- LU2IN006 -

Compte-rendu intermédiaire du projet :

Blockchain appliquée à un processus électoral

Sommaire

I. Reformulation du sujet : 2

II. Description globale du code : 3

1. premier.c , premier.h , test_premier.c : 3

2. Deuxieme.c, deuxieme.h, test_deuxieme.c : 4

3. troisième.c, troisieme.h, test_troisieme.c : 5

4. quatrieme.c, quatrieme.h, test_quatrieme.c : 6

III. Description des jeux d’essais : 8

IV. Réponses aux questions : 8

I. Reformulation du sujet :

L’objectif de ce projet est de simuler un processus électoral à distance, dans lequel chaque participant peut déclarer sa candidature ou voter pour un candidat. Pour assurer une élection transparente et sans interférences, les déclarations de vote seront chiffrées en s’appuyant sur un algorithme de cryptographie, le protocole RSA.
Le processus électoral doit conserver sa fonctionnalité d’anonymat ainsi que sa sécurité en cas de fraude. Cela peut être implémenté à travers plusieurs structures de données dont chaqu’un ses avantages et ses inconvénients.

Dans ce compte-rendu, nous allons aborder toutes les parties du projet :

- Partie 1 : Implémentation d’outils de cryptographie.
- Partie 2 : Création d’un système de déclarations sécurisées par chiffrement asymétriques.
- 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.

II. Description globale du code :

Les fichiers de notre projet sont organisés selon leur partie correspondante. Nous avons ainsi trois groupes de fichiers :

1. premier.c , premier.h , test_premier.c :

- Le fichier *premier.c* contient les outils de cryptographie qui servent comme base pour la sécurisation des déclarations de votes.

- Le fichier *premier.h* correspond a son fichier header, et *test_premier.c* contient le main du premier.c, ainsi que les jeux d'essais effectués.

- Les fonctions importants sont :

Fonctions :	**Explications** :
void generate_key_values(long p, long q, long* n, long* s, long* u)	Création des couples de valeurs (s,n) et (u,n) correspondant a la clé publique et secrète du protocole RSA.
long* encrypt(char* chaine, long s, long n)	Chiffrement d'une chaîne, grâce à la clé publique, en un tableau d'entiers.
char* decrypt(long* crypted, int size, long u, long n)	Déchiffrement d'un tableau d'entiers, grâce à la clé secrète, en une chaîne de caractères.

2. Deuxieme.c, deuxieme.h, test_deuxieme.c :

- Le fichier *deuxieme.c* manipule les structures de clé du protocole RSA (Key), de signature (Signature) et de déclaration de vote (Protected). De plus, elle possède des fonctions permettant d'initialiser ces structures, ainsi que de convertir de celles-ci en chaine de caractère et vice versa.

- Le fichier *deuxieme.h* correspond a son fichier header, et contient la définition des structures Key, Signature, et Protected. Le fichier *test_deuxieme.c* contient le main du deuxieme.c, ainsi que les jeux d'essais effectués.

- Les structures utilisées sont :

Key	**Signature**	**Protected**
long val;	int taille;	Key* pKey;
long n;	long* tab;	char* mess;
		Signature* s;

- Dans ce projet, une signature d'un électeur correspond simplement à un tableau d'entiers créé en chiffrant la clé publique du candidat (qui est en chaîne de caractères) avec la clé secrète de l'électeur.

- La structure de declaration de vote est composée alors de :
  - la clé publique de l'électeur, pKey.
  - la chaîne de caractères de la clé publique du candidat, mess.
  - la signature de l'électeur, s.

