CHETOUANI Younes (28632942)
SALEHUDDIN Ishrat (28610536)

Projet : Blockchain appliquée à un processus électoral

Ce projet se base sur l'organisation d'une élection inspirée du modèle français (à deux tours et au scrutin uninominal), ou plus précisément son déroulement.

L'objectif du projet est d'essayer de comprendre les protocoles tout en approfondissant notre capacité à manipuler les structures de données, dans le but de désigner le vainqueur de l'élection, en toute intégrité et fiabilité de l'élection.

Notre dossier sera composé principalement, pour chaque groupement d'exercices :

- d'un fichier *.h* qui va contenir les prototypes de chaque fonction ainsi que la définition des structures ;
- du fichier *.c* qui va contenir le corps des fonctions ;
- d'un fichier *main.c* qui va contenir le main nous permettant d'effectuer les tests pour vérifier le bon fonctionnement du code.

Le projet se divise ainsi en 5 parties que nous détaillerons ici-bas :

Développement d'outils cryptographiques

Dans un premier temps, nous allons créer des fonctions permettant de chiffrer un message, grâce au **protocole RSA**, un algorithme de cryptographie asymétrique.

Exercice 1 – Résolution du problème de primalité

Nous allons nous concentrer sur la génération des nombres premiers.

IMPLEMENTATION PAR UNE METHODE NAIVE

`is_prime_naive` (**Q 1.1**), étant donné un entier impair p, renvoie 1 si p est premier et 0 sinon. Concernant la complexité, dans le meilleur cas, p est divisible par 1, donc on a une complexité de 1. Dans le pire des cas, on effectue p tours de boucle. Donc on a une complexité de O(p).

(**Q 1.2**) Après plusieurs lancements, le plus grand nombre premier qu'on arrive à tester en moins de 0.002s avec cette fonction est 203 431, donc d'ordre de grandeur à 200 000 à peu près.

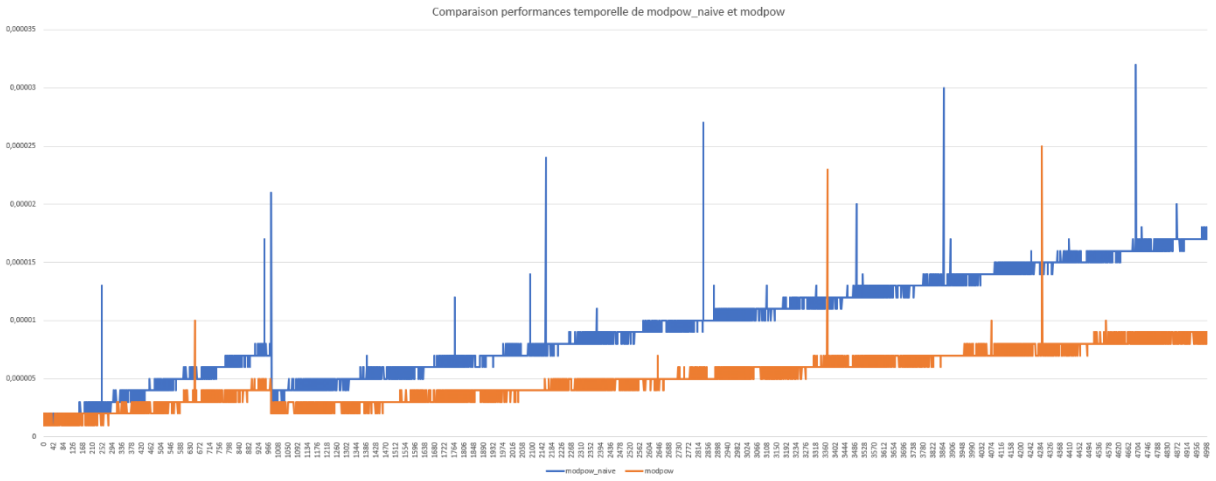
EXPONENTIATION MODULAIRE RAPIDE

Nous avons défini deux fonctions dans cette partie, à savoir :

- `modpow_naive` (**Q 1.3**) retournant la valeur $a^b \ mod \ n$ par la méthode naïve, avec une complexité de O(m).
- `modpow` (**Q 1.4**) effectuant la même chose que la fonction précédente, mais par une succession d'élévations au carré, avec une complexité de $O(\log_2(m))$.

CHETOUANI Younes (28632942)
SALEHUDDIN Ishrat (28610536)

Q 1.5 :



Dans le main, nous avons analysé les performances de `modpow` et de `modpow_naive`.

