. L'application de π à t est le n-uplet $t_\pi = (t_{\pi(0)}, t_{\pi(1)}, \ldots, t_{\pi(n-1)})$. On représente une permutation π par le n-uplet $(\pi(0), \pi(1), \ldots, \pi(n))$.

(a) Déterminer l'image de (a, b, c, d, e) par la permutation $\pi = (4, 2, 0, 3, 1)$.

(b) Écrire une fonction `permutation` qui reçoit en entrée un n-uplet `t` et une permutation π et renvoie le n-uplet t_π.

(c) Écrire une fonction `rand_perm` qui reçoit en entrée un entier n et renvoie une permutation aléatoire π de $\{0, \ldots, n-1\}$ sous forme de n-uplet. *On utilisera les fonctions du module `random` décrites en annexe.*

Question 1.2. Substitutions Soit $t = (t_0, t_1, \ldots, t_{n-1})$ une liste de bits de longueur n. Une *substitution* de t est un n-uplet $t \oplus \sigma = (t_0 \oplus \sigma_0, t_1 \oplus \sigma_1, \ldots, t_{n-1} \oplus \sigma_{n-1})$ où $\sigma = (\sigma_0, \ldots, \sigma_{n-1})$ est un n-uplets de bits et le symbole $\oplus$ représente le « ou exclusif » entre deux bits [2].

(a) Écrire une fonction `oplus` qui prend en entrée deux bits et renvoie leur « ou exclusif ».

(b) Déterminer σ tel que $(0, 0, 1, 1) \oplus \sigma = (0, 1, 0, 1)$.

(c) Déterminer, pour tous n-uplets de bits σ et t, les n-uplets $\sigma \oplus \sigma$ et $t \oplus \sigma \oplus \sigma$.

(d) Écrire une fonction `substitution` qui reçoit en entrée deux n-uplets de bits t et σ de même taille et renvoie le n-uplet $t \oplus \sigma$.

Question 1.3. Le schéma de Feistel L'objectif de cette question est de construire un système chiffrement par réseau de substitution-permutation suivant un schéma de Feistel. Un tel schéma opère sur une paire d'octets [3] (L, R) suivant le protocole de chiffrement à n tours défini de la manière suivante :

$$\begin{cases} (L_0, R_0) & = (L, R) \\ (L_{i+1}, R_{i+1}) & = (R_i, L_i \oplus F(k_i, R_i)) \text{ pour } 0 \leq i < n \end{cases}$$

1. Pour *Substitution Permutation Network* en anglais.
2. Rappel : $0 \oplus 0 = 1 \oplus 1 = 0$ et $0 \oplus 1 = 1 \oplus 0 = 1$.
3. Un octet est un 8-uplet, c'est-à-dire un uplet de 8 bits.

où k_i est une partie de la clé de chiffrement et F est une fonction que l'on précisera. Un *tour* de ce schéma de chiffrement est représenté par la figure 1.

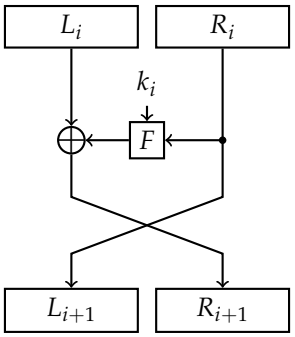

FIGURE 1 – Un tour du schéma de Feistel

Dans la suite, on implémente un schéma à 8 tour avec une clé k de 64 bits :

— k est une liste de 8 octets, que l'on note k_0 à k_7.

— $F(R_i, k_i) = \pi(R_i \oplus k_i)$ où π est la permutation représentée par $(1, 2, 3, 7, 6, 5, 4, 0)$.

(a) Écrire une fonction de chiffrement `feistel` qui reçoit en entrée les octets L, R et la clé k et renvoie le couple d'octets (L_7, R_7).

(b) Pour $i = 0$ à 7, exprimer (L_i, R_i) en fonction de L_{i+1}, R_{i+1} et k_i. Représenter cette relation sous forme d'un schéma du même type que la figure 1.

(c) En déduire une fonction de déchiffrement `feistel_inv` de paramètres L, R et k.

(d) Écrire enfin une fonction `chiffrement_SPN` qui reçoit en entrée une liste d'octets L de taille paire, un booléen b et une clé de 64 bits k et renvoie la liste L chiffrée par le protocole précédent si b vaut `True` ou la liste L déchiffrée si b vaut `False`.

1.2 Le cryptosystème RSA

Le chiffrement RSA (du nom de ses inventeurs R. Rivest, A. Shamir et L. Adleman) est un chiffrement asymétrique dont la clé publique est un couple (N, e) et la clé privée est le couple (N, d) où N, e et d sont de grands entiers (d'au moins 1024 bits) vérifiant $N = pq$ pour deux nombres premiers p et q et $ed = 1 \mod \phi(N)$ où $\phi(N) = (p-1)(q-1)$ est l'indicatrice d'Euler. Les fonctions de chiffrement et de déchiffrement sont respectivement

$$\Phi(m) = m^e \mod N$$

et

$$\Phi^{-1}(c) = c^d \mod N.$$

On admet que $\Phi^-1 \circ \Phi$ *est effectivement l'identité.*

Il s'agit d'un chiffrement par blocs : le message clair est d'abord découpés en blocs de même taille avant d'être chiffrés. Chaque bloc est vu comme un entier dans l'ensemble $\{0, \ldots, N-1\}$.

