



**LU2IN006**

**COMPTE RENDU PROJET :**  
**BLOCKCHAIN APPLIQUÉE À UN PROCESSUS ÉLECTORAL**

HORCHANI Samy – DORMANT Maéva

Avril 2022

# Table des matières

<b>Table des matières</b>	<b>p.2</b>
<b>Introduction</b>	<b>p.3</b>
1. <u>Développement d'outils cryptographiques</u>	<u>p.4-6</u>
a. Résolution du problème de primalité	p.4
b. Implémentation du protocole RSA	p.6
2. <u>Déclarations sécurisées</u>	<u>p.7-8</u>
a. Manipulations de structures sécurisées	p.7
b. Création de données pour simuler le processus de vote	p.8
3. <u>Base de déclarations centralisée</u>	<u>p.9-10</u>
a. Lecture et stockage des données dans des listes chaînées	p.9
b. Détermination du gagnant de l'élection	p.10
4. <u>Blocs et persistance des données</u>	<u>p.11-13</u>
a. Structure d'un bloc et persistance	p.11
b. Structure arborescente	p.13
c. Simulation du processus de vote	p.13
<b>Conclusion</b>	<b>p.14</b>

# Introduction

**Cadre du projet :** Dans un processus électoral, chaque participant peut donner sa candidature ou donner sa voix à un candidat déclaré. Néanmoins, les élections présentent des conflits d'intérêts car elles sont organisées par le gouvernement en place. Le vote décentralisé permettrait de réduire les risques de fraude et de réduire le taux d'abstention.

**Objectif :** 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

**Remarque :**

- Nous testerons nos fonctions dans un fichier `< tests.c >`
- Le programme de tests fourni sera dans un fichier `< main.c >`
- La simulation finale de nos élections sera dans un fichier `< simulation.c >`
- Le Makefile permet de réaliser automatiquement des commandes (exécution, compilation, valgrind, création de dossiers et de répertoires) sur Linux
- Nous avons essayé de gérer au maximum les fuites mémoires.

## Développement d'outils cryptographiques

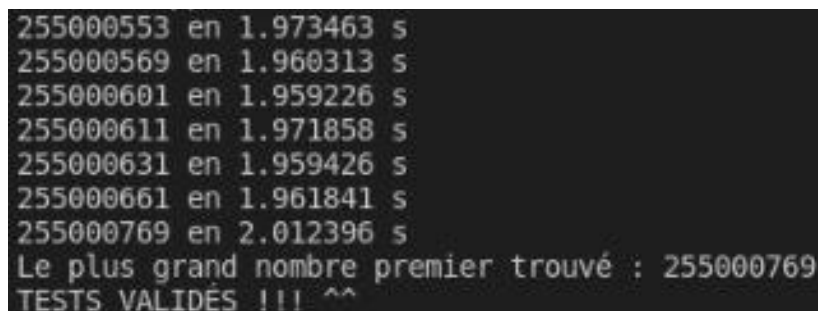
La cryptographie asymétrique consiste en la création de deux clés : une clé publique et une clé privée. La clé publique permet à l'envoyeur de chiffrer son message et la clé secrète permet au récepteur de déchiffrer le message reçu. Pour générer ces clés, le protocole RSA utilise de grands nombres premiers.

**Remarque :** Nos fonctions, nos structures et nos prototypes seront contenus dans les fichiers suivants : < dev\_out\_crypto.h > et < dev\_out\_crypto.c >.

