PROJET DE LU2IN006
BLOCKCHAIN ET ÉLECTIONS

POIRET Bastien 28000789
DIAB Dana 28705848

Introduction :

Ceci est notre compte rendu de projet de l'enseignement de structures de données. Nous y implémentons les outils de cryptographie à clé publique (RSA) nécessaires à la création d'une structure Blockchain permettant d'effectuer une élection fiable. Nous traitons ensuite des déclarations sécurisées, puis des structures chaînées de la base de données. La gestion du compte des votes et des listes électorales fonctionne grâce à deux tables de hachage. Nous créerons finalement les Blockchains dont la validité est garantie grâce à une preuve de travail et à un chiffrage, SHA256 de la valeur du bloc et de la valeur hachée du bloc précédent. La simulation finale utilisera une arborescence de ces blocs pour déterminer la chaîne la plus longue et désigner le vainqueur de l'élection.

Séance 1 : Développement des outils cryptographiques

Dans cette première séance, nous avons implémenté un algorithme d'exponentiation modulaire rapide permettant de réaliser des tests de miller rabin afin de générer de grands nombres premiers. Nous avons ensuite implémenté un protocole RSA consistant en une génération de clé publique et secrète permettant de crypter (avec la clé secréte) ou décrypter un message (avec la clé publique).

Fichiers dédiés à la séance 1 :

- `seance_1.c` : Code source des fonctions.
- `seance_1.h` : Fichier header contenant les déclarations de fonctions et de structures
- `main_seance_1.c` : Main qui principalement produit des courbes comparant les fonctions naïves et rapides d'exponentiation modulaire, tout en testant l'ensemble des fonctions.
- `test_main.c` : main fourni dans le sujet du projet permettant de tester l'encodage et décodage en RSA.

Structures manipulées :

On se contente ici de manipuler des entiers long, int et des chaînes de caractères.

Fonctions principales :

- `long modpow(long a, long m,long n)` et sa version récursive `long modpow_rec(long a, long m,long n)` : Servent à calculer l'exponentiation modulaire rapide, l'algorithme fonctionne par dichotomie sur l'exposant.

- `int witness ( long a , long b , long d , long p )` : pour p = 2bd + 1, retourne 1 si a est témoin de miller de p et 0 sinon.
- `int is_prime_miller ( long p , int k )` : vérifie par k tests de miller-rabin si p est un nombre premier. Avec une probabilité de (¼)^k qu'il ne le soit pas. Très peu de tests sont donc nécessaires pour obtenir une probabilité très forte que le nombre soit vraiment premier lorsque la fonction retourne 1.
- `long random_prime_number(int low_size, int up_size, int k )` : Retourne un nombre premier compris entre `low_size` et `up_size` et `k` correspond au nombre de tests de Miller-Rabin à réaliser.
- `long extended_gcd(long s, long t, long *u, long *v)` : Algorithme d'euclide étendu fourni qui retourne le PGCD et stocke les coefficients de Bezout.
- `void generate_key_values(long p, long q, long *n, long *s, long *u)` : Génère le trio de valeurs `n`, `s` et `u`, les couples `(s,n)` formant la clé secrète et `(u,n)` la clé publique.
- `long *encrypt(char *chaine, long s, long n)` : chiffre la chaîne fournie grâce à la clé fournie, retourne un tableau de `long` chaque case correspondant au cryptage d'un caractère de la chaîne.
- `char* decrypt(long *crypted, int size, long u, long n)` : Décrypte un tableau de `long` et retourne la chaîne obtenue.

