PROJET DE LU2IN006
BLOCKCHAIN ET ÉLECTIONS

POIRET Bastien 28000789
DIAB Dana 28705848

Introduction :

Ceci est la première partie de notre compte rendu de projet de l'enseignement de structures de données. Nous y implémentons les outils de cryptographie à clé publique (RSA) nécessaires à la création d'une structure Blockchain permettant d'effectuer une élection fiable. Nous traitons ensuite des déclarations sécurisées, puis des structures chaînées de la base de données.

Séance 1 : Développement des outils cryptographiques

Dans cette première séance, nous avons implémenté un algorithme d'exponentiation modulaire rapide permettant de réaliser des tests de miller rabin afin de générer de grands nombres premiers. Nous avons ensuite implémenté un protocole RSA consistant en une génération de clé publique et secrète permettant de crypter ou décrypter un message.

Fichiers dédiés à la séance 1 :

- `seance_1.c` : Code source des fonctions.
- `seance_1.h` : Fichier header contenant les déclarations de fonctions et de structures
- `main_seance_1.c` : Main étudiant produisant des courbes comparant les fonctions naïves et rapides d'exponentiation modulaire.
- `test_main.c` : main fourni dans le sujet du projet permettant de tester l'encodage et décodage en RSA.

Structures manipulées :

On se contente ici de manipuler des entiers long, int et des chaînes de caractères.

Fonctions principales :

- `long modpow(long a, long m,long n)` et sa version récursive `long modpow_rec(long a, long m,long n)` : Servent à calculer l'exponentiation modulaire rapide, l'algorithme fonctionne par dichotomie sur l'exposant.
- `int witness ( long a , long b , long d , long p )` : pour p = 2bd + 1, retourne 1 si a est témoin de miller de p et 0 sinon.
- `int is_prime_miller ( long p , int k )` : vérifie par k tests de miller-rabin si p est un nombre premier. Avec une probabilité de (¼)^k qu'il ne le soit pas. Très peu de tests sont donc nécessaires pour obtenir une probabilité très forte que le nombre soit vraiment premier lorsque la fonction retourne 1.

- `long random_prime_number(int low_size, int up_size, int k )` : Retourne un nombre premier compris entre `low_size` et `up_size` et `k` correspond au nombre de tests de Miller-Rabin à réaliser.
- `long extended_gcd(long s, long t, long *u, long *v)` Algorithme d'euclide étendu fourni qui retourne le PGCD et stocke les coefficients de Bezout.
- `void generate_key_values(long p, long q, long *n, long *s, long *u)` Génère le trio de valeurs `n`, `s` et `u`, les couples `(s,n)` formant la clé secrète et `(u,n)` la clé publique.
- `long *encrypt(char *chaine, long s, long n)` crypte la chaîne fournie grâce à la clé fournie, retourne un tableau de `long` chaque case correspondant au cryptage d'un caractère de la chaîne.
- `char* decrypt(long *crypted, int size, long u, long n)` Décrypte un tableau de `long` et retourne la chaîne obtenue.

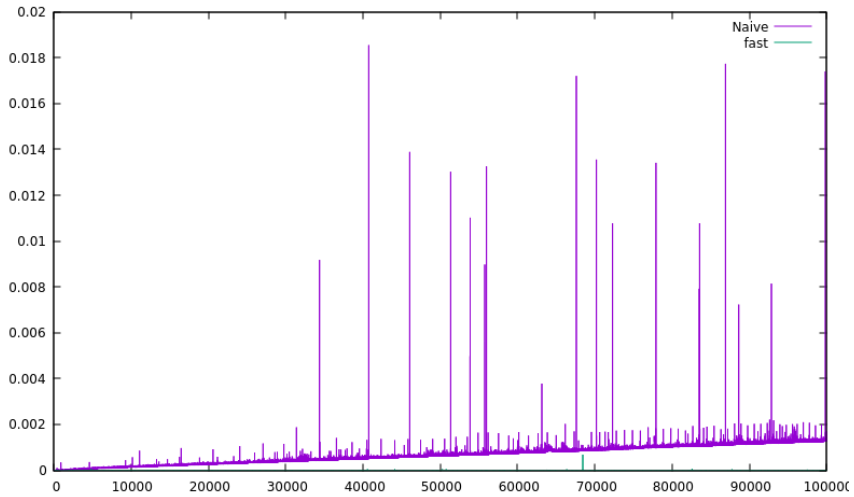
Réponse aux Questions :

Q1.1 - La complexité de la fonction `is_prime_naive` est de complexité `n`, `n` étant la valeur fourni en paramètre.

