IN006 - Structures de Données

Compte Rendu

Projet - Blockchain appliquée à un processus éléctoral

COCAULT Éléa - CARBONEAU Danaël

Contents

1 Outils cryptographiques

Dans cette partie, nous allons développer des fonctions permettant de chiffrer un message de façon asymétrique. La cryptographie asymétrique est une cryptographie qui fait intervenir deux clés :

- Une clé publique que l'on transmet à l'envoyeur et qui lui permet de chiffrer son message.
- Une clé secrète (ou privée) qui permet de déchiffrer les messages à la réception.

L'algorithme de cryptographie asymétrique que nous allons implémenter est le protocole RSA. Ce protocole s'appuie sur de (très grands) nombres premiers pour la génération des clés publiques et secrètes. Nous allons donc commencer par traiter le problème de la génération de nombres premiers et de l'implémentation de puissances modulaires (calculer pour de très grandes valeurs $a^m \ mod \ n$), plus nous les utiliserons afin d'implémenter les fonctions de génération de clé, de chiffrage et de déchiffrage de messages.

Le code est organisé entre quatre fichiers .c, deux fichiers .h, un makefile et un fichier de commandes pour gnuplot:

- `tests_primalité.c` contient le code des fonctions de l'exercice 1 (`tests_primalité.h` les répertorie)
- `main.c` contient un menu nous permettant de tester les fonctions du premier exercice.
- `gestion_cryptage.c` contient le code des fonctions de l'ecercice 2 (`gestion_cryptage.h` les répertorie)
- `main.h` est le code fourni par l'énoncé, servant à tester nos fonctions de chiffrage avec le protocole RSA
- `commande.txt` sert à exécuter de manière automatisée dans notre programme (à l'aide de la fonction `system(char* commande)`) les commandes gnuplot permettant de tracer les courbes de la question 5 de l'exercice 1.

Exercice 1.1 Résolution du problème de primalité

Implémentation par une méthode naïve

Q.1.1.1 Complexité en fonction de p

On fait un tour de boucle en itérant sur p, d'où une complexité en $\Theta(p)$.

Q.1.1.2 Temps de calcul

En exécutant le main avec la première option du menu, on exécute `is_prime_naive` jusqu'à trouver un temps de calcul supérieur à 0.002 secondes, le programme nous affiche alors 110039, qui n'est pas une très grande valeur au regard des long ints que nous voulons manipuler pour chiffrer nos données.

Q.1.1.3 Exponation modulaire

On va chercher à coder une fonction modpow qui calcule $a^m mod\ n$ de façon efficace, puisqu'elle nous servira dans le chiffrage RSA.

Version naïve de modpow

La complexité de la version naïve de modpow est en $\Theta(m)$ puisqu'on itère un nombre fini d'opérations m fois. Lorsque m est très grand, ce qui est le cas ici, puisque nous travaillons sur de grandes valeurs, il nous faut donc une fonction plus optimale.

Version récursive

Calculons la complexité de notre fonction en fonction de m tel que $m = 2^{k1}$

$$
\begin{aligned}
u_m &= 1 + u_{\frac{m}{2}} && (1.1)\\
&= 1 + (u_{\frac{m}{4}} + 1) && (1.2)\\
u_m &= u_{\frac{m}{2^k}} + k && (1.3)\\
&= u_{\frac{m}{m}} + log_2(m) && (1.4)\\
&= u_1 + log_2(m) \qquad \in \Theta(log(m)) && (1.5)
\end{aligned}
$$

Donc la complexité de notre fonction est en $\Theta(log(m))$, ce qui est bien plus efficace et rapide, surtout pour les grandes valeurs que nous souhaitons manipuler, pour calculer la puissance modulaire.

Notre fonction étant amenée à calculer de grandes valeurs, nous avons changé le prototype fourni dans l'énoncé : `long modpow(long a, long m, long n)`*, et non plus* `int modpow(long a, long m, long n)`*.*

Q.1.1.4 Comparaison des deux fonctions

Pour de grandes valeurs de m, il devient évident et visible que l'implémentation naïve est énormément couteuse temporairement parlant, contrairement à la fonction abordée à la question 4.

