

---

# Rapport : Projet

**LETELLIER Yohann**  
**GOGRITCHIANI Lasha**

Groupe 8  
2021-2022

---

## Introduction

Le but de ce projet est de proposer une piste de réflexion sur les protocoles et sur 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.

L'organisation d'un processus électoral par scrutin est considérée uninominal majoritaire à deux tours (comme ici en France). Chaque participant peut déclarer sa candidature au scrutin et/ou donner sa voix au candidat déclaré.

Le projet est divisé en 5 parties :

- Partie 1 : Implémentation d'outils de cryptographie.
- Partie 2 : Création d'un système de déclarations sécurisés par chiffrement asymétrique.
- Partie 3 : Manipulation d'une base centralisée de déclarations.
- Partie 4 : Implémentation d'un mécanisme de consensus.
- Partie 5 : Manipulation d'une base décentralisée de déclarations.

Chaque partie est divisée en exercices, que nous suivons pour organiser le projet dans les fichiers décrits dans le Table 1 ci-dessous.

| Fichier                           | Description  |
|-----------------------------------|--|
| Makefile                          | Make est un utilitaire permettant d'automatiser la compilation utile lorsque les sources sont organisés en plusieurs fichiers. Il permet de ne recompiler que les fichiers ayant été modifiés depuis la dernière compilation. Il se base sur un fichier décrivant les relations entre les fichiers et les actions à effectuer : le Makefile. Une fois le terminal est ouvert dans le répertoire, il suffit d'y taper : "make" ou "make all" pour compiler le programme, ou "make clean" pour effacer les fichiers.o et les exécutable. |
| main.c                            | Ce fichier est le fichier main de notre programme, qui contient les tests de tous les fonctions que nous écrivons dans le cadre de ce projet.  |
| primalite.c / primalite.h         | Exercice 1 : Les fonctions pour résoudre les problèmes de primalité.   |
| protocole_rsa.c / protocole_rsa.h | Exercice 2 : Les fonctions pour implémenter le protocole du RSA, pour chiffrer et déchiffrer les messages.   |
| keys.c / keys.h                   | Exercice 3 : Les fonctions pour manipuler des structures sécurisées, telles que les clés, les signatures et les déclarations signées.  |
| generate_data.c / generate_data.h | Exercice 4 : La fonction generate_random_data pour simuler le processus de vote.   |
| rw_data.c / rw_data.h             | Exercice 5 : Les fonctions pour lire et stocker les données dans les listes chaînées afin de déterminer le gagnant de l'élection.  |

TABLE 1 – Tableau descriptif des fichiers. Chaque fichier.c possède son header, fichier.h (excepté "main.c"), qui contient la définition des structures et les signatures des fonctions écrites dedans.

## Partie 1 : Développement d'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 implémentons est le pro-

protocole RSA (Rivest, Shamir, Adleman), très utilisé actuellement sur internet pour transmettre des données confidentielles (notamment dans le cadre du e-commerce). Ce protocole s'appuie sur des nombres premiers pour la génération de clés publiques et secrètes.

## Résolution du problème de primalité

Pour générer efficacement des nombres premiers, il faut avoir un moyen d'effectuer rapidement des tests de primalité. Nous essayons de résoudre ce problème par les méthodes différents dont nous choisissons le plus efficace. Le problème de primalité se définit comme suit : étant donnée un entier  $p$  impair,  $p$  est-il un nombre premier ?

### Implémentation par une méthode naïve

Une méthode naïve consiste à énumérer tous les entiers entre 3 et  $p - 1$ , et conclure que  $p$  est premier si et seulement si aucun de ces entiers ne divise  $p$ . Pour effectuer ça, nous implémentons la fonction `is_prime_naive(long p)` qui, étant donné un entier impair  $p$ , renvoie 1 si  $p$  est premier et 0 sinon. La fonction contient une boucle avec les opérations élémentaires, avec initialement  $i = 3$  et la condition d'arrêt  $i < m$  (la boucle s'arrête dès que  $i \geq m$ ). Comme  $i$  est incrémenté de 2 à la fin de chaque boucle, au total nous effectuons  $m/2 - 1$  tour de la boucle quand nous faisons appel à cette fonction. Nous en constatons que la complexité de `is_prime_naive(long p)` est en  $\Theta(n)$ .

