Projet : Blockchain appliquée au processus électoral

Yuxin XUE et Zhirui CAI

April 25, 2022

Contents

1 Introduction

Ce projet vise à garantir le secret du vote lors de l'élection et à déterminer le vainqueur de manière équitable. Le déroulement du processus électoral a traditionnellement soulevé des questions de confiance et de transparence, et il est bien connu que le taux d'abstention aux élections est souvent relativement élevé. Dans ce projet, nous souhaitons proposer une piste de réflexion sur les protocoles et sur les structures de données afin de mettre en œuvre efficacement le processus de détermination du vainqueur d'une élection, tout en garantissant l'intégrité, la sécurité et la transparence de l'élection.

2 Description du code général

2.1 Code

Nous utilisons make pour vérifier les dépendances et générer des exécutables ou des fichiers de bibliothèque. Le répertoire racine contient deux dossiers, 'src' et 'test', 'src' contenant le code source et 'test' contenant les fichiers de test.

```
Project:
     makefile
     scr
         Makefile     utility.h
         key.c               key.h             rsa.c               rsa.h
         lcc.c                lcc.h               sgn.c               sgn.h
         lcp.c               lcp.h              hash.c             hash.h
         prime.c           prime.h           blo.c                blo.h
         pro.c               pro.h                 blo_t.c              blo_t.h
        sml.c               sml.h
     test
        Makefile
        key.c      lcp.c    lck.c      prime.c     pro.c      sgn.c      rsa.c
        blo.c   blo_t.c    sml.c   hash.c
         blockchain
             block1.txt
           ... ...
         temp
           blocks.txt    candidates.txt  declarations.txt   keys.txt
           Pending_block   Pending_votes.txt
           proof_of_work.csv  blocks_test.txt
```

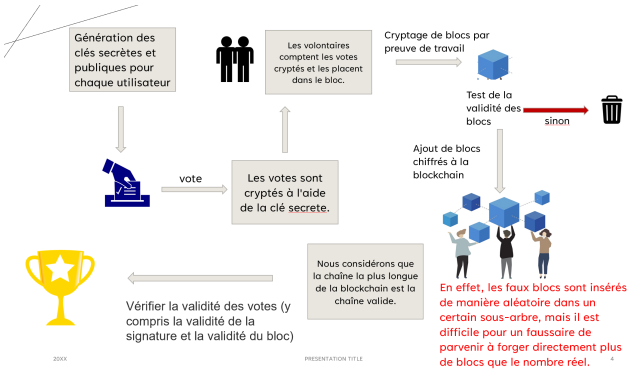
- prime.c et prime.h contient des fonctions mathématiques qui génèrent des premiers dans un intervalle donné.
- rsa.c et rsa.h contient les fonctions liées au protocole RSA, génère les valeurs des clés publiques et secrètes, encode et décode les messages.
- key.c key.h contient la structure et les fonctions connexes des clés secrètes et publiques, initialise des clés secrètes et publiques, passe de la variable de la clé à sa représentation sous forme de chaîne de caractères et l'inverse
- sgn.c et sgn.h contient la structure de la signature(Contient la clé secrète et le message codé) et les fonctions associées, initialise la signature, passe de la variable de la signature à sa représentation sous forme de chaîne de caractères et l'inverse.
- pro.c et pro.h contient des structures protégées (c.-à-d. donnees protegees) et des fonctions connexes, initialise protected, passe des variables protected à leur représentation sous forme de chaîne et l'inverse. ils contient aussi Vérification de la validité de la signature (si le codage de la signature, une fois décodé, est le même que l'information stockée).