[1]Toute valeur de m est comprise entre deux puissances de deux, donc on peut approximer la suite en ne considérant que ce cas

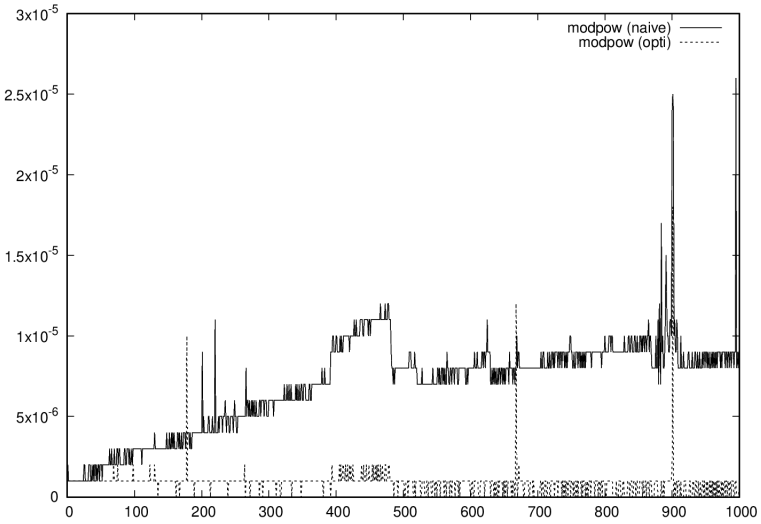
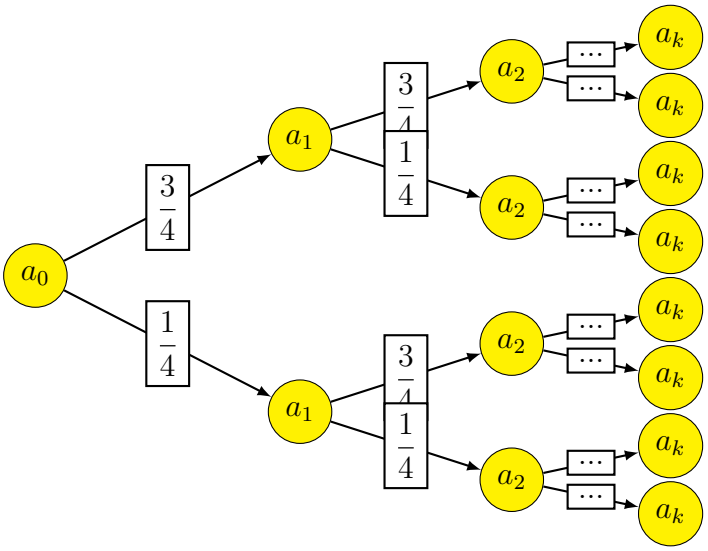


Figure 1.1: Évolution du temps de calcul entre les deux implémentations de modpow (version naïve en traits pleins, version récursive en pointillés). [a]

[a] fichier .ps dans le dossier COURBES)

Q.1.1.5 Probabilité d'erreur du test de Miller-Rabin

La probabilité à chaque fois qu'on tire un pontentiel témoin a est d'au plus $\frac{1}{4}$ (car il y a au moins les $\frac{3}{4}$ des nombres entre 2 et p-1 qui sont des témoins de p), en répétant k fois l'opération, d'après l'arbre de probabilités (on prend la probabilité de la seule branche représentant l'événement "aucun nombre a tiré n'a été témoin", dont tous les arcs sont étiquettés par $\frac{1}{4}$), la probabilité qu'on ne tire en k expérience aucun témoin à un nombre p qui n'est pas premier, c'est à dire la probabilité d'erreur de notre algorithme, est alors d'au plus $(\frac{1}{4})^k$.



Q.1.1.6 Fonction renvoyant un entier premier aléatoire

On prend des valeurs de `long` comprises entre deux tailles : low_size et up_size (exclu[2]) pour générer des nombres entiers jusqu'à en trouver un qui réussisse le test de Miller.

En répétant un grand nombre de fois le test, on minimise nos chances que le nombre retourné ne soit pas premier (voir Q.5). Si on ne trouve pas de nombre premier au bout de 1000 essais, un message d'erreur s'affiche et on retourne la valeur -1.

Exercice 1.2 Implémentation du protocole RSA

En utilisant `modpow_naive`, les temps de calculs pour notre main deviennent beaucoup trop longs, on utilise donc la version optimisée de la question 4 de l'exercice 1.

