

Projet de Structures des données: Blockchain appliquée à un processus électoral

04.25.2022



Chadi Tawbi - Said Bougjdj

Numero etudiant:28706563-28622769

2ème année Licence Informatique

Mono-disciplinaire Groupe n°1 2021-2022




Introduction

Le projet consiste de l'implémentation et la manipulation d'outils de cryptographie pour l'organisation d'un processus électoral par scrutin uninominal majoritaire à deux tours. Le but de ce projet est de faire une election sécurisée mais aussi de voir la différence entre une base centralisée et décentralisée et de pouvoir déduire si une implémentation d'une blockchain est préférable dans le cadre d'un processus de vote.

Voici les différentes étapes qu'on a pris pour aboutir ce sujet:

Base centraliser:

- **étape 1:** l'implémentation de codes à la base de certains outils de chiffrement. Plus spécifiquement, nous avons traité le problème de la génération de nombres premiers.
- **étape 2:** nous avons utilisé les codes précédents pour implémenter des fonctions à l'origine du chiffrement et du déchiffrement du protocole RSA, où des couples clés publiques et clés privées du destinataire recevant le message crypté sont nécessaires.
- **etape 3:** après avoir étudié un cas singulier, nous élargissons ce mécanisme de cryptographie à une situation de vote où les électeurs utilisent leur clés privées pour signer lors du vote, dont l'authenticité peut être vérifiée par n'importe qui à l'aide de la clé publique. Ainsi, des fonctions de clés et de signatures sont manipulées.
- **etape 4:** nous simulons un scrutin avec un ensemble de clés publiques et privées et de candidats votables.
- **etape 5:** nous procédons à la comptabilisation des déclarations de vote à l'aide des clés publiques, regroupées dans des listes chaînées manipulables.



etape 6: ensuite nous utilisons les tables de hachages pour pouvoir comptabiliser le nombre de vote de chaque candidat pour pouvoir enfin trouver le candidat gagnant.

Base decentraliser:

Pour la mise en place d'une base decentraliser, nous avons commencer par la creation d'une blockchain

etape 1: grace a la fonction de hachage cryptographique d'SHA-256 nous avons mis en place un un mecanisme de consensus dit 'proof of work' (preuve de travail)

etape 2: pour limiter les fraudes nous avons creer une structure arborescente de blocs. Avec celle-ci, ne pouvons deduire quelle est la plus longue chaine de blocs en partant de la racine et donc conclure que cette blockchain est probablement la moins frauduleuse.

etape 3: Enfin, nous avons simuler les soumissions de vote par les citoyens et les créations de blocs valides par des assesseurs (citoyens volontaires). Puis,nous avons créer l'arbre de blocs correspondant, et calculer le gagnant de l'élection en faisant confiance à la plus longue chaîne de l'arbre.

Descriptions globales du code et des structures:

Les structures utilises:

De nombreuses structures ont été manipulé:

- **Une structure Key**, identifiant les clés publiques et secrètes, dont les valeurs sont générées à partir des algorithmes initiaux (key->valeur, key->taille)

- **Une structure Signature** correspondant au message chiffré dans la situation du vote (sgn->content (tableau de long) , sgn->taille (int))

- **Une structure Protected** qui contient la clé publique de l'électeur, sa déclaration devote et la signature associée. (p->pkey , p->mess (décla) ,p->sgn (signature associé))

- **Une structure CellKey** regroupant les clés publiques sous forme de liste chaînée


- **Une structure CellProtected**, listes chaînées de déclarations signées

-**Une structure HashCell** qui contient une clé et un entier val. Val pour un citoyen represente s'il a voter ou non (1 s'il a voter et 0 sinon). Pour un candidat, val represente le nombre de votes gagnés.

-**Une structure HashTable** correspondant a une Table de Hash qui contient des HashCell placer dans la table grace a une fonction de hachage. Elle contient aussi un entier size qui correspond a la taille de la table.

