Projet LU2IN006 - Blockchain appliquée à un processus électoral

Camille VANG (28701368), Wassim MUSSARD (28706762), Sebastian SZEBRAT (28706848)

Description

La tenue d'un processus électoral pose des questions de confiance et de transparence épineuses. Le compte des voix fait appel à des assesseurs, ce qui en fait un travail avec peu de garanties de fiabilité. De plus, un candidat ne peut pas vérifier à posteriori que sa voix a été comptabilisée chez le bon candidat. Le processus peut aussi éprouver un manque d'ergonomie (par exemple, en ne proposant pas de vote par correspondance).

Notre objectif est donc de réflechir sur les protocoles et les structures de données à mettre en place pour permettre d'implémenter efficacement le processus de désignation du vainqueur de l'élection, tout en garantissant l'intégrité, la sécurité et la transparence de l'élection.

Organisation des fichiers

Le code pour chaque partie du projet est divisé dans un répartoire par partie, puis par exercice. Chaque exercice a un fichier .c définissant les fonctions et un header correspondant, et un main pour tester ces fonctions. Un makefile global situé dans la racine du projet permet de tout compiler.

Développement d'outils cryptographiques

Cette partie comporte les fonctions permettant de chiffrer un message de façon asymétrique. Nous avons implémenté le protocole RSA.

Exercice 1 - Résolution du problème de primalité

Fonctions principales

Exponentiation modulaire rapide

- `int is_prime_naive(long p)` étant donné un nombre impair p, renvoie 1 si p est premier et 0 sinon
- `long modpow_naive(long a, long m, long n)` retourne la valeur $a^b \bmod n$

- `int modpow(long a, long m, long n)` retourne la même valeur que `modpow_naive`, mais en réalisant des élévations au carré pour obtenir une complexité logarithmique

Test de Miller-Rabin Fonctions fournies :
* `int witness(long a, long b, long d, long p)` 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)` réalise 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

Génération de nombres premiers

- `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`

Exercice 2 - Implémentation du protocole RSA

Fonctions principales

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

- `void generate_key_values(long p, long q, long *n, long *s, long *u)` permet de générer la clé publique $pkey = (s, n)$ et la clé secrète $skey = (u, n)$ à partir des nombres premiers p et q (suit le protocole RSA)

Chiffrement et déchiffrement des messages

- `long* encrypt (char* chaine, long s, long n)` chiffre la chaîne de caractères `chaine` avec la clé publique $pKey = (s, n)$
- `char* decrypt(long* crypted, int size, long u, long n)` déchiffre un message à l'aide de la clé secrète $skey = (u, n)$

Fonction de test Fonctions fournies :
* `void print_long_vector(long *result, int size)` affiche la représentation chiffrée d'une chaîne de caractères * `int main()` contient des tests de cryptage et décryptage

Réponses aux questions

Q1.1 : La complexité de `is_prime_naive` en fonction de p est $O(p)$.
Q1.2 : Le plus grand nombre premier généré moins de 2 millièmes deseconde est 2 .
Q1.3 : La complexité de `modpow_naive` est de $O(m)$.
Q1.5 : Comme prévue la courbe modpow naive est strictement croissante car elle est en $O(m)$. Alors que la courbe associée à la fonction modpow est presque confondue avec l'axe des abscisses car elle est en $O(\log 2(m))$. On en déduis qu'en comparant leur complexité `modpow` est bien plus performante que `modpow naive` *(cf. Description schématique des algorithmes)* .

Q1.7 : On sait que au moins 3/4 des valeurs entre 2 et p − 1 sont des témoins de Miller pour p . Donc pour k tirages on aura une probabilité $(1/4)^k$ pour avoir une erreur de l'algorithme .

Description schématique des algorithmes

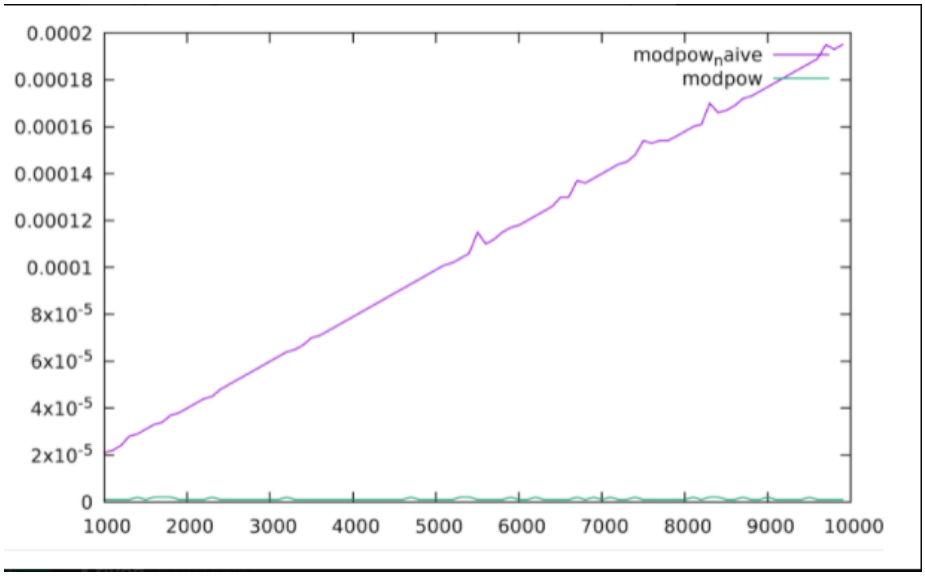

Figure 1: Courbe du temps en fonction de m

Jeux d'essais

Les jeux d'essais ont été concluant. En effet, nous avons obtenu les résultats qui étaient attendus, nous avons également réglé les problèmes de fuites mémoires grâce à Valgrind.