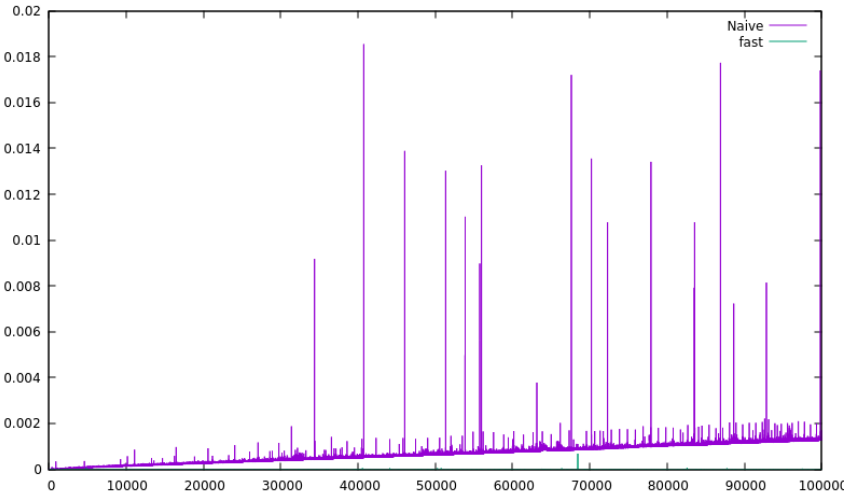
Réponse aux Questions :

Q1.1 - La complexité de la fonction `is_prime_naive` est de complexité linéaire (en omega de `n`, `n` étant la valeur fourni en paramètre).

Q1.2 - Avec la fonction `naive_is_prime`, on arrive vite à des valeurs très longues à calculer. Quand on atteint n=1000000000, on mesure en secondes le temps pour calculer si un nombre premier, ce qui n'est pas étonnant au vu de la fréquence du processeur.

Q1.3 - La complexité de `modpow_naive` est linéaire , en omega de `m`, `m` étant la deuxième valeur fourni en paramètre.

Q1.5 - Les courbes de ces deux fonctions montrent clairement la complexité pire cas de chacune d'elles, O(m) pour la fonction naïve et O(log(m) pour l'autre.



Ce graphique représente le temps d'exécution des deux fonctions en fonction de i (entre 0 et 100000) tel que i soit le deuxième paramètre des fonctions modpow. Dans ce graphe, on remarque que pour la fonction naive, la courbe est, en effet, linéaire. Par contre, la courbe de la deuxième fonction est quasi-invisible. La différence du temps d'exécution de ses deux fonctions augmente proportionnellement avec i.

Q1.7 - Dans le pire des cas, on ne teste l'existence d'un témoin qu'une seule fois, donc k=1. Et comme, au moins 3/4 des valeurs entre 2 et p-1 sont témoins de p, alors au plus 1/4 ne le sont pas. Donc la borne supérieur sur la probabilité d'erreur que p soit premier alors qu'il ne l'est pas est (1/4)^k = 1/4 pour k=1

Description des tests :

Tous les tests de *main* ont été réalisés avec valgrind les fonctions fournies ne provoquent donc en principe aucune fuite mémoire. Le fichier *test_main.c* utilise les fonctions de cryptographie pour créer un couple de clé, encode un message avec *encrypt* et le décrypte avec *decrypt*. Le test est concluant. D'autres tests non fournis ont été menés avec différentes valeurs de *low_size* et *up_size* dans *random_prime_number* et différents messages.

Séance 2 : Déclarations sécurisées

La seconde séance est consacrée à la formulation de déclarations sécurisées. Nous y définissons toutes les structures nécessaires à l'implémentation du système de vote. Une déclaration de vote sera signée grâce à la clé secrète d'un électeur. Il sera ensuite possible de certifier son authenticité via un décryptage utilisant la clé publique de ce dernier. Nous finirons par créer un jeu aléatoire de données afin de simuler une session de vote.

Fichiers dédiés à la séance 2 :

- *seance_2.c* : Code source des fonctions.
- *seance_2.h* : Déclarations des structures utilisées et prototypes de fonctions

- *main_seance_2.c* : Main testant l'ensemble des fonctionnalités importantes de la séance 2. Ce main doit être exécuté car il génère, sous forme de fichiers textes, le jeu de données utilisé dans la séance 3.
- *test_main_2.c* : Ce main correspond à celui fourni par les enseignants. Il sert à tester les diverses fonctions de sérialisation.

