Rapport Toychain

Le but de ce projet est de créer des outils qui serviront à simuler une élections.

Choix de structure

- Afin d'avoir une meilleure arborescence nous avons choisit, d'une part de séparer les différents fichiers. Les fichiers définnissant les structures et les fonctions de manipulation de ces dernières sont présentes dans `/lib`. Tandis que les jeux de tests sont générés dans `/test`. Ca devient vite pratique quand on a pleins d'éxécutables.

Nous avons d'ailleurs choisi d'utiliser **`Doxygen`**, outil très utile pour faire de la documentation. Les fichiers qui permettent de générer la doc sont contenus dans `/docs`.

- Le projet contient plusieurs fichiers **`Makefile`**, un principal, et d'autres dans différents dossier (`lib` et `test` pour l'instant). Cela nous permet depuis un makefile principal, de séparer les tâches, mais surtout de pouvoir accéder à des fichiers qui peuvent être contenus dans des dossiers différents. A titre d'exemple, les fichiers de tests font appel aux fichiers de `lib` lors de la compilation. Ainsi on a une structure très fléxible. Dès qu'on veut rajouter un nouveau fichier, il suffit de le spécifier dans son makefile localement, sans avoir besoin de le donner dans le makefile principal(qui sinon serait trop rempli et illisible). On peut également, grâce à cette structure exporter des variables. C'est le cas avec le file directory qu'on exporte et qui est accessible par les makefiles des sous-dossiers ou encore la constante `C_INCLUDE_PATH` qui nous permet d'ecrire des chemins absolus a partir de tout fichier faisant partie du projet.

- `/assignement` contient notre rapport et le sujet du projet.

- Pour les tests, nous avons défini dans test.h des fonctions qui vérifient le résultat d'un appel à une fonction.

    - **TEST_SECTION()** annonce le début d'un jeu de test dont on donne le nom en paramètre.
    - **TEST_SECTION_END()** annonce la fin du jeu de test en cours.
    - **TEST()** et **TEST_MSG()** vérifient si le résultat attendu est bien obtenus par la fonction donnée en paramètre. Dans le cas d'un succès, un compteur de succès est incrémenté, sinon un compteur d'échec est incrémenté. A chaque appel à une des 2 fonctions un compteur général est incrémenté.
    - **TEST_SUMMARY()** fait le bilan du dernier jeu de test : le nombre des tests réussis et échoués.

- Vérification de la présence d'overflow. Dans `overflow.h` on définit des fonctions qui vérifient si une opération(addition, multiplication ...) va causer un dépassement. La raison de cette implémentation vient du fait que des entiers trops grands peuvent amener à des mauvaises clés. Ce problème est détaillé dans la partie 1.

Partie 1

Dans cette première partie nous nous intéressons au chiffrement et dechiffrement d'un message à l'aide du protocole `RSA`. Dans le premier exercice on implémente des fonctions permettant de calculer des nombres premiers dont le test de `Miller-Rabin`. Dans l'exercice suivant, on utilise les fonctions précédentes pour déterminer des clés publiques et privées grâce à `l'Algorithme d'Euclide`.

Exercices 1 et 2

Q1.1

Dans le cas où p est premier, c'est-à-dire le pire cas, la complexité de **is_prime_naive** est O(p).

Q1.2

Le plus grand nombre premier qu'on parviens a verifier avec **is_prime_naive** en moins de `0.002s` est `5279`.

