Projet : Blockchain appliquée à un processus éléctoral

Nous considérons dans ce projet l'organisation d'un processus électoral par scrutin uninominal majoritaire à deux tours (comme ici en France).

- Partie 1 : Implémentation d'outils de cryptographie.
- Partie 2 : Création d'un système de déclarations sécurisés par chiffrement asymétrique.
- Partie 3 : Manipulation d'une base centralisée de déclarations.
- Partie 4 : Implémentation d'un mécanisme de consensus.
- Partie 5 : Manipulation d'une base décentralisée de déclarations.

Chaque partie est divisée en exercices, qui vont vous permettre de concevoir progressivement le programme final. Il est impératif de travailler régulièrement afin de ne pas prendre de retard et de pouvoir profiter des séances correspondantes pour chaque partie du projet.

Cadre du projet

Dans ce projet, nous considérons la problématique de désignation du vainqueur d'un processus électoral. Dans un processus électoral, chaque participant peut déclarer sa candidature au scrutin et/ou donner sa voix à un candidat déclaré. La tenue d'un processus électoral a, depuis toujours, posé des questions de confiance et de transparence épineuses, dans la mesure où les élections sont généralement organisées par le système exécutif en place, qui est souvent candidat à sa réélection et donc soupçonné d'interférences. De plus, le compte des voix fait appel à des assesseurs, ce qui en fait un travail long et avec peu de garanties de fiabilité, dans la mesure où tout le monde ne peut pas vérifier que le compte a eu lieu dans des conditions honnêtes. Enfin, le caractère anonyme de la désignation par bulletin fait que personne ne peut vérifier a posteriori que sa voix a été comptabilisée chez le bon candidat.

Un autre aspect à considérer est celui de l'ergonomie pour le votant. Plus précisément, un système décentralisé permettrait un vote à distance, et le vote à distance (tout comme le vote par correspondance) a longtemps été envisagé comme un outil pour combattre l'abstention, qui a atteint un record historique lors des des élections régionales et départementales de juin 2021 (66,7 % et allant jusqu'à 87 % chez les jeunes de moins de 25 ans). Un projet de loi a par ailleurs été déposé à l'assemblée nationale le 21 septembre 2021, qui permettrait le vote par correspondance. Ce projet de loi serait un premier pas vers le vote électronique, avec comme argument principal le fait que le vote postal a été instauré en Outre-Rhin en 1957, justifiant possiblement l'écart entre le taux d'abstention en France et celui de l'Allemagne (qui ne dépasse pas les 33%).

L'objectif de ce projet est donc de proposer une piste de réflexion sur les protocoles et sur les structures de données à mettre en place pour permettre d'implémenter efficacement le processus de désignation du vainqueur de l'élection, tout en garantissant l'intégrité, la sécurité et la transparence de l'élection.

Développement d'outils cryptographiques

Dans cette partie, nous allons développer des fonctions permettant de chiffrer un message de façon asymétrique. La cryptographie asymétrique est une cryptographie qui fait intervenir deux clés :

— Une clé publique que l'on transmet à l'envoyeur et qui lui permet de chiffrer son message.
— Une clé secrète (ou privée) qui permet de déchiffrer les messages à la réception.

Par exemple, Bob souhaite envoyer un message à Alice. Pour cela, il utilise la clé publique d'Alice pour chiffrer le message avant de lui envoyer. Une fois reçu, Alice peut utiliser sa clé secrète pour déchiffrer le message. Ces clés peuvent aussi servir à signer des messages (pour vérifier la provenance). Plus précisément, Alice peut utiliser sa clé secrète pour signer des messages qu'elle envoie, et sa clé publique permet aux destinataires de vérifier la signature.

L'algorithme de cryptographie asymétrique que nous allons implémenté est le protocole RSA, très utilisé actuellement sur internet pour transmettre des données confidentielles (notamment dans le cadre du e-commerce). Ce protocole s'appuie sur des nombres premiers pour la génération des clés publiques et secrètes. Nous allons donc commencer par traiter le problème de la génération de nombres premiers.

Exercice 1 – Résolution du problème de primalité

Pour générer efficacement des nombres premiers, il faut avoir un moyen d'effectuer rapidement des tests de primalité. Le problème de primalité se définit comme suit : étant donné un entier p impair, p est-il un nombre premier ?