Structures manipulées :

```
typedef struct key
{
      long k; //clé
      long n; //modulo
}Key;
```

Structure du type *Key* servant à stocker un couple (k,n), formant une clé publique ou privée de la forme de celle générée à la séance 1 par *generate_key_values.*

```
typedef struct signature
{
      int size;      //taille du message
      long *content; //contenu crypté
}Signature;
```

Structure du type Signature stockant la taille en caractères ainsi que la signature d'un message crypté par encrypt (tableau de long).

```
typedef struct protected
{
      Key* pkey;       //clé publique de l'émetteur
      char * mess;     //message en clair
      Signature* sign; //signature du message
}Protected;
```

Structure du type Protected stockant une déclaration signée sous forme d'une clé publique de votant, du message correspondant a la déclaration de vote et de la signature de ce dernier.

Fonctions Principales :

Les trois structures ci-dessus disposent chacune d'une fonction d'initialisation, de sérialisation et de désérialisation.

De plus :

- *void init_pair_keys(Key* pk, Key* sk, long low_size, long up_size) :* Génère une paire de clé publique et privée a l'aide des outils de la

séance 1 et les stocke dans deux structures de type *Key* déjà allouées données en argument.
- *Signature* sign(char* mess, Key* sKey)* : Prend une chaîne correspondant à un message et la crypte utilisant la clé secrète fournie. Cette fonction utilise *init_signature* effectuant une allocation.
- *int verify(Protected *pr)* : Retourne 1 si la structure *Protected* fournie est authentique, c'est-à-dire si le message retrouvé par décryptage de la signature *pr->sign* avec la clé publique *pr->pkey* donne le même message que celui stocké dans *pr->mess*. Retourne 0 sinon.
- *void generate_random_data(int nv, int nc)* : fonction plus conséquente qui génère une série de données. Elle crée un fichier *candidates.txt*, *declarations.txt* et *keys.txt* en utilisant les fonctions de sérialisation précédentes pour former des chaînes de caractères. *nv* est le nombres d'individus et *nc* le nombre de candidats qui seront sélectionnés aléatoirement parmi les votants. Les votes seront ensuite effectués aléatoirement également.

Description des tests :

Le fichier *test_main_2.c* teste les fonctions de sérialisation dans les deux sens et ne produit aucune fuite mémoire. Le fichier *main_seance_2.c* teste principalement *generate_random_data* (fonction qui utilise toutes les fonctionnalités précédemment implémentées dans cette séance), et ne produit aucune fuite. Il fournit des fichiers ayant le format attendu qui ont pu être utilisés pour la suite.

Séance 3 : Base de déclarations centralisées

Cette troisième séance est consacrée à la création de listes chaînées à partir des fichiers créés précédemment par la fonction *generate_random_data*. On se servira ensuite de la liste chaînée des déclarations signées pour implémenter un algorithme qui en supprime les déclarations frauduleuses.

Fichiers dédiés à la séance 3 :
- *seance_3.c* : code source des fonctions
- *seance_3.h* : déclaration des structures chaînées et prototypes des fonctions
- *main_seance_3.c* : Test

Structures manipulées :

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

structure d'un élément de liste de clés.

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

structure d'un élément de liste de déclaration signée.

Fonctions Principales :

Chacune des deux structures ci-dessus dispose d'une fonction de création d'élément, d'une fonction d'ajout en tête, d'une fonction d'affichage de liste, d'une fonction de création de liste à partir des fichiers (*candidates.txt, keys.txt ou declarations.txt*), de suppression d'élément et de suppression de liste complète.

De plus :

- *void delete_non_valide(CellProtected ** LCP)* Fonction qui parcourt la liste de *Protected* fournie et qui supprime toutes celles qui sont non valides, c'est-à-dire, qui ne passent pas le test de la fonction *verify*.

Description des tests :

Le fichier *main_seance_3.c* lit le fichier *declarations.txt* et *keys.txt*, affiche les listes créées. Il supprime ensuite les déclarations non valides de la liste de déclarations. Il ne produit pas de fuite mémoire lors d'un test avec *valgrind* et des tests avec des déclarations frauduleuses ont été concluants, elles ont bien été supprimées.

