Itminene EL GARAH Matthias AIRED Kacem JORHO

Projet : Blockchain appliquée à un processus électoral

L'objectif de ce projet est d'implémenter de manière efficace le processus de désignation du vainqueur de l'élection, tout en garantissant l'intégrité, la sécurité et la transparence de l'élection.Pour cela, le projet est divisé en 5 parties, chacune centrée sur un aspect en particulier.

La première partie concerne la cryptographie asymétrique et, plus précisément le protocole RSA. La deuxième partie s'intéresse aux déclarations, de vote ou de candidature, que peuvent effectuer les citoyens. La troisième partie se concentre sur la centralisation de ces déclarations ainsi que sur leur intégrité. La quatrième partie, quant à elle, s'ouvre à l'implémentation d'un mécanisme de consensus. Enfin, la cinquième partie s'intéresse à la décentralisation des déclarations.

TABLE DES MATIÈRES

Gestion des fichiers et Makefile

Chaque partie est regroupée dans des fichiers contenant les fonctions à réaliser.

Les fichiers <nom>.h contiennent les signatures des fonctions des fichiers <nom>.c

Chaque partie peut être compilée et exécutée à l'aide du Makefile et de la commande

make main puis ***./main***

Les fichiers .txt sont les fichiers utilisés pour récupérer et traiter des données.

Partie 1 (exercice 1-2) :

- **prime_number.c** (fichier contenant les méthodes servant à la génération de nombre premiers)
- **chiffrement.c** (fichier contenant les méthodes servant à la génération de paire de clés)

Partie 2 (exercice 3-4) :

- **key.c** (fichier contenant les méthodes servant à la création de clés)

Partie 3 (exercice 5-6) :

- **Base.c** (fichier contenant les fonctions permettant de manipuler les structures cellKey et cellProtected)
- **hachage.c** (fichier contenant les fonctions permettant de manipuler les tables de hachage)

Partie 4 (exercice 7-8) :

- **block.c** (fichier contenant les fonctions permettant de manipuler les blocs)
- **blocktree.c** (fichier contenant les fonctions permettant de manipuler les chaînes de blocs, structure arborescente)

Partie 5 (exercice 9) :

- **simulation.c** (fichier contenant les fonctions permettant de simuler un processus de vote)

Partie 1 : Implémentation d'outils cryptographiques

La première partie est centrée sur l'implémentation d'outils cryptographiques, en commençant par développer une méthode de génération de nombres premiers efficace, puis en réalisant le cryptage et décryptage de messages à l'aide de clés publiques et privées.

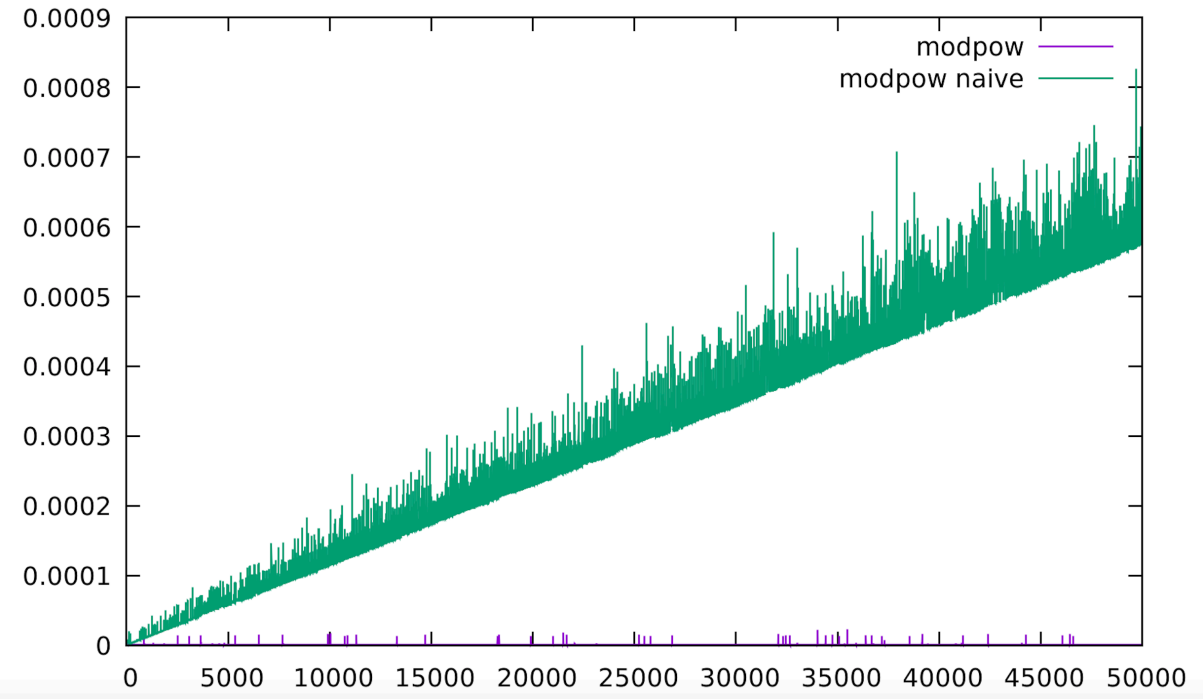
Exercice 1 :

