Compte-Rendu

IORDACHE Paul-Tiberiu, LUONG Ethan

Semestre 4

Introduction

Contexte

On considère le processus électoral français en deux tours. Un scrutin est supposé désigner un et un seul candidat. L'objectif de ce projet est d'implémenter les outils et les structures nécessaires à la désignation du vainqueur, tout en garantissant intégrité, sécurité et transparence. De tels outils faciliteraient non seulement le décompte des scrutins, mais également le vote en lui-même. Le contexte proposé rendrait en outre le vote à distance envisageable, contribuant ainsi à la diminution du taux d'abstention, estimée à 66,7 % aux élections de juin 2021).

Structure générale du code

D'une manière générale, code et résultats sont répartis dans divers dossiers, nommés à l'égard de leur finalité. Par exemple, les graphes présentés peuvent être retrouvés dans le dossier Charts. D'une manière générale, on a les correspondances suivantes :

- `CryptographyTools` : Partie 1
- `ProtectedDeclarations` : Partie 2
- `Lists` : Partie 3
- `HashTables` : Partie 4
- `Blocks` : Partie 5
- `Trees` : Partie 5

Chacun de ces dossiers comprend un fichier `main.c` et un exécutable `main`. La compilation est facilitée par l'ajout d'un `Makefile`.

Table des matières

Chapitre 1

Développement d'outils cryptographiques

Dans cette partie, on vise à crypter et décrypter des chaînes de caractères au moyen du protocole RSA. L'exécutable main permet d'exécuter chacun des tests associés.

1.1 Résolution du problème de primalité

1.1.1 Implémentation par une méthode naïve

Cette méthode consiste en énumérer tous les entiers impairs avant de vérifier leur primalité à l'aide de la fonction :

– `int is_prime_naive(long p)`

dont la complexité est en $O(p)$. Il est possible, avec cette fonction, de déterminer en moins de 0.002 seconde que 84467 est un nombre premier.

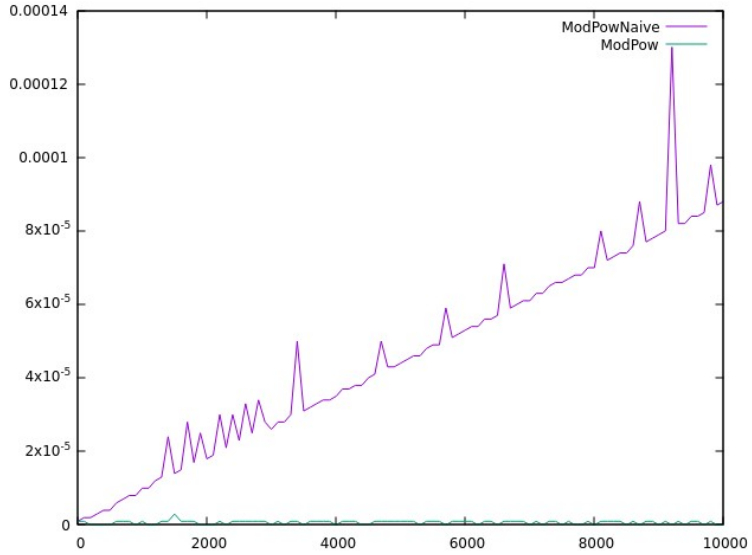
1.1.2 Exponentiation modulaire rapide

On cherche dans cette section à calculer efficacement $a^m \mod n$. Deux fonctions sont prévues à cet effet :

– `long modpow_naive(long a, long m, long n)`
– `long modpow(long a, long m, long n)`

Des résultats graphiques permettent de comparer la rapidité de ces deux algorithmes.

Comparaison des temps de calculs
$0 \leq m \leq 10000, a = 100,\ n = 10$



Les données proviennent du fichier SortieExpMod.txt constitué après exécution du `main`. Il est évident que la seconde méthode en $O(\log_2(m))$ est significativement plus rapidement que la première en $O(m)$.

