Projet : Blockchain appliquée à un processus électoral

BOUCHAAL Samia | 28610654

NOGUEIRA MENDES Magali | 28709071

On veut, à l’aide d’outils cryptographique, implémenter un système sécurisé permettant la désignation d’un vainqueur dans le cadre d’un élection.

Développement d’outils cryptographiques

On met d’abord en place des fonctions introduisant la cryptographie asymétrique : elle chiffre et déchiffre les messages à l’aide de deux clés, une clé publique, transmise à l’envoyeur et permettant de chiffrer le message, et une clé privée, permettant de déchiffrer les message. On utilise ici le protocole RSA.

1-Résolution du problème de primalité

On crée pour résoudre le problème deux fichiers primalite.c et primalite.h, le header.

Implémentation par une méthode naïve

1. On implémente une fonction int is_prime_naive(long p) de complexité O(p-3) dans le pire cas et en Ω(1) dans le meilleur cas pour tester la primalité d'un entier.

```
int is_prime_naive(long p) {
  /*
  Renvoie 1 si p est premier, 0 sinon
  */
    if(p%2 == 0 || p < 3) {
        printf("%ld n'est pas un nombre premier\n", p);
        return 0;
    }
    else{
        for(int i = 3; i < p; i++) {
            if(p%i == 0) {
                printf("%ld n'est un pas nombre premier\n", p);
                return 0;
            }
        }
        printf("%ld est un nombre premier\n", p);
        return 1;
    }
}
```

2. Le plus grand nombre premier que l'on arrive à tester à l'aide de cette fonction en moins de deux minutes est 262897.

Exponentiation modulaire rapide

On veut calculer a^m mod n sans passer par le calcul de la valeur de a^m. On implémente deux algorithme pour le calcul, une méthode naïve et une méthode récursive se basant sur l'élévation au carré.

3. La complexité de modpow_naive est en Θ(m).

```
long modpow_naive(long a, long m, long n) {
  /*
  Calcule a^m mod n par iteration avec la methode naive
  */
    long res = a;
    for(int i = 1; i < m; i++) {
        res = res * a;
    }
    res = res % n;
    return res;
}
```

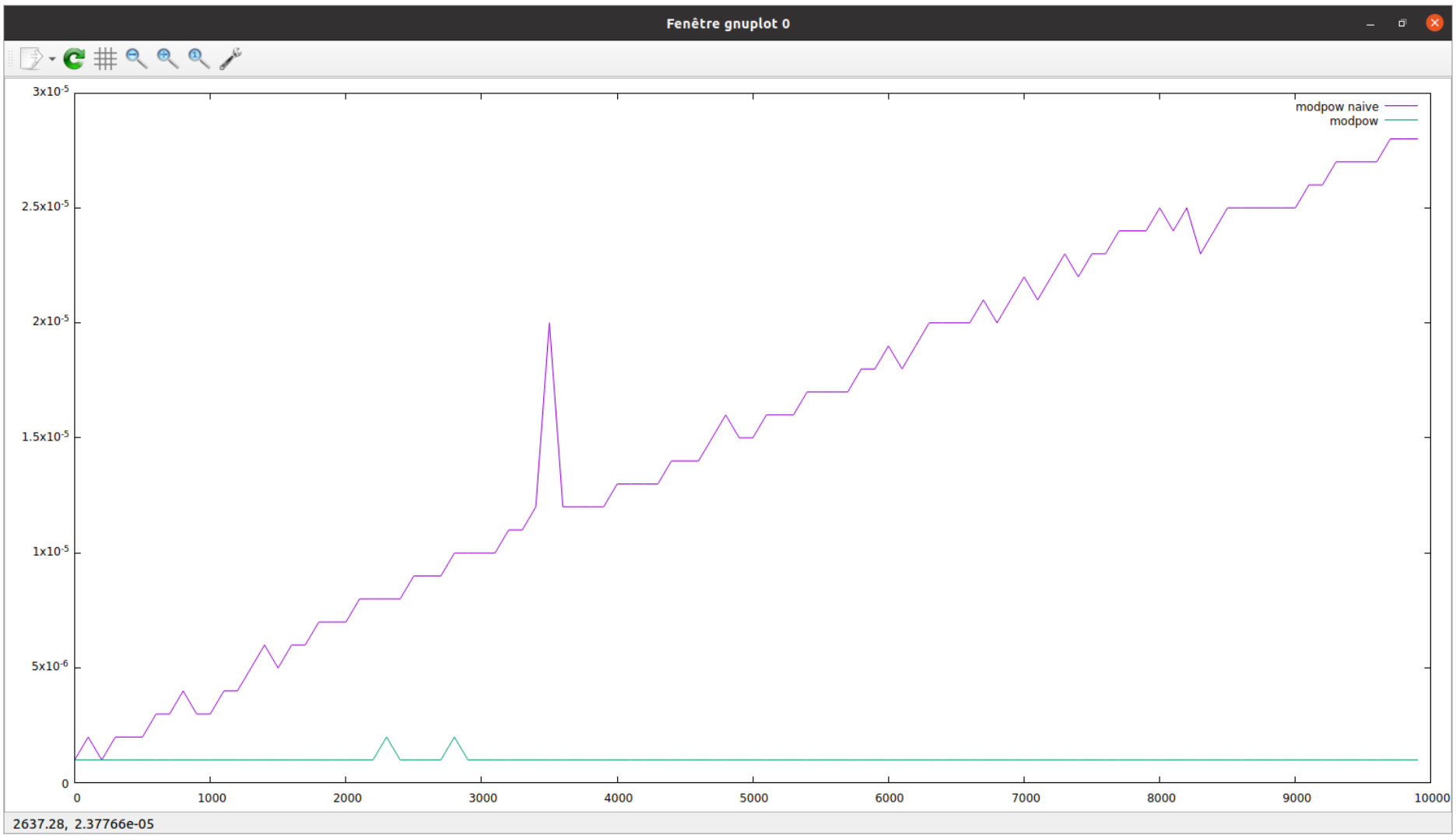
4.

```
long modpow(long a, long m, long n) {
  /*
  Calcule a^m mod n par recursion
  */
    a = a % n;
    if(m == 0) return 1;
    if(m == 1) return a;
    if(m%2 == 0) {
        long x = modpow(a, m/2, n);
        return (x * x) % n;
    } else {
        long x = modpow(a, m/2, n);
        return (((x * x) %n) *a) %n;
    }
}
```