### Résolution du problème de primalité

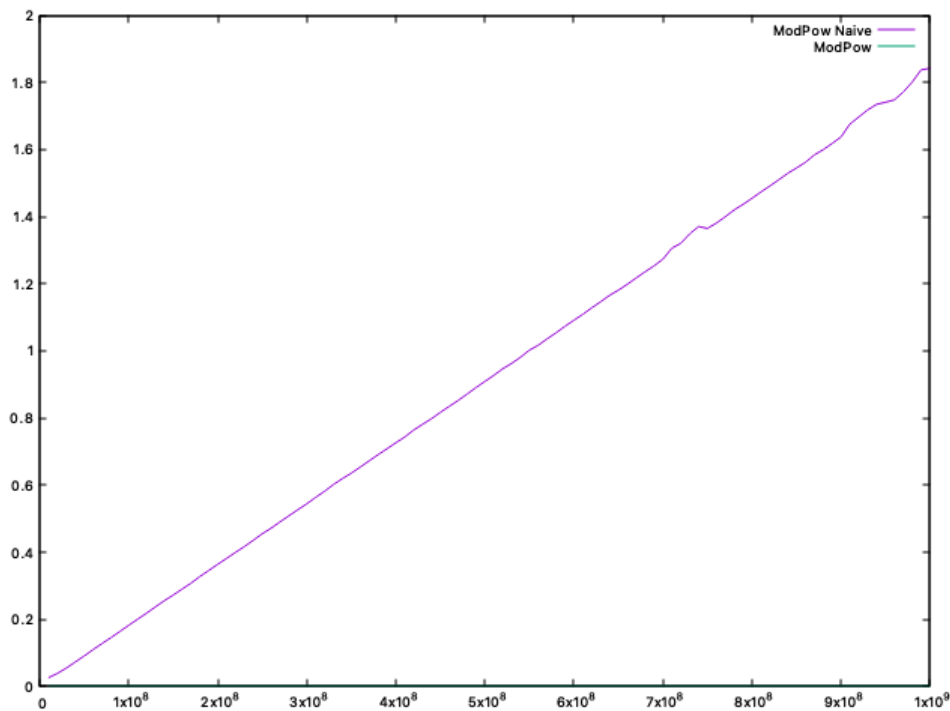
**Objectif :** Effectuer rapidement des tests de primalité

Nous cherchons à réaliser des tests de primalité avec une méthode naïve. Étant donné un entier impair  $p$ , < is\_prime\_naive > retourne 1 si  $p$  est premier et 0 sinon. Dans le meilleur des cas, la fonction ne réalisera qu'un seul tour de boucle :  $\Omega(1)$ . Dans le pire des cas, elle en réalisera  $p : \mathcal{O}(p)$ . La fonction nous permet de tester la primalité de 255 00 661, en moins de 2 secondes (cf : photo du terminal ci-dessous).



```
255000553 en 1.973463 s
255000569 en 1.960313 s
255000601 en 1.959226 s
255000611 en 1.971858 s
255000631 en 1.959426 s
255000661 en 1.961841 s
255000769 en 2.012396 s
Le plus grand nombre premier trouvé : 255000769
TESTS VALIDÉS !!! ^^
```

Nous utilisons deux méthodes pour l'exponentiation modulaire. Étant donnés trois entiers  $a$ ,  $m$  et  $n$ , < modpow\_naive > et < modpow > retournent  $a^m \bmod n$ . < modpow\_naive > est moins performante que < modpow > car elle utilise une méthode naïve, et non la méthode de l'élévation au carré. Son temps d'exécution est très proche de 0 s, donc la courbe est très difficile à voir sur le graphe.



**Nous implémentons le test de Miller-Rabin.** Dans un premier temps, nous cherchons les témoins de Miller. Étant donnés quatre entiers  $a$ ,  $b$ ,  $d$  et  $p$ , `< witness >` retourne 1 si  $a$  est un témoin de Miller pour  $p$  et 0 sinon. Dans un second temps, nous générons des nombres aléatoires. Étant donnés deux entiers  $low$  et  $up$ , `< rand_long >` retourne (aléatoirement) un entier compris entre  $low$  et  $up$ . Le test de Miller-Rabin est, donc, implémenté à l'aide des fonctions précédentes. Étant donnés deux entiers  $p$  et  $k$ , `< is_prime_miller >` retourne 1 dès qu'un témoin de Miller a été trouvé et 0 sinon. Cependant, cet algorithme a une probabilité d'erreur de :  $\frac{1}{4}$ . En effet, s'il retourne 1, nous sommes sûrs que  $p$  n'est pas un nombre premier. Dans l'autre cas, il y a une forte probabilité qu'il soit premier.

**Nous voulons, à présent, générer de très grands nombres en utilisant `< rand_long >`.** Étant donnés trois entiers  $low\_size$ ,  $up\_size$  et  $k$ , `< random_prime_number >` retourne un nombre premier de taille comprise entre  $low\_size$  bits et  $up\_size$  bits.