1.1.3 Test de Miller-Rabin

Pour déterminer efficacement la primalité d'un `long`, nous nous sommes appuyés sur le test de Miller-Rabin de complexité pire-cas en $O(k \log_2^3(p))$. En ce qui concerne sa fiabilité : considérant que $\frac{3}{4}$ des valeurs entre 2 et p-1 sont un témoin du test de Miller-Rabin, ce dernier a ainsi une probabilité d'erreur majorée par $(\frac{1}{4})^k$. A cette fin, les fonctions suivantes ont été implémentées :

- `int witness(long a, long b, long d, long p)`
- `long rand_long(long low, long up)`
- `int is_prime_miller(long p, int k)`

1.1.4 Génération de nombres premiers

Les sections précédentes contribuent finalement à la génération aléatoire d'un nombre premier dont le nombre de bits est compris entre `low_size` et `up_size` :

- `long quickexp(long a, long b)`
- `long random_prime_number(int low_size, int up_size, int k)`

1.2 Implémentation du protocole RSA

1.2.1 Génération d'une paire (clé publique, clé secrète)

Partant du fonctionnement du chiffrement asymétrique, on se donne p et q deux nombres premiers générés aléatoirement. On détermine alors $n = p \times q$, $t = (p-1)(q-1)$, $s \in \mathbb{N}, PGCD(s,t) = 1$ et $u \in \mathbb{N}, s \times u \mod t = 1$. Leur donnée permet de générer une clé publique (s,n) et une clé secrète (u,n) :

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

1.2.2 Chiffrement et déchiffrement des messages

Les fonctions suivantes :

- long *encrypt(char *chaine, long s, long n)
- char *decrypt(long *crypted, int size, long s, long n)

permettent de chiffrer des chaînes de caractères en stockant chaque caractère sous forme de long dans un tableau, et de décrypter un tel tableau afin de retrouver la chaîne de caractères originelle. Les fonctions ont été testées à l'aide du `main` avec des chaînes de caractères saisies par l'utilisateur.

Chapitre 2

Déclarations sécurisées

Ce chapitre est dédié à la génération d'une base de données de clés et scrutins, retrouvable dans le dossier `Database`.

2.1 Manipulation de structures sécurisées

2.1.1 Manipulation de clés

On implémente dans cette section une nouvelle structure représentant une clé issue du protocole RSA, ainsi que des outils permettant de la manipuler :

- Key {long val, long n}
- Key *create_key()
- void init_key(Key *key, long val, long n)
- init_pair_keys(Key *pKey, Key *sKey, long low_size, long up_size)
- char *key_to_str(Key *key)
- Key *str_to_key(char *str)

2.1.2 Signature

On procède de même pour la représentation d'une signature qui consiste en la clé secrète d'un individu, et de son vote crypté :

- Signature {int size, long *content}
- 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)

2.1.3 Déclarations signées

On fait finalement de même pour représenter les scrutins des électeurs (triplet clé publique, message, signature) :

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

La fonction `verify` joue un rôle déterminant pour garantir de l'authenticité des déclarations. En outre, l'exécution du `main` permet d'éprouver les fonctions listées dans ce chapitre.

2.2 Création de données

Afin de simuler le processus de vote, un certain nombre de données est nécessaire. Pour générer un certain nombre nv de couples de clés (parmi lesquelles les nc candidats sont choisis aléatoirement), et des déclarations signées aléatoires, on élabore également une nouvelle structure stockant les couples de clés et indiquant si ces derniers correspondent à des candidats au moyen d'un champ de marquage `b` :

- KArray (Key *pKey, Key *sKey, int b)
- void generate_random_data (int nv, int nc)

En outre, on a les fichiers suivants :

- `Database/keys.txt` : sortie des couples de clés
- `Database/candidates.txt` : sortie des clés publiques des candidats
- `Database/declarations.txt` : sortie des déclarations signées

La fonction `generate_random_data` a été testée dans le cas $nv = 500$ et $nc = 20$ par l'exécutable `main`.