IMPLÉMENTATION PAR UNE MÉTHODE NAÏVE

Un méthode naïve consiste à énumérer tous les entiers entre 3 et $p-1$, et conclure que p est premier si et seulement si aucun de ces entiers ne divise p.

Q 1.1 Implémentez la fonction `int is_prime_naive(long p)` qui, étant donné un entier impair p, renvoie 1 si p est premier et 0 sinon. Quelle est sa complexité en fonction de p ?

Q 1.2 Quel est le plus grand nombre premier que vous arrivez à tester en moins de 2 millièmes de seconde avec cette fonction ?

Pour parvenir à générer de très grands nombres premiers, nécessaires au bon fonctionnement du protocole RSA en pratique, nous allons implémenter un test de primalité plus efficace : le test de primalité de Miller-Rabin. Ce test probabiliste nécessite de pouvoir calculer efficacement une exponentiation modulaire, c'est-à-dire la valeur $a^m \bmod n$, étant donnés trois entiers a, m et n. C'est l'objet des questions suivantes.

EXPONENTIATION MODULAIRE RAPIDE

Pour calculer $a^m \bmod n$, sans passer par le calcul de la valeur a^m (qui peut être très grande), une méthode naïve consiste à itérer les étapes suivantes : on multiplie la valeur courante par a puis on applique le modulo n sur le résultat, avant de passer à l'itération suivante. Il s'agit de répéter ces opérations m fois.

Q 1.3 Implémenter la fonction `long modpow_naive(long a, long m, long n)` qui prend en entrée trois entiers a, m et n, et qui retourne la valeur a^b mod n par la méthode naïve. Quelle est sa complexité ?

Q 1.4 Au lieu de multiplier par a à chaque itération, on peut réaliser des élévations au carré (directement suivies de modulo) pour obtenir un algorithme de complexité logarithmique (c-à-d en $O(log_2(m))$). Donnez le code d'une fonction `int modpow(long a, long m, long n)` réalisant cette succession d'élévation au carré.

Indication : pour une version récursive, il suffit de remarquer que a^b mod n est égal à :
- a mod n quand $m = 0$ (cas de base).
- $b * b$ mod n avec $b = a^{m/2}$ mod n, quand m est pair.
- $a * b * b$ mod n avec $b = a^{\lfloor m/2 \rfloor}$ mod n, quand m est impair.

Attention, à chaque appel, au plus un appel récursif est à effectuer.

Q 1.5 Comparez les performances des deux méthodes d'exponentiation modulaire en traçant des courbes de temps en fonction de m. Qu'observez-vous ?

Pour implémenter le test de Miller-Rabin, nous utiliserons la fonction `modpow` ci-après.

TEST DE MILLER-RABIN

Le test de primalité de Miller-Rabin est un algorithme randomisé qui utilise la propriété suivante. Soit p un nombre impair quelconque. Soient b et d deux entiers tels que $p = 2^b d + 1$. Étant donné un entier a strictement inférieur à p, on dit que a est un témoin de Miller pour p si :
- a^d mod $p \neq 1$,
- et $a^{2^r d}$ mod $p \neq -1$ pour tout $r \in \{0, 1, \ldots, b-1\}$.

Si a est un témoin de Miller pour p, alors il est possible de prouver que p n'est pas premier. Malheureusement, dans le cas contraire, on ne peut pas dire que p est premier. Par contre, si on répète ce test suffisamment de fois, pour des valeurs de a tirées au hasard entre 1 et $p-1$, et qu'aucune de ces valeurs générées ne correspond à un témoin de Miller pour p, alors on peut dire que p est très probablement premier.