Question 1.4. Exponentiation modulaire On définit une fonction `exp_mod` qui réalise le calcul de $a^b \mod N$ en décomposant l'exposant en base 2.

```
def exp_mod(a, b, N):
    res = 1
    puiss = a
    while b > 0:
        if b % 2 == 1:
            res *= puiss
```

```
        puiss *= puiss
        b //= 2
    return res % N
```

(a) Déterminer la complexité en temps de `exp_mod` *en supposant que les multiplications et les réductions modulaires se font en temps constant*.

(b) En quoi l'hypothèse de la question précédente est irréaliste ? Comment modifier l'algorithme pour rendre l'hypothèse crédible ?

(c) Écrire une fonction récursive `exp_mod_rec` qui réalise le même calcul et utilise aussi la décomposition de l'exposant en base 2.

Question 1.5. Chiffrement et déchiffrement

(a) À l'aide des fonctions précédentes, écrire une fonction `chiffre_bloc` qui prend en entrée un entier $< N$ et un couple (N, e) et renvoie l'entier chiffré avec la clé publique (N, e).

(b) Écrire de même une fonction `dechiffre_bloc`, qui prend en entrée un entier $< N$ et une clé privée (N, d) et déchiffre l'entier.

(c) En déduire des fonctions `chiffre_RSA` et `dechiffre_RSA` qui effectuent respectivement le même travail que `chiffre_bloc` et `dechiffre_bloc` mais sur des listes d'entiers de longueur quelconque.

(d) Supposons qu'au lieu de manipuler des listes d'entiers, on manipule des entiers de taille quelconque (c'est-à-dire sans la condition « $< N$ »). Proposer une solution pour transformer ces grands entiers en liste d'entiers $< N$, et réciproquement.

Question 1.6. Test de primalité de Miller-Rabin Pour générer des clés RSA, il faut savoir tester si un nombre est premier. Pour cela, on étudie le test de primalité de Miller-Rabin : c'est un algorithme probabiliste de type Monte Carlo qui répond `True` si son entrée est un nombre premier, et qui répond `False` avec bonne probabilité dans le cas contraire.

Il est basé sur le petit théorème de Fermat (pour tout nombre premier p et tout $a \in \{1, \dots, p-1\}$, $a^{p-1} = 1 \mod p$) et sur le fait que *les deux seules racines carrés de* 1 *modulo p sont* 1 *et* $p-1$ (autrement dit, $a^2 = 1 \mod p$ si et seulement si $a = 1 \mod p$ ou $a = p-1 \mod p$).

(a) Montrer que pour tout entier $n > 1$, il existe un unique couple (k, t) tel que $n - 1 = 2^k t$ avec t impair.

(b) En déduire que si n est un nombre premier avec $n - 1 = 2^k t$ où t est impair, alors pour tout $a \in \{1, \dots, n-1\}$, l'une des deux conditions suivantes est vérifiée :

— $a^{2^s t} = 1 \mod n$ pour tout $s \in \{0, \dots, k\}$,

— il existe $s \in \{0, \dots, k-1\}$ tel que $a^{2^s t} = n - 1 \mod n$ et $a^{2^\ell t} = 1 \mod n$ pour $\ell > s$.

Le test de Miller-Rabin est l'algorithme de la figure 2.

(c) Quel est le rôle des lignes 3 à 7 ?

(d) Quelle est la complexité de l'algorithme, en supposant que les opérations arithmétiques se font en temps constant ?

(e) Démontrer la correction de la fonction, c'est-à-dire que si `miller_rabin(n)` renvoie `False` alors n n'est pas premier.

(f) Comme le test de Miller-Rabin est un test probabiliste, l'exécuter plusieurs fois minimise la probabilité de déclarer premier un nombre qui ne l'est pas. Écrire une fonction `test_mr` qui reçoit en entrée deux entiers n et k et renvoie un booléen qui indique si n est premier ou non, après k tests de Miller-Rabin.

```python
def miller_rabin(n):
    a = randint(2, n-2)
    k = 0
    t = n-1
    while t % 2 == 0:
       t //= 2
       k += 1
    b = exp_mod(a, t, n)
    if b == 1 or b == n-1:
        return True
    while k > 1:
        b = b*b % n
        if b == n-1:
            return = True
        k -= 1
    return False
```

FIGURE 2 – Algorithme de Miller-Rabin

Question 1.7. Génération aléatoire d'une clé RSA 1024 bits Pour générer une clé RSA, il faut choisir deux nombres premiers p et q aléatoirement, d'à peu près la même taille, et tels que $N = pq$ possède 1024 bits.

(a) Écrire une fonction `generateur` qui ne reçoit aucun argument et renvoie (N, p, q) tel que p et q sont choisis aléatoirement, premiers et de même ordre de grandeur, et la représentation binaire de N est exactement de longueur 1024. *Justifier les choix effectués.*

(b) Soit P_n le nombre de nombres premiers inférieurs ou égaux à n. On sait que $P_n \sim n/\ln n$ quand n tend vers $+\infty$. En déduire une estimation du nombre moyen d'essais nécessaires pour trouver un nombre premier lorsqu'on tire des entiers de b bits aléatoirement. *On utilisera l'approximation* $\ln 2 \simeq 0,7$.