1.1 La première étape pour trouver un nombre premier est de déterminer si un nombre p est premier. Nous commençons par réaliser **is_prime_naive()** , dont la complexité est en $O(p)$.

1.2 En utilisant cette fonction, le plus grand nombre premier que l'on arrive à tester en moins de 2 millièmes de secondes est de l'ordre de 200 000.

1.3 La complexité de la fonction **modpow_naive** est en theta(m).

1.5 Après affichage des courbes à l'aide de gnuplot nous obtenons le graphe suivant :



Nous observons alors que la méthode naïve a un temps de calcul croissant en fonction de m, tandis que la méthode rapide a un temps de calcul environ constant (car sa complexité est en O(log m)) .

1.7 Soit O l'ensemble des entiers entre 2 et p – 1.

Soit A l'ensemble des entiers de O qui ne sont pas témoins de Miller.
Soit a_i le numéro tiré au i-ème tour de boucle, pour i entre 1 et k.

Alors, on a : $P(échec) = P(a_1 \in A, a_2 \in A, \dots, a_k \in k)$
Les tirages sont indépendants, donc :

$$P(échec) = \prod_{i=1}^{k} a_i \in A$$

Les tirages sont aléatoires donc chacun des a_i suit la loi uniforme, ainsi :

$$P(a_i \in A) = \frac{card(A)}{card(O)}$$

$$P(échec) = \left(\frac{card(A)}{card(O)}\right)^k$$

Or, $card(A) \leq \left(1 - \frac{3}{4}\right) card(O)$ d'après l'énoncé, donc on a :

$$P(échec) \leq \frac{1^k}{4}$$

Donc la borne supérieure sur la probabilité d'erreur de l'algorithme est $\frac{1}{4}^k$.
Cela montre que pour des grands nombres, ce test est fiable.

Exercice 2 :

Une fois en capacité de générer un grand nombre premier, nous nous intéressons au chiffrement et déchiffrement de messages à l'aide du protocole RSA. C'est un algorithme de cryptographie asymétrique basé sur la génération de deux clés : une clé publique et une clé privée. De cette manière, on crypte le message avec notre clé privée, et le message peut être décrypté à l'aide de la clé publique, ou inversement. Pour réaliser cela, nous commençons par générer le couple de clés publique/privé (fonction **generate_key_values**), à l'aide de deux nombre premiers p et q, générés aléatoirement, puis nous calculons :

$n = p \times q$

$t = (p - 1) \times q - 1)$

s tel que $PGCD(s, t) = 1$

u tel que $s \times u \ mod \ t = 1$

On a alors $pKey = (s,n)$ et $sKey = (u,n)$

Une fois ces deux clés générées, nous pouvons implémenter des fonctions permettant de crypter ou décrypter un message (qui est une chaine de caractères) à l'aide d'une clé (publique ou privée) :

Pour encrypter, il suffit d'allouer un tableau d'entiers longs de la même taille que le message, puis de calculer $m^s \ mod \ n$, pour m un caractère du message (converti automatiquement en entier).

Pour décrypter, le principe est le même à ceci près que le calcul est maintenant $m^u \ mod \ n$.

Chaque fonction peut être testée à l'aide du fichier **main.c**

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

Exercice 3 :

La deuxième partie est centrée sur les déclarations sécurisées, en créant des structures adaptées : des clés publiques et privées représentant un électeur, des signatures permettant d'identifier l'électeur, et des déclarations sécurisées permettant à l'électeur de voter.

Chaque personne, électeur ou candidat, possède une clé publique et une clé privée. Si un électeur veut voter pour un candidat, il doit réaliser une déclaration sécurisée, comportant la clé publique du candidat ($mess$), la version cryptée de cette clé, à l'aide de sa clé privée ($sign$), et enfin sa clé publique. De cette manière, si on décrypte $sign$ avec la clé publique de l'électeur, on devrait obtenir $mess$.