Ci-dessous, vous trouverez le code des fonctions :
- `int witness(long a, long b, long d, long p)` qui teste si a est un témoin de Miller pour p, pour un entier a donné.
- `long rand_long(long low, long up)` qui retourne un entier `long` généré aléatoirement entre `low` et `up` inclus.
- `int is_prime_miller(long p, int k)` qui réalise le test de Miller-Rabin en générant k valeurs de a au hasard, et en testant si chaque valeur de a est un témoin de Miller pour p. La fonction retourne 0 dès qu'un témoin de Miller est trouvé (p n'est pas premier), et retourne 1 si aucun témoin de Miller n'a été trouvé (p est très probablement premier).

```
1
2  int witness(long a, long b, long d, long p) {
3      long x = modpow(a,d,p);
4      if(x == 1){
5          return 0;
6      }
7      for(long i = 0; i< b; i++){
8          if(x == p-1){
```

```
            return 0;
        }
        x = modpow(x,2,p);
    }
    return 1;
}

long rand_long(long low, long up){
    return rand() % (up - low +1)+low;
}

int is_prime_miller(long p, int k) {
    if (p == 2) {
        return 1;
    }
    if (!(p & 1) || p <= 1) { //on verifie que p est impair et different de 1
        return 0;
    }
    //on determine b et d :
    long b = 0;
    long d = p - 1;
    while (!(d & 1)){ //tant que d n'est pas impair
        d = d/2;
        b=b+1;
    }
    // On genere k valeurs pour a, et on teste si c'est un temoin :
    long a;
    int i;
    for(i = 0; i < k; i++){
        a = rand_long(2, p-1);
        if(witness(a,b,d,p)){
            return 0;
        }
    }
    return 1;
}
```

Q 1.6 Intégrez ces fonctions dans votre programme.

On s'intéresse maintenant à la fiabilité du test de Miller-Rabin, autrement dit à sa probabilité d'erreur. L'algorithme fait une erreur quand il déclare qu'un entier p est premier alors qu'il ne l'est pas. Cela se produit quand, pour un entier p non premier, l'algorithme ne trouve pas de témoin de Miller pour p parmi les k valeurs de a générées.

Q 1.7 En utilisant le fait que, pour tout entier p non premier quelconque, au moins $\frac{3}{4}$ des valeurs entre 2 et $p-1$ sont des témoins de Miller pour p, donner une borne supérieure sur la probabilité d'erreur de l'algorithme.

Comme la probabilité d'erreur de cet algorithme devient rapidement très faible quand k augmente, et que sa complexité pire-cas est en $O(k(log_2(p))^3)$, alors il est plus intéressant d'utiliser cet algorithme pour effectuer des tests de primalité que la méthode naïve implémentée au tout début de ce projet. On utilisera donc le test de primalité de Miller-Rabin dans la suite du projet.

GÉNÉRATION DE NOMBRES PREMIERS

On souhaite à présent utiliser le test de Miller-Rabin pour générer des nombres premiers de grande taille, où la taille d'un entier est donnée par son nombre de bits. L'idée est de générer aléatoirement

des entiers de la bonne taille (avec la fonction `rand_long` définie précédemment), jusqu'à en trouver un qui réussit le test de Miller-Rabin. Le théorème des nombres premiers assure que l'on trouve par cette méthode un nombre premier au bout d'un nombre d'essais raisonnable.

Q 1.8 Écrire une fonction `long random_prime_number(int low_size, int up_size, int k)` qui étant donnés :

- deux entiers `low_size` et `up_size` représentant respectivement la taille minimale et maximale du nombre premier à générer,
- et un entier k représentant le nombre de tests de Miller à réaliser,

retourne un nombre premier de taille comprise entre `low_size` et `up_size`.

Rappel utile : 2^{t-1} est le plus petit entier à t bits, tandis que $2^t - 1$ est le plus grand.

Remarque : Dans le cadre de ce projet, on considérera uniquement des entiers avec au plus 7 bits. Pour pouvoir travailler avec des nombres plus grands, il faudrait arrêter d'utiliser des `long` et passer plutôt sur des nombres de 128 bits qu'on implémenterait avec une librairie de modélisation mathématique.

Exercice 2 – Implémentation du protocole RSA

Le chiffrement RSA, nommé ainsi par les initiales de ses trois inventeurs (Rivest, Shamir et Adleman), est un algorithme de cryptographie asymétrique qui a été décrit en 1977 et breveté en 1983.

GÉNÉRATION D'UNE PAIRE (CLÉ PUBLIQUE, CLÉ SECRÈTE)

Pour pouvoir envoyer des données confidentielles avec le protocole RSA, il faut tout d'abord générer deux clés : une clé publique permettant de chiffrer des messages et une clé secrète pour pouvoir les déchiffrer. Pour que les échanges soient sécurisés, le couple (clé secrète, clé publique) doit être engendré de manière à ce qu'il soit calculatoirement impossible de retrouver la clé secrète à partir de la clé publique. Le fonctionnement du protocole RSA est fondé sur la difficulté de factoriser de grands entiers. Plus précisément, pour générer un couple (clé secrète, clé publique), le protocole RSA a besoin de deux (grands) nombres premiers p et q distincts (générés aléatoirement), et réalise les opérations suivantes :

1. Calculer $n = p \times q$ et $t = (p-1) \times (q-1)$.
2. Générer aléatoirement des entiers s inférieur à t jusqu'à en trouver un tel que $\text{PGCD}(s,t) = 1$.
3. Déterminer u tel que $s \times u \bmod t = 1$.