- lck.c et lck.h contient la structure et les fonctions connexes pour cellKey (c'est-à-dire la liste chaînée de key), initialise CellKey, lit le fichier et génère la liste chaînée de key.
- lcp.c et lcp.h contiennent la structure et les fonctions connexes de cellProtected (c'est-à-dire la liste chaînée de donnee Protégé), qui initialise cellProtected, lit le fichier et génère la liste chaînée de donnee Protégé. Le fichier contient également les fonctions utilisées pour générer les tests, telles que 'generate_random_data'(qui est inclus dans pro.c) pour générer des données aléatoires.
- blo.c et blo.h contiennent la structure et les fonctions associées pour 'Block' (c'est-à-dire les blocs) et les fonctions pour générer les chaînes de hachage sha256 par chaîne, qui initialisent le Block, lisent et écrivent dans le fichier et génèrent les blocs.
- blo_t.c et blo_t.h contiennent la structure du CellTree (c'est-à-dire l'arbre à blocs) et les fonctions associées qui initialisent l'arbre à blocs, lisent et écrivent les fichiers et génèrent l'arbre à blocs. Elles contiennent également des fonctions permettant d'ajouter des enfants à l'arbre parent et de mettre à jour la hauteur, de sélectionner l'enfant ayant la hauteur la plus élevée, le dernier nœud, de mélanger tous les CellProtected dans la plus longue chaîne
- sml.c et sml.h contiennent la structure de la hashcell (i.e. case de hachage) hashtable (i.e. table de hachage) et les fonctions associées qui recherchent la position correspondante de la Clé et initialisent la table de hachage à partir de la CellKey. Finalement, le vainqueur est déterminé.
- sml.c et sml.h contiennent des fonctions liées à la simulation du processus d'élection.
- utility.h contien toutes les constantes utilisées dans le projet et peut être modifié en changeant le fichier pour changer les paramètres du projet et du test.
- Dans le dossier de test, le nom du fichier de test est le même que le nom du fichier source.

2.2 Description du processus d'élection

Au début, pour les candidats et les électeurs, nous générons les clés secrètes et publiques pour chaque utilisateur. Ici, nous avons également besoin des volontaires pour compter les votes et nous les générons également.

1. Les électeurs envoient leurs votes. L'envoi se compose de la clé publique de l'électeur, du vote et de la signature, qui est chiffrée avec la clé privée de l'électeur. Une signature est considérée comme valide si et seulement si elle contient la même information que le vote après qu'il ait été décodé par une clé publique
2. Les informations relatives au vote sont cryptées par les clés secrètes de l'électeur. Nous chiffrons le vote avec la clé secrète de l'électeur afin que n'importe qui puisse vérifier le vote avec la clé publique de l'électeur.
3. Les volontaires comptent les votes et les placent dans le bloc. Les blocs sont identifiés par le hachage et le hachage du bloc précédent.
4. Cryptage de blocs par preuve de travail. Un bloc n'est valable que si le travail s'avère conforme aux attentes, et nous préciserons plus tard comment cela fonctionne.
5. Ajout de blocs chiffrés à la blockchain. La blockchain est un arbre car le bloc peut avoir le même hash du bloc précédent que le bloc réel.
6. La chaîne la plus longue d'une blockchain est considérée comme la chaîne valide. En effet, les faux votes sont insérés de manière aléatoire dans un certain sous-arbre, mais il est difficile pour un faussaire de parvenir à forger directement plus de blocs que le nombre réel.
7. Vérifier la validité des votes (y compris la validité de la signature et la validité du bloc)



3 Développement d'outils cryptographiques

Dans cette section, nous allons développer quelques fonctions permettant de chiffrer des messages de manière asymétrique, à savoir : le système de cryptage à clé publique RSA. - Une clé publique qui est transmise à l'expéditeur et qui lui permet de chiffrer son message.
- Une clé secrète (ou privée), qui permet de décrypter le message à sa réception.
RSA (Rivest-Shamir-Adleman) est un système de cryptage à clé publique largement utilisé pour la transmission sécurisée de données. Un utilisateur crée et publie une clé publique basée sur deux grands nombres premiers, ainsi qu'une valeur auxiliaire. Les nombres premiers sont tenus secrets. Les messages peuvent être cryptés par n'importe qui, via la clé publique, mais ne peuvent être décodés que par quelqu'un qui connaît les nombres premiers.

3.1 Résolution du problème de primarité

Les clés publiques et privées nécessitent des nombres premiers. Nous allons donc commencer par générer des nombres premiers. La méthode que nous utilisons est très simple, nous générons d'abord un nombre aléatoire, puis si le nombre généré n'est pas un nombre premier, nous générons un autre nombre aléatoire... jusqu'à ce que le nombre soit un nombre premier. Il est donc particulièrement important de choisir la fonction qui détermine si c'est un nombre premier. Deux méthodes ont été développées. Commençons par la méthode classique.

3.1.1 Implémentions par une méthode naıve

par le calcul de a^m valeurs (qui peuvent être très grandes), une approche naïve serait de 1. multiplier la valeur actuelle par a. 2. appliquer modulo n au résultat avant de passer à l'itération suivante Répétez ces opérations m fois.

```
int  is_prime_naive(long p);
long modpow_naive(long a, long m, long n);
```

-int is_prime_naive(long p) teste si p est premier. Sa complexité est en O(p/2).
-long modpow_naive(long a, long m, long n) qui retourne la valeur a^b mod n en la multipliant par a à chaque itération
Pour tester la vitesse de cette fonction, nous exécutons is_prime_naive pour chaque entier afin de trouver le plus grand nombre premier qui peut être exécuté en deux millièmes de seconde.