En suivant le protocole RSA pour coder nos fonctions d'encryptage et de décryptage, on arrive à exécuter le main fourni par l'énoncé, et à chiffrer puis déchiffrer le message `"Hello"`.

Il faut cependant faire attention à la taille des nombres demandés : de trop grandes valeurs, dépassant la taille de stockage des long, ne donneront pas les bons résultats au moment de `modpow`, rendant impossible le chiffrage et le déchiffrage.

[2] si on exclue pas cette taille, on va générer sur une plage de nombre trop importantes, et grande, ce qui génère des erreurs de dépassement mémoire

2 Déclarations sécurisées

Il s'agira dans cette partie d'appliquer les outils cryptographiques du protocole RSA développés en partie 1 au problème concret du vote électoral.

Tout citoyen participant au vote, simple votant ou candidat votant, est ainsi associé à une carte électorale qui correspond en RSA à l'ensemble d'une clé publique et une clé privée.

Chaque citoyen doit alors émettre une certaine déclaration de vote, qui consiste techniquement en l'association de la clé publique du citoyen à la représentation hexadécimale de la clé publique du candidat choisi.

L'enjeu est alors de pouvoir authentifier la déclaration de vote du citoyen, c'est à dire être en mesure de vérifier que le candidat apparaissant dans sa déclaration a bien été choisi par le citoyen et ne relève pas d'une fraude ou d'un problème technique extérieur. C'est tout l'enjeu de l'utilisation du protocole RSA que de pouvoir alors associer à chaque déclaration de vote une signature permettant cette authentification : la valeur de la clé publique du candidat choisi exprimée en hexadécimal est cryptée à l'aide de la clé privée du citoyen, et cette signature est associée à sa déclaration initiale. Ainsi, il devient possible en utilisant la clé publique du citoyen votant de décrypter cette signature pour vérifier que le candidat déclaré correspond bien au candidat choisi par le citoyen lui-même, doté de sa clé privée.

La sécurité apportée par le protocole RSA fournit donc un moyen d'implémenter une situation de vote juste : chaque vote pour un candidat peut être associé à un unique citoyen qui l'a réellement choisi.

Le code nous permettant d'appliquer le protocole RSA à la situation de vote est organisé en quatre fichiers `.c`, trois headers et un `makefile`.

- `gestion_clef.h` contient la structure de clé utilisée en RSA, et les prototypes de `gestion_clef.c`. Ce sont toutes les fonctions permettant d'initialiser nos clés, et les exprimer comme nombres hexadécimaux ou chaînes de caractères.

- `signature.h` contient la structure `Signature` et celle `Protected` qui regroupe la clé publique du votant, sa déclaration de candidat choisi et la signature permettant d'authentifier son choix. Les prototypes sont ceux de `signature.c` qui permet d'initialiser ces structures et les manipuler en chaînes de caractères. On retrouve aussi ici les fonctions permettant de générer une signature et de vérifier la concordance entre celle-ci et la déclaration de vote apparente.

- `gestion_votants_candidats.h` contient la structure représentant le votant comme ensemble de deux clés RSA et les protypes de `gestion_votants_candidats.c` qui permettent une première simulation de vote complète.

Point sur la compilation

Dans cette seconde partie nous sommes donc amenés à réutiliser du code de la partie précédente. Pour s'y repérer, il nous semble cependant nécessaire de séparer en deux fichiers distincts ces parties. Cela implique de compiler ensemble des fichiers issus de dossiers différents. Pour ce faire nous avons retenu la solution suivante :

- Chaque header associé à un fichier `.c` utilisant des fonctions d'un `.c` se trouvant dans un autre dossier doit comprendre un `#include "../Partie_1/nom_fichier_a_inclure"`.
- Dans le `makefile`, à chaque fois qu'un fichier exterieur au dossier apparait il doit aussi être précédé de `../ Partie_1`.

Exercice 2.1 Manipulations de structures sécurisées

Manipulation de clés

Voir `gestion_clef.c`.

Q.2.1.1

Simple définition de structure que nous plaçons dans le header `gestion_clef.h`.

Q.2.1.2

Nous vérifions que la structure est bien déjà allouée en mémoire. Si c'est le cas, l'initialisation consiste simplement à donner pour valeur aux champs de la structure les données passées en paramètres.