## Implémentation du protocole RSA

**Objectif :** Générer des clés, chiffrer et déchiffrer des messages

Les entiers  $s$ ,  $n$  et  $u$  nous permettront de constituer les clés publiques  $pkey = (s, n)$  et privées  $skey = (u, n)$ .

On a :  $n = p \cdot q$  et  $t = (p - 1) \cdot (q - 1)$

On cherche :

- $s$  et  $t$ , tels que  $s < t$  et  $\text{PGCD}(s, t) = 1$ . Étant donnés deux entiers  $s$  et  $t$ , `< extended_gcd >` retourne les entiers  $u$ ,  $v$  et retourne la valeur  $\text{PGCD}(s, t)$ .
- $u$  tel que  $s \cdot u \bmod t = 1$ . Étant donnés deux entiers  $p$  et  $q$ , `< generate_key_values >` retourne les entiers  $n$ ,  $s$  et  $u$  pour générer la clé publique et la clé secrète.

**Nous chiffrons un message**  $m$  en calculant  $c = m^s \bmod n$ . Étant donnés une chaîne de caractère et deux entiers, `< encrypt >` retourne un tableau d'entiers représentant le chiffrement de la chaîne).

**Nous déchiffrons un message**  $c$  en calculant  $m = c^u \bmod n$ . Étant donnés une chaîne de caractère et deux entiers, `< decrypt >` retourne une chaîne de caractères représentant le message décrypté).

## Déclarations sécurisées

Dans notre système de vote, un citoyen peut soit déclarer sa candidature, soit déclarer un vote. On suppose que tous les candidats sont connus. On ne s'intéresse donc qu'à la soumission de votes.

**Remarque :** Nos fonctions, nos structures et nos prototypes seront contenus dans les fichiers suivants : `< dec_sec.h >` et `< dec_sec.c >`.

## Manipulation de structures sécurisées

**Objectif :** Signer une déclaration par chiffrement de contenu et vérifier une signature par déchiffrement

L'électeur vote pour un candidat et signe sa déclaration. Ensuite, il publie sa déclaration avec le message, la signature et la clé publique.

**Nous manipulons des clés** représentées par la structure `Key`. Étant donnés une clé et deux entiers, `< init_key >` initialise la clé déjà allouée. Étant donnés, deux clés et deux entiers, `< init_pair_keys >` initialise une clé publique et une clé secrète selon le protocole RSA. Étant donné une clé, `< key_to_str >` retourne la présentation sous forme de chaîne de caractères de la clé en paramètres. Étant donné une chaîne de caractères, `< str_to_key >` retourne la présentation sous forme de clé de la chaîne de caractères en paramètres.

**Nous produisons les signatures des déclarations de vote car la signature atteste l'authenticité de la déclaration.** Une signature est représentée par la structure `Signature`. Étant donnés un contenu et sa taille, `< init_signature >` initialise une signature. Étant donnés un message et une clé secrète, `< sign >` retourne la signature de l'émetteur. Étant donné une signature, `< signature_to_str >` retourne la présentation sous forme de chaîne de caractères de la signature en paramètres. Étant donné une chaîne de caractères, `< str_to_signature >` retourne la présentation sous forme de signature de la chaîne de caractères en paramètres.

**Nous créons des déclarations signées.** En tant que données protégées, elles sont représentées par la structure `Protected`. Étant donnés une clé publique, une clé secrète, un message et une signature, `< init_protected >` initialise une déclaration signée. Étant donnée une donnée protégée, `< verify >` retourne 1 si la signature correspond au message et à la personne de la déclaration et 0 sinon. Étant donné une déclaration, `< protected_to_str >` retourne la présentation sous forme de chaîne de caractères de la déclaration en paramètres. Étant donné une chaîne de caractères, `< str_to_protected >` retourne la présentation sous forme de déclaration de la chaîne de caractères en paramètres.

## Création de données pour simuler le processus de vote

**Objectif :** Simuler un processus de vote à l'aide d'une base de données de citoyens, de candidats et de déclarations signées

Pour mettre en place un scrutin, il faut générer pour chaque citoyen une carte électorale. Le système de vote collectera chaque déclaration et vérifiera leur authenticité.

**Nous générons des données aléatoires.** Étant données deux entiers, `< generate_random_data >` crée des fichiers .txt pour les couples de clés, les clés publiques, les déclarations.