- Les fonctions importants sont :

Fonctions :	**Explications** :
void init_pair_keys(Key* pKey, Key* sKey, long low_size, long up_size)	Initialisation du couple de clés publique et secrète selon le protocole RSA.
Signature* sign(char* mess, Key* sKey)	Initialisation d'une signature en chiffrant la chaîne *mess* avec la clé secrète de l'électeur.
Protected* init_protected(Key* pKey, char* mess, Signature* s)	Initialisation d'une déclaration.
int verify(Protected *pr)	Vérification de l'authenticité d'une déclaration. En déchiffrant pr->s->tab avec pr->pKey, nous devrons obtenir une chaîne de caractères identique a pr->mess.
void generate_random_data(int nv, int nc)	Génération aléatoire de nv électeurs (= nv couple de clés), qui votent aléatoirement pour nc candidats (= nv déclarations de votes).

3. troisième.c, troisieme.h, test_troisieme.c :

- Le fichier *troisieme.c* manipule les structures de listes chaînées et une table de hachage afin d'implémenter une base centralisée pour vérifier et compter les votes de manière efficace.

- Le fichier *troisieme.h* correspond a son fichier header, et contient la définition des structures CellKey, CellProtected, HashCell, et HashTable. Le fichier *test_troisieme.c* contient les jeux d'essais effectués pour nos fonctions.

- Les structures utilisées sont :

CellKey	**CellProtected**
Key* data;	Protected* data;
cellKey* next;	cellProtected* next;

HashCell	**HashTable**
Key* key;	HashCell** tab;
int val;	int size;

- Les fonctions importants sont :

Fonctions :	**Explications :**
CellKey* read_public_key(char* nomfic)	Renvoie une liste chaînée de clés depuis les données dans le fichier text nomfic.
CellProtected* read protected()	Renvoie une liste chaînée de déclarations depuis les données dans "declarations.txt".
Key* compute_winner(CellProtected* decl, CellKey* candidates, CellKey* voters, int sizeC, int sizeV)	Renvoie la clé du candidat élu en vérifiant l'authenticité de chaque vote.

4. quatrieme.c, quatrieme.h, test_quatrieme.c :

La partie 4 et 5 du projet.

- Le fichier *troisieme.c* manipule les structures de Blockchain afin d'implémenter une base centralisée pour vérifier et compter les votes de manière efficace.

- Le fichier *quatrieme.h* correspond à son fichier header, et contient la définition des structures Block et CellTree. Le fichier *test_quatrieme.c* contient les jeux d'essais effectués pour nos fonctions.

- Les structures utilisées sont :

Block	**CellTree**
Key* author;	Block * block
CellProtected* votes;	struct block_tree_cell* father;
unsigned char* hash;	struct block_tree_cell* firstChild;
unsigned char* previous_hash;	struct block_tree_cell* nextBro;
int nonce;	int height;

- Les fonctions importants sont :

Fonctions :	**Explications :**
CellTree* read_tree()	Parcourt le tableau T et trouve sa racine puis elle l'envoie
void add_block(int d, char* name)	Vérifie que le bloc représenté par le fichier "Pending block" est valide.
Key* compute_winner_BT(CellTree* tree, CellKey* candidates, CellKey* voters, int sizeC, int sizeV)	Détermine le gagnant de l'élection en se basant sur la plus longue chaîne de l'arbre.

III. Description des jeux d'essais :

Les jeux d'essais utilisés sont les tests données par l'énoncé:

- Dans *test_premier,* les fonctions principales sont testées en générant un couple de clés (pKey , sKey). Une chaîne de caractères est ensuite chiffrée avec encrypt(), puis déchiffrée avec decrypt().

- Dans *test_deuxieme*, le jeu d'essais consiste à générer un couple de clés, une signature et une déclaration (Protected). Pour chaque structure, nous vérifions le bon fonctionnement de sa sérialisation, avec les fonctions to_str() et vice versa.
  La simulation d'une déclaration de vote est aussi effectuée, et son authenticité est vérifiée avec verify().

- Dans *test_troisieme*, après avoir récupéré notre liste chaînée de keys.txt, nous affichons la liste et créons une nouvelle clé. Ensuite nous affichons une déclaration non valide et testons que notre fonction de suppression de déclarations non valides fonctionne en affichant la nouvelle liste modifiée.