Séance 4 : Tables de hachage et Détermination du gagnant de l'élection

Dans cette séance nous implémenterons une structure de table de hachage. La détermination du gagnant du vote fonctionnera via deux tables. La première contiendra les clés des candidats et le nombre de votes qu'ils ont obtenus. On vérifiera en *O(1)* qu'ils sont bien candidats. La deuxième contiendra les clés des votants et permettra de vérifier également en *O(1)* qu'ils sont bien inscrits sur la liste électorale et qu'ils n'ont voté qu'une seule fois. Nous gérerons les collisions par probing linéaire.

Fichiers dédiés à la séance 4 :

- *seance_4.c* : code source des fonctions
- *seance_4.h* : déclaration des structures chaînées et prototypes des fonctions
- *main_seance_4.c* : Test

Structures manipulées :

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

structure d'un élément de table de hachage

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

structure de la table de hachage

Fonctions Principales :

- `HashCell *create_hashcell(Key* key)` **:** Alloue une `HashCell` et initialise sa valeur a `0`.
- `int hash_function(Key* key, int size)` **:** Fonction de hachage qui retourne la somme des champs `k` et `n` de key modulo `size`.
- `int find_position(HashTable* t, Key* key)` **:** Cherche dans la table `t` si il existe une cellule qui a `key` pour clé. Si elle existe, retourne sa position, si elle n'existe pas, retourne la position où elle devrait être insérée. En cas de collision, on applique un probing linéaire. En cas d'échec de l'insertion, notamment si la table est pleine, on retourne `-1`.
- `HashTable * create_hashtable(CellKey* keys, int size)` **:** Alloue une table de hachage qui contient une cellule pour chaque clé de la liste chaînée.
- `void delete_hashtable(HashTable* t)` **:** Libère la table de hachage.
- `Key* compute_winner(CellProtected* decl, CellKey* candidates, CellKey* voters, int sizeC, int sizeV)` **:** Retourne la clé du vainqueur de l'élection. Crée la table de hachage des candidats et celle des votants à partir des listes `CellKey*`. Vérifie ensuite que les candidats ont le droit de voter, qu'ils ne l'ont pas déjà fait et qu'ils votent bien pour un candidat de la liste des candidats. Comptabilise alors le vote dans la table de hachage des candidats à la case correspondante. Le champ `val` des candidats correspond à leur nombre de voies, le champ `val` des votants est une valeur binaire qui indique s'ils ont déjà votés.

Description des tests :

Notre main de test récupère les fichiers de la séance 2 pour créer les listes de clés et de déclarations et supprime les déclarations non valides. Il exécute ensuite `compute_winner`, ce qui permet de tester le fonctionnement de nos tables de hachage. Une exécution avec `valgrind` et l'option `--leak-check=full` nous permet de traquer les fuites mémoire

pour les corriger. Des tests intermédiaires ont été réalisés en cours de développement mais ne figurent pas dans le main final. Ce main ne contient pas de fuite de mémoires.

Analyse :

La table de hachage permettant d'accéder aux candidats et aux votants et de modifier *val* en *O(1),* on remarque que c'est une structure particulièrement adaptée car elle permet d'éviter de parcourir entièrement la liste des candidats et des votants chaque fois qu'on émet un vote.

Séance 5 : Blockchain

Durant cette séance, nous allons à présent nous tourner vers une blockchain (chaîne de blocs), qui est une base de données décentralisée et sécurisée dans laquelle toutes les personnes sur le réseau possèdent une copie de la base ; dans notre contexte, les données sont les déclarations de vote signées, et les personnes sur le réseau sont les citoyens. Chaque block contiendra une preuve de travail qui nous permettra de vérifier si ce block est valide, de façon à ce que la valeur hachée du block commence par un nombre spécifique de zéros. Chaque block contiendra la valeur hachée du block précédent. Le hachage se fait à l'aide de la fonction SHA256.