## Base de déclarations centralisée

On considère un système de vote centralisé dans lequel les déclarations de vote sont envoyées au système de vote. Le système de vote collecte tous les votes et annonce le vainqueur de l'élection.

**Remarque :** Nos fonctions, nos structures et nos prototypes seront contenus dans les fichiers suivants : `< bdc.h >` et `< bdc.c >`.

### Lecture et stockage des données dans des listes chaînées

**Objectif :** Lire et stocker des données sous forme de listes simplement chaînées

**Nous manipulons des clés avec des listes chaînées** représentées par `CellKey`. Étant donnée une clé, `< create_cell_key >` initialise une cellule de liste chaînée. Étant donnée une clé et une liste chaînée, `< add_key >` ajoute la clé correspondante en tête de la liste chaînée. Étant donné un fichier, `< read_public_keys >` retourne la liste chaînées contenant les clés publiques du fichier. Étant donnée une liste chaînée, `< print_list_keys >` affiche une liste chaînée de clés. Étant donnée une liste chaînée, `< delete_cell_key >` supprime une cellule de la liste chaînée en paramètres. Étant donnée une liste chaînée, `< delete_list_keys >` supprime une liste chaînée de clés.

**Nous manipulons des déclarations signées** avec des listes signées représentées par `CellProtected`. Étant donnée une déclaration signée, `< create_cell_protected >` initialise une cellule de liste chaînée. Étant donnée une déclaration signée et une liste chaînée, `< add_protected >` ajoute la déclaration signée correspondante en tête de la liste chaînée. Étant donné un fichier, `< read_protected >` retourne la liste chaînée contenant les déclarations signées du fichier. Étant donnée une liste chaînée, `< print_list_protected >` affiche une liste chaînée de déclarations signées. Étant donnée une liste chaînée, `< delete_cell_protected >` supprime une cellule de la liste chaînée en paramètre. Étant donnée une liste chaînée, `< delete_list_protected >` supprime une liste chaînée de déclarations signées.

## Détermination du gagnant de l'élection

**Objectif :** Déterminer le gagnant de l'élection en évitant les fraudes

**Nous cherchons à retirer les fausses déclarations.** Étant donnée une liste chaînée de déclarations signées, `< anti_fraude >` supprime toutes les déclarations dont la signature n'a pas été validée.

**Nous manipulons des cellules pour les tables de hachage.** HashCell est une cellule de table de hachage qui permet de compter le nombre de vote en fonction d'un candidat ou de vérifier si le votant a déjà voté. Étant donnée une clé, `< create_hashcell >` alloue une cellule de la table de hachage et initialise ses champs. Étant donné une clé et la taille d'un tableau, `< hash_function >` retourne la position de l'élément. Étant donnés une table de hachage et une clé, `< find_position >` cherche la clé dans la table. Si elle existe, on renvoie sa position, sinon on renvoie la position où elle aurait dû être.

**Nous manipulons des tables de hachage.** HashTable contient la table de hachage et sa taille. Étant donnés une liste chaînée de clés et une taille, `< create_hashtable >` contient une cellule pour chaque clé de la liste chaînée. Étant donnée une table de hachage, `< delete_hashtable >` supprime la table de hachage.

**Nous cherchons à déterminer le gagnant de l'élection de manière efficace.** Étant données trois listes et deux tailles, `< compute_winner >` calcule le vainqueur de l'élection.

## Blocs et persistance des données

Pour des questions de confiance, nous souhaitons utiliser un système de vote décentralisé. Il permettra à chaque citoyen de vérifier le résultat du vote. Dans notre système, chaque citoyen possède une copie de la base de déclarations de votes signés.

**Remarque :** Nos fonctions, nos structures et nos prototypes seront contenus dans les fichiers suivants : < bpd.h > et < bpd.c >.