En utilisant la boucle `for` et les outils disponibles dans la librairie `<time.h>`, nous calculons et ensuite transformons en secondes le temps mis par le processeur pour tester si le nombre est premier à l'aide de la fonction `is_prime_naive(long p)`. Nous remarquons que à partir d'un nombre codé sur 30 bits, la fonction commence à prendre plus que 2 secondes pour s'exécuter.

### Exponentiation modulaire rapide

Pour parvenir à générer de très grands nombres premiers, nécessaires au bon fonctionnement du protocole RSA en pratique, nous implémentons un test de primalité plus efficace : le test de primalité de Miller-Rabin. Ceci nécessite de calculer efficacement une exponentiation modulaire, c.à.d. la valeur  $a^m \bmod n$ .

La fonction `long modpow_naive(long a, long m, long n)` prend en entrée trois entiers  $a$ ,  $m$  et  $n$  et retourne la valeur  $a^m \bmod n$  par la méthode naïve, la fonction effectue  $m$  tour de la boucle, donc sa complexité est en :  $\Theta(m)$ .

Au lieu de multiplier par  $a$  à chaque itération, nous implémentons une version récursive de la fonction `long modpow(long a, long m, long n)` qui réalise des élévations au carré, qui nous donne en conséquence un algorithme de complexité logarithmique (c.à.d. en  $\mathcal{O}(\log_2(m))$ ).

Nous comparons les performances des deux méthodes d'exponentiation modulaire en traçant des courbes de temps en fonction de  $m$  à l'aide d'un logiciel

gnuplot (Cf. Figure 1 ) et nous remarquons que la fonction `modpow` est bien plus avantageuse à utiliser.

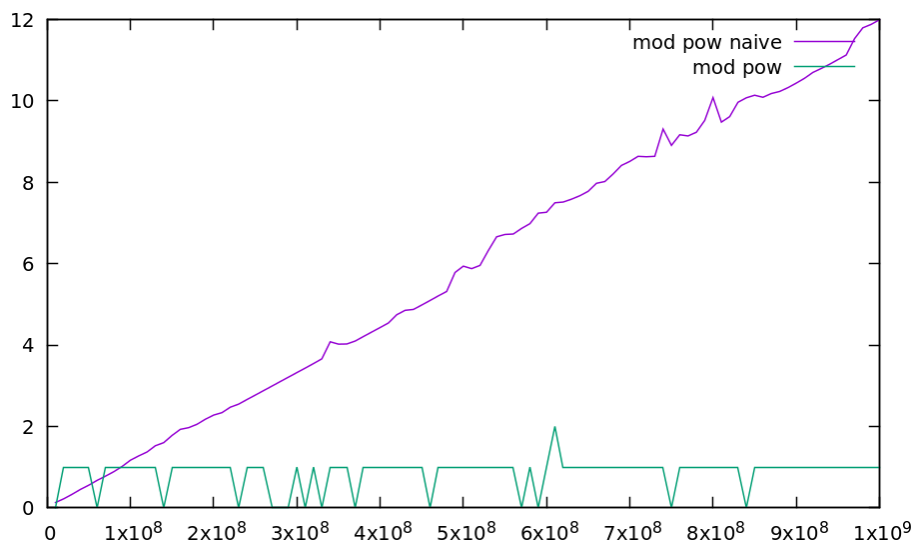


FIGURE 1 – Comparaison des temps mit par les deux méthodes pour effectuer le calcul d’une exponentiation modulaire.

### Test Miller-Raban

Nous intégrons dans notre code les fonctions suivantes :

- `int witness(long a, long b, long d, long p)` qui teste si `a` est un témoin de Miller pour `p`, pour un entier `a` donné.
- `long rand_long(long low, long up)` qui retourne un entier `long` générée aléatoirement entre `low` et `up` inclus (la taille dans le mémoire).
- `int is_prime_miller(long p, int k)` qui 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`. 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).