Q1.2 Après 6 tentatives, nous sommes finalement parvenus à un nombre maximal de nombres premiers pouvant être testés en 2 millisecondes d'environ 32329 qui est beaucoup plus petit que la limite supérieure de la fourchette de nombres souhaitée(2^16).

fois	1	2	3	4	5	6
premier	32329	31261	30639	31515	30973	36153

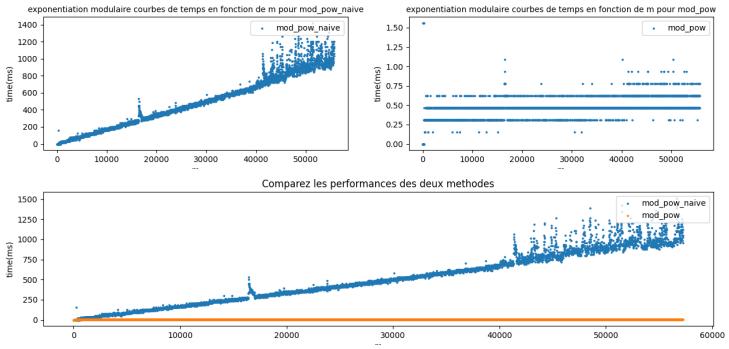
Table 1: le plus grand nombre premier a tester en moins de 2 millièmes

Nous souhaitons donc améliorer cet algorithme. Au lieu de multiplier par a à chaque itération, il est en fait possible d'élever au carré (directement avec modulo), ce qui donne un algorithme d'une complexité logarithmique (c'est-à-dire O(log2(m)).

```
long modpow(long a, long m, long n);
```

-int modpow(long a, long m, long n) retourne la valeur a^b mod n par des élévations au carré
ab mod n est égal à .
- 1 lorsque m = 0 (cas de base). - b b mod n avec b = am/2 mod n, lorsque m est pair. - a b b mod n avec b = am/2 mod n, lorsque m est impair.
Nous allons comparer la vitesse de ces deux méthodes à l'aide de calculs pratiques.

Q1.3 Pour modpow_naive qui retourne la valeur a^b mod n, on fait m tours de boucle. Comme les opérations à l'intérieur de la boucle sont à nombre constant et sont en O(1),alors on peut conclure que la complexité est O(m).



Q1.5 Sur la base de ces deux courbes, nous pouvons voir que modpow est beaucoup plus rapide que modpow_naive, dont la vitesse varie moins. Nous pouvons conclure que modpow est plus efficace que modpow_naive.

3.1.2 Implémentions du test de Miller-Rabin

Le test de primauté de Miller-Rabin utilise un algorithme de randomisation pour déterminer si un nombre est premier ou non. Le test de Miller-Rabin s'appuie sur le fait que dans un corps, ce qui est le cas de /p si p est premier, l'équation $X^2 = 1$ n'a pour solutions que 1 et –1. Soit p un nombre impair quelconque. Soit s et d deux entiers tels que $p = 2^b \cdot d + 1$. Supposons que a soit un entier strictement inférieur à p.
Si nous pouvons trouver un tel a satisfaisant les deux équations ci-dessous:
1. $a^d mod n = 1$
2. $a^{2^r \cdot d} mod n = 1$
alors n n'est pas un nombre premier. On dit que a est la témoin de Miller de p.

```
int  witness(long a, long b, long d, long p);
int  is_prime_miller(long p, int k);
long random_prime_number(int low_size, int up_size, int k);
```

-int witness(long a, long b, long d, long p) utilise la fonction modpow pour teste si a est un témoin de Miller pour p, pour un entier a donné.

-long rand_long(long low, long up) retourne un entier long généré aléatoirement entre low et up inclus.
— int is_prime_miller(long p, int k) utilise witness et rand_long pour réaliser le test de Miller-Rabin en générant k valeurs de a au hasard, et en testant si chaque valeur de a est un témoin de Miller pour p. 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). -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 en utilisant rand_long et is_prime_miller .
En raison du caractère aléatoire utilisé dans cette méthode, nous devons connaître son erreur

Q1.7 En utilisant le fait que, pour tout entier p non premier quelconque, au moins 3/4 des valeurs entre 2 et p-1 sont des témoins de Miller pour p.
Nous pouvons conclure que la probabilité que l'un des nombres choisis au hasard ne soit pas un témoin de Miller est de 1/4. Étant donné que dans l'algorithme nous testons k fois de suite, la probabilité que tous les nombres testés ne soient pas des témoins de Miller est de $(\frac{1}{4})^k$.
une borne supérieure sur la probabilité d'erreur de l'algorithme est $(\frac{1}{4})^k$. L'erreur décroît de manière exponentielle, et est déjà faible lorsque k est supérieur à 20.