### Structure d'un bloc et persistance

**Objectif :** Gérer des blocs

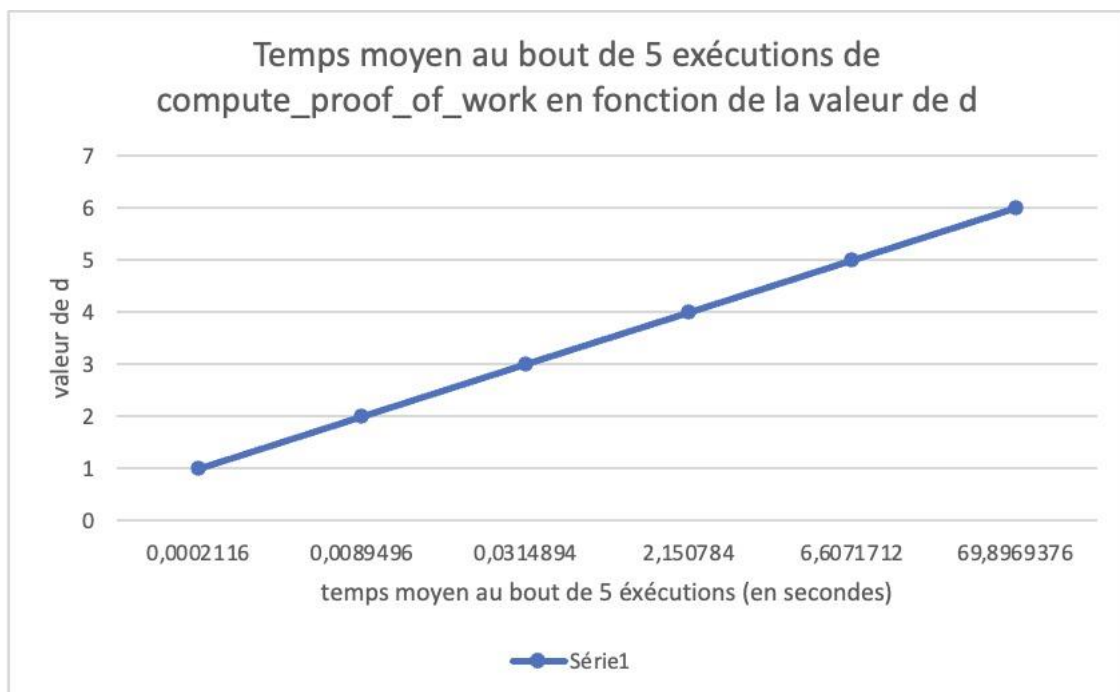
**Nous manipulons des blocs représentés par une structure Block. Nous souhaitons lire et écrire dans des fichiers contenant des blocs.** Étant donné le nom d'un fichier et un bloc, < write\_block > écrit tous les champs d'un bloc dans un fichier. Étant donné le nom d'un fichier, < read\_block > crée un bloc à partir d'un fichier déjà existant.

**Nous créons des blocs valides à l'aide d'un mécanisme *proof of work*.** Un bloc est valide si la valeur hachée de ses données commence par d zéro successifs. Étant donné un bloc, < block\_to\_str > génère la chaîne de caractères correspondant au bloc en paramètres. Nous utilisons une fonction de hachage cryptographique *SHA-256*. Elle est standard. Elle prend en entrée des messages et produit leur valeur hachée. Étant donné une chaîne de caractères, < hash\_function\_SHA56 > produit cette valeur hachée. Étant donné un bloc et un entier, < compute\_proof\_of\_work > vérifie la validité d'un bloc. Analysons, de plus près cette fonction. Nous faisons varier la valeur de l'entier d :

Valeurs de d	Temps d'exécution (en secondes)
1	0,000478 0,000139 0,000156 0,000136 0,000149
2	0,012836 0,009406 0,008781 0,007516 0,006209
3	0,08121 0,028463 0,016072 0,016234 0,015468
4	2,15179 2,140249

	2,139908 2,140934 2,181039
5	6,624453 6,576402 6,551826 6,616244 6,666931
6	68.946701 70.493446 70.084885 69.555

Nous remarquons avec les temps moyens d'exécution selon la valeur de  $d$ , que plus il y a de zéros dans le hash, plus la vérification de la du bloc est longue. Cela est visible sur le graphe :



Le temps d'exécution dépasse une seconde à partir de  $d = 4$  sur la machine virtuelle Linux Ubuntu d'un MacOS M1.

**Nous supprimons des blocs.** Étant donné un bloc, `< delete_block >` supprime un bloc.

## Structure arborescente

**Objectif :** Éviter la triche

**Nous manipulons des arbres de blocs représentés par une structure CellTree.** Étant donné un bloc, `< create_node >` initialise un nœud avec une hauteur égale à 0. Étant donné un nœud-père et un nœud-fils, `< update_height >` met à jour la hauteur du nœud-père, si nécessaire, et renvoie 1, dans ce cas (et 0 sinon). Étant donné un nœud-père et un nœud-fils, `< add_child >` ajoute un fils à un nœud. Étant donné un arbre, `< print_tree >` affiche chaque nœud de l'arbre.