Comme la probabilité d’erreur de cet algorithme devient rapidement très faible quand `k` augmente, et que sa complexité pire-cas est en  $O(k(\log_2(p))^3)$ , alors il est plus intéressant d’utiliser cet algorithme pour effectuer des tests de primalité que la méthode naïve.

### Génération de nombres premiers

Nous écrivons une fonction `long random_prime_number(int low_size, int up_size, int k)` qui étant donnée :deux entiers `low_size` et `up_size` respectivement la taille minimale et maximale du nombre premier à générer, et un entier

$k$  représentant le nombre de tests de Miller à réaliser, retourne un nombre premier de taille comprise entre `low_size` et `up_size`. Dans la suite de projet, nous prenons  $k = 5000$ .

## Implémentation du protocole du RSA

Le chiffrement RSA, nommé ainsi par les initiales de ses trois inventeurs (Rivest, Shamir et Adleman), est un algorithme de cryptographie asymétrique qui a été décrit en 1977 et breveté en 1983.

### Génération d'une paire (Clé publique, Clé secrète).

Pour pouvoir envoyer des données confidentielles avec le protocole RSA, nous avons besoin de générer une paire des clés (clé publique, clé secrète).

dans le fichier `protocole_rsa.c` nous implémentons la fonction `void generate_key_values(long p, long q, long* n, long *s, long *u)` qui permet de générer la clé publique  $pKey = (s, n)$  et la clé secrète  $sKey = (u, n)$ , 'a partir des nombres premiers  $p$  et  $q$ , en suivant le protocole RSA.

### Chiffrement et déchiffrement de messages

On s'intéresse maintenant à l'envoi de message. Supposons que la personne C2 souhaite envoyer un message à la personne C1 en utilisant le protocole RSA. Dans ce cas, la personne C2 utilise la clé publique de la personne C1 pour chiffrer le message avant son envoi. À sa réception, la personne C1 déchiffre le message à l'aide de sa clé secrète (Cf.Figure 2).

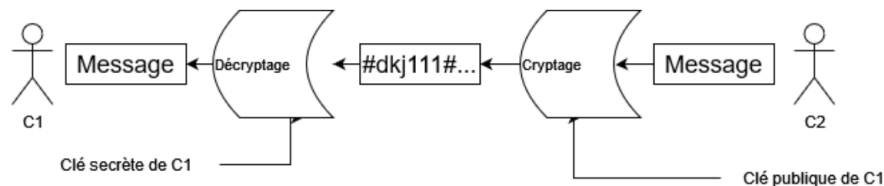


FIGURE 2 – Le principe de fonctionnement du protocole RSA pour l'envoi d'une message chiffrée.

Nous Implémentons la fonction `long* encrypt(char* chaine, long s, long n)` qui chiffre la chaîne de caractères `chaine` avec la clé publique  $pKey = (s,n)$ . Nous chiffons le message  $m$  en calculant  $c = m^s \bmod n$ . Pour effectuer cette operation, la fonction convertit chaque caractère en un entier de type `int` (sauf le caractère spécial `'\0'`), et retourne le tableau de `long` obtenu en chiffrant ces entiers. Pour réaliser une exponentiation modulaire efficace, nous utilisons la fonction `modpow`.

Afin de déchiffrer  $c$  pour retrouver  $m$  en calculant  $m = c^u \bmod n$ , nous implémentons la fonction `char* decrypt(long* crypted, int size, long u,`

`long n)` qui effectue cette opération à l'aide d'une clé secrète `sKey = (u, n)`, en connaissant la taille du tableau d'entiers. Cette fonction renvoie la chaîne de caractères obtenue, sans oublier le caractère spécial `'\0'` à la fin.