Q.2.1.3

L'initialisation demandée est ici plus complexe en ce qu'elle demande de générer les valeurs des entiers qui composeront ensuite les clés. Ces entiers sont générés en RSA selon un protocole que nous avions écrit en partie 1. On reprend donc ici la fonction `random_prime_number` de `test_primalite.c` pour générer deux grands nombres premiers de taille donnée en paramètre et la fonction `generate_key_values` de `gestion_cryptage.c` pour de là se donner par exponentiation modulaire les valeurs des clés publiques et privées. Avec ces valeurs, nous pouvons réutiliser `init_key` et disposer enfin de deux clés initialisées à partir de leur seule taille en mémoire.

Q.2.1.4

Pour écrire notre clé en chaine de caractères, on se donne un `buffer` statique de grande taille dans lequel on écrit en hexadécimal les valeurs de notre clé suivi du caractère de séparaison `'\0'`. On "rogne" ensuite selon la longueur réelle de la chaine nécessaire à l'aide de `strdup` qui renvoie une chaine allouée dynamiquement en s'arrêtant au premier caractère d'arrêt rencontré. Pour passer de notre chaine de caractère à des valeurs numériques, on fait attention à ce que le format fourni soit bien celui attendu par un simple test, puis on récupère les valeurs avant de les utiliser pour initialiser une clé qu'on alloue préalablement.

Signature

Note sur le code : des fonctions ont été ajoutées pour gérer la mémoire. Dans cette optique, nous avons également fait le choix d'ajouter une fonction `deep_copy` *des signatures, et de toujours dupliquer les structures lorsqu'elles sont référencées dans d'autres (toujours dans un soucis de gestion de la mémoire facilité dans nos mains).*

Voir `signature.c`.

Q.2.1.5

Simple définition de structure que nous plaçons dans le header `signature.h`.

Q.2.1.6

Attention, l'initialisation comprend ici l'allocation mémoire.

Q.2.1.7

Créer une signature revient à coder selon la clé privée du votant un certain message (sa déclaration de vote, c'est à dire la clé publique du candidat choisi). On utilise donc la fonction `encrypt` définie en partie 1 pour coder le message donné en argument. On connaît sa taille grâce à la bibliothèque `string` qu'on inclue. On peut alors initialises une signature grâce à la fonction précédente `init_signature`.

Q.2.1.8

Recopie du code donné.

Déclarations signées

Voir `signature.c`.

Q.2.1.9

On définit de nouveau simplement dans le header associé la structure `Protected` à partir des structures déjà codées.

Q.2.1.10

L'initialisation avec allocation de mémoire ne pose pas de problème. On note juste l'emploi de `strdup` pour allouer dynamiquement la place reservée à la déclaration de vote (le message). On se garantit ainsi de garder cette donnée en mémoire, au cas où la chaine de caractère donnée en paramètre serait dépilée par la fonction appelante. Mais il faut alors penser à libérer cet espace supplémentaire.

Q.2.1.11

La vérification de la signature consiste à vérifier l'égalité de deux chaines de caratères : celle correspondant au champ `mess` de la structure `Protected`, la déclaration de vote apparente du votant, et celle correspondant à la signature décryptée du votant. On se sert à nouveau d'une fonction de la partie 1 : `decrypt`. Attention, nous utilisons `strcmp` pour comparer deux chaines de caractères : cette fonction renvoie 0, c'est à dire "faux" si les chaines sont identiques. On choisi d'utiliser la négation pour renvoyer plutôt 1, "vrai", dans ce cas.

Q.2.1.12

Dans `protected_to_str`, les différentes fonctions de conversion en chaine de caractères développées jusqu'ici sont utilisées. L'allocation en mémoire est permise par l'utilisation de `strlen` de la bibliothèque `string` pour connaître la taille de la chaine résultat. Dans `str_to_protected`, on fait simplement attention au format avant de récupérer nos données et les utiliser pour initialiser la structure.

Fonction de tests

Voir `test.c`.

On recopie tel quel le code proposé. On vérifie d'une part que l'on génère bien des clés aux valeurs affichées en hexadécimales aléatoires, et que leur traduction en chaine de caratère se fait sans encombre ni modifications. On se donne ensuite une situation de vote où un citoyen vote pour un candidat : on vérifie que la génération de la signature associée se fait bien, et que sa traduction aussi. Enfin, on vérifie que l'on peut construire une structure `Protected`, la traduire, et nous servir pour authentifier ce vote : la signature doit être valide et est annoncée l'être.