On observe que `modpow` (en orange) est plus rapide donc plus efficace que `modpow_naive` (en bleu) et que la différence des performances se creuse au fur et à mesure.

TEST DE MILLER-RABIN
(**Q 1.7**) On s'intéresse maintenant à la fiabilité du test de Miller-Rabin.
La borne supérieure sur la probabilité d'erreur de l'algorithme est de $\frac{1}{4}^k$, sachant qu'au moins $\frac{3}{4}$ des valeurs entre 2 et p−1 sont des témoins de Miller pour p.

On en déduit que le test de Miller-Rabin est préférable à la méthode naïve, vu qu'ici la probabilité d'erreur devient rapidement très faible quand k augmente.

GENERATION DE NOMBRES PREMIERS
`random_prime_number` (**Q 1.8**) retourne un nombre premier de taille comprise entre *low_size* et *up_size*, avec k le nombre de tests de Miller à effectuer. Plus k est grand (dans le raisonnable), plus c'est précis et fiable.

Exercice 2 – Implémentation du protocole RSA

GENERATION D'UNE PAIRE (CLE PUBLIQUE, CLE SECRETE)
On souhaite, avec le protocole RSA, générer deux clés :

- *pKey = (s, n)*, une clé publique permettant de chiffrer des messages
- *sKey = (u, n),* une clé secrète pour pouvoir les déchiffrer.

`generate_key_values` (**Q 2.1**) génère *pKey* et *sKey*, à partir de *p* et *q* choisis par nous-mêmes, en suivant le protocole RSA.

CHIFFREMENT ET DECHIFFREMENT DE MESSAGES
Le chiffrement et le déchiffrement d'un message se feront selon le schéma suivant, à l'aide des fonctions `encrypt` (**Q 2.2**) et `decrypt` (**Q 2.3**).

CHETOUANI Younes (28632942)
SALEHUDDIN Ishrat (28610536)

FONCTION DE TESTS

Dans cette batterie de tests fournie par le sujet, nous avons pu vérifier la compilation ainsi que le bon fonctionnement du code.

Ainsi, on a pu tester `random_prime_number`, `generate_ket_values`, et si le chiffrement et le déchiffrement d'un message se faisait correctement et sans fuite mémoire.

Déclarations sécurisées

Exercice 3 – Manipulations de structures sécurisées

MANIPULATION DE CLES

Nous avons défini une structure `Key` (**Q 3.1**) contenant deux *long* représentant une clé (publique ou secrète), ainsi que :

- `init_key` (**Q 3.2**), initialisant une clé déjà allouée ;
- `init_pair_keys` (**Q 3.3**) ;
- `key_to_str / str_to_key` (**Q 3.4**) permettent de passer d'une variable de type *Key* à une chaîne de caractères de type "(x,y)" et inversement.

SIGNATURE

Q 3.5 : Nous avons défini une structure `Signature` qui contient un tableau de long et sa taille.

De plus, nous avons créé `init_signature` (**Q 3.6**) qui initialise une signature ainsi que `sign` (**Q 3.7**) va nous permettre générer la signature associée à sa déclaration de vote par chiffrement avec la clé secrète de l'émetteur.

DECLARATIONS SIGNEES

Pour voter, une personne lambda se doit de faire une déclaration de vote signée pour garantir l'authenticité de celle-ci.

Nous avons ainsi dû définir une structure `Protected` (**Q 3.9**) qui contient le *pKey* de l'émetteur, sa déclaration de vote (*mess*), et la signature associée.

- `init_protected` (**Q 3.10**) allouant et initialise la structure ci-dessus.
- `verify` (**Q 3.11**) vérifiera que la signature contenue dans le *Protected* en paramètre correspond bien au message et à la personne contenus dans celui-ci.
- `protected_to_str / str_to_protected` (**Q 3.12**)

CHETOUANI Younes (28632942)
SALEHUDDIN Ishrat (28610536)

FONCTION DE TESTS

Dans cette batterie de tests fournie par le sujet, nous avons pu vérifier la compilation ainsi que le bon fonctionnement du code.

Ainsi, nous avons-nous avons initialisé une paire de clefs, `sign`, `init_protected`, `verify`, si le passage d'un élément de type *Key, Signature* ou *Protected* à une chaîne de caractères et inversement se faisait correctement.

Et ce tout en veillant à ce qu'il n'y ait pas de fuite mémoire, en rajoutant les `free()` manquants dans le sujet.