Q1.2 - Avec la fonction `naive_is_prime`, on arrive vite à des valeurs très longues à calculer. Quand on atteint n=1000000000, on mesure en secondes le temps pour calculer si un nombre premier, ce qui n'est pas étonnant au vu de la fréquence du processeur.

Q1.3 - La complexité de `modpow_naive` est `m`, `m` étant la deuxième valeur fourni en paramètre.

Q1.5 - Les courbes de ces deux fonctions montrent clairement la complexité pire cas de chacune d'elles (m et log(m)).



Q1.7 - Dans le pire des cas, on ne teste l'existence d'un témoin qu'une seule fois, donc k=1. Et comme, au moins 3/4 des valeurs entre 2 et p-1 sont témoins de p, alors au plus 1/4 ne le

sont pas. Donc la borne supérieur sur la probabilité d'erreur que p soit premier alors qu'il ne l'est pas est (1/4)^k = 1/4 pour k=1

Description des tests :

Tous les tests de *main* ont été réalisés avec valgrind les fonctions fournies ne provoquent donc en principe aucune fuite mémoire. Le fichier *test_main.c* utilise les fonctions de cryptographie pour créer un couple de clé, encode un message avec *encrypt* et le décrypte avec *decrypt*. Le test est concluant. D'autres tests non fournis ont été menés avec différentes valeurs de *low_size* et *up_size* dans *random_prime_number* et différents messages.

Séance 2 : Déclarations sécurisées

La seconde séance est consacrée à la formulation de déclarations sécurisées. Nous y définissons toutes les structures nécessaires à l'implémentation du système de vote. Une déclaration de vote sera signée grâce à la clé RSA secrète d'un électeur. Il sera ensuite possible de certifier son authenticité via un décryptage utilisant la clé publique de ce dernier. Nous finirons par créer un jeu aléatoire de données afin de simuler une session de vote.

Fichiers dédiés à la séance 2 :

- *seance_2.c* : Code source des fonctions.
- *seance_2.h* : Déclarations des structures utilisées et prototypes de fonctions
- *main_seance_2.c* : Main testant l'ensemble des fonctionnalités importantes de la séance 2. Ce main doit être exécuté car il génère, sous forme de fichiers textes, le jeu de données utilisé dans la séance 3.
- *test_main_2.c* : Ce main correspond à celui fourni par les enseignants. Il sert à tester les diverses fonctions de sérialisation.

Structures manipulées :

```
typedef struct key
{
      long k; //clé
      long n; //modulo
}Key;
```

Structure du type *Key* servant à stocker un couple (k,n), formant une clé publique ou privée de la forme de celle générée à la séance 1 par *generate_key_values.*

```
typedef struct signature
{
      int size;      //taille du message
```

```
    long *content; //contenu crypté
}Signature;
```

Structure du type `Signature` stockant la taille en caractères ainsi que la signature d'un message crypté par `encrypt` (tableau de `long`).

```
typedef struct protected
{
    Key* pkey;       //clé publique de l'émetteur
    char * mess;     //message en clair
    Signature* sign; //signature du message
}Protected;
```

Structure du type Protected stockant une déclaration signée sous forme d'une clé publique de votant, du message correspondant a la déclaration de vote et de la signature de ce dernier.

Fonctions Principales :

Les trois structures ci-dessus disposent chacune d'une fonction d'initialisation, de sérialisation et de désérialisation.