Fichiers dédiés à la séance 5 :

- *seance_5.c* : code source des fonctions
- *seance_5.h* : déclaration des structures chaînées et prototypes des fonctions
- *main_seance_5.c* : Test l'ensemble des fonctions

Structures manipulées :

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

structure d'un block d'une blockchain

Fonctions Principales :

- *Block * create_block(Key* author, CelProtected * votes, unsigned char * hash, unsigned char * prev_hash, int nonce):* Cette fonction n'est pas demandée dans le sujet. Elle alloue un *Block* et initialise

ses champs à l'aide des valeurs en paramètre, en allouant dynamiquement les champs *hash* et *previous_hash.*

- *unsigned char * hashage (const char * s) :* Fonction qui hache la valeur en paramètre à l'aide de la fonction SHA256 et affiche la valeur hachée en hexadecimal.
- *void compute_proof_of_work (Block *b , int d) :* Fonction qui incrémente le *nonce* de *Block* b jusqu'à ce que la valeur hachée du *Block* commence par *d* zéros consécutifs.
- *void delete_block(Block * b) :* Fonction qui libère le *Block* mais pas son champ *author.* Et libère la liste *CellProctected* mais pas son champ *data.*

Description des tests :

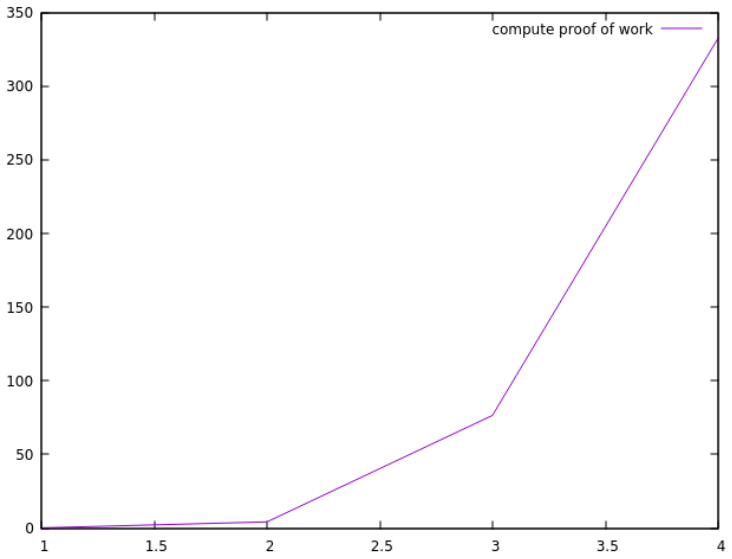
Notre main de test récupère les fichiers de la séance 2 pour créer les listes de clés et de déclarations. Il crée ensuite un *Block*, ce qui permet de tester le fonctionnement de toutes nos fonctions, notamment les fonctions qui sérialise un *Block* et inversement. Le main contient aussi une boucle qui chronomètre le temps d'exécution de la fonction *compute_proof_of_work* pour un nombre de zéro allant de 1 à 4 et stock ces valeurs dans le fichier *sortie_compute_proof.txt*, qui va nous permettre de tracer la courbe *coubres_compute_proof.png.* Une exécution avec *valgrind* et l'option *--leak-check=full* nous permet de traquer les fuites mémoire pour les corriger. Ce main ne contient pas de fuite de mémoires.

Réponse aux Questions :

Q-7.8 Le temps d'exécution de la fonction compute_proof_of_work dépasse 1 seconde pour d=2.

