Rapport de projet

Réda Boudrouss

Contents

Introduction

Ce projet a pour but de faire un système de vote “electoral” de manière décentralisé avec de la blockchain. Toute notre crypotgraphie et sécurité est assurée par une approche naïve du chiffrement RSA.

Description du projet

Organisation des dossiers

Le projet se divise comme suit :

- **`bin/`** qui contient les fichiers compilés
- **`blockchain/`** qui contient les données stockées et les données aléatoires générés
- **`data/`** qui contient les données d’analyses des différents graphiques
- **`img/`** qui contient les graphiques stockés sous format image
- **`report/`** qui contient tout les fichiers nécessaires à la réalisation du présent rapport
- **`src/`** Ce dossier contient l’essentiel du code source du projet, nous expliqueront plus tard l’organisation du code.

- **include/** contient tout les headers du projet
- **tests/** contient tout les jeux de test unitaires par fonction. Tout est assurée sans memory-leak !

Organisation du code

Chaque header dans le dossier **include/** a son code dans le fichier portant le même nom dans **src/** et a ses tests unitaires dans les fichiers portant le même noms suivi de **_tests.c** dans le dossier **tests**. Excépté le fichier **files.h** et le fichier **main.c** qui n'a pas de header.

- **prime**

Ces fichiers contiennent toutes les fonctions de manipulation des nombres premiers et de l'aléatoire.

- **rsa**

Ces fichiers contiennt une approche naïve du RSA avec une manière de générer les clés publiques & privés et de décrypter/crypter des messages.

- **keys**

Ces fichiers reposent sur la structure suivante :

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

Cette structure permet de stocker et gérer les informations nécessaires aux clés publiques et privés RSA. Les fichiers contiennent donc les fonctions permetant d'initialiser les clés, de les libérer et de les convertire en chaine de caractère et vice versa.

- **signs**

Ces fichiers reposent sur la structure suivante :

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

Cette structure permet de stocker les informations nécessaire à une signature. Dans notre code cela remprésente à la cle publique du candidat (en chaine de caractère) encrypté en RSA avec la clé privé du votant.

Les fichiers contiennent alors les différentes fonctions d'initialiser et de libérer une structure **Signature** et donc de signer une chaine de caractère, sans oublier la conversion en chaine de caractère et vice versa.

- **protec**

Ces fichiers reposent sur la structure suivantes :

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

Cette structure permet de stocker les informations d'une déclaration de vote. **pKey** étant la clé publique du votant, **mess** la chaine de caractère de la clé publique du candidat et **sgn** la signature.

Les fichiers contiennent donc les différentes fonctions pour initialiser, vérifier et libérer une **Signature** et convertir une chaine de caractère.

- **cells**

J'aurais du découper ce fichier en plusieurs fichiers.

Ces fichiers regroupent les fonctions dépendantes des 4 structures suivantes :

- **CellKey** & **CellProtected**

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

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

Cette structure est une liste chainée de `Key` ou de `Protected`. Cela nous permettera de gérer une grande quantité de clé ou de déclaration de vote dont on ne connait pas la taille en avance. Nous avons donc des fonctions pour les manipuler. (Ajouter un élément, libérer la liste, l'afficher et lire les données d'un fichier).

- **HashCell** & **HashTable**

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

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

Cette structure est une table de hashage. Cela permet de récupérer et modifier les informations sur les votants et les candidats de manière rapide. Cela est nécessaire car notre programme doit être capable de gérer une très grand nombre de votant.

Nous avons donc aussi toutes les fonctions nécessaires pour cela. la fonction de hashing, et les fonctions permettant de créer libérer insérer une **`HashCell`**.

Nous avons aussi une fonction importante :

```
Key *compute_winner(CellProtected *decl, CellKey *candidates, CellKey
    *voters, int sizeC, int sizeV);
```

Cette fonction détermine le gagnant de l'élection à partir de la liste chainées de vote et des clées des votant et des candidats.