Le couple $pKey = (s, n)$ constitue alors la clé publique, tandis que le couple $sKey = (u, n)$ forme la clé secrète. Par définition, u est l'inverse de s modulo t (c'est ce qui permettra le déchiffrement).

Pour déterminer rapidement la valeur $\text{PGCD}(s,t)$ et l'entier u vérifiant $s \times u \bmod t = 1$, on peut utiliser l'algorithme d'Euclide étendu. En effet, étant deux nombres entiers s et t, cet algorithme calcule la valeur $\text{PGCD}(s,t)$ et détermine les entiers u et v vérifiant l'équation de Bezout :

$$s \times u + t \times v = \text{PGCD}(s,t) \tag{1}$$

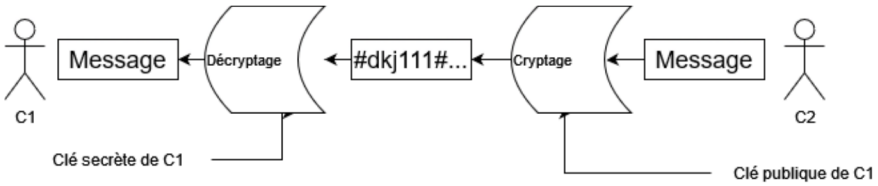
En particulier, quand $\text{PGCD}(s,t) = 1$, on a bien $s \times u \bmod t = 1$. Un version récursive de l'algorithme d'Euclide étendu vous est donné ci-dessous :

```c
long extended_gcd(long s, long t, long *u, long *v){
    if (s == 0){
        *u = 0;
        *v = 1;
        return t;
    }
    long uPrim, vPrim;
    long gcd = extended_gcd(t%s, s, &uPrim, &vPrim);
    *u = vPrim -(t/s)*uPrim;
    *v = uPrim;
    return gcd;
}
```

Q 2.1 Implémentez la fonction `void generate_key_values(long p, long q, long* n, long *s, long *u)` qui permet de générer la clé publique $pkey = (s, n)$ et la clé secrète $skey = (u, n)$, à partir des nombres premiers p et q, en suivant le protocole RSA.

CHIFFREMENT ET DÉCHIFFREMENT DE MESSAGES

On s'intéresse maintenant à l'envoi de message. Supposons que la personne C2 souhaite envoyer un message à la personne C1 en utilisant le protocole RSA. Dans ce cas, la personne C2 utilise la clé publique de la personne C1 pour chiffrer le message avant son envoi. À sa réception, la personne C1 déchiffre le message à l'aide de sa clé secrète.



Soient $pKey = (s, n)$ et $sKey = (u, n)$ la clé publique et la clé secrète du destinataire, et m le message à lui envoyer représenté par un entier (inférieur à n).

- **Chiffrement :** on chiffre le message m en calculant $c = m^s \bmod n$ (c est la représentation chiffrée de m).
- **Déchiffrement :** on déchiffre c pour retrouver m en calculant $m = c^u \bmod n$.

Q 2.2 Implémentez une fonction `long* encrypt(char* chaine, long s, long n)` qui chiffre la chaîne de caractères `chaine` avec la clé publique $pKey = (s, n)$. Pour cela, la fonction convertit chaque caractère en un entier de type `int` (sauf le caractère spécial '\0'), et retourne le tableau de `long` obtenu en chiffrant ces entiers.

Rappel : il faut utiliser la fonction `modpow` pour réaliser une exponentiation modulaire efficace.

Q 2.3 Implémentez une fonction `char* decrypt(long* crypted, int size, long u, long n)` qui déchiffre un message à l'aide de la clé secrète $skey = (u, n)$, en connaissant la taille du tableau d'entiers. Cette fonction renvoie la chaîne de caractères obtenue, sans oublier le caractère spécial '\0' à la fin.