Analyse :

Notre analyse de cette séance se concentre sur la fonction *compute_proof_of_work*. Cette fonction permet de rendre un *Block* valide, en incrémentant son *nonce* jusqu'à ce que sa valeur hachée commence par *d* zéros. D'après le graphique ci-dessus, on remarque que le temps d'exécution de la fonction *compute_proof_of_work* en fonction du nombre de zéros est exponentiel. Cela nous montre que la fraude est extrêmement difficile comme on utilise une fonction de hachage cryptographique, c'est à dire une fonction de hachage facile à calculer mais difficile à inverser (one way function). Cependant une fois qu'on a accès au *nonce,* il est très facile de s'assurer de la validité du *Block.*



Séance 6 : Structures arborescentes

Durant cette séance, nous allons travailler sur une structure d'arbre de block. Dans une blockchain, chaque block contient la valeur hachée du bloc qui le précède, ce qui forme une chaîne. Cependant, en cas de triche, on peut se retrouver avec plusieurs blocs indiquant le même bloc précédent, ce qui conduit à une structure arborescente. Dans ce cas, la règle à suivre est de faire 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.

Fichiers dédiés à la séance 6 :

- *seance_6.c* : code source des fonctions
- *seance_6.h* : déclaration des structures chaînées et prototypes des fonctions
- *main_fusion_HC.c* : Test de la fonction *fusion_highest_child*
- *main_seance_6.c* : Tests de toutes les autres fonctions

Structures manipulées :

```
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;
```

structure du noeud d'un arbre, contenant un block

Fonctions Principales :

- *CellTree* create_node(Block * b)* : Fonction qui alloue et retourne une *CellTree* avec ses champs *Block* = *b* et `height` = 0.
- *void add_child(CellTree* father, CellTree* child)* : Fonction qui ajoute *child* au noeud *father.*
- *CellTree* highest_child(CellTree* cell)* : Retourne le fils de *cell* qui a le plus grand *height.*
- *void fusion_cell_protected(CellProtected ** fst, CellProtected ** snd)* : Fusionne les listes fst et snd dans fst.
- *CellProtected* fusion_highest_child(CellTree * tree)*: Fonction qui fusionne les listes de *votes* des *Block* des nœuds de la plus longue chaîne.

Description des tests :

Notre main du fichier *main_seance_6.c* récupère les fichiers de la séance 2 pour créer les listes de clés et de déclarations. Il crée 4 *Block*, avec lesquels on crée, ensuite, un *CellTree.* Ceci nous permet de tester le fonctionnement de toutes nos fonctions, notamment les fonctions d'affichage. Notre main du fichier *main_fusion_HC.c* crée un arbre *CellTree* avec lequel on test la fonction *fusion_highest_child*. La liste retournée par cette fonction contient bien la fusion des listes de votes des *Block* constituant la plus longue chaîne. Une exécution avec *valgrind* et l'option *--leak-check=full* nous permet de traquer les fuites mémoire pour les corriger. Ces deux mains ne contiennent pas de fuite de mémoires.

Analyse :

Q- 8.8 La fonction *fusion_cell_protected* a une complexité linéaire sur la longueur de son argument *fst.* On pourrait obtenir une fusion en temps constant en modifiant la structure de façon à avoir un pointeur sur le dernier élément de la liste, ce qui nous éviterait de la parcourir jusqu'au bout pour y accéder. Cela est possible avec une liste doublement chaînée circulaire. Nous n'implémenterons pas cette amélioration ici.

Séance 7 : Simulation du processus de vote

La séance finale de ce projet porte sur la simulation d'une Blockchain. On substituera des fichiers au véritable réseau. Un répertoire Blockchain simulera la copie locale d'un utilisateur, chaque fichier représentant un bloc. Un main final permettra de tester l'entièreté du projet et d'afficher le gagnant de l'élection.

Fichiers dédiés à la séance 7 :

- *Blockchain* (Répertoire) : contient les blocs valides ajoutés. ATTENTION : LE CONTENU DE CE RÉPERTOIRE DOIT ÊTRE VIDE AVANT EXÉCUTION DU MAIN_FINALE . Dans le cas contraire des erreurs et fuites de mémoire ont lieu.
- *Pending_block.txt* : Contient le dernier bloc fabriqué avant qu'il soit vérifié et ajouté au répertoire *Blockchain*.
- *Pending_votes.txt* : Contient un vote en attente d'être ajouté dans un bloc.
- *seance_7.c* : code source des fonctions.
- *seance_7.h* : déclaration des structures chaînées et prototypes des fonctions.
- *main_seance_7.c* : Test.
- main_finale.c : Main final du projet générant 1000 votants et 5 candidats.