Exercice 2.2 Création de données pour simuler le processus de vote

Voir `gestion_votants_candidats.c`

Q.2.2.1

L'énoncé semblait demander une unique fonction remplissant plusieurs fonctionnalités. Pour plus de clarté nous avons décidé de découper ces tâches en plusieurs sous-fonctions, dont la réunion nous permet d'écrire la `generate_random_data` demandée.

- Pour permettre de naviguer entre ces fonctions sans avoir à procéder à de longues lectures de fichiers à chaque fois, nous nous donnons tout d'abord une structure `Votant` dont on connait les deux clés et s'il est ou non candidat. On manipulera alors des tableaux de votants de fonction en fonction et non seulement des fichiers.

- `generation_fichier_votants` génère un nombre donné de votants. On stocke ceux ci dans un tableau tout en écrivant dans un fichier `keys.txt` la valeur de leurs deux clés transformées en chaines de caractères.

- `generation_fichier_candidats` génère un nombre donné de candidats choisi parmis les votants créés précedemment. On manipule ici le tableau de `Votant` retourné par la fonction précédente sans avoir besoin de reparcourir le fichier de stockage. De même, pour se faciliter les choses par la suite on créé ici un tableau des clés publiques des candidats générés tout en la stockant plus durablement dans `candidates.txt`. Le champ est_candidat de notre structure `Votant` est alors très pratique pour s'assurer que tous les candidats soient bien distincts, c'est à dire qu'un votant déjà candidat ne peut être choisi une seconde fois.

- `emission_vote` choisi pour un `Votant` donné le candidat qui aura son vote. On manipule le tableau de clé de la fonction précédente sans avoir à réouvrir le fichier de stockage. L'émission de vote correspond à une structure `Protected` regroupant la clé publique du votant, celle du candidat choisi, et la signature associée.

- `generate_random_data` permet de simuler aléatoirement une situation de vote où `nc` électeurs doivent voter pour `nc` candidats désignés parmi eux. On génère les votants et les candidats, puis on créée les `nv` émissions de vote qu'on stocke dans le fichier `declarations.txt`. On fait bien attention à libérer toute la mémoire utilisée : les tableaux, les chaines de caractères, les structures et certains de leurs champs intermédiaires utilisés sont maintenant inutiles et à libérer.

On rajoute à `test.c` un appel à textttgenerate_random_data et l'étude des différents fichiers générés nous permet de conclure sur son bon fonctionnement.

3 Base de déclarations centralisée

Dans cette partie, nous allons mettre en place les outils permettant un système de vote centralisé. Il s'agit de construire un système permettant de traiter les données que nous avons pu récolter grâce aux outils des parties précédentes, dans le but de désigner in fine le vainqueur de l'élection.

Les déclarations de vote sont enregistrées dans le fichier appelé `declarations.txt` généré en partie 2. Il faut maintenant les charger dans une liste chaînée qui permettra de les manipuler. On récupère également l'ensemble des clés publiques des citoyen.ne.s et des candidat.e.s, stockées respectivement dans `keys.txt` et `candidates.txt`.

Pour ce faire, nos disposons de deux fichiers `.c` et `.h` associés qui nous permettent de créer et manipuler des listes chaînées de structures de données crées et stockées dans des fichiers dans la partie précédente.

`chaine_cle.c` nous permet de manipuler des `Key`, son header inclut donc `gestion_cle.h`. `chaine_protected.c` quand à lui gère des listes de `Protected` à l'aide de `signature.h`. C'est également dans ce dernier fichier que se trouve la première fonction d'intérêt particulier quant à notre problème de vote, celle qui permet la suppression des déclarations de votes falsifiées.

Enfin, `test.c` permet quelques tests rudimentaires de ces nouvelles fonctions.

Exercice 3.1 Lecture et stockage des données dans des liste chaînées

Cet exercice consiste en l'implémentation des fonctions de manipulations usuelles de listes, ainsi que la lecture de documents formatés, appliquées à nos structures de données : une liste chaînée de cellules contenant un pointeur vers une clé et une liste chaînée de déclarations signées (`Protected`). Ces fonctions étant classiques, nous pensons que notre code suffit à comprendre ce que nous avons fait.