- Dans test_quatrieme, nous testons les fonctions save_block() et load_block() en cherchant si les valeurs sont équivalentes. Nous comparons ensuite nos deux votes (liste_protected) et leur taille. Après nous comparons nos différents blocks et la taille de notre blockchain_list. Enfin nous appliquons les jeux de test demandés de la question.

IV. Réponses aux questions :

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 ?

La complexité de *is_prime_naive(long p)* est O(p/2), soit une complexité d'ordre O(p).

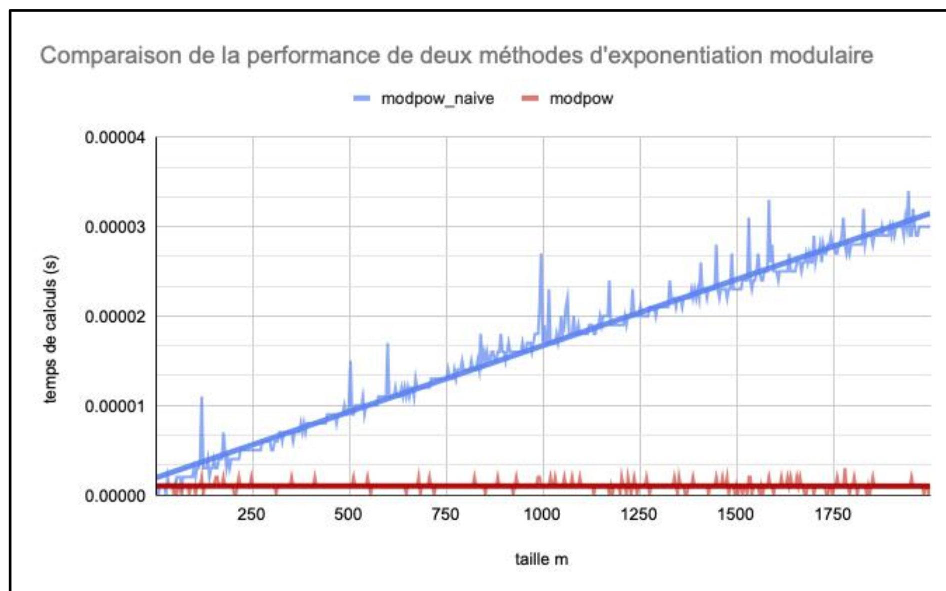
Q 1.2 : Quel est le plus grand nombre premier que vous arrivez à tester en moins de 2 millièmes de seconde avec cette fonction ?

En moins de 2 millièmes de seconde, nous arrivons à tester jusqu'à 37 223.

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^m mod n par la méthode naïve. Quelle est sa complexité ?

Sa complexité est O(m).

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



On observe que modpow() est nettement plus rapide que modpow_naive(). En effet, la complexité temporelle de modpow() est O(ln(m)), ce qui est meilleure que O(m) de modpow_naive().

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

La borne supérieure de la probabilité d'erreur est ¼.

Q 8.8 : Comparer les performances des deux méthodes d'exponentiation modulaire en traçant des courbes de temps en fonction de m. Qu'observez-vous ?

Cette fonction est de complexite O(2n), ou encore O(n) pour n elements.

Pour une complexite de O(1), la structure doit etre une liste chainee circulaire.

Q 9.7 Que pensez-vous de l'utilisation d'une blockchain dans le cadre d'un processus de vote ?

Le système de blockchain résout la plupart des problèmes rencontrés avec les autres structures de données centralisées. Ça permet notamment de bien sécuriser le processus, mais ça le rend plus long.

Est-ce que le consensus consistant à faire confiance à la plus longue chaîne permet d'éviter toutes les fraudes ?

Non, le consensus ne permet pas cela puisqu'il y a toujours un élément de hasard.