-**Une structure Block** qui represente un bloc de la blockhaine qui est chaînée à l'aide de la valeur hachée du bloc précédent.

Celle-ci contient:

- 
- La clé publique de son créateur.
 - Une liste de d'éclarations de vote.
 - La valeur hachée du bloc.
 - La valeur hachée du bloc précédent.
 - Une preuve de travail.

-Une structure CellTree qui correspond a un arbre de blocs. Celle-ci nous permet de représenter des noeuds avec un nombre arbitraire de fils sans avoir a utiliser une liste chaînée. On peut donc parcourir cette arbre avec l'aide de firstChild qui représente le premier fils et puis nextBro pour accéder les fils jusqu'à qu'on obtient le pointure NULL. Enfin on a le champ height qui représente la hauteur du nœud.

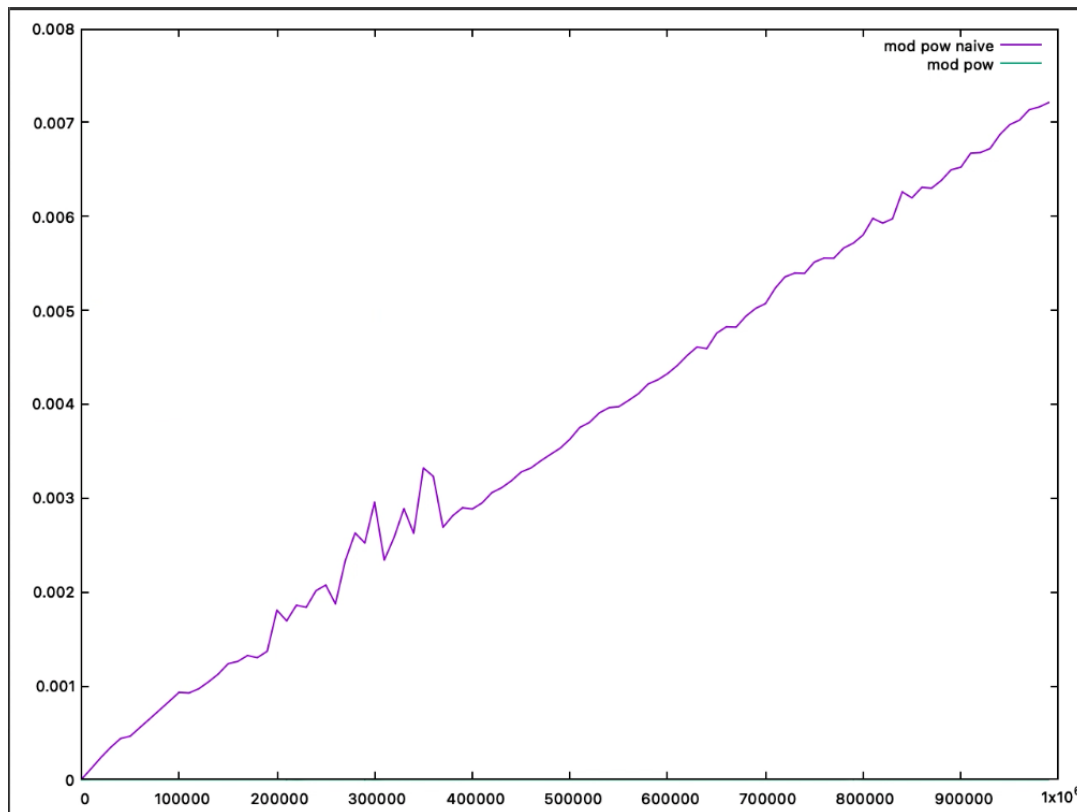
Description des codes des différentes fonctions:

Exponentiation modulaire rapide:

int is_prime_naive(long p): teste si p est un nombre premier ou non.

long modpow_naive(long a, long m, long n): retourne la valeur $a^b \bmod n$ par la méthode naïve.

long modpow(long a, long m, long n): réalise une élévation au carré.