3.2 Implémentions du protocole RSA

3.2.1 Génération d'une clé publique et d'une clé secrète

Pour pouvoir envoyer des données confidentielles à l'aide du protocole RSA, il est d'abord nécessaire de générer deux clés : une clé publique pour chiffrer le message et une clé secrete pour le déchiffrer.Afin de sécuriser l'échange, une couple (clé secrète, clé publique) doit être générée de telle sorte qu'il soit impossible, de récupérer la clé secrète à partir de la clé publique. Le fonctionnement du protocole RSA est basé sur la difficulté de factoriser de grands nombres entiers. Plus précisément, afin de générer une couple(clé secrète, clé publique), le protocole RSA requiert deux (grands) nombres premiers distincts p et q (générés au hasard) et effectue les opérations suivantes
1. Calculer n = p × q et t = (p-1) × (q-1).
2. Générer aléatoirement des entiers s inférieur à t jusqu'à en trouver un tel que PGCD(s, t) = 1.
3. Déterminer u tel que s × u mod t = 1.
Le couple pkey = (s, n) constitue alors la clé publique, tandis que le couple skey = (u, n) forme la clé secrète.

```
void generate_key_values(long p, long q, long *n, long *s, long *u);
long extended_gcd(long s, long t, long *u, long *v);
```

-La fonction generate_key_values(long p, long q, long* n, long *s,long *u)utilise rang_long(dans libp.c) et extended_gcd pour génère la clé publique pkey = (s, n) et la clé secrète skey = (u, n), à partir des nombres premiers p et q, en suivant le protocole RSA.

3.2.2 Chiffrement et déchiffrement de messages

Dans cette section, nous nous concentrons sur la manière de déchiffrer des messages à l'aide d'une clé secrète sKey=(u,n) et de les chiffrer à l'aide d'une clé publique pKey=(s,n).
-Chiffrement : on chiffre le message m en calculant c = m^s mod n (c est la représentation chiffrée de m).
-Déchiffrement : on déchiffre c pour retrouver m en calculant m = c^u mod n.

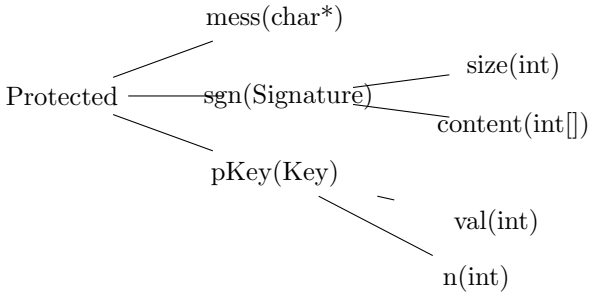
```
long *encrypt(char *chaine, long s, long n);
char *decrypt(long *crypted, int size, long u, long n);
void print_long_vector(long *result, int size);
```

- La fonction encrypy chiffre la chaîne de caractères avec la clé publique.
- La fonction decrypy déchiffre la chaîne de caractères avec la clé secrète.
- La fonction print_long_vector est de tester la validité de la fonction ci-dessus

Pour le tester, nous créons une paire de clés secrètes et publiques, nous codons et décodons un message donné (par exemple "hello") et nous comparons les messages codés et décodés pour voir s'ils correspondent au message original.

4 Déclarations sécurisée

Dans cette partie,on s'intéresse au problème de vote.On va supposer que l'ensemble de candidats est déjà connu, et que les citoyens ont juste à soumettre des déclarations de vote.



4.1 Manipulations de structures sécurisées

Dans notre modèle, chaque citoyen possède une carte électorale, qui est définie par un couple de clés :
— Une clé secrète (ou privée) qu'il utilise pour signer sa déclaration de vote. Cette clé ne doit être connue que par lui.
— Une clé publique permettant aux autres citoyens d'attester de l'authenticité de sa déclaration (vérification de la signature). Cette clé est aussi utilisée pour l'identifier dans une déclaration de vote, non seulement quand il vote, mais aussi quand quelqu'un souhaite voter en sa faveur.