Exercice 3.2 Détermination du gagnant de l'élection

Afin de déterminer le gagnant de l'élection, on va d'abord éliminer toutes les déclarations de vote fausses (cas de tentative de fraude). Pour cela on parcourt simplement la liste en éliminant les déclarations fraduleuses à l'aide de la fonction `verify` de la partie 2.

Puis on va se servir de deux tables de hachage, dont chaque cellule contient un pointeur vers une clé et une valeur.

Dans la table de hachage contenant les clés des candidats, la valeur correspond au nombre de voix, dans celle contenant les clés des votants potentiels, la valeur est un booléen indiquant si le votant a déjà émis son vote.

Nous les remplissons par *probing linéaire*, c'est à dire qu'en cas de collision avec notre fonction de hachage, nous nous déplaçons dans la table de hachage jusqu'à trouver une case libre.

Attention : en faisant tourner notre code, nous avons remarqué qu'en générant un grand nombre de votes avec une trop petite plage de valeurs pour les clés, on générait plusieurs fois le même jeu, ce qui résultait, dans le tableau, à l'écrasement de ces doublons, il faut donc bien faire attention à prendre des plages de valeurs adaptées au nombre de votants.

Pour obtenir le résultat de l'élection, on parcourt la chaîne de `protected` contenant les déclarations signées en retirant les fraudes puis en vérifiant que *a)* le candidat choisi est bien candidat *b)* le votant ayant émis le vote n'a pas déjà voté, puis on rajoute une voix au compteur dans la table de hachage des candidats. Il suffit alors de trouver le candidat dans la table de hachage ayant le plus grand nombre de voix, qui sera déclaré comme étant le gagnant de l'élection.

4 Blocs et persistance des données

Dans la partie précédente, l'élection est gérée de manière centralisée par une autorité régulatrice. Nous souhaitons maintenant procéder de manière décentralisée, pour cela, nous utiliserons le principe d'une blockchain.

Le code nous permettant de gérer les blocks servant à la réception des déclarations de vote est organisé en trois fichiers .c, un fichier header et un makefile.

- `block.h` contient contient la déclaration de la structure d'un block ainsi que les fonctions nécessaires à leur manipulation.
- `test_SHA256.c` contient des sommaires manipulations de la fonction de hachage cryptographique utilisée.
- `test.c` contient nos tests sur la structure de block.
- `commande.txt` nous permet de tracer la courbe correspondant à la question 7.8 de l'énoncé.
- `ecriture_bloc.txt` est le fichier dans lequel est écrit le block sur lequel nous travaillons dans nos tests
- `comparaison_d.txt` est le fichier où sont notés les temps de minage selon le nombre de 0 en début de valeur de hachage demandé.

Exercice 4.1 Structure d'un block et persistance

Lecture et écriture d'un block

Note : Pour mieux visualiser les valeurs de hachage lorsque nous voulons les afficher, ainsi que pour alléger nos fonctions d'écriture et de lecture, nous avons fait deux fonctions : `char * hash_to_str (unsigned char * hash)` qui effectue le passage d'une valeur de hachage à une chaîne de caractères où elle est écrite en succession de valeurs hexadécimales, ainsi qu'une fonction effectuant l'opération inverse : `unsigned char * str_to_hash (char * st)`.

Pour la première, nous faisons une simple boucle ajoutant dans la chaîne de caractères, case par case, les valeurs du tableau en hexadéicmal à l'aide de `sprintf`.

Pour la seconde, il faut faire un buffer intermédiaire en itinérant sur ses cases trois par trois (donc sur trois fois la constante de la taille des valeurs de hachage fournies par la bibliothèque) afin d'y récupérer le format de l'hexadécimal, le conserver dans un `unsigned int stock` puis le mettre dans le tableau de la valeur de hachage que nous voulons. [1]

[1] *Note : On peut aisément itérer sur cette taille là car les valeurs de hachage ont une taille fixe avec la fonction de hachage cryptographique SHA 256*

Q.4.1.1

Pour l'écriture du block, nous avons choisi le format :

.
`clé publique de l'émetteur du block`
`Valeur de hachage` affichée en héxadécimal avec un espace entre chaque caractère
`Valeur de hachage` précédente avec le même format
`Valeur de la preuve de travail`
Toutes les `déclarations protégées` enregistrées par le block (une par ligne)
Un tilde (∼) à la fin des déclarations afin de voir où s'arrête la liste
.