Q1.3

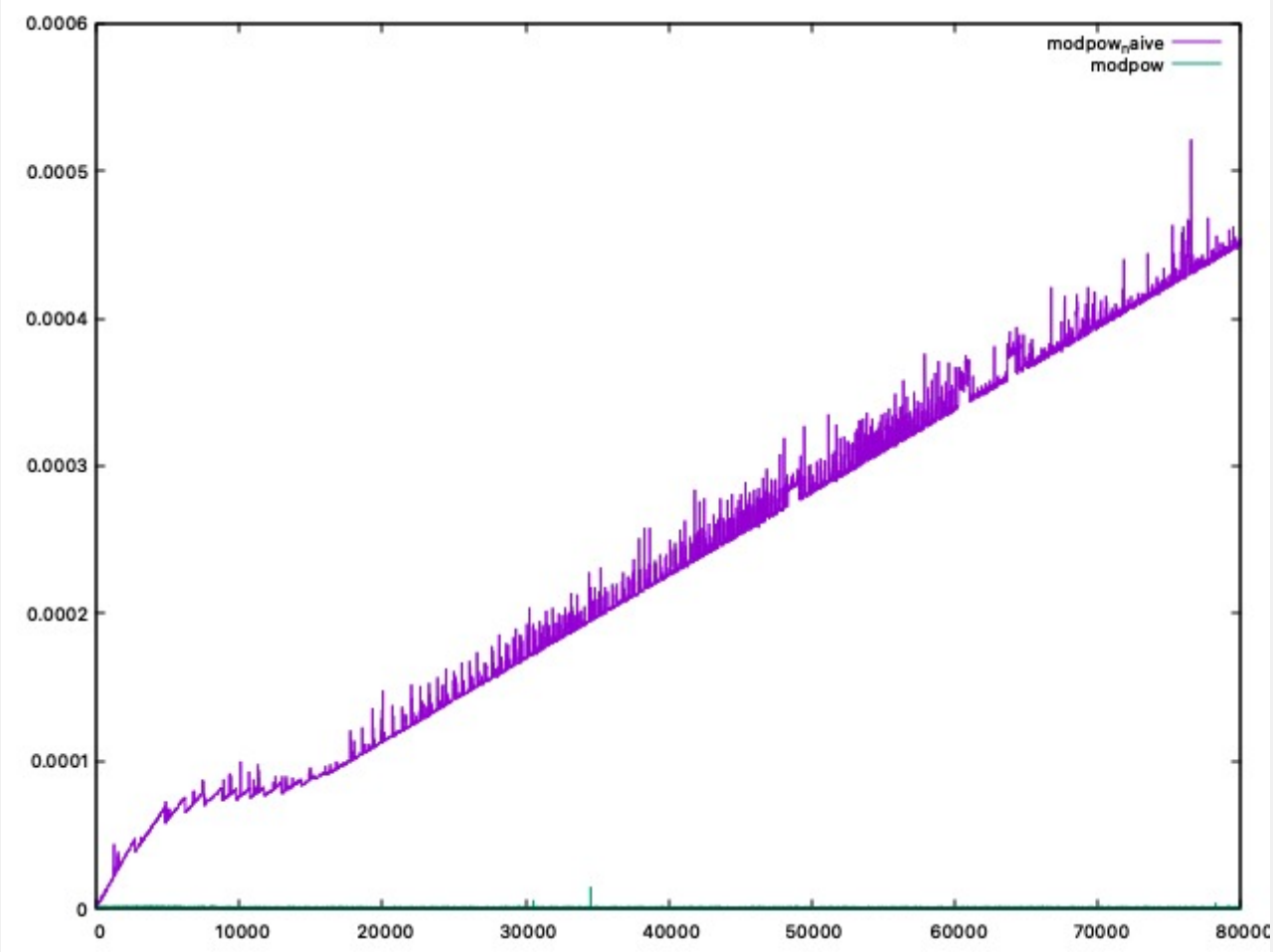
La complexité de **modpow_naive(a, m, n)** est O(m).