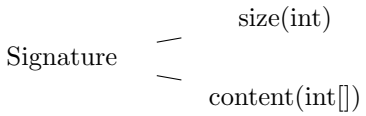
4.1.1 Manipulation de clés

Dans le protocole RSA, la clé public et la clé secrète d'un individu sont des couples d'entiers, notés respectivement pKey = (s, n) et sKey = (u, n). On écrit des fonctions pour initialiser des clés et passer d'un clé à sa représentation sous forme de chaîne de caractères et inversement.

```
typedef struct _Key {
    long val;
    long n;
} Key;

void init_key(Key *key, long val, long n);
void init_pair_keys(Key *pKey, Key *sKey, int low_size, int up_size);
char *key_to_str(Key *key);
Key  *str_to_key(char *str);
```

4.1.2 Signature



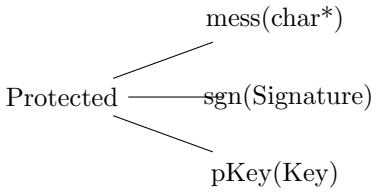
- La fonction init_key attribuer des valeurs à la clé qui est déjà allouée.
- La fonction init_pair_keys utilise random_prime_number (dans libp.c) pour générer deux premiers aléatoires, puis utiliser generate_key_values (dans rsa.c) pour créer les valeurs des clés secrète et publique, et enfin utilise init_key pour créer les clés secrète et publique.
- La fonction key_to_str passe d'un clé à sa représentation sous forme de chaîne de caractères.
- La fonction str_to_key passe d'un chaîne de caractères à clé.
Dans cette section, chaque électeur doit produire une déclaration de vote signée pour garantir son authenticité. Cette signature consiste en un tableau de long, gérée par l'émetteur de la déclaration au moyen de sa clé secrète, qui peut être vérifiée par d'autres personnes au moyen de la clé publique de l'émetteur.Le protocole de déclaration de vote:
Avant de publier la déclaration, on génère la signature associée à sa déclaration de vote. Cette signature prendra la forme d'un tableau de long obtenu par chiffrement du message mess avec la clé secrète de l'électeur.
L'électeur peut ensuite publier une déclaration sécurisée, composée de sa déclaration mess, de la signature associée, et de sa cl´e publique.Ceux qui souhaitent vérifier l'authenticité de la déclaration peuvent déchiffrement la signature en utilisant la clé publique de l'électeur.

```
typedef struct signature {
    int  size;
    long *content;
} Signature;

Signature *init_signature(long *content, int size);
Signature *sign(char *mess, Key *sKey);
char     *signature_to_str(Signature *sgn);
Signature *str_to_signature(char *str);
void      free_signature(Signature *sgn);
```

Les fonctions ont presque la même structure que la clé. - free_signature libere la memoire de Signature.

4.1.3 Déclarations signées



On crée des déclarations signées en utilisant la structure Protected qui contient la clé publique de l'électeur, sa déclaration de vote, et la signature associée.

```
typedef struct _Protected {
    Key      *pKey;
    Signature *sgn;
    char     *mess;
} Protected;

Protected *init_protected(Key *pKey, char *mess, Signature *sgn);
int       verify(Protected *pr);
char     *protected_to_str(Protected *sgn);
Protected *str_to_protected(char *str);
void      free_protected(Protected *pr);
```

les fonctions ont aussi presque la même structure que la clé. - La fonction verify vérifie que la contenue dans signature correspond bien au message contenus apres decoder.
Pour tester la validité de ces fonctions, nous générons d'abord les clés secrètes et publiques, puis les signatures

et les déclarations, et nous vérifions que les informations stockées dans les déclarations correspondent aux informations originales lorsqu'elles sont décodées.

4.2 Création de données pour le processus de vote

Nous allons simuler une séance de vote.Chaque citoyen aura une carte électorale unique contenant sa clé secrète et une clé publique. Le citoyen votera avec sa clé secrète pour garantir son anonymat. Le système de vote recueillera ces déclarations signées et utilisera toutes les clés publiques recueillies pour vérifier leur authenticité.

```
void generate_random_data(int nbCitoyen, int nbCandidate);
```

5 Base de déclarations centralisée

Dans cette section, nous allons créer un système de vote centralisé qui collectera toutes les déclarations de vote et annoncera ensuite le gagnant aux citoyens.Toutes les déclarations de vote seront incluses dans un fichier appelé declarations.txt, qui sera ensuite stocké dans un tableau lié. Afin de pouvoir vérifier l'intégrité des données et compter les votes, le système doit également récupérer toutes les clés publiques des citoyens et des candidats, qui sont stockées dans des fichiers appelés respectivement keys.txt et candidents.txt.

