

Projet : Blockchain appliquée au processus électoral

Yuxin XUE et Zhirui CAI

March 29, 2022



Contents

1	Introduction	3
2	Description du code général	3
3	Développement d'outils cryptographiques	4
3.1	Résolution du problème de primarité	4
3.1.1	Implémentations par une méthode naive	4
3.1.2	Implémentations du test de Miller-Rabin	5
3.2	Implémentations du protocole RSA	6
3.2.1	Génération d'une clé publique et d'une clé secrète	6
3.2.2	Chiffrement et déchiffrement de messages	6
4	Déclarations sécurisée	7
4.1	Manipulations de structures sécurisées	7
4.1.1	Manipulation de clés	7
4.1.2	Signature	7
4.1.3	Déclarations signées	8
4.2	Création de données pour le processus de vote	8
5	Base de déclarations centralisée	9
5.1	Lecture et stockage des données dans des listes chaînées	9
5.1.1	Liste chaînées de clés	9
5.1.2	Liste chaînées de déclarations signées	9
5.2	Détermination du gagnant de l'élection	10

1 Introduction

Ce projet s'intéresse à la problématique de la désignation du vainqueur du processus électoral.

Au cours du processus électoral, chaque participant peut déclarer sa candidature ou voter pour le candidat élu. Le déroulement du processus électoral a traditionnellement soulevé des difficultés en matière de confiance et de transparence, tandis que le taux d'abstention très élevé en France est un problème qui pourrait être amélioré. 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

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.

C:

makefile					
scr					
Makefile					
key.c		key.h			
lcc.c		lcc.h			
lcp.c		lcp.h			
libp.c		libp.h			
pro.c		pro.h			
rsa.c		rsa.h			
sgn.c		sgn.h			
test					
Makefile	candidates.txt	declarations.txt	keys.txt		
key.c	lcc.c	libp.c	pro.c	sgn.c	rsa.c

- libp.c et libp.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 et 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 contiennent 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 donnée Protégé), qui initialise cellProtected, lit le fichier et génère la liste chaînée de donnée 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.
- Dans le dossier de test, le nom du fichier de test est le même que le nom du fichier source.

3 Développement d'outils cryptographiques

Dans cette section, nous allons développer quelques fonctions pour chiffrer les messages de manière asymétrique. La cryptographie asymétrique est un type de cryptographie qui utilise deux clés.

- 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. On a développé deux méthodes.

3.1.1 Implémentations par une méthode naïve

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 $a^b \bmod n$ 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.

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

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.

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 $O(\log_2(m))$).

$ab \bmod n$ est égal à .

- 1 lorsque $m = 0$ (cas de base). - $b \bmod n$ avec $b = am/2 \bmod n$, lorsque m est pair. - $a \cdot b \bmod n$ avec $b = am/2 \bmod n$, lorsque m est impair.

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

Nous allons comparer la vitesse de ces deux méthodes à l'aide de calculs pratiques.

Q1.3

Pour `modpow_naive` qui retourne la valeur $a^b \bmod n$, 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 Implémentations du test de Miller-Rabin

Le test de primalité 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 $\mathbb{Z}/p\mathbb{Z}$ 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 $p = 2^s \cdot d + 1$. Supposons que a soit un entier strictement inférieur à p .

Si nous pouvons trouver un tel a satisfaisant les deux équations ci-dessous:

1. $a^d \bmod p = 1$
2. $a^{2^{r-1} \cdot d} \bmod p = -1$

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 modpow(long a, long m, long n) retourne la valeur $a^b \bmod n$ par des élévations au carré
 -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 Implémentations 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 secrète 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 $n = p \times q$ et $t = (p-1) \times (q-1)$.
2. Générer aléatoirement des entiers s inférieur à t jusqu'à en trouver un tel que $\text{PGCD}(s, t) = 1$.
3. Déterminer u tel que $s \times u \bmod t = 1$.

Le couple $\text{pkey} = (s, n)$ constitue alors la clé publique, tandis que le couple $\text{skey} = (u, n)$ 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 rand_long(dans libp.c) et extended_gcd pour génère la clé publique $\text{pkey} = (s, n)$ et la clé secrète $\text{skey} = (u, n)$, 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 $\text{sKey}=(u,n)$ et de les chiffrer à l'aide d'une clé publique $\text{pKey}=(s,n)$.

-Chiffrement : on chiffre le message m en calculant $c = m^s \bmod n$ (c est la représentation chiffrée de m).

-Déchiffrement : on déchiffre c pour retrouver m en calculant $m = c^u \bmod n$.

```
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 encrypt chiffre la chaîne de caractères avec la clé publique.
- La fonction decrypt 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);
```

- 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 utilise 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 passer d'un chaîne de caractères a clé.

4.1.2 Signature

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:

- 1.
2. 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.
3. L'électeur peut ensuite publier une déclaration sécurisée, composée de sa déclaration mess, de la signature associée, et de sa clé publique. Ceux qui souhaitent vérifier l'authenticité de la déclaration peuvent déchiffrer 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 generate_random_data(int nv, int nc);
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 `candidates.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 chaînée 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);
int           verifyForList(CellProtected **LCP);
```

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

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.

```
int          verifyForList(CellProtected **LCP);
```

-La fonction verifyForList utilise la fonction verify(dans pro.h) vérifie la validité de message contenue dans la déclaration signée et renvoie le nombre de signatures invalides et les supprime.