FONCTION DE TESTS

Pour vérifier le bon fonctionnement de votre programme, reproduisez le programme principal suivant et compilez le en utilisant les fonctions que vous avez implémentées dans les questions précédentes.

```c
void print_long_vector(long *result, int size){
   printf("Vector: [");
   for (int i=0; i<size; i++){
      printf("%lx \t", result[i]);
   }
   printf("]\n");
}

int main()
{
    srand(time(NULL));

    //Generation de cle :
    long p = random_prime_number(3,7, 5000);
    long q = random_prime_number(3,7, 5000);
    while(p==q){
        q = random_prime_number(3,7, 5000);
    }
    long n, s, u;
    generate_keys_values(p,q,&n,&s,&u);
    //Pour avoir des cles positives :
    if (u<0){
        long t = (p-1)*(q-1);
        u = u+t; //on aura toujours s*u mod t = 1
    }

    //Afichage des cles en hexadecimal
    printf("cle publique = (%lx , %lx) \n",s,n);
    printf("cle privee = (%lx , %lx) \n",u,n);

    //Chiffrement:
    char mess[10] = "Hello";
    int len = strlen(mess);
    long* crypted = encrypt(mess, s, n);

    printf("Initial message: %s \n", mess);
    printf("Encoded representation : \n");
    print_long_vector(crypted, len);

    //Dechiffrement
    char* decoded = decrypt(crypted, len, u, n);
    printf("Decoded: %s\n", decoded);

    return 0;
}
```

Déclarations sécurisées

Revenons à notre problème de vote. Dans ce problème, un citoyen interagit pendant les élections en effectuant des *déclarations*. En pratique, ces déclarations peuvent soit être des déclarations de candidature soit des déclarations de vote. Pour simplifier le projet, on va supposer que l'ensemble des candidats est déjà connu, et que les citoyens ont juste à soumettre des déclarations de vote.

Exercice 3 – Manipulations de structures sécurisées

Dans notre modèle, chaque citoyen possède une carte électorale, qui est définie par un couple de clés :

- Une clé *secrète* (ou privée) qu'il utilise pour signer sa déclaration de vote. Cette clé ne doit être connue que par lui.
- Une clé *publique* permettant aux autres citoyens d'attester de l'authenticité de sa déclaration (vérification de la signature). Cette clé est aussi utilisée pour l'identifier dans une déclaration de vote, non seulement quand il vote, mais aussi quand quelqu'un souhaite voter en sa faveur.

Dans cet exercice, nous verrons que signer une déclaration se fait par chiffrement du contenu (avec la clé secrète), puis vérifier une signature se fait simplement par déchiffrement (avec la clé publique).

MANIPULATION DE CLÉS

On rappelle que, dans le protocole RSA, la clé publique et la clé secrète d'un individu sont des couples d'entiers, notés respectivement $pKey = (s, n)$ et $sKey = (u, n)$ dans l'exercice précédent.

Q 3.1 Définissez une structure `Key` qui contient deux `long` représentant une clé (publique ou secrète).

Q 3.2 Écrivez une fonction `void init_key(Key* key, long val, long n)` permettant d'initialiser une clé déjà allouée.

Q 3.3 Écrivez une fonction `void init_pair_keys(Key* pKey, Key* sKey, long low_size, long up_size)` qui utilise le protocole RSA pour initialiser une clé publique et une clé secrète (déjà allouées).

Indication : Cette fonction fait appel aux fonctions `random_prime_number`, `generate_keys_values` et `init_key`. Inspirez-vous des lignes 14 à 25 de la fonction `main` de l'exercice précédent.

Q 3.4 Écrire deux fonctions `char* key_to_str(Key* key)` et `Key* str_to_key(char* str)` qui permettent de passer d'une variable de type `Key` à sa représentation sous forme de chaîne de caractères et inversement. La chaîne de caractères doit être de la forme `"(x,y)"`, où `x` et `y` sont les deux entiers de la clé exprimés en hexadécimal.

Indication : les fonctions `sprintf` et `sscanf`, et le spécificateur `%lx` peuvent vous être utiles.

Remarque : l'hexadécimal est un système très souvent utilisé en informatique, notamment parce qu'il permet une conversion sans aucun calcul avec le système binaire (système employé par nos ordinateurs), et qu'il possède l'avantage de rendre des entiers très grands plus compactes et plus lisibles.