Fonctions Principales :

- *void submit_vote(Protected *p) :* Prend une déclaration et l'écrit dans le fichier *Pending_votes.txt.* Si le fichier n'existe pas, elle le crée.
- *void createBlock(CellTree* tree,Key* author, int d) :* Crée un bloc valide à partir de la déclaration contenue dans *pending_votes.txt* et de la valeur hachée de *last_node(tree)*. Place ce bloc dans le fichier *Pending_block.txt* via la fonction *block_to_file*. Supprime *pending_votes.txt* dans tous les cas.
- *void add_block(int d,char * name) :* Si le bloc contenu dans *Pending_block.txt* est valide, l'ajoute au répertoire *Blockchain*. Supprime *Pending_block.txt.*
- *CellTree* read_tree() :* Cette fonction lit tous les fichiers du répertoire Blockchain et construit un arbre correspondant.
- *Key * compute_winner_BT(CellTree* tree, CellKey * candidates, CellKey* voters,int sizeC, int sizeV) :* Calcule le vainqueur de l'élection en se basant sur la plus longue chaîne de l'arbre.

Description des tests :

Le main du fichier *main_seance_7.c* récupère les fichiers de la séance 2 pour créer les listes de clés et de déclarations. Il soumet un vote dans le fichier *Pending_vote.txt* . Ensuite, un *Block* est créé à partir du vote soumis et écrit dans le fichier *Pending_Block.txt*. Après validation du *Block*, il l'écrit dans */Blockchain/Block1.txt.* Ce main ne contient pas de fuite mémoire.

Le main du fichier *main_finale.c* constitue le programme final de ce projet. Ceci nous permet de tester le fonctionnement de l'ensemble de nos fonctions. Ce main génère 1000 votants (donc 1000 votes) dont 5 candidats. Puis, il créé des *Block* valides, chaque 10 votes soumis, pour enfin construire un *CellTree* qui nous aidera à déterminer le vainqueur des élections. Une exécution avec *valgrind* et l'option *--leak-check=full* nous

permet de traquer les fuites mémoire pour les corriger. Ce main ne contient pas de fuite de mémoires.

Analyse :

Q-9.7) L'utilisation du Blockchain dans le cadre d'un processus de vote semble pertinent. La cryptographie RSA pouvant garantir l'authenticité d'une déclaration avec une bonne fiabilité pour des nombres générés assez grands. Le consensus consistant à toujours choisir la chaîne de blocs la plus longue ne semble à première vue pas imparable, on peut imaginer qu'une fraude puisse être possible avec une puissance de calcul suffisante, avec un grand nombre de machines ou d'individus par exemple.

Conclusion :

Un projet intéressant et dans l'air du temps que nous avons apprécié. Très formateur sur la méthodologie d'un travail plus conséquent et sur plusieurs mois.

Bastien : Comme réflexion personnelle j'ajouterais que notre travail en tant qu' informaticiens repose sur l'implémentation de technologies mais qu'il ne faut pas oublier de questionner nos choix de conceptions qui ne sont pas anodins et sont en réalité politiques et philosophiques, surtout sur un sujet aussi important que le vote dans une démocratie. Le scrutin uninominal a été choisi ici, j'imagine, pour des raisons de simplicité. Cependant, si on devait informatiser notre système de vote en France, il serait judicieux de choisir un mode de scrutin plus robuste, plus démocratique, avec de meilleures propriétés. J'aurais apprécié à titre personnel que le sujet sensibilise les étudiants sur la question des modes de scrutin. Le scrutin uninominal à deux tours nous semble tellement naturel qu'on ne le remet pas en question en tant que citoyens. Pourtant, ses mauvaises propriétés notamment quant au vote utile, aux alternatives non pertinentes et à l'absence de prise en compte du vote blanc, sont consensuelles.