**Nous supprimons des arbres de blocs.** Étant donné un nœud, `< delete_node >` supprime un nœud d'un arbre. Étant donné arbre, `< delete_tree >` supprime un arbre.

**Nous déterminons le dernier bloc.** Pour éviter la triche, nous feront confiance à la chaîne de blocs, la plus longue, en partant de la racine de l'arbre. Étant donné un arbre, `< highest_child >` renvoie le nœud-fils avec la plus grande hauteur. Étant donné un arbre, `< last_node >` renvoie la valeur hachée du dernier bloc de la plus longue chaîne.

**Nous voulons extraire des déclarations de votes.** Pour déterminer le vainqueur d'une élection, il faut fusionner les déclarations signées de chaque bloc de la plus longue chaîne en une liste. Étant donné deux listes de déclarations signées, `< fusion_list_protected >` fusionne de liste de déclarations de votes. Étant donné un arbre, `< fusion_votes >` utilise la fonction précédente pour fusionner chaque liste de déclarations de la plus longue chaîne de l'arbre. En considérant la complexité comme le nombre de comparaison :

- Dans le meilleur des cas, la fonction ne réalisera aucun tour de boucles :  $\Omega(1)$ . En effet, si une des deux listes est NULL, nous renverrons directement la liste non NULL.
- Dans le pire des cas, elle en réalisera  $p : \partial(p)$ , pour  $p = \text{taille de la liste}$ .

Pour avoir une fusion en  $\partial(p)$ , il faudrait mettre un pointeur vers le dernier élément de la liste de déclarations de votes.

## Simulation du processus de votes

**Objectif :** Simuler une élection

**Nous souhaitons voter tout en créant des blocs valides.** Étant donné un arbre, une clé et un entier, `< create_block >` crée un bloc valide content  $d$  déclarations à partir d'un arbre et de la clé. Il met ce bloc dans un fichier .txt. Étant donné un entier et un nom de fichier, `< add_block >` crée un fichier représentant le bloc valide et l'ajoute dans un répertoire.

**Nous lisons un arbre et calculons le gagnant de l'élection.** `< read_tree() >` crée un arbre à partir de l'ensemble des fichiers contenus dans un répertoire prédéterminé. Il renvoie la tête de l'arbre. Étant donné un arbre, deux listes de clés et leur taille, `< compute_winner_BT >` renvoie le gagnant de l'élection.

## Conclusion

**Les élections traditionnelles sont complexes à organiser. Elles mobilisent énormément de personnel.** Il faut s'inscrire sur les listes électorales, aller en bureau de vote, avoir des agents pour comptabiliser les votes, etc. Cette solution est fastidieuse et manque de fiabilité. En effet, les élections sont organisées par le gouvernement en place et rien ne permet de déterminer si notre vote a bien été comptabilisé.

**Pour pallier cela, nous avons, dans un premier temps, chercher à réaliser une élection centralisée.** Dans ce système de vote, toutes les déclarations de vote sont collectées et gérées par une autorité régulatrice. C'est-à-dire, qu'une entité annexe vérifie et comptabilise les votes de chaque citoyen. Avec cette méthode, le vote est plus simple pour le votant. Cependant, nous n'avons toujours pas réglé notre problème de fiabilité.

**Pour ces questions de confiance, il est donc préférable d'utiliser des blockchains.** Mais pour quelle raison ? Les blockchains permettent à chaque citoyen de vérifier eux-mêmes le résultat du vote. De plus, ce système empêche les déclarations de votes frauduleuses car nous ne « faisons confiance » qu'aux plus longues chaînes de votes pour désigner le gagnant. Un vote s'ajoutant à la base de déclarations, in extremis, ne serait pas pris en compte car son ajout est considéré comme « étrange ». Le système de vote décentralisé par blockchain est donc fiable, sécurisé et ergonomique pour le votant.

**Nous pouvons donc conclure que les blockchains sont une meilleure alternative aux systèmes de vote traditionnels que les systèmes de votes centralisés.**