Pour réaliser cela, nous avons utilisé plusieurs structures différentes :
Une pour représenter les clés, une pour les signatures, et une pour les déclarations.

Chaque structure comporte les mêmes types de fonctions :

- **init_struct** : alloue la taille de la structure et l'initialise avec les données en paramètre.
  - **key_to_str/str_to_key** : transforme la clé en paramètre en chaine de caractères et inversement.
  - **signature_to_str/str_to_signature** : transforme la signature (tableau d'entiers long) en chaine de caractères séparés par un « # » et inversement.
  - **protected_to_str/str_to_protected** : transforme la déclaration en chaine de caractères sous la forme « pkey mess sign » et inversement.
  - **init_pair_keys** : génère une clé publique et une clé privée à l'aide de nombre premiers compris entre low_size et up_size bits et de la fonction generate_key_values vu à la première partie.

- **verify** : vérifie que la signature décryptée à l'aide de la clé publique corresponde bien au message de la déclaration à l'aide de la fonction decrypt, renvoie 1 si c'est le cas, 0 sinon.

De cette manière, nous pouvons à présent réaliser une déclaration sécurisée et l'afficher.

Exercice 4 :

Maintenant que nous pouvons créer des déclarations, il est important de pouvoir les stocker, dans le but de les utiliser pour faire des simulations de vote.

Ainsi, la fonction **generate_random_data** permet cela :
Cette fonction prend en paramètres le nombre d'électeurs et de candidats.

Tout d'abord, elle génère aléatoirement autant de couple de clés publique/privée que d'électeurs, les stocke dans un tableau, et les écrit dans le fichier « *keys.txt* » au fur et à mesure (à l'aide de **key_to_str**).

Ensuite, elle sélectionne aléatoirement les candidats parmis les électeurs, stocke leur clés publiques dans un deuxième tableau et les écrit dans le fichier « *candidates.txt* ».

Enfin, pour chaque électeur, elle sélectionne aléatoirement un candidat pour qui voter, génère une déclaration sécurisée, la stocke dans un tableau et l'écrit dans le fichier « *déclarations.txt* ».

Partie 3 : Manipulation d'une base centralisée de déclarations

La troisième partie est centrée sur la gestion des électeurs, candidats et déclarations dans des fichiers, en transformant le contenu des fichiers en listes chainées, puis sur la détermination du gagnant de l'élection.

Exercice 5 :

Nous voulons désormais pouvoir lire des données, telles que des clés ou des déclarations, dans des fichiers afin de les stocker dans des listes chaînées. Rappelons que les données lues seront soit des clés, soit des déclarations. Pour ce faire, nous devons d'abord créer des structures associées à une donnée. On obtient ainsi une structure pour ranger les clés

(cellKey) et une autre pour y ranger les déclarations(cellProtected).

À partir de ces structures, nous voulons créer des fonctions afin de pouvoir créer un élément, ajouter un élément à une liste, afficher/supprimer cet élément ou lire le fichier.

Exercice 6 :

Dans cet exercice, nous commencerons par créer une fonction **filter** permettant de limiter les fraudes.
En effet, une fois toutes les données collectées, le système commence par retirer toutes les déclarations contenant une fausse signature.

Pour cela, il nous suffit de parcourir la liste de toutes les déclarations obtenues, vérifier la signature de chaque déclaration et supprimer la déclaration en question de la liste si sa signature n'est pas valide. On veillera à faire la distinction entre un élément à supprimer en début ou en milieu de chaîne.

L'objectif de cet exercice est de réaliser une fonction **compute_winner** permettant de déterminer le gagnant d'une élection en utilisant des fonctions traitant les tables de hachage (vues durant la totalité l'exercice) ainsi que la fonction **filter** mentionnée plus haut.

Chaque fonction peut être testée dans le fichier **main.c**

Partie 4 : Implémentation d'un mécanisme de consensus

Après avoir utilisé un système de vote centralisé dans la partie précédente, nous opterons maintenant pour un système de vote décentralisé à l'aide d'une blockchain.
Cette décentralisation permettra à chaque personne de vérifier le résultat du vote par elle même ce qui est préférable au niveau de la fiabilité du vote

Pour utiliser une blockchain (chaîne de blocs) nous aurons besoin d'une nouvelle structure (**block**)
Le bloc contiendra la clé publique de son créateur (Key* **author**),
une liste de déclarations de vote (CellProtected* **votes**),
la valeur hachée du bloc (unsigned char* **hash**),
la valeur hachée du bloc précédent (unsigned char* **previous_hash**),
un entier comme « preuve de travail » (int **nonce**),