5.1 Lecture et stockage des données dans des listes chaînées

Dans cette section, nous allons lire les fichiers keys.txt et candidates.txt afin de récupérer les clés. Nous allons ensuite lire le fichier declarations.txt pour récupérer les déclarations signées.

5.1.1 Liste chaînées de clés

Nous allons créer des fonctions pour lire les fichiers contenant des clés et les stocker sous forme de listes chaînées.
Enfin, nous testons toutes les déclarations contenues dans ce fichier, dans le but de supprimer toutes les déclarations invalides (c'est-à-dire que le codage dans la signature est décodé différemment de l'information correcte)

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

CellKey *create_cell_key(Key *key);
CellKey *read_public_keys(char *fichier);
void    print_list_keys(CellKey *LCK);
void    delete_cell_key(CellKey *c);
void    delete_list_key(CellKey *LCK);
```

Pour tester la validité de la fonction, nous lisons le fichier généré par generateRandomData (keys.txt), générons une liste chainee et l'imprimons au terminal.

5.1.2 Liste chaînées de déclarations signées

Nous allons créer des fonctions pour lire les fichiers contenant des déclarations signées et les stocker sous forme de listes chaînées.

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

CellProtected *create_cell_protected(Protected *pr);
void          add_head_LCP(CellProtected **LCP, Protected *p);
CellProtected *read_protected(char *fileName);
void          print_list_protected(CellProtected *LCP);
void          delete_cell_protected(CellProtected *c);
void          delete_list_protected(CellProtected *LCP);
```

Pour tester la validité de la fonction, nous lisons le fichier généré par generateRandomData (declarations.txt), générons une liste chainee ,l'imprimons au terminal et le verifier.

5.2 Détermination du gagnant de l'élection

Une fois que toutes les déclarations et clés publiques signées ont été collectées, celles qui contiennent des signatures incorrectes sont retirées du système.

```
typedef struct hashcell {
    Key *key;
    int val;
} HashCell;

typedef struct hashtable {
    HashCell **tab;
    int size;
} HashTable;

int       verify_for_list_protected(CellProtected **LCP);
HashCell* create_hashcell(Key *key);
int       hash_function(Key *key, int size);
int       find_position(HashTable *t, Key *key);
HashTable* create_hashtable(CellKey *keys, int size);
void      delete_hashtable(HashTable *t);
Key*      compute_winner(CellProtected *decl, CellKey *candidates, CellKey *voters, int sizeC, int
     sizeV);
```

-Cette structure de donnée va nous permettre de construire deux tables de hachage:
1.Une table de hachage qui contient les clés publiques des candidats, et permet de compter le nombre de votes en faveur des candidats.
2.Une table de hachage qui contient les clés publiques des citoyens inscrits sur la liste électorale, et les valeurs sont égales à zéro pour les citoyens n'ayant pas (encore) voté, et sont égales à un pour ceux qui ont déjà voté.

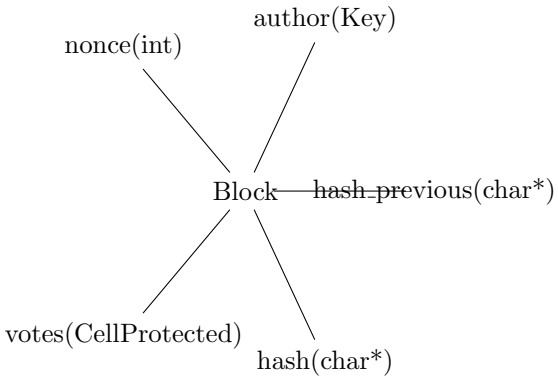
- La fonction verifyForList utilise la fonction verify(dans pro.h) verifie la validité de message contenue dans la déclaration signée et renvoie le nombre de signatures invalides et les supprime
- La fonction create_hashcell alloue une cellule de la table de hachage, et initialise ses champs en mettant la valeur à zéro.
- La fonction hash_function retourne la position d'un élément dans la table de hachage
- La fonction find_position cherche dans la table t s'il existe un élément dont la clé publique est key, en sachant que les collisions sont gérées par probing linéaire. Si l'élément a été trouvé, la fonction retourne

sa position dans la table, sinon la fonction retourne la position où il aurait dû être.

- La fonction create_hashtable crée et initialise une table de hachage de taille size contenant une cellule pour chaque clé de la liste chaînée keys.
- La fonction delete_hashtable supprime une table de hachage
- La fonction compute_winner calcule le vainqueur de l'élection.