SIGNATURE

Dans notre contexte, une déclaration de vote consiste simplement à transmettre la clé publique du candidat sur qui porte le vote. Dans un processus de scrutin, il faut que chaque personne puisse produire

des déclarations de vote *signée* pour attester de l'authenticité de la déclaration. Cette signature consiste en un tableau de `long` qui ne peut être généré que par l'émetteur de la déclaration (avec sa clé secrète), mais qui peut être vérifié par n'importe qui (avec la clé publique de l'émetteur). Plus précisément, nous allons utilisé le protocole de déclaration de vote suivant :

1. Un électeur E souhaite voter pour le candidat C. Dans ce cas, la déclaration de vote de E est le message `mess` obtenu en transformant la clé publique du candidat C en sa représentation sous forme de chaîne de caractères (obtenu par la fonction `key_to_str`).
2. Avant de publier sa déclaration, l'électeur E utilise une fonction de signature (appelée `sign` ci-après) qui permet de générer la signature associée à sa déclaration de vote. Cette signature prendra la forme d'un tableau de `long` obtenu par chiffrement du message `mess` avec la clé secrète de l'électeur E (à l'aide de la fonction `encrypt`).
3. L'électeur peut ensuite publier une déclaration sécurisée, composée de sa déclaration `mess`, de la signature associée, et de sa clé publique. De cette manière, toute personne souhaitant vérifier l'authenticité de la déclaration peut le faire en déchiffrant la signature avec la clé publique de E : le résultat obtenu doit correspondre exactement au message `mess`.

Q 3.5 Une signature est donc simplement un tableau de `long` dont on connaît la longueur. Définissez la structure `Signature`.

Q 3.6 Écrivez une fonction `Signature* init_signature(long* content, int size)` qui permet d'allouer et de remplir une signature avec un tableau de `long` déjà alloué et initialisé.

Q 3.7 Écrivez une fonction `Signature* sign(char* mess, Key* sKey)` qui crée une signature à partir du message `mess` (déclaration de vote) et de la clé secrète de l'émetteur.

Q 3.8 On vous donne ci-dessous les fonctions `signature_to_str` et `str_to_signature`, qui permettent de passer d'une `Signature` à sa représentation sous forme de chaîne de caractères et inversement. La chaîne de caractères est de la forme "$\#x_0\#x_1\#\ldots\#x_n\#$" où x_i est le $i^{\text{ème}}$ entier du tableau de la signature donné en hexadécimal. Intégrez ces fonctions dans votre programme.

```
1  char* signature_to_str(Signature* sgn){
2      char* result = malloc(10*sgn->size*sizeof(char));
3      result[0]='#';
4      int pos = 1;
5      char buffer[156];
6      for (int i=0; i<sgn->size; i++){
7          sprintf(buffer, "%lx", sgn->content[i]);
8          for (int j=0; j< strlen(buffer); j++){
9              result[pos] = buffer[j];
10             pos = pos+1;
11         }
12         result[pos] = '#';
13         pos = pos+1;
14     }
15     result[pos] = '\0';
16     result = realloc(result, (pos+1)*sizeof(char));
17     return result;
18 }
19
20 Signature* str_to_signature(char* str){
21     int len = strlen(str);
22     long* content = (long*)malloc(sizeof(long)*len);
23     int num = 0;
24     char buffer[256];
```

```
    int pos = 0;
    for (int i=0; i<len; i++){
        if (str[i] != '#'){
            buffer[pos] = str[i];
            pos=pos+1;
        }else{
            if (pos != 0){
                buffer[pos] = '\0';
                sscanf(buffer, "%lx", &(content[num]));
                num = num + 1;
                pos = 0;
            }
        }
    }
    content=realloc(content,num*sizeof(long));
    return init_signature(content, num);
}
```

DÉCLARATIONS SIGNÉES

On peut maintenant créer des déclarations signées (données protégées).

Q 3.9 Définissez la structure `Protected` qui contient la clé publique de l'émetteur (l'électeur), son message (sa déclaration de vote), et la signature associée.

Q 3.10 Écrivez une fonction `Protected* init_protected(Key* pKey, char* mess, Signature* sgn)` qui alloue et initialise cette structure (cette fonction ne vérifie pas si la signature est valide).