Q1.5

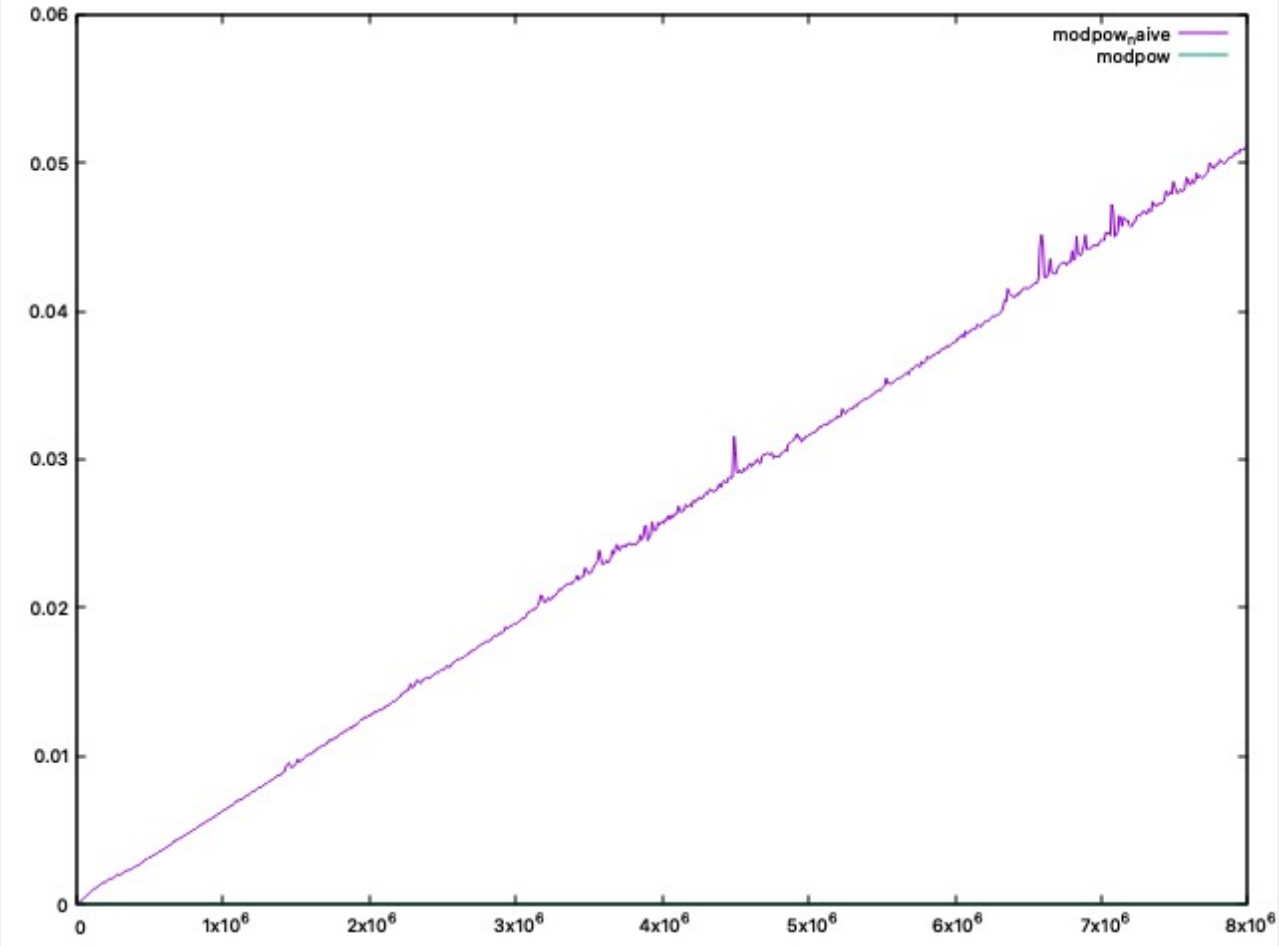
Pour les 2 fonctions, les temps sont très faibles même pour des grands m. La courbe de **modpow_naive** est bien lineaire et correspond bien a la complexité de O(n). La courbe de **modpow** est négligeable devant celle de **modpow_naive** et ne depasse jamais `0.0001s`

Exemple:

Values (a, m, n)	**modpow_naive**	**modpow**
(9435000329, 16, 1093)	0.000001 sec	0.000001 sec
(160986, 134217728, 25000)	0.817935 sec	0.000001 sec
(9435000329, 134217728, 1093)	1.491000 sec	0.000001 sec



Graphe d'execution de `modpow` et `modpow_naive` pour a = 1234 (la base), n = 667 (le module) et m (l'exposant) de 1 à 8000000 par pas de 10000



Graphe d'execution de `modpow` et `modpow_naive` pour a = 123456789 (la base), n = 66666 (le module) et m (l'exposant) de 1 à 8000000 par pas de 10000

Q1.7

Parmi les valeurs de `2` a `p-1`, cad `p-3` valeurs. On a une probabilité de `1/4` de ne pas tomber sur un temoin de Miller. De plus on effectue le test `k` fois avec remise (la fonction **rand_long** peut tomber deux fois sur la meme valeur). Autrement dit la probabilité ne change pas entre les lancers. Donc la probabilité de ne pas tomber sur un temoin de Miller pour `k` tests consecutifs est de `(1/4)^k`.

> Rmq: Il serait preferable d'adapter notre fonction de tirage aléatoire pour qu'elle ne répète pas deux fois la meme valeur. Cela augmenterait la probabilité de tomber sur un temoin de Miller à chaque iteration consecutive et par consequent diminuerait celle de faire un faux positif pour un même nombre des tests k. Cependant il se trouve que le gain de cette alteration serait négligeable devant la perte pour l'implementation de la fonction de tirage.

Jeu de tests

Les sets utilisés pour les tests de chiffrement sont générés aléatoirement. On implémente dans `test/test.h` des fonctions qui permettent de faires des tests simplements ainsi que `lib/overflow.h` pour detecter des overflows (voir la doc). Nous avons fournis un assez large set de tests, dont les résultats sont valides. Libre au lecteur de rajouteur ses propres tests. Cela se fait très simplement avec un appel à **TEST()** ou tout simplement avec des **printf**.

Difficultés rencontrées

Nous nous sommes rendus compte, dans la fonction **generate_key_values()**, que certaines valeurs générées pouvaient causer des overflows et ainsi perturber le chiffrement/dechiffrement.

C'est le cas de l'argument `n` de cette même fonction. En effet, si `n` est trop grand, sa multiplication par lui-même peut causer un *overflow*. Et il en va de même pour le produit de certains nombres modulo `n`. Par exemple : `x = (a % n) * (b % n)` *(où `a` et `b` entiers positifs quelconques)* peut depasser la taille d'un entier.

