Projet : Blockchain appliquée à un processus électoral
Rapport du mi-projet

Description globale du code

Le projet se divise en 3 parties (pour l'instant):

1. Implémentation d'outils de cryptographie
2. Création d'un système de déclarations sécurisés par chiffrement asymétrique
3. Manipulation d'une base centralisé de déclarations.

Partie 1 : Implémentation d'outils de cryptographie

Le but de cette partie est de générer efficacement de très grand nombre premiers qui nous serviront à créer des paires de clé publique et secrète selon le protocole RSA. A l'aide de la publique nous pourrons crypter un message puis le décrypter à l'aide de la clé privé

Fichiers

prime_number.c: Fichier contenant les méthodes servant à la génération de nombre premiers
prime_number.h: Fichiers contenant les prototypes des fonctions de prime_number.c

chiffrement.c :Fichier contenant les méthodes servant à la génération de paire de clés
chiffrement.h :Fichiers contenant les prototypes des fonctions de chiffrement.c

Makefile: fichier permettant la compilation des différents fichiers.c
main.c: fichier principal contenant les tests des différents fichiers

Partie 2 :Création d'un système de déclarations sécurisés par chiffrement asymétrique

Le but de cette partie est de mettre en place les déclarations de vote des électeurs. Chaque électeur possède une carte électorale définie par un couple de clé secrète et privée, pour voter il doit publier une déclaration sécurisée.Celle ci est composé:

- *Un message, c'est la clé du candidat pour qui il vote, transformée en chaîne de caractère*
- *Une signature, c'est le message chiffré à l'aide de la clé secrète de l'électeur*
- *La clé publique de l'électeur*

Structure

Key : Structure représentant une clé (public ou secrète) contenant deux longs correspondant à ses valeurs.
Signature: Structure représentant un message(la déclaration de vote) chiffré, elle contient un tableau de long(le message chiffré) dont on connaît la longueur
Protected : Structure contenant la clé publique de l'émetteur, son message(la déclaration de vote) et la signature associé.

Fichiers

key.c :Fichier contenant les méthodes servant à la création de clé...
key.h :Fichiers contenant les prototypes des fonctions de key.c

Makefile: fichier permettant la compilation des différents fichiers.c
main.c: fichier principal contenant les tests des différents fichiers

Partie 3 :Manipulation d'une base centralisé de déclarations.

Structure
cellKey: Liste chainée de clés

Description des fonctions principales

Partie 1 : Implémentation d'outils de cryptographie

Génération de nombre premiers

- **is_prime_naive(long)**:Renvoie 1 si p est premier sinon renvoie 0. Pour cela parcourt les entiers de 3 à p-1 inclus et test si p est divisible par l'entier. Si oui p n'est pas premier

- **modpow_naive(long a, long m, long n)** : Renvoie $(a^m \bmod n)$ en itérant m fois le résultat multiplié par a auquel on applique le modulo n

- **modpow(long a, long m, long n) :** Renvoie $(a^m \bmod n)$ en calculant le résultat récursivement par disjonction de cas en fonction de la parité de m :
  $(a^m \bmod n) = 1$ quand $m = 0$ (cas de base)
  $(a^m \bmod n) = (b * b \bmod n)$ avec $b = a\ m/2 \bmod n$, quand m est pair
  $(a^m \bmod n) = a * b * b \bmod n$ avec $b = a\ \lfloor m/2 \rfloor \bmod n$, quand m est impair

- **witness(long a, long b, long d, long p) :** Retourne 1 si a est un témoin de Miller pour p, (fait appelle à **modpow()**)

- **rand_long(long low, long up) :** Génère un long compris entre low et up inclus

- **is_prime_miller(long p, int k) :** Réalise le test de Miller-Rabin en générant k valeurs de a au hasard (**rand_long()**), et teste si chaque valeur de a est un témoin de Miller pour p (**witness**()). 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).

- **random_prime_number(int low_size, int up_size, int k) :**Retourne un nombre premier aléatoire de taille comprise entre low_size et up_size (**rand_long()**) en effectuant k tests de Miller pour tester la primalité du nombre généré (**is_prime_miller()**).

Génération d'un paire de clé

- **extended_gcd(long s, long t, long *u, long *v) :** version récursive de l'algorithme d'Euclide, Retourne le PGCD(s,t) et affecte à u et v les valeur vérifiant l'équation de Bézout : s*u +t*v =PGCD(s,t)

- **generate_key_values(long p, long q, long *n, long *s, long *u) :**génère les clés publiques pkey(s,n) et privées skey(u,n) à partir des nombres premier p et q en mettant à jours les pointeurs n, s et u (en arguments) grâce à la fonction **extended_gcd()** définie précédemment.