6 Blocs et persistance des données

Dans cette partie, on s'intéresse à la gestion de blocs. un bloc contiendra :
— La clé publique de son créateur.
— Une liste de déclarations de vote.
— La valeur hachée du bloc.
— La valeur hachée du bloc précédent.
— Une preuve de travail.




```
typedef struct block {
    Key *author;
    CellProtected *votes;
    unsigned char *hash;
    unsigned char *previous_hash;
    int nonce;
} Block;
```

un bloc ne sera considéré comme valide que si la valeur hachée sting

```
unsigned char *str_to_SHA256(const char *str);
void    write_block(char *fileName, Block *block);
Block  *read_block(char *fileName);
char *block_to_str(Block *block);
void compute_proof_of_work(Block *B, int d);
int verify_block(Block *, int d);
void delete_block(Block *b);
void delete_block_partial(Block *b);
void free_block(Block *b);
Block *create_random_block(Key *author);
Block *init_block(Key *author, CellProtected *lcp);
```

-La fonction create_random_block génère un random bloc. -La fonction ini_block initialise un bloc avec nonce=0, l'auteur et les notes comme valeurs d'entrée, toutes les autres sont nulles. -La fonction block_to_str génère une chaîne de caractères représentant un bloc.
-La fonction str_to_SHA256 passe une chaîne de caractère à sa valeur hachée obtenue par l'algorithme SHA256.

-La fonction compute_proof_of_work rendre un bloc valide.
-La fonction verify_block vérifie qu'un bloc est valide.
-La fonction delete_block supprime un bloc.
-La fonction write_block permet d'écrire dans un fichier un bloc.
-La fonction read_block lire un bloc depuis un fichier.

6.1 Structure arborescente

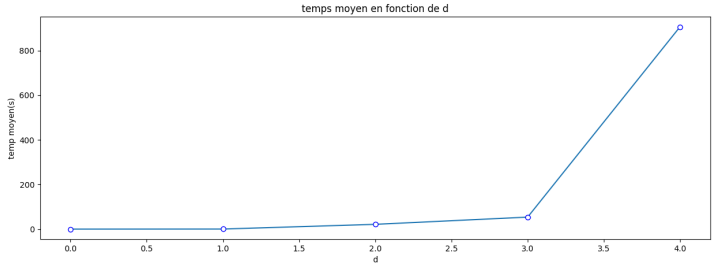
Dans une blockchain, chaque bloc contient la valeur hachée du bloc qui le précède,en cas de triche, on peut se retrouver avec plusieurs blocs indiquant le même bloc précédent.Dans cette section on fait confiance à la chaîne la plus longue (en partant de la racine de l'arbre), ce qui permet de retomber sur une chaîne de blocs.

6.1.1 Manipulation d'un arbre de blocs

```
typedef struct block_tree_cell {
    Block *block;
    struct block_tree_cell *father;
    struct block_tree_cell *firstChild;
    struct block_tree_cell *nextBro;
    int height;
} CellTree;
CellTree *create_node(Block *b);
int update_height(CellTree *father, CellTree *child);
void add_child(CellTree *father, CellTree *child);
void print_node(CellTree *node);
void print_tree(CellTree *ct);
void delete_node(CellTree *node);
void delete_tree(CellTree *node);
void delete_tree_partial(CellTree *ct);
CellTree *highest_child(CellTree *cell);
CellTree *last_node(CellTree *tree);
CellProtected *fusion(CellProtected *lcp1, CellProtected *lcp2);
CellProtected *longestList(CellTree *tree);
```

- La fonction longestList cree de la plus longue chaîne d'un arbre.

Q7.8 Pour tester la fonction computes_proof_of_work et obtenir la meilleure valeur de d, nous générons aléatoirement un bloc et traçons d en fonction du temps de calcul