Le problème vient du fait que n est ensuite utilisé pour l'exponentiation modulaire **modpow()** et peut donc induire un mauvais chiffrement ou dechiffrement. Effectivement, si n a "*débordé*", l'opération modulo n donnera un résultat non valide pour notre utilisation. Or n est calculé à partir du produit de `p` et `q`. Donc si `p`, `q` sont trop grands, `p*q` peut causer un overflow.

Pour empêcher cela, nous avons :

1. Créé des runtime warnings dans les fonctions de base comme **modpow_r**.
2. Créé des define qui vérifient si une opération entre deux entiers cause un overflow (voir `overflow.h`).
3. Généré un nouveau couple `(p,q)` d'entiers premiers tant qu'un overflow a eu lieu pendant la génération, ie tant que `n = p*q` est trop grand. *(Qu'on vérifie grace aux fonctions dans `overflow.h`)*

Ainsi, on s'assure que le couple `(p,q)` ne cause pas d'overflow et qu'il permet bien de chiffrer/dechiffrer le message.

Partie 2

Dans cette partie, nous implémentons des structures qui permettront par la suite de générer des intentions de votes. L'un des objectifs est de rendre les déclarations de tous les participants uniques, de cette façon il devient impossible de voter à la place d'une autre personne sans sa clé secrète (dont on supposera qu'elle n'est jamais transmise). Ici pas de questions auxquelles répondre, donc on va plutôt faire une présentation de notre code, des choix de structures et des difficultés rencontrées. Et possiblement des tests de performance.

Exercice 3

Ici sont implémentées 3 structures(et leurs fonctions de manipulation) qui représentent :

1. Tous les citoyens, ansi que les candidats(1 clé publique et une clé privée) en utilisant une structure **_Key**.
2. Les signatures(tableau de `long`), qui permettent d'attester de l'authenticité d'une déclaration. Dont la structure est **_Signature**.
3. Des déclarations, contenant la clé publique du votant, un message(clé publique d'un candidat), et une signature. C'est représenté par la structure **_Protected**.

Tests

Les jeux de tests de cet exercice sont disponibles dans `test/rsa.c` et `test/sign.c`.

- `key_to_str()` et `str_to_key()` : les tests sont effectués dans une fonction `test/test_key_str_conversion()`, on vérifie la conversion `Key` -> `string` -> `Key`.
- De même pour `signature_to_str()`, `str_to_signature()` et `protected_to_str()`, `str_to_protected()`.

- On vérifie que la signature est valide avec `verify()` pour laquelle on génère un set aléatoire de clés.

Exercice 4

Cet exercice est une application directe des fonctions définies dans l'exercice 3. Avec `generate_random_data()`, on simule une élection et on sauvegarde les résultats dans 3 fichiers txt contenus dans /temp.

> Nous avons fait attention à ce qu'il ne soit pas possible qu'un citoyen soit plusieurs fois candidat, ce qui est possible quand on fait plusieurs tirages aléatoires dans une liste.

Nous ferons des tests pour vérifier que la sauvegarde des données s'est bien déroulée dans la partie 3 dans laquelle sont définies des fonctions de lecture.

Partie 3

Ici on implémente des fonctions de lecture et des structures qui peuvent stocker le contenu des *.txt* générés précédemment. Concrètement on à défini des listes chaînées pour les structures **_Key** et **_Protected**. Lors de la lecture d'un fichier texte. On ajoute en tête d'une liste chaînée chaque clé lue(après l'avoir convertie en clé bien sur).

Exercices 5 et 6

Tests

Dans l'exercice il est demandé de vérifier que **read_public_keys()** puis **print_list_keys()** donnent les mêmes clés que dans `keys.txt`. Faire des tests pour comparer des strings n'aurait pas beaucoup d'intérêt ici. En effet, pour faire les tests il faudrait redéfinir une autre fonction de lecture qui ferait exactement la même chose que celles demandées, qui serait alors elle aussi sujet à de potentielles erreurs. De plus les tests de conversions de clés on déjà été effectués plus tôt, le refaire ici serait rendondant. Cependant, en affichant le résultat de **print_list_keys()** on observe que la lecture puis conversion s'est bien déroulée. Pour s'en convaincre, ces clés sont affichées lors d'un appel à make test.

Résumé

Dans ces trois grandes parties nous avons développé des outils qui nous permettent de chiffrer/dechiffrer des messages. Pour cela nous avons utilisé le protocole `RSA` et les algorithmes permettant de générer des clés publiques et privées. Ces dernières sont l'élément clé *(ahah)* de notre simulation d'élection car nécessaires quand on veut créer un message unique et identifiable, de sorte qu'il est impossible d'usurper une autre personne(en considérant que les clés sont gardés bien secrètement).