Exercice 7 :

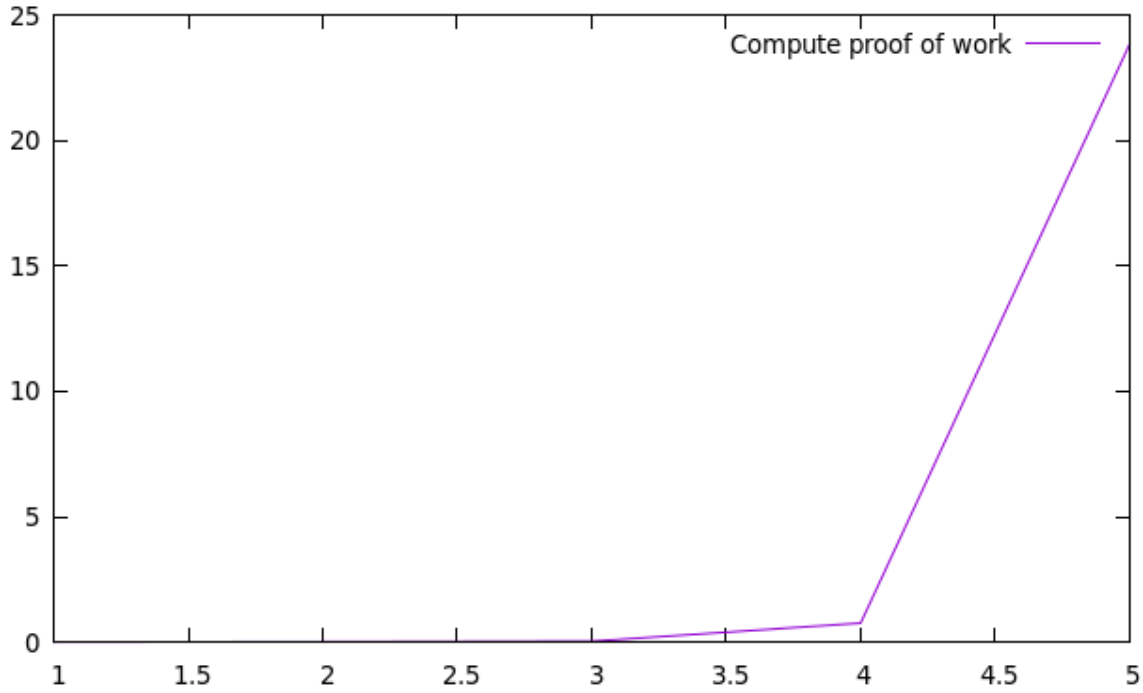
Dans cet exercice nous manipulerons des blocs, nous allons d'abord écrire un bloc dans un fichier à l'aide de la fonction **ecrire_block** , cette fonction prend en paramètre le nom du fichier ainsi que le bloc.
Ensuite, avec la fonction **lire_block** nous réaliserons le procédé inverse, lire un bloc grâce à un fichier.

Pour la suite nous devrons calculer la valeur hachée du bloc, c'est pour cette raison que nous allons utiliser une fonction **block_to_str** qui va générer une chaîne de caractères comme représentation du bloc.Cette fonction ne prend que le bloc en paramètre et devra contenir les attributs du bloc (voir la structure au dessus).

Un bloc est considéré comme valide si sa valeur hachée commence par *d zéros* successifs.

La fonction **verify_block** utilisera le hachage cryptographique SHA-256 pour vérifier la validité d'un bloc, elle retournera 1 si le bloc est valide.

7.8 _____ajout graphe de compute_proof_of_work_____



Ce graphe montre qu'avec cet algorithme, il est possible de trouver en moins d'une seconde des preuves de travail correspondant à d < 4.

L'allure du graphe nous montre la complexité exponentielle de **compute_proof_of_work.**

Remarque : Le nombre d de zéros exigé sur la représentation hexadécimale revient à exiger que la représentation binaire débute par au moins 4d zéros.

Pour la fonction **delete_block** qui permet de supprimer un bloc prit en paramètre, nous utiliserons une fonction vue précédemment (**delete_list_protected**)
Qui supprime entièrement une liste chaînée.
La libération de la mémoire se fait à l'aide de free( ).

Exercice 8 :

Dans cet exercice nous traiterons des structures arborescente de blocs.
Un arbre de bloc se crée lorsque plusieurs blocs possèdent le même père (bloc précédent).
En effet, dans une blockchain, chaque bloc contient la valeur hachée du bloc précédent, cependant, s'il y a de la triche alors certains blocs peuvent annoncer avoir le même bloc précédent —> arbre de blocs.