Analyse des performances

Partie 2

Dans cette nouvelle partie, nous allons essayer de modéliser des déclarations de vote .

Exercice 3 - Manipulations de structures sécurisées

Fonctions principales

Manipulation de cles

- `void init_key(Key * key , long val , long n )`, initialise une clé déjà allouée .
- `void init_pair_keys(Key * pKey , Key *sKey , long low_size , long up_size )`, initialise une clé publique et une clé secrète .
- `char * key_to_str(Key * key )`, convertit la clé passée en paramètre en `str`. Afin d'optimiser l'espace mémoire nous avons fait un `realloc`.
- `char * str_to_key(char * str )`, convertit le `str` passé en paramètre en clé.

Signature

- `Signature * init_signature(long *content , int size )` ,permet d'allouer et de remplir une `Signature`.
- `Signature * sign(char *mess, Key *sKey )` , retourne une `Signature` à partir de `mess` et de la clé secrète . Pour cela nous utilisons la fonction `long* encrypt(char * chaine, long s , long n)` vu dans l'exercice 2 de la partie 2 .

Fonctions fournies :
* `char* signature_to_str(Signature *sgn)`, convertit une `Signature` en `str` .

- `Signature* str_to_signature(char *str)`, convertit un `str` en `Signature` .

Déclarations signées

- `Protected * init_protected(Key * pKey , char * mess , Signature * sgn)`, alloue et initialise la structure `Protected` .
- `int verify(Protected *pr)` , vérifie s'il y a correspondance entre la `Signature` contenue dans `pr` et le message . Nous utilisons la fonction `char *decrypt(long * crypted, int size , long u , long n)` ce qui va nous permettre de comparer la chaîne en sortie (mess_decrypt) avec `pr->mess`. Si `strcmp(pr->mess,mess_decrypt) == 0` alors il y a correspondance .
- `char* protected_to_str`, convertit une structure `Protected` en `str` . Pour cela on fait appel aux fonctions `char* key_to_str(Key* key)` et `Key* str_to_key(char* str)`. Puis on alloue l'espace mémoire nécessaire pour notre `char*` et enfin on libère la mémoire allouée par nos variables locales `char* pKey` et `char* sgn` .
- `Protected * str_to_protected` , convertit un `str` en une structure `Protected` . Pour cette fonction , notre stratégie était de déclarer une variable `char buffer[256]` puis de parcourir `str` et de stocker dans `buffer` les caractères de `str` jusqu'à rencontrer les différents délimiteurs : `' '` ou `\0` représentant respectivement le délimiteur de `pKey`, le `message` et la `signature`. On garde en mémoire ces informations , enfin on déclare une `Signature *` et un `Protected *` avant de les initialiser .

Jeux d'essais

Exercice 4 – Création de données pour simuler le processus de vote

Fonction principale

- `void generate_random_data(int nv, int nc)` , qui va donc nous permettre de modéliser tout le déroulement d'un vote . Pour cela , on suit tout simplement les différentes étapes données .

Jeux d'essais La fonction `void generate_random_data(int nv, int nc)` fonctionne . En effet elle retourne le resultat attendu sans erreurs de gestion de mémoire . *(cf. candidats.txt , keys.txt , déclarations.txt)* .

Partie 3

Description

Cette partie va nous permettre de modéliser une Base de déclarations centralisée. Nous aurons besoin d'utiliser la fonction `void generate_random_data(int nv, int nc)` vu dans l'exercice 4.

Exercice 5 – Lecture et stockage des données dans des listes chaînees

Fonctions principales

Liste chaînée de clés

- `CellKey* create_cell_key(Key* key)`, alloue et initialise une cellule de liste chaînée.
- `CellKey* insert_cell_key(CellKey* LCK, Key* data)`, permet d'ajouter la clé `data` en tête de la liste `LCK`.
- `CellKey* read_public_keys(char* nomFic)`, retourne le contenu de `nomFic` sous la forme d'une liste chaînée.
- `void print_list_keys(CellKey* LCK)`, affiche `LCK->data` tant que `LCK != NULL`.
- `void delete_cell_key(CellKey* c)`, libère une cellule.
- `void delete_list_keys(CellKey* LCK)`, libère toute la liste chaînée.

Liste chaînée de déclarations signées

- `CellProtected* create_cell_protected(Protected* pr)`, alloue et initialise un `CellProtected`.
- `CellProtected* insert_cell_protected(CellProtected* LCP, Protected* pr)`, ajoute `pr` en tête de `LCP`.
- `CellProtected* read_protected()`, retourne un `CellProtected*` avec toute les déclarations du fichier `'declarations.txt'`.
- `void print_list_protected(CellProtected* LCP)`, affiche `LCP->data` tant que `LCP != NULL`.
- `void delete_cell_pr(CellProtected* cp)`, libère un élément de type `CellProtected*`.
- `void delete_list_pr(CellProtected* LCP)`, libère toute la liste chaînée.

Jeux d'essais

Partie 4

Dans cette nouvelle partie, nous allons faire des fonctions qui pourront déterminer le gagnant de l'élection. Pour cela nous allons nous utiliser la notion de table de hachage.

Exercice 6 - Détermination du gagnant de l'élection

Fonctions principales

- `void delete_fraud(CellProtected* cp)`, supprime de `cp` toutes les signatures non valides.