Chapitre 3

Base de déclarations centralisée

Dans ce chapitre, on considère un système de vote centralisé pour lequel on s'efforce d'implémenter différentes structures (Listes Chaînées et Tables de Hachage) afin de manipuler les données générées à l'issue du chapitre précédent. Chaque citoyen envoie ainsi un vote, dont on vérifie l'intégrité, au système de vote.

3.1 Lecture et stockage des données

Pour faciliter le stockage des données, on utilisera des structures de listes chaînées.

3.1.1 Liste chaînée de clés

On implémente une structure de liste chaînée de clés et des fonctions permettant de les manipuler et d'interagir avec les flux d'entrée et de sortie :

- CellKey {Key *data, CellKey *next}
- CellKey *create_cell_key(Key *key)
- void insert_cell_key(CellKey **list, CellKey *cell)
- CellKey *read_public_keys(char *filename)
- void print_list_keys(CellKey *LCK)
- void delete_cell_key(CellKey *c)
- void delete_cell_list(CellKey *list)

3.1.2 Liste chaînée de déclarations signées

De même pour les déclarations signées :

- CellProtected {Protected *data, CellProtected *next}
- CellProtected *create_cell_key(Protected *pr)
- void insert_cell_protected(CellProtected **list, CellProtected *cell)

– CellProtected *read_public_protected(char *filename)
– void print_list_protected(CellProtected *LCP)
– void delete_cell_protected(CellProtected *c)
– void delete_cell_protected(CellProtected *list)

L'exécutable permet de lire et d'afficher le contenu des fichiers `keys.txt` et `declarations.txt` dans le dossier `Lists/Tests`. Les contenus étant égaux, nous en avons conclu que les fonctions étaient correctes.

3.2 Détermination du gagnant de l'élection

Après avoir récolté les données nécessaires, il s'agit de supprimer les déclarations signées frauduleuses :

– void delete_false_protected(CellProtected **list)

Cette fonction a été éprouvée par le `main` qui falsifie une déclaration signée quelconque d'une liste chaînée et vérifie qu'elle ait bien été enlevée (sortie dans les dossier `Lists/Tests`.

On s'attèle par la suite à l'élaboration d'un mécanisme de consensus. Pour ce faire, on implémente une structure de table de hachage et des fonctions de manipulation :

– HashCell {Key *key, int val}
– HashTable {HashCell **tab, int size}
– HashCell *create_hashcell(Key *key)
– 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)

A noter que la fonction de hachage `hash_function` est, dans notre cas, particulièrement efficace. Calculons sa probabilité de collisions. Notant les évènements A : "même valeur hachée", B : "même clé", C : "collision après hachage multiplicatif uniforme", pour $size \in \mathbb{N}$ fixé :

$$\mathbb{P}(B) = \frac{1}{|\{p \in \mathbb{P},\ 2 < p < 256\}|^2} = \frac{2}{53^2} \approx 0,0007$$

$$\mathbb{P}(A) = \mathbb{P}(C \cap B) + \mathbb{P}(C \cap B^c) = \mathbb{P}(B) + \mathbb{P}(C \cap B^c) \text{ car } (B \cup B^c = \Omega)$$

$$Or, \mathbb{P}(C \cap B^c) = \mathbb{P}(C) * \mathbb{P}(B^c) = \frac{1}{size} * 0,9993 \text{ car C idp. de B}$$

$$\text{Donc } \mathbb{P}(A) \approx 0,0007 + \frac{0,9993}{size} \approx \frac{1}{size}$$

On peut donc supposer notre fonction de hachage uniforme, ce qui revient à considérer la probabilité de collisions faible, pour une valeur de *size* suffisamment élevée.

Par ailleurs, la fonction `compute_winner` correspond à l'aboutissement du mécanisme de consensus. Elle permet en effet de déterminer le candidat majoritaire suite à la soumission des votes. Elle s'appuie pour cela sur le champ de `HashCell` : `val` permettant aussi bien de déterminer si un citoyen a voté, ou de compter le nombre de votes pour un candidat.