Afin de représenter les arbres de blocs, nous utiliserons une nouvelle structure (**block_tree_cell**).

L'importance de déterminer le dernier bloc est grande car le consensus est de faire confiance à la chaine de blocs la plus longue.

Pour y parvenir nous allons réaliser une fonction **highest_child** qui renvoie le noeu fils avec la plus grande hauteur ainsi qu'une fonction **last_node** qui retourne le dernier bloc de la chaîne la plus longue de **highest_child**.

En remontant de père en père depuis le CellTree* renvoyé par **last_node**, il est possible de récupérer la plus longue chaîne de blocs et de garantir la bonne gestion des données frauduleuses.

8.8

Il serait possible d'obtenir une fonction de fusion en O(1) en remplaçant l'implémentation actuelle de CellProtected par une structure de liste doublement chaînée, garantissant une insertion en fin de liste en O(1) et non en O(n) où n est le nombre d'éléments de la liste à laquelle on fusionne une autre liste chaînée.

Partie 5 : Manipulation d'une base décentralisée de déclarations

On utilisera dans cette section le répertoire **Blockchain** contenant les fichiers temporaires suivants : *Pending_votes.txt* et *Pending_block.txt*. Cela permet d'assurer un processus de vote en trois temps :

Votes des citoyens
Création de blocs à intervalles réguliers
Mise à jour de la base de données.

Exercice 9 :

Vote et création de blocs valides

On intègre les fonctionnalités de soumission de vote et d'ajout de blocs, en se servant des fichiers *Pending_votes.txt* et *Pending_block.txt* comme transition pour les blocs et votes en cours d'enregistrement :

void submit_vote(Protected *p)
void create_block(CellTree *tree, Key *author, int d)
void add_block(int d, char *name)

En particulier, **create_block** permet de créer un bloc temporaire, que l'on ajoute à la base de données dans le répertoire Blockchain après vérification avec **add_block**.

Lecture de l'arbre et calcul du gagnant

Pour lire un arbre depuis un fichier, et calculer le gagnant des élections à partir d'un arbre, on utilise les fonctions :

CellTree ***read_tree**()
Key ***compute_winner_BT**(CellTree *tree , CellKey *candidates , CellKey *voters, int sizeC, int sizeV)

La bibliothèque ***<dirent.h>*** contribue largement à l'implémentation de **read_tree**, de même que **compute_winner** pour **compute_winner_BT**.
Notons également que **read_tree** lit l'intégralité des blocs présents dans le dossier **Blockchain**.

Pour finir, l'exécutable ***main*** permet de générer une base de données.
On ajoute alors un Block tous les 10 votes avec les fonctions **submit_vote, create_block** et **add_block**, avant de constituer l'arbre final grâce à **read_tree**, affiché par la suite avec **print_tree** dans *tree.txt.*
On peut finalement déterminer le vainqueur de l'élection à l'aide de **compute_winner_BT**.

9.7

Conclusion sur l'utilisation d'une blockchain dans un processus électoral

L'utilisation d'une blockchain dans le cadre d'un processus de vote s'avère être intéressant en terme de transparence des élections.
En effet chacun pourrait vérifier le décompte des voix et suivre l'avancé de l'élection.
Cependant la notion de consensus consistant à faire confiance à la plus longue chaîne ne permet pas d'éviter toutes les fraudes car cela suppose qu'on fait confiance à la plus grande puissance de calcul.
Une tierce personne avec d'important moyen informatique est en mesure de générer une chaîne plus longue ce qui lui permettrait de modifier le résultat de l'élection.
Le système que nous avons donc réaliser se révèle encore imparfait.

Conclusion

Projet avec une thématique intéressante et actuelle, l'utilisation des *blockchains* sera de plus en plus présente à l'avenir.
Cependant, au cours du projet on a du faire face à un problème récurrent:
le temps passé à débugger les fonctions est + important que le temps mit à profit pour manipuler les structures ce qui est un peu dommage.

Pour finir, l'objectif du projet est atteint :
on a bien mis en pratique l'utilisation des structures de données vues dans le cours/TD.
On a également une vision plus précise du fonctionnement d'une *blockchain .*

Malheureusement, on a manqué de temps pour finaliser et corriger l'exercice 9 : certaines fonctions ne sont pas encore fonctionnelles et doivent être ajustées.
On a passé énormément de temps à retirer les fuites mémoires et avoir des exercices propres.