En regardant ce graphique, (la courbe de modpow est bien présente mais est difficile à voir) on aperçoit que modpow_naive a une complexité de $O(n)$ alors que modpow a une complexité de $O(\log n)$. Donc on peut conclure que modpow est beaucoup plus rapide et c'est cette fonction qu'on utilisera pour la suite.

Test de Miller-Rabin:

int witness(long a, long b, long d, long p): celle-ci renvoie 1 si a est un témoin de Miller pour p, pour un entier a donné.

long rand_long(long low, long up): retourne un long généré aléatoirement entre low et up inclus.

int is_prime_miller(long p, int k): celle-ci regarde k fois si on trouve un témoin de Miller pour p. Elle renvoie 0, Si on trouve un témoin pour p, on peut donc dire que p n'est pas premier. Sinon elle renvoie 1, si on ne trouve pas de témoin, il est donc probable que p est premier.

long random_prime_number(int low_size, int up_size, int k): retourne un nombre premier de taille comprise entre low_size et up_size.

Protocole RSA:

long extended_gcd(long s, long t, long *u, long *v): Renvoie le plus grand diviseur commun entre s et t.

void generate_key_values(long p, long q, long *n, long *s, long *u): La variable n étant la multiplication de p et q. Cette fonction génère une clé publique (s, n) et une clé secrète (u, n).

long* encrypt(char* chaine, long s, long n): Celle-ci chiffre la chaîne de caractères avec la clé publique pKey = (s, n). Pour cela, la fonction réalise une exponentiation modulaire (cf modpow) et convertit chaque caractère en un entier et retourne le tableau de long obtenu en chiffrant ces entiers.

char* decrypt(long* crypted, int size, long u, long n): A l'aide de la clé secrète sKey=(u, n), celle-ci déchiffre un message (en utilisant modpow) et renvoie la chaîne de caractère obtenue.

void print_long_vector(long *result, int size): affiche un tableau de long de longueur size.

Manipulation des clés:

void init_key(Key* key, long val, long n): initialise une clé déjà allouer.

void init_pair_keys(Key* pKey, Key* sKey, long low_size, long up_size); Celle-ci fait appel aux fonctions random_prime_number, generate_keys_values et init_key pour initialiser les clés publique et privé via le protocole RSA.

char* key_to_str(Key* key): passage en chaîne de la clé.

Key* str_to_key(char* str): passage d'une chaîne à une clé.

Key *copie_key(Key *key): Fait une copie d'une clé.

Manipulation des signatures:

Signature* init_signature(long* content, int size): alloue et initialise une signature avec un tableau de long.

Signature* sign(char* mess, Key* sKey): retourne un pointeur vers une signature qui contient une mess de forme encryptée (donc d'un tableau de long, cf encryptée) en utilisant la clé secrète pour crypter.

char* signature_to_str(Signature *sgn): passage en chaîne de la signature avec un '#' entre chaque char crypté.

Signature* str_to_signature (char* str): passage de chaîne à signature.

Manipulation des déclarations:

Protected* init_protected(Key* pKey, char* mess, Signature* sgn): initialise une déclaration de vote signée

int verify(Protected* pr): renvoie 1 si la signature contenue dans pr correspond bien au message à la personne contenue dans pr, 0 sinon.

char* protected_to_str(Protected *pr): passage en chaîne de la structure

Protected* str_to_protected(char* str): passage de structure à chaîne

Simulation d'un scrutin de vote:

void generate_random_data(int nv, int nc): Celle-ci génère des couples de clés publiques et secrete nv fois. Celles-ci sont enregistrer dans le fichier keys.txt. Ensuite on choisi aleatoirement nc (candidats) cle publique de keys.txt different et on les enregistre dans le fichier candidates.txt. Enfin, on creer nv declarations et on les enregistre dans le fichier déclaration.txt .