Chapitre 4

Blocs, persistance des données

Nous allons dans ce chapitre proposer un système de vote décentralisé, préférable au système centralisé du chapitre précédent pour des raisons de confiance. On s'intéresse ainsi à l'implémentation d'une blockchain, structure décentralisée et sécurisée pour laquelle tous les acteurs possèdent une copie de la base.

4.1 Structure d'un bloc et persistance

4.1.1 Lecture et écriture de blocs

Il s'agit dans un premier temps d'implémenter la structure de bloc, et ses fonctions d'écriture et lecture :

- Block {Key *author, CellProtected *votes, unsigned char *hash, unsigned char *previous_hash, int nonce}
- void write_block(char *filename, Block *b)
- Block *read_block(char *filename)

4.1.2 Création de blocs valides

Pour pouvoir créer un bloc, on demande à l'utilisateur de fournir une preuve de travail, garantissant la validité du bloc. Pour ce faire, en plus d'utiliser la bibliothèque `<openssl/sha.h>` et sa fonction `SHA_256`, on ajoute les fonctions suivantes :

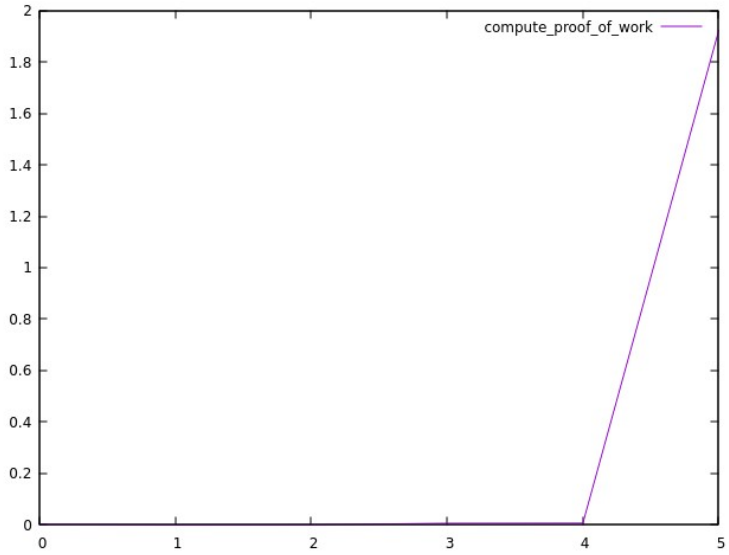
- char *block_to_str(Block *b)
- unsigned char *hash_SHA256(char *str)
- void compute_proof_of_work(Block *B, int d)
- int verify_block(Block *b)
- void delete_block(Block *b)
- void free_block(Block *b)

La fonction `hash_SHA256` permet de fournir une valeur hachée au format hexadécimal du bloc, à partir de `block_to_str`. Il est ensuite possible d'implémenter une fonction calculant la preuve de travail à partir de l'algorithme suivant :

1. Initialiser le champ `B->hash`, puis `B->nonce` $= 0$
2. Construire la chaîne `B->hash` $\cdot$ `B->nonce`
3. Hacher la chaîne obtenue avec `hash_SHA256`
4. Vérifier que la chaîne commence par un nombre de zéros $\geq d$
5. Répéter si ce n'est pas le cas en incrémentant `B->nonce` de 1

Le graphe suivant montre qu'à l'aide de cet algorithme, il est possible de trouver, en moins d'une seconde, des preuves de travail correspondant à $d < 4$. Ces résultats ainsi que l'allure du graphe permet de conjecturer une complexité exponentielle de `compute_proof_of_work`.

Performances de compute_proof_of_work
$0 \leq m \leq 10000, a = 100,\ n = 10$



Il est heureusement plus facile de vérifier la validité du bloc en réalisant uniquement les étapes 2 à 4 de l'algorithme précédent. A noter également que le nombre de zéros d exigé sur la représentation hexadécimale, revient à exiger que la représentation binaire débute par au moins $4d$ zéros.