### Fonctions de tests

Pour vérifier le bon fonctionnement de notre programme, nous reproduisons le programme principal suivant nous compilons et exécutons la fonction `test_protocole_rsa()` défini dans le fichier `main.c`. Nous vérifions l'absence de fuite mémoire à l'aide de Valgrind (Cf. Figure3).

```
Vecteur: [38a  119  435  435  4a8  ]
72
101
108
108
111
Decoded : Hello
==95706==
==95706== HEAP SUMMARY:
==95706==    in use at exit: 0 bytes in 0 blocks
==95706==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==95706==
==95706== All heap blocks were freed -- no leaks are possible
```

FIGURE 3 – Vérification d'un bon fonctionnement et l'absence d'une fuite mémoire avec Valgrind d'une fonction `test_protocole_rsa()`

## Partie 2 : Déclarations sécurisées

Un citoyen interagit pendant les élections en effectuant des déclarations. En pratique, ces déclarations peuvent soit être des déclarations de candidature soit des déclarations de vote. Dans le cadre de notre projet, nous avons supposé que l'ensemble des candidats est déjà connu, et que les citoyens ont juste à soumettre des déclarations de vote.

### Manipulations de structures sécurisées

Dans le modèle de notre projet 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 de vote, non seulement quand il vote, mais aussi quand quelqu'un souhaite voter en sa faveur.

Nous effectuons signature de la déclaration par chiffrement du contenu (avec la clé secrète), puis vérifions une signature simplement par déchiffrement (avec la clé publique).

## Manipulation des clés

Nous définissons une structure `Key` dans le fichier `keys.h`. Ceci contient deux long représentant une clé (publique ou secrète). Ensuite nous écrivons dans le fichier `keys.c` les fonctions nécessaires pour manipuler cette structure. La fonction `void init_key(Key* key, long val, long n)` permet d'initialiser une clé déjà allouée. La fonction `void init_pair_keys(Key* pKey, Key* sKey, long low_size, long up_size)` utilise le protocole RSA pour initialiser une clé publique et une clé secrète (déjà allouées) en faisant appel aux fonctions `random_prime_number`, `generate_keys_values` et `init_key`.

Nous écrivons ensuite deux fonctions `char* key_to_str(Key* key)` et `Key* str_to_key(char* str)` qui permettent de passer d'une variable de type `Key` à sa représentation sous forme de chaîne de caractères et inversement. La chaîne de caractères doit être de la forme "(x,y)", où x et y sont les deux entiers de la clé exprimées en hexadécimal. Nous avons l'intérêt de les exprimer en hexadécimal parce qu'il permet une conversion sans aucun calcul avec le système binaire, et qu'il possède l'avantage de rendre des entiers très grands plus compactes et plus lisibles.

## Signature

Une déclaration de vote consiste à transmettre la clé publique du candidat sur qui porte le vote. Dans un processus de scrutin, il faut que chaque personne puisse produire des déclarations de vote signée pour attester de l'authenticité de la déclaration.

Nous implémentons une signature simplement comme un tableau de long dont on connaît la longueur. Nous définissons la structure `Signature` dans le fichier `keys.h` et ensuite les fonctions `Signature* init_signature(long* content, int size)` et `Signature* sign(char* mess, Key* sKey)` dans le fichier `keys.c` qui permettent de d'allouer et de remplir une signature avec un tableau de long déjà alloué et initialisé et de créer une signature à partir du message `mess` (déclaration de vote) et de la clé secrète de l'émetteur.

Les fonctions `signature_to_str` et `str_to_signature`, qui permettent de passer d'une `Signature` à sa représentation sous forme de chaîne de caractères et inversement sont déjà données et nous les intégrons simplement dans notre programme, dans le fichier `keys.c`. La chaîne de caractères est de la forme "`#x0#x1#...#xn#`" où  $x_i$  est le  $i$ ème entier du tableau de la signature donnée en hexadécimal.

## Déclarations signées

Nous avons maintenant tous les outils pour créer des déclarations signées (données protégées).

Dans le fichier `keys.h` nous définissons la structure `Protected` qui contient la clé publique de l'émetteur (l'électeur), son message (sa déclaration de vote), et la signature associée. Ensuite, dans le fichier `keys.c` nous écrivons une fonction `Protected* init_protected(Key* pKey, char* mess, Signature* sgn)` qui

alloue et initialise cette structure et pour vérifier si la signature est bien valide nous ajoutons une fonction `int verify(Protected* pr)` qui vérifie que la signature contenue dans `pr` correspond bien au message et à la personne contenus dans `pr`.