Liste Chaînée de clés:

CellKey* create_cell_key(Key* key): créé une structure CellKey vide

void inserer(CellKey LCK, Key* key):** insère une CellKey a la tete de la liste

CellKey* read_public_keys(char* nomFichier): crée une liste contenant toutes les clés du fichier.

void print_list_keys(CellKey* LCK): affiche une liste de clés.

void delete_cell_key(CellKey* LCK): supprime une cellule de la liste de clés.

void delete_list_keys(CellKey* LCK): supprime une liste de clés.

int list_keys_length(CellKey *c): retourne la taille de la liste des clés.

Liste Chaînée de déclarations:

CellProtected* create_cell_protected(Protected* pr): créé une structure de liste Protected vide

void inserer_CellProtected(CellProtected* cpr, Protected* data): ajoute une déclaration signée en tête de liste.

CellProtected* read_protected(): crée une liste contenant toutes les déclarations signées du fichier

void print_list_protected(CellProtected* c): affichage de la liste des declarations

void delete_cell_protected(CellProtected* c): supprime une cellule de liste chaînées de déclarations signées

void delete_list_protected(CellProtected* c): supprime toute la liste des declarations

CellProtected *fusionner_list_protected(CellProtected *votes1, CellProtected *votes2): fusionne deux listes de declarations.

Manipulation de Table de Hachage:

void fraude(CellProtected LCP):** supprime toutes les déclarations non valides d'une liste de déclarations.

HashTable* create_hashtable(CellKey* keys,int size): alloue et initialise une table de Hachage en plaçant chaque cellule (cf create_hashcell) dans leur position respective grâce à la fonction de hachage(cf hash_function).

HashCell* create_hashcell(Key* key): alloue et initialise à 0 une cellule HashCell qui va être placée dans la table de hachage.

int hash_function(Key* key,int size): calcule la position de la cellule avec les valeurs de la clé en utilisant la formule $(v+1)*(n+1)\%size$

int find_position(HashTable *t ,Key* key): cherche dans la table de hachage s'il existe un élément dont la clé publique est key et si on le trouve, la fonction retourne sa position dans la table sinon, celle-ci retourne la position où il aurait dû être.

void delete_hashtable(HashTable* t): supprime une table de hachage

Key* compute_winner(CellProtected* decl, CellKey* candidates, CellKey* voters, int sizeC, int sizeV): créer deux tables de hachages (une table de taille sizeC pour la liste des candidats et une table de taille sizeV pour la liste des votants). Vérifie bien que toutes les déclarations (decl) sont valides(cf fraude). Vérifie aussi que la personne qui vote a le droit de voter et n'a pas déjà voté et que la personne sur qui porte le vote est bien un candidat. Quand les conditions sont respectées, on comptabilise le vote dans la table des candidats et la table des voteurs est mise à jour pour indiquer que le votant vient de voter. Enfin la fonction parcourt la table de hachage des candidats pour déterminer le gagnant(celui qui a le plus de votes).

Manipulation des blocs:

Block *creerBlock(Key *author, CellProtected *votes, unsigned char *hash, unsigned char *previous_hash, int nonce): alloue et initialise un bloc.