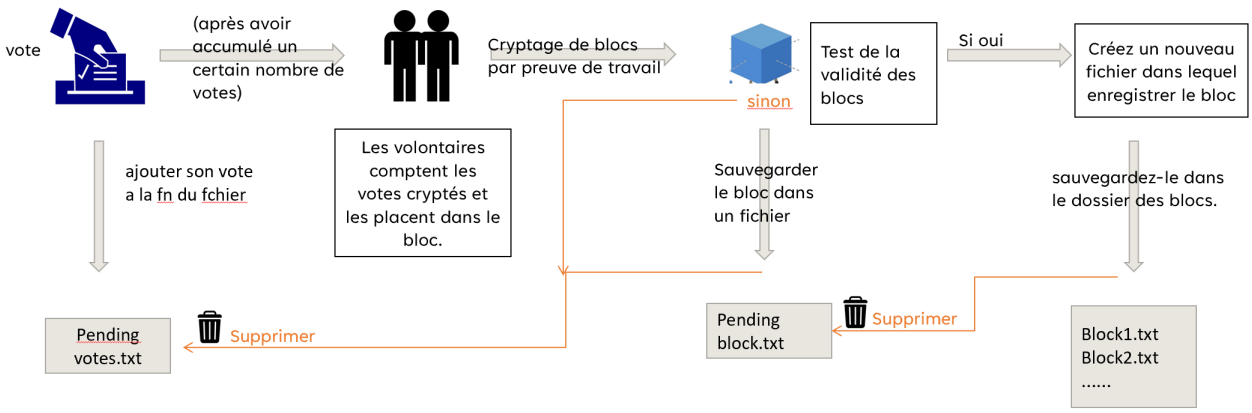


Le diagramme montre que pour que le bloc soit infaillible, il faut que d soit supérieur à 4, mais dans nos tests nous avons choisi d=3 afin de réduire le temps d'exécution.

Q8.8 La fonction 'fusion(CellProtected *lcp1,CellProtected *lcp2)' mélange deux CellProtected en un seul, dans notre schéma leur complexité est liée à la longueur de la première variable, en supposant que la longueur de 'lcp1'' est n, sa complexité est O(n).

Afin de rendre cette fonction O(1), nous pouvons créer une autre structure dans laquelle nous stockons le début et la fin de la liste des chaînes.

7 Simulation du processus de vote



SIMULATION

Dans cette section, nous allons simuler un citoyen qui soumet un vote et un assesseur (citoyen volontaire) qui crée un bloc valide. Nous allons ensuite créer l'arbre de blocs correspondant et calculer le gagnant de l'élection à partir de la plus longue chaîne de l'arbre de confiance. La chaîne la plus longue de la chaîne de blocs est considérée comme la chaîne valide. En effet, les faux votes sont insérés de manière aléatoire dans un certain sous-arbre, mais il est difficile pour un faussaire de forger directement un plus grand nombre de blocs que le vrai.

```
void submit_vote(Protected *p);
void create_block(CellTree *tree, Key *author, int d);
void add_block(int d, char *name);
CellTree *read_tree();
Key *compute_winner_BT(CellTree *tree, CellKey *candidates, CellKey *voters, int sizeC, int sizeV);
void Simulation(int d,int sizeC,int sizeV);
```

8 Conclusion

Dans notre conception, nous utilisons le cryptage rsa pour garantir la confidentialité du vote, la technologie blockchain pour parvenir à un système de vote décentralisé afin de garantir la confiance des électeurs, et la preuve de travail pour réduire considérablement la possibilité de fraude.

Q9.7 Les systèmes de vote électronique basés sur la blockchain permettent aux électeurs de voter à distance, ce qui, d'une part, protège l'identité des électeurs et garantit la confidentialité et, d'autre part, empêche les électeurs de recevoir des interférences pendant le processus de vote. La vérification absolue de tous les bulletins de vote entrant sur la plateforme blockchain est possible, ce qui se traduit par un mécanisme de vote et un contrôle des bulletins sécurisés et transparents, ainsi que par une forte protection contre la fraude électorale.
Si la blockchain peut assurer un certain degré de confidentialité des électeurs et de sécurité des élections après le vote, la nature anonyme des systèmes de vote électronique rend l'identification des électeurs problématique. Des pirates informatiques pourraient potentiellement exploiter les identités des électeurs et interférer avec le processus de vote, tandis que la fiabilité et la sécurité de la plateforme de vote sont également remises en

question.
Supposons que les données enregistrées soient valides. Après l'enregistrement du bloc, le pirate peut pirater la base de données et modifier les données du bloc ou ajouter un faux bloc. Le faux bloc a le hash du bloc existant comme hash_précédent afin d'être ajouté à l'arbre des blocs, supposons que le bloc est A. Si le pirate veut créer un bloc plus long, il doit créer plus de blocs que le sous-arbre de plus grande hauteur de ce bloc A.
Comme nous utilisons la preuve de travail pour valider les blocs, il faut beaucoup de temps pour que ces blocs falsifiés prennent effet, à moins que le bloc A choisi ne se trouve à la fin de l'arbre des blocs.
La combinaison de la preuve du travail et de l'option de la chaîne la plus longue est donc efficace pour réduire la fraude et minimiser son impact sur les résultats, mais n'empêche pas complètement la fraude. Il est tout à fait possible pour les pays et les groupes disposant d'une plus grande puissance de calcul d'interférer avec le déroulement des scrutins s'ils le souhaitent.