À la fin, nous écrivons les fonctions `protected_to_str` et `str_to_protected` qui permettent de passer d'un `Protected` à sa représentation sous forme de chaîne de caractères et inversement.

## Fonctions de tests

Pour vérifier le bon fonctionnement de notre programme, nous reproduisons le programme principal suivant nous compilons et exécutons la fonction `test_keys()` défini dans le fichier `main.c`. (Cf. Figure)

```
pKey: 213, 14e9
sKey: 55b, 14e9
key to str:(213,14e9)
str to key: 213, 14e9
(213,14e9) votepour (615,1705)
signature:Vecteur: [17a 10c6 d1a 2e6 81d d1a 1318 14c6 2e6 68c ]
signature to str:#17a#10c6#d1a#2e6#81d#d1a#1318#14c6#2e6#68c#
str to signature:Vecteur: [17a 10c6 d1a 2e6 81d d1a 1318 14c6 2e6 68c ]
40
54
49
53
44
49
55
48
53
41
Signature valide
protected to str:(213,14e9) (615,1705) #17a#10c6#d1a#2e6#81d#d1a#1318#14c6#2e6#68c#
str to protected: (10b6fce70,7ff7b48047a8) 1318#14c6#2e6#68c#05)14e9) #
```

FIGURE 4 – Vérification d’une bonne fonctionnement des fonctions pour manipuler les déclarations sécurisées que nous avons défini dans le fichier `keys.c`

## Création de données pour simuler le processus de vote

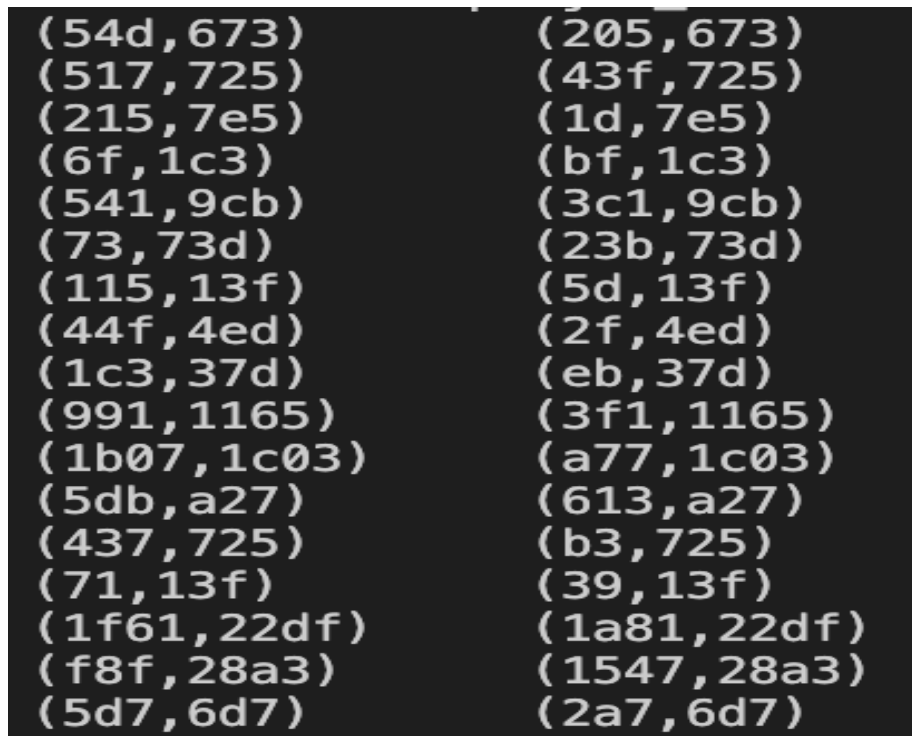
Nous simulons le processus de vote à l’aide de trois fichiers : un fichier contenant les clés de tous les citoyens, un fichier indiquant les candidats et un fichier contenant des déclarations singées. Pour cela nous écrivons le fichier `generate_data.c` et son header, contenant une fonction `void generate_random_data(int nv, int nc)` qui :

- génère `nv` couples de clés (publique, secrète) différents représentant les `nv` citoyens.
- crée un fichier `keys.txt` contenant tous ces couples de clés (un couple par ligne).
- sélectionne `nc` clés publiques aléatoirement pour définir les `nc` candidats.
- crée un fichier `candidates.txt` contenant la clé publique de tous les candidats (une clé publique par ligne).
- génère une déclaration de vote signée pour chaque citoyen (candidat choisit aléatoirement).