On remplit alors le fichier ligne par ligne avec nos différentes fonctions subsidiaires et `fprintf`.

Q.4.1.2

Pour la lecture, il nous faut donc lire le fichier ligne par ligne, conservant à chaque fois cette dernière dans le buffer. Pour nos deux valeurs de hachage, on utilise la fonction `str_to_hash` afin d'obtenir le hash correspondant à la ligne lue au format hexadécimal.

On peut ensuite lire la valeur de la preuve de travail puis tous les protected déclarés ajoutés dans la liste chaînée de protected du block. Le tilde nous permet d'avoir une condition d'arrêt sur notre boucle sans savoir le nombre de déclarations.

Lorsque toutes ces opérations ont été réalisées, on peut retourner le résultat.

Q.4.1.3

Pour cette fonction, on ne s'est pas souciés de la mise en page, puisqu'elle n'a pas pour but de produire une chaîne lisibile mais une chaîne à laquelle appliquer la fonction de hachage. Des optimisations sont possibles, mais nous ne les avons pas jugées nécessaires car elle ne sert que lors du minage, qu'on souhaite être long.

Q.4.1.4

L'affichage proposé par le main de tests `tests_SHA256` est celui que nous avons utilisé dans les questions précédentes.

Q.4.1.5

La fonction de hachage utilisée sera SHA256 appliquée à notre chaîne de caractères avec sa longueur.

Q.4.1.6

Pour implémenter `compute_proof_of_work`, nous avons fait une fonction `int verifie_nb_d (unsigned char * hash, int d)` qui vérifie qu'il y a bien d 0 au début de la représentation hexadécimale du `unsigned char`.

Puisqu'un `unsigned char` est codé sur 8 bits, on peut sub-diviser chaque case du tableau en deux. Ainsi, la case 0a représente le premier chiffre du couple de la représentation hexadécimale et la case 0b représente le second. Pour plus de simplicité, nous utiliserons la représentation binaire du nombre.

En effet, un `unsigned char` est donc fait d'une suite de 8 chiffres (soit des 0 , soit des 1), qu'on subdivise aisément en deux.

Lorsque d est pair, la fonction n'est pas complexe : il suffit de vérifier que les $n/2$ cases sont bien nulles (deux 0 par cases dans la représentation hexadécimale). Lorsqu'il est impair, nous utilisons un cache et une comparaison bit à bit avec l'opérateur & et le nombre binaire 0b11110000. En comparant, bits à bits, ce nombre avec un autre nombre de 8 bits, si un des 4 premiers n'est pas nul, alors le résultat ne sera pas nul, les 4 derniers 0 font que le reste (c'est à dire le deuxième chiffre de la représentation hexadécimale) n'est pas pris en compte.

Ainsi, il suffit de vérifier les valeurs jusqu'à $d/2$, et si d est impair, vérifier la condition avec le cache pour la case $d/2$.

Exemple sur une petite valeur :

Héxadécimal	`0 8`	`4 a`
Binaire	`0000 1000`	`0100 1010`
comparé avec &	`1111 0000`	`1111 0000`
=	`0000 0000`	`0100 0000`

`compute_proof_of_work` consiste alors à appliquer la fonction de hachage sur la conversion en chaîne de caractères du block jusqu'à ce qu'il vérifie `verifie_nb_d`. Pour modifier la chaîne de caractères du block, on modifie alors l'attribut nonce (incrémenté de 1)

Q.4.1.7

Un block est alors vérifié par `verify_block` si le nombre de 0 dans sa fonction de hachage est égal à d et si la valeur de hachage est égale à l'application de la fonction de hachage sur le block.

Q.4.1.8

Le temps de calcul croît rapidement du fait de notre fonction `block_to_str` est très couteuse.

Q.4.1.9

Concernant la suppression en mémoire des valeurs de hachage, cela dépend de comment le block a été initialisé (par exemple, pour le block b, la chaîne est statique, mais pas pour b_bis car il a été alloué dynamiquement), la solution choisie fut de les libérer hors de la fonction.