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

Q 4.1 : `generate_random_data` va simuler le processus de vote à l'aide de trois fichiers : un fichier contenant les clés de *nv* citoyens (*keys.txt*), un fichier indiquant les *nc* candidats (*candidates.txt*) et un fichier contenant des déclarations signées (*declarations.txt*), tous issus d'une génération aléatoire des données citées précédemment.

Concernant le jeu de tests, nous avons testé dans le main `generate_random_data(20,5)`, et les fichiers nécessaires se sont créées correctement avec les données qu'il fallait.

Base de déclarations centralisée

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

LISTE CHAINEE DE CLES

Une fois les citoyens ayant tous voté, on cherche à centraliser toutes les données récoltées, pour ensuite désigner le vainqueur. Les déclarations de vote sont ainsi enregistrées au fur et à mesure dans declarations.txt (vu précédemment), et une fois le scrutin clos, ces données sont chargées dans une liste chaînée.
Pour cette liste chainée, nous avons d'abord défini une structure `CellKey` qui contient une clé et une cellule de liste chainée pointant vers la suivante. Nous avons ensuite créé les fonctions suivantes :

- `create_cell_key` (**Q 5.1**) va allouer et initialiser une cellule de liste chaînée de clés
- `delete_cell_key` (**Q 5.5**) va la supprimer,
- `ajout_en_tete` (**Q 5.2**) ajoutera une clé en tête de liste,
- `delete_list_keys` (**Q 5.5**) supprimera une liste chaînée de clés.

En parallèle, pour vérifier la validité des données et s'en servir, nous avons également besoin des *pKey* des citoyens et des candidats (dans *keys.txt* et *candidates.txt*). Pour se faire, nous avons créé `read_public_keys` (**Q 5.3**) qui prend en entrée les deux fichiers pour lire les clés publiques, et `print_list_keys` (**Q 5.4**) qui les affiche.

CHETOUANI Younes (28632942)
SALEHUDDIN Ishrat (28610536)

LISTE CHAINEE DE DECLARATIONS SIGNEES
Même principe que précédemment mais avec des *Protected* au lieu de clés.

[tests dans le main pour vérifier le bon fonctionnement du code et les fuites mémoire]

Exercice 6 – Détermination du gagnant de l'élection

Dans cet exercice, nous avons créé la fonction `compute_winner` (**Q 6.7**) qui calcule le vainqueur de l'élection :

- en créant deux tables de hachage *Hc* (table des candidats) et *Hv* (table des votants) à l'aide de `create_hashtable`(**Q 6.5**) ;
- en supprimant toutes les déclarations dont la signature n'est pas valide à l'aide de `delete_no_valid` (**Q 6.1**), pour éviter les tentatives de fraude ;
- en vérifiant ensuite si la personne a bien le droit de voter, si elle ne l'a pas déjà fait, et si la personne sur qui porte le vote est bien candidat de l'élection
    - avec `find_position` (**Q 6.4**) qui retourne la position d'un élément *Key* dans la table, ou la position dans laquelle il aurait dû être
    - et `hash_function` (**Q 6.3**) qui retourne la position d'un *Key* dans la table de hachage ;
- en mettant à jour à chaque fois (vote comptabilisé/droit de vote du votant déduit) ;
- puis en parcourant la table des candidats pour récupérer le gagnant.

[tests dans le main pour vérifier le bon fonctionnement du code et les fuites mémoire]

CHETOUANI Younes (28632942)
SALEHUDDIN Ishrat (28610536)

Blocs et persistance des données

Exercice 7 – Structure d'un bloc et persistance

Nous avons, dans un premier temps, défini une structure `Block`, afin de nous intéresser à la gestion des blocs.

Cette structure contiendra le *pKey* de son créateur, une liste de déclarations de vote, la valeur hachée du bloc, et du précédent, ainsi qu'une preuve de travail.

LECTURE ET ECRITURE DE BLOCS

`block_to_script` (**Q 7.1**) et `script_to_block` (**Q 7.2**) vont nous permettre de passer d'un block à sa transcription dans un fichier et inversement.

CREATION DE BLOCS VALIDES

`block_to_str` (**Q 7.3**) va générer une chaîne de caractères représentant un bloc, pour ensuite retourner sa valeur hachée obtenue par l'algorithme SHA256 avec `valhach` (**Q 7.5**).

Nous allons aussi pouvoir supprimer un bloc avec `delete_block` (**Q 7.9**), vérifier si un bloc est valide avec `verify_block` (**Q 7.7**), `compute_proof_of_work` (**Q 7.6**) pour rendre un bloc valide.