- crée un fichier `declarations.txt` contenant toutes les déclarations signées (une déclaration par ligne).

Nous vérifions la bonne fonctionnement de cette fonction en la faisant appeler dans le fichier `main.c`.



```
(54d,673) (205,673)
(517,725) (43f,725)
(215,7e5) (1d,7e5)
(6f,1c3) (bf,1c3)
(541,9cb) (3c1,9cb)
(73,73d) (23b,73d)
(115,13f) (5d,13f)
(44f,4ed) (2f,4ed)
(1c3,37d) (eb,37d)
(991,1165) (3f1,1165)
(1b07,1c03) (a77,1c03)
(5db,a27) (613,a27)
(437,725) (b3,725)
(71,13f) (39,13f)
(1f61,22df) (1a81,22df)
(f8f,28a3) (1547,28a3)
(5d7,6d7) (2a7,6d7)
```

FIGURE 5 – Vérification de la bonne fonctionnement de la fonction `void generate_random_data(int nv, int nc)`.

### Partie 3 : Base de déclarations centralisée

Dans cette partie, on considère un système de vote centralisé, dans lequel toutes les déclarations de vote sont envoyées au système de vote, qui a pour rôle de collecter tous les votes et d'annoncer le vainqueur de l'élection à tous les citoyens. En pratique, les déclarations de vote sont enregistrées au fur et à mesure dans un fichier appelé `declarations.txt`, et une fois que le scrutin est clos, ces données sont chargées dans une liste chaînée. Pour pouvoir vérifier l'intégrité des données et comptabiliser les votes, le système doit aussi récupérer l'ensemble des clés publiques des citoyens et des candidats, qui sont stockées respectivement dans les fichiers appelés `keys.txt` et `candidates.txt`.

## Lecture et stockage des données dans des listes chaînées

Dans cette partie, nous nous intéressons à la lecture et au stockage des données sous forme de listes (simplement) chaînées. Nous nous occupons d'abord des fichiers `keys.txt` et `candidates.txt`, qui conduisent à des listes chaînées de clés (publiques), puis nous nous occupons du fichier `declarations.txt`, qui correspond à une liste chaînée de déclarations signées.

### Liste chaînée des clés

Pour les listes chaînées de clés, nous allons définir la structure `CellKey` dans le fichier `rw_data.h`. Nous écrivons les fonctions de manipulation de cette structure dans le fichier `rw_data.c`.

La fonction `CellKey* create_cell_key(Key* key)` alloue et initialise une cellule de liste chaînée. La fonction `void push_lst_key(CellKey **lst_key, Key* key)` ajoute une clé en tête de la liste.

Nous écrivons ensuite une fonction `read_public_keys(char *file_Name)` qui prend en entrée le fichier `keys.txt` ou le fichier `candidates.txt`, et qui retourne une liste chaînée contenant toutes les clés publiques du fichier.

La fonction `void print_list_keys(CellKey* LCK)` permet d'afficher une liste chaînée de clés, puis nous utilisons cette fonction pour vérifier notre fonction de lecture.

À la fin, nous ajoutons la fonction `void delete_cell_key(CellKey* c)` qui supprime une cellule de liste chaînée de clés et la fonction `delete_list_keys(CellKey* c)` qui supprime une liste chaînée de clés.

### Liste chaînée de déclarations signées

Pour la liste chaînée de déclarations signées, nous allons ajouter la structure `CellProtected` dans le fichier `rw_data.h` et ensuite nous définissons les mêmes fonctions que nous avons défini pour la structure `CellKey` pour cette structure aussi.

### Détermination du gagnant de l'élection

Une fois toutes les données collectées, le système commence par retirer toutes les déclarations contenant une fausse signature (tentative de fraude). Pour effectuer cela, nous définissons la fonction la fonction `void remove_fraud(CellProtected **LCP)`.