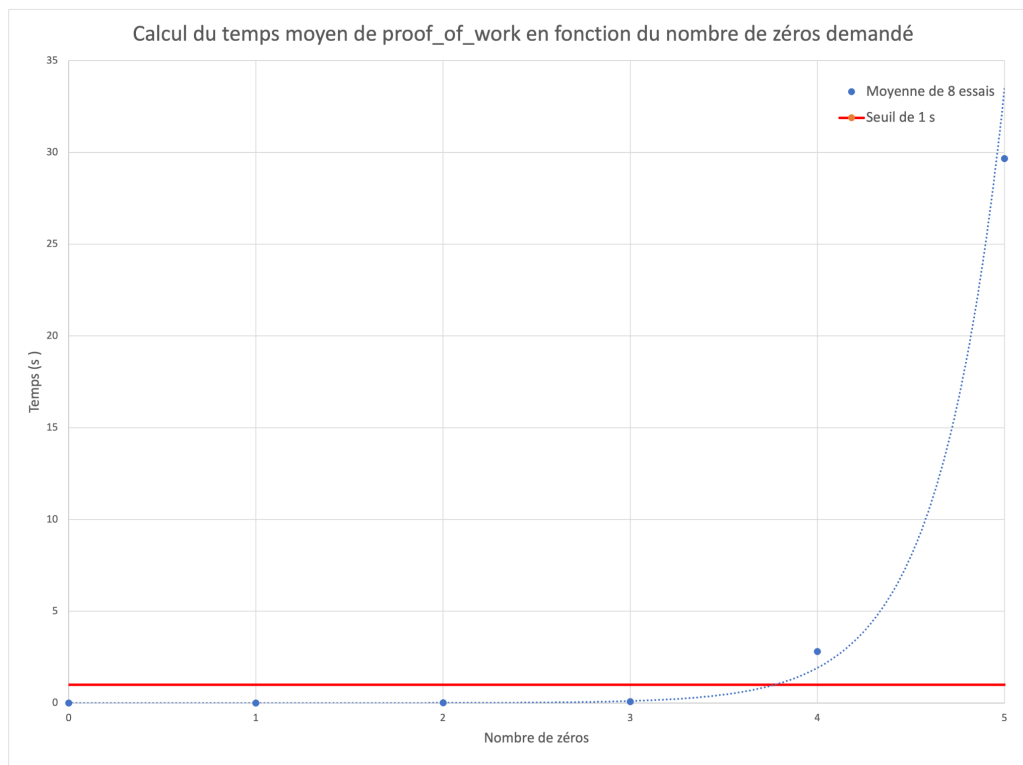
void enregistrerBlock(char *filename, Block *block): écrire dans un fichier un bloc.

Block *lireBlock(char *filename): lire un bloc depuis un fichier.

char *block_to_str(Block *block): génère une chaîne de caractère représentant un bloc qui contient la clé de l'auteur, la valeur hachée du bloc précédent, tous ses votes et une preuve de travail.

unsigned char* hash_function_block(const char* str): c'est la fonction de hachage SHA-256 qui en prend en entrée des messages de taille variable et produit des valeurs hachées de taille fixe(256 bits).

void compute_proof_of_work(Block *B, int d): Pour rendre un bloc valide, on procède par brute force. Cela veut dire qu'on commence avec l'attribut nonce à zéro, puis on incrémente l'attribut nonce jusqu'à ce que la valeur hachée du bloc commence par d zéros successifs.



On peut observer une croissance exponentielle du temps quand le nombre de 0 d augmente. Sa complexité est donc de $O(2^d)$.

On peut voir qu'à partir de 3 zéros le temps d'exécution de la fonction dépasse 1 seconde (Tout dépend du processeur sur lequel s'effectue l'exécution).

int verify_block(Block *B, int d): vérifie qu'un bloc est valide (si la valeur hachée de ses données commence par d zéros successifs).

void delete_block(Block *b): supprime le bloc mais ne libère pas la mémoire associée aux champs author et ne libère pas le contenu de la liste chaînée de votes.

Manipulation des Arbres de blocs:

CellTree* create_node(Block* b): permet de créer et initialiser un nœud avec une hauteur de 0.

int update_height(CellTree *father, CellTree *child): met à jour la hauteur du nœud father quand l'un de ses fils a été modifié.

void add_child(CellTree *father, CellTree *child): ajoute un fils à un père et actualise la hauteur de tous les ascendants (avec l'aide de update_height).

void print_tree(CellTree *tree): affiche l'arbre et plus précisément la hauteur et la valeur hachée du bloc de chaque nœud.

void delete_node(CellTree *node): Supprime un nœud

void delete_tree(CellTree *tree): Supprime un arbre

CellTree* highest_child(CellTree* cell): Récupère le nœud fils avec la plus grande hauteur.