Pour terminer, précisons que nous avons choisi de libérer la totalité des données contenues dans un `Block` avec la fonction `free_block` contrairement à ce que préconisait le sujet, afin d'éviter toute sorte de fuite mémoire. La totalité de ces fonctions est testée par `Blocks/main`.

4.2 Structure arborescente

4.2.1 Manipulation d'un arbre de blocs

Il s'agit ici d'implémenter une structure d'arbre de blocs et ses primitives élémentaires :

- CellTree {Block *block, CellTree *father, CellTree *firstChild, CellTree *nextBro, int height}
- CellTree *create_node(Block *b)
- int update_height(CellTree *father, CellTree *child)
- void add_child(CellTree *father, CellTree *child)
- void print_tree(char *filename, CellTree *tree)
- void delete_node(CellTree *node)
- void delete_tree(CellTree *tree)
- void free_node(CellTree *node)
- void free_tree(CellTree *tree)

A noter que les fonctions `free_node` et `free_tree` sont à `CellTree` ce que `free_block` est à `Block`.

4.2.2 Détermination du dernier bloc

Le consensus étant de toujours faire confiance à la chaîne la plus longue, c'est un enjeu important que de pouvoir en déterminer le dernier bloc :

- CellTree *highest_child(CellTree *cell)
- CellTree *last_node(CellTree *tree)

En remontant de père en père depuis le `CellTree*` renvoyé par `last_node`, à l'aide du champ `T->father`, il est ainsi possible de récupérer la plus longue chaîne de blocs et de garantir la bonne gestion des données frauduleuses.

4.2.3 Extraction des déclarations de vote

On peut, de la même manière, récupérer la totalité des listes chaînées de déclarations signées en fusionnant les listes contenues d'un bloc à l'autre à l'aide des fonctions :

- void merge_list_protected(CellProtected **list1, CellProtected **list2)
- CellProtected *longest_chain_protected(CellTree *tree)

A noter qu'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.

4.3 Simulation du processus de vote

On utilisera dans cette section le répertoire `Blockchain` contenant les fichiers temporaires suivants : `Blockchain/Pending_votes.txt`, et `Blockchain/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.

4.3.1 Vote et création de blocs valides

On intègre les fonctionnalités de soumission de vote et d'ajout de blocs, en se servant des fichiers `Blockchain/Pending_votes.txt`, et `Blockchain/Pending_block.txt` comme transition :

- `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 après vérification avec `add_block`.

4.3.2 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 :

- `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`. On peut finalement déterminer le vainqueur de l'élection à l'aide de `compute_winner_BT`.

Conclusion

A travers les différents chapitres proposés, nous avons pu proposer une implémentation de Blockchain appliqué à un processus électoral. Nous avons pu en déduire les avantages suivants :

- Anonymat
- Décentralisation du système de vote
- Gestion des fraudes efficace
- Infalsifiabilité
- Sécurité
- Transparence et traçabilité
- Vote en distanciel

Mais également les inconvénients suivants :

- Cas où ≥ 2 chaînes de longueur maximale ont été trouvées
- Difficile à appréhender
- Impossible de supprimer un bloc ajouté
- Manque d'accessibilité (en raison de la complexité exponentielle de la preuve de travail, des machines puissantes sont requises pour miner)
- Panne de serveurs
- Vulnérable à l'attaque à 51% (un pirate détient plus de la moitié de la puissance de calcul du réseau)

Au final, le concept de Blockchain promet de belles perspectives d'avenir, notamment pour sa transparence et sa sécurité, nécessaires au bon déroulement d'une élection. La possibilité de voter en ligne est également attirante dans le cadre de la réduction du taux d'abstention.

Sa complexité et son fort impact écologique nous invitent néanmoins à attendre qu'elle progresse davantage avant de l'implémenter définitivement.