Q 3.11 Écrivez une fonction `int verify(Protected* pr)` qui vérifie que la signature contenue dans `pr` correspond bien au message et à la personne contenus dans `pr`.

Q 3.12 Écrivez les fonctions `protected_to_str` et `str_to_protected` qui permettent de passer d'un `Protected` à sa représentation sous forme de chaîne de caractères et inversement. La chaîne doit contenir dans l'ordre :

- la clé publique de l'émetteur,
- son message,
- sa signature,

séparés par un espace.

Indication : utilisez les fonctions `key_to_str` et `signature_to_str`.

FONCTION DE TESTS

Pour tester vos fonctions, vous pouvez vous inspirer de ce main.

```
int main (void) {

    srand(time(NULL));

    //Testing Init Keys
    Key* pKey = malloc(sizeof(Key));
    Key* sKey = malloc(sizeof(Key));
    init_pair_keys(pKey, sKey,3,7);
    printf("pKey: %lx, %lx \n", pKey->val, pKey->n);
    printf("sKey: %lx, %lx \n", sKey->val, sKey->n);

    //Testing Key Serialization
```

```
13      char* chaine = key_to_str(pKey);
14      printf("key_to_str: %s \n", chaine);
15      Key* k = str_to_key(chaine);
16      printf("str_to_key: %lx , %lx \n", k->val, k->n);
17
18      //Testing signature
19      //Candidate keys:
20      Key* pKeyC = malloc(sizeof(Key));
21      Key* sKeyC = malloc(sizeof(Key));
22      init_pair_keys(pKeyC, sKeyC,3,7);
23      //Declaration:
24      char* mess = key_to_str(pKeyC);
25      printf("%s vote pour %s\n",key_to_str(pKey), mess);
26      Signature* sgn = sign(mess, sKey);
27      printf("signature: ");
28      print_long_vector(sgn->content,sgn->size);
29      chaine = signature_to_str(sgn);
30      printf("signature_to_str: %s \n", chaine);
31      sgn = str_to_signature(chaine);
32      printf("str_to_signature: ");
33      print_long_vector(sgn->content,sgn->size);
34
35
36      //Testing protected:
37      Protected* pr = init_protected(pKey,mess, sgn);
38      //Verification:
39      if (verify(pr)){
40          printf("Signature valide\n");
41      }else{
42          printf("Signature non valide\n");
43      }
44      chaine = protected_to_str(pr);
45      printf("protected_to_str: %s\n", chaine);
46      pr = str_to_protected(chaine);
47      printf("str_to_protected: %s %s %s\n",key_to_str(pr->pKey),pr->mess,
            signature_to_str(pr->sgn));
48
49      free(pKey);
50      free(sKey);
51      free(pKeyC);
52      free(sKeyC);
53      return 0;
54  }
```

Exercice 4 – Création de données pour simuler le processus de vote

Pour pouvoir mettre en place le scrutin, il faut d'abord générer pour chaque citoyen une carte électorale unique, comprenant sa clé publique et sa clé secrète, et recenser toutes les clés publiques de ces cartes électorales. Pour que le vote soit anonyme, le système ne doit pas savoir à qui correspondent ces clés publiques. Par ailleurs, les citoyens ont la responsabilité de garder leur clé secrète, et doivent l'utiliser au moment de voter, pour transmettre une déclaration de vote signée. Le système de vote collecte les déclarations signées au fur et à mesure qu'elles arrivent, et vérifie l'authenticité de chaque vote avant de l'enregistrer dans sa base de données.

Dans le cadre de ce projet, nous allons simuler ce processus de vote à l'aide d'une base de données de citoyens, de candidats et de déclarations signées.

Q 4.1 Écrivez une fonction `void generate_random_data(int` nv`, int` nc`)` qui :
- génère nv couples de clés (publique, secrète) différents représentant les nv citoyens,
- crée un fichier `keys.txt` contenant tous ces couples de clés (un couple par ligne),
- sélectionne nc clés publiques aléatoirement pour définir les nc candidats,
- crée un fichier `candidates.txt` contenant la clé publique de tous les candidats (une clé publique par ligne),
- génère une déclaration de vote signée pour chaque citoyen (candidat choisi aléatoirement),
- crée un fichier `declarations.txt` contenant toutes les déclarations signées (une déclaration par ligne).