CellTree *last_node(CellTree *tree): Renvoie la feuille du plus grand tronc de l'arbre.

CellProtected* get_trusted_list_protected(CellTree* tree): Récupère la liste des Déclarations de la plus longue branche (avec l'aide de highest_child) en partant de la racine de l'arbre. Toutes les listes de déclarations sont fusionnées grâce à la fonction fusionner_list_protected vu précédemment.

Simulation du processus de vote:

void submit_vote(Protected* p): Permet à un citoyen de soumettre un vote, donc d'ajouter son vote a la fin du fichier "Pending_votes.txt"

void create_block(CellTree tree, Key* author, int d):** Celle-ci creer un bloc valide qui contient les votes en attente dans le fichier "Pending_votes.txt" et ecrit le bloc obtenu dans un fichier appele "Pending_block".Le fichier "Pending_votes.txt" est supprimer apres la creation du bloc.

void add_block(int d, char* name): Le bloc dans "Pending_block" passe par une verification pour voir s'il est valide (cf verify_block).Si c'est le cas alors un fichier name représentant le bloc est créé et est ajouté au repertoire "Blockchain". Le fichier "Pending_block" est supprimer dans tout les cas.

CellTree *read_tree(): parcour le répertoire "Blockchain" et pour chaque bloc,créé un noeud et donc creer un arbre de bloc.

Key* compute_winner_BT(CellTree* tree, CellKey* candidates, CellKey* voters, int sizeC, int sizeV): Celle-ci commence par extraire la liste des declarations de vote de la plus grande hauteur de la racine(cf get_trusted_list_protected). Ensuite elle supprime toute les declarations de votes non valides avec la fonction fraude.Enfin avec la fonction compute_winner, elle trouve le gagnant et renvoie la clé publique de celui-ci.

Réponse aux Questions:

Exercice 1:

1)Quelle est la complexité de is_prime_naive?

Sa complexité est en $O(n)$.

2)Quelle difference remarquez vous entre is_prime et is_prime naive?

(cf page 5)

3)Quelles sont les probabilités d'erreur de l'algorithme pour l'obtention des temoins de Miller(couple de nombre premier)?

On parle d'une complexité $O(m)$ vu qu'elle ne dispose que d'opération élémentaires. Via le graphe, on remarque que le temps d'exécution de modpow_naive est croissante en fonction du N tandis que modpow est constante sur N . On utilise la loi de Bernoulli avec $p \geq \frac{3}{4}$ (du coup min $p = \frac{3}{4}$). Si on a k valeurs de a (k tirages), on a $(\frac{1}{4})^k$ (maximal) de prob d'avoir que des échecs.

Conclusion:

L'utilisation d'une blockchain est un modèle très intéressant dans ce cadre pour la réalisation d'un processus de vote.

Toutefois, il n'est pas possible de prévenir entièrement les tentatives de fraudes car l'or de nos choix de parcours, on se base sur la branche la plus longue qu'on considère comme étant la véridique et si suite à un noeud, on trouve 2 freres de meme longueur , il est impossible de distinguer la branche de déclarations vérifié et celle qui est frauduleuse.

Par ce fait, la création de longue chaine n'est synonyme de sécurité , meme si le système est assez efficace pour traiter les informations .

Donc on ne recommanderais vraiment pas l'utilisation de blockchain pour une élection où il est possible ,avec des compétences assez avancé en informatique, de corrompre les votes et de trafiquer des informations dites confidentielles.

Pour Executer le code:

Cf le Makefile et choisir le main que vous voulez.

Ensuite executez la commande suivante:

make all && ./run_(le main que vous voulez)_main && make clean