- **encrypt(char * chaine, long s, long n) :** chiffre la chaine en argument caractère par caractère ($c = m^s$)

- **decrypt(long* crypted, int size, long u, long n) :**déchiffre l'entier long en argument chiffre par chiffre ($m = c^u$)

<u>Partie 2 :Création d'un système de déclarations sécurisés par chiffrement asymétrique</u>

Manipulation de clés

- **void init key(Key* key, long val, long n):**Initialise une clé key avec les valeurs n et val

- **void_init_pair_keys(Key* pKey, Key* sKey, long low_size, long up_size) :**Initialise un couple de clé en générant deux nombres premiers de taille entre low_size et upe_size (**random_prime_number()**). A partir de ces nombres les valeurs de clés sont générés (**generate_key_values()**), enfin la clé public pKey est initialisé avec les valeurs (s,n) et la clé privé sKey est initialisé avec les valeurs (u,n).

- **key_to_str(Key* key):** Retourne la chaîne de caractère représentant la clé, en créant un tableau de char dans lequel on stocke les valeurs de la clé(**sprintf()**)

- **str_to_key(char* str):**Retourne la clé représenté par la chaîne de caractère passé en argument.Pour cela on crée une variable de type key, on récupère les valeurs de la chaîne de caractère (**sscanf()**) et on initialise la clé crée (**init_key()**)

Signature

- **init_signature(long* content, int size):**Crée une signature et l'alloue avec la taille et le tableau de long passé en argument

- **sign(char* mess, Key* sKey) :**chiffre le message mess(**encrypt()**) à l'aide de la clé privée passé en argument puis retourne une signature qu'on a initialisé avec mess chiffré et la longueur de mess(**init_signature()**)

- **signature_to_str(signature* sgn) :**Retourne la chaîne de caractère représentant la signature sgn avec la clé publique de l'émetteur, son message et sa signature dans l'ordre

- **str_to_protected(char* str) :**Retourne la signature représenté par la chaîne de caractère passé en argument.

Déclarations signées

- **init_protected(Key* pKey, char* mess, Signature* sgn):**Crée et initialise la structure avec les données en argument

- **verify(protected* pr):**Vérifie si la signature contenue dans la structure protected correspond bien à celle de l'émetteur. Pour cela on récupère la clé public, le message et la signature de pr ensuite on décrypte la signature avec la clé public, si le message déchiffré correspond bien à celui de la structure protected on retourne 1 .

- **protected_to_str(Protected* pr) :**Retourne la chaîne de caractère associé à pr, en créant une tableau de char qu'on initialise (**sprintf()**)avec la clé transformé(**key_to_str()**) le message et la signature (**signature_to_str()**) dans l'ordre

- **str_to_protected(char* str) :**Retourne la structure protected associé a une chaîne de caractère en récupérant ses éléments dans la chaîne (**sscanf()**) à l'aide des fonctions **str_to_key(), strdup()** et **str_to_signature()**, puis initialise la nouvelle structure.

Création de données pour une simulation

- **generate_random_data(int nv, int nc) :**

Partie 3 :Manipulation d'une base centralisé de déclarations.

Liste chaînée de clés

- **create_cell_key(Key* key):**Alloue une cellule d'une liste chaînée de clés avec une clé qui pointe sur NULL
- **add_cell_key(CellKey** cellKey, Key* key) :**Ajoute une clé en tête de liste en vérifiant que la clé key n'est pas déjà en tête de liste, puis crée une cellule y ajoute key et l'ajoute en tête de liste
- **read_public_keys(char* fichier) :**
- **print list keys(CellKey* LCK) :**
- **delete cell key(CellKey* c) :**

Liste chainée de déclarations signées

- **create cell protected(Protected* pr) :**
- **add_cell_protected(CellProtected** cellPro, Protected* pr) :**
- **read protected(char* fic) :**
- **print_list_protected(CellProtected* LCP) :**
- **delete cell protected(CellProtected* c) :**

Réponses aux questions

Partie 1:

1.1 La complexité de **is_prime_naive()** est en O(n).

1.2 Le plus grand nombre généré en moins de 2 millième de secondes est 215 000 363 .

1.3 La complexité de **modpow_naive()** est en theta(n)

1.5

1.7