De plus :

- `void init_pair_keys(Key* pk, Key* sk, long low_size, long up_size)` Génère une paire de clé publique et privée a l'aide des outils de la séance 1 et les stocke dans deux structures de type `Key` déjà allouées données en argument.
- `Signature* sign(char* mess, Key* sKey)` Prend une chaîne correspondant à un message et la crypte utilisant la clé secrète fournie. Cette fonction utilise `init_signature` effectuant une allocation.
- `int verify(Protected *pr)` Retourne 1 si la structure `Protected` fournie est authentique, c'est-à-dire si le message retrouvé par décryptage de la signature `pr->sign` avec la clé publique `pr->pkey` donne le même message que celui stocké dans `pr->mess`. Retourne 0 sinon.
- `void generate_random_data(int nv, int nc)` fonction plus conséquente qui génère une série de données. Elle crée un fichier `candidates.txt`, `declarations.txt` et `keys.txt` en utilisant les fonctions de sérialisation précédentes pour former des chaînes de caractères. `nv` est le nombres d'individus et `nc` le nombre de candidats qui seront sélectionnés aléatoirement parmi les votants. Les votes seront ensuite effectués aléatoirement également.

Description des tests :

Le fichier `test_main_2.c` teste les fonctions de sérialisation dans les deux sens et ne produit aucune fuite mémoire. Le fichier `main_seance_2.c` teste principalement `generate_random_data` (fonction qui utilise toutes les fonctionnalités précédemment

implémentées dans cette séance), et ne produit aucune fuite. Il fournit des fichiers ayant le format attendu qui ont pu être utilisés pour la suite.

Séance 3 : Base de déclarations centralisées

La dernière séance de ce rendu de mi-projet est consacrée à la création de listes chaînées à partir des fichiers créés précédemment par la fonction *generate_random_data*. On se servira ensuite de la liste chaînée des déclarations signées pour implémenter un algorithme qui en supprime les déclarations frauduleuses.

Fichiers dédiés à la séance 3 :

- *seance_3.c* : code source des fonctions
- *seance_3.h* : déclaration des structures chaînées et prototypes des fonctions
- *main_seance_3.c* : Test

Structures manipulées :

```
typedef struct cellKey
{
      Key* data;
      struct cellKey* next;
}CellKey;
```

structure d'un élément de liste de clés.

```
typedef struct cellProtected
{
      Protected * data;
      struct cellProtected * next;
}CellProtected;
```

structure d'un élément de liste de déclaration signée.

Fonctions Principales :

Chacune des deux structures ci-dessus dispose d'une fonction de création d'élément, d'une fonction d'ajout en tête, d'une fonction d'affichage de liste, d'une fonction de création de liste à partir des fichiers (*candidates.txt, keys.txt ou declarations.txt*), de suppression d'élément et de suppression de liste complète.

De plus :

- *void delete_non_valide(CellProtected ** LCP)* Fonction qui parcourt la liste de *Protected* fournie et qui supprime toutes celles qui sont non valides, c'est à -dire qui ne passent pas le test de la fonction *verify*.

Description des tests :

Le fichier *main_seance_3.c* lit le fichier *declarations.txt* et *keys.txt*, affiche les listes créées. Il supprime ensuite les déclarations non valides de la liste de déclarations. Il ne produit pas de fuite mémoire lors d'un test avec *valgrind* et des tests avec des déclarations frauduleuses ont été concluants, elles ont bien été supprimées.

Conclusion de mi-parcours :

Les algorithmes d'exponentiation modulaire et les outils de cryptographie mis à part, la plupart des fonctions implémentées ont des boucles simples et une complexité en O(1) ou linéaire. Nous n'avons pas jugé pertinent, étant donné le temps déjà investi dans la programmation, de développer l'analyse plus loin que dans les questions explicitement posées dans le sujet, principalement à la séance 1.

C'est un projet intéressant et instructif, la réussite des exercices est gratifiante. La cryptographie intéresse, je pense, beaucoup d'entre nous.

Merci donc pour ce sujet.