- **`Block`**

Ces fichiers reposent sur la structure suivantes :

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

Cette structure représente un block d'une blockchaine. Elle contient la liste des déclarations de votes, le hash du block et le hash des block précédent, le `nonce` et la clé de l'auteur.

Nous avons donc des fonctions pour écrire/lire un block dans un fichier. convertir en str et vice versa et faire l'algorithme de proof of work.

- **`btree`**

Ces fichiers reposent sur la structure suivantes :

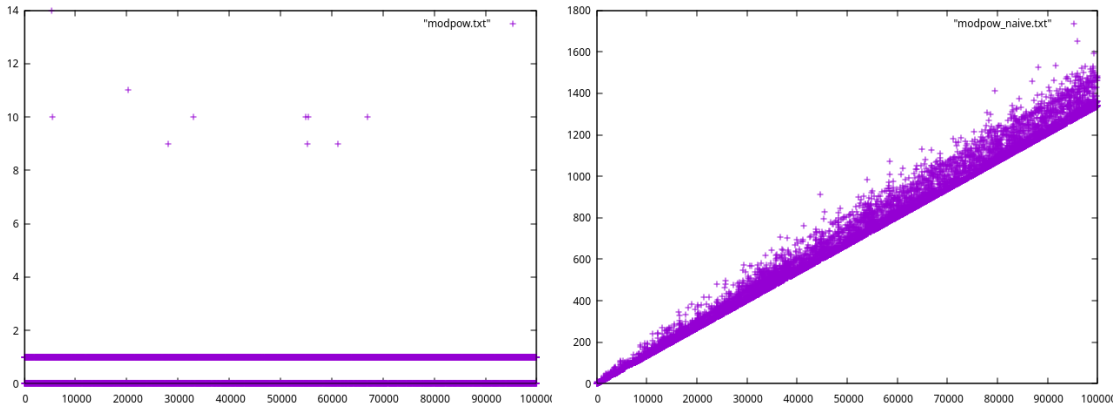
```
typedef struct block_tree_cell
{
    Block *block;
    struct block_tree_cell *father;
    struct block_tree_cell *firstChild;
    struct block_tree_cell *nextBro;
    int height;
} CellTree;
```

Cette structure représente l'arbre de tout les blocks de la blockchain. Chaque bloc a un block qui le précéde, et plusieurs blocks suivant.

Dans ces fichiers nous avons donc toutes les fonctions d'insertion d'un block dans un arbre, d'écriture de l'arbre dans le dossier **`blockchain/`**.

Réponses aux questions

1. 1. La complexité de la fonction **`is_prime_naif`** est $\mathcal{O}(n)$.
   2. 6163 est le plus petit nombre premier calculé en 2 milisecondes ou plus.
   3. **'modpow_naif"** a pour complexité $\mathcal{O}(n)$.
   4. Performance des fonctions **`modpow`**:



   Dans les graphes ci-dessous on voit bien que la fonction **`modpow`** est linéaire (du moins jusqu'à 10000) alors que le temps d'execution de **`modpow_naive`** est linéaire et augmente avec n. On peut donc conclure que **`modpow`** est beaucoup plus rapide surtout pour les grands nombres.

7. 8. À partir de $d = 4$, la fonction **`compute_proof_of_work`** prend plus d'une seconde sur ma machine. Le temps de calcule monte très vite cependant.

8. 8. la fonction **`fusion_chaines`** est en $\mathcal{O}(n)$. On aurait pu introduire des listes doublement chaine pour que ça soit en $\mathcal{O}(1)$.

Conclusion

Pour conclure, utiliser un la technologie de la blokchain pour faire un système électorale a ses avantages. Elle permet une assez grande transparence vu que n'importe qui peut vérifier les résutlats lui-même et une bonne sécurité grâce aux chiffrement RSA et grâce au principe du proof of work.

Cependant, notre code tel qu'il est ne peut pas du tout être utilisé dans un vrai système électorale. Notre approche du RSA par exemple est assez rudimentaire et on se limite aux long int, à cause de ça nous pouvons retrouver en moins d'une heure une clé privée à partir d'une clé publique.