5. On compare les performances des deux fonctions à l'aide du code suivant, contenu dans primalite_main.c :

```
int main(void) {
    // temps d'exécution is_prime_naive
    double temps_cpu;
    long p = 262897;
    int startTime = clock();

    is_prime_naive(p);
    int endTime = clock();
    temps_cpu = ((double) (endTime - startTime)) / CLOCKS_PER_SEC;
    printf("temps d'exécution: %f s\n", temps_cpu);

    FILE *f=fopen("Comparaison.txt","w");

    clock_t temps_initial,temps_final,temps_initial2,temps_final2;
    double  temps_cpu,temps_cpu2;

    if( (f) != NULL ) {
      for( long i = 0 ;i< m ; i+=100 ){
        //On prend le temps mis pour l'execution de mod_naive(a,i,n)
        temps_initial=clock();
        modpow_naive(a,i,n);
        temps_final=clock();

        //on convertit les "ticks consommes par modpow_naive en secondes"
        temps_cpu=((double)(temps_final-temps_initial))/CLOCKS_PER_SEC;

        //On prend le temps mis pour l'execution de modpow(a,i,n)
        temps_initial2=clock();
        modpow(a,i,n);
        temps_final2=clock();

        //on convertit les "ticks consommes par modpow en secondes"
        temps_cpu2=((double)(temps_final2-temps_initial2))/CLOCKS_PER_SEC;


        fprintf(f,"%ld %f %f \n",i,temps_cpu,temps_cpu2);
       }
    }else{
      printf("erreur\n");
    }
    fclose(f);

    dessiner_courbes();
    /* gnuplot
    set terminal png
    set output "courbes.png"
    plot "courbe.txt" using 1:2 title 'modpow_naive' with lines
    replot "courbe.txt" using 1:3 title 'modpow' with lines */

    return EXIT_SUCCESS;
}
```

On remarque que la courbe de modpow_naive est plutôt linéaire tandis que la courbe de modpow est constante.



Test de Miller-Rabin

On veut utiliser le test de primalité de Miller-Rabin dans la suite du projet. On implémente donc les fonctions suivantes données dans le sujet :

int witness(long a, long d, long p);

long rand_long(long low, long up);

int is_prime_miller(long p, int k);

7. La fonction is_prime_miller renvoie 1 si p est très probablement premier, en testant k fois si a est un témoin de Miller (si a l'est, la fonction renvoie 0). Or, si p non premier et au moins 3/4 des valeurs entre 2 et p-1 sont des témoins de Miller, il y'a une probabilité de 1/4 que a ne soit pas un témoin de Miller. Pour k tours de boucles, la probabilité s'élève à (1/4)^k.

Génération de nombres premiers

On veut générer des nombres premiers de grande taille de façon aléatoire entre deux entiers en utilisant le test de primalité précédent.

8. Voici l'algorithme créé pour faire cela :

```
long random_prime_number(int low_size, int up_size, int k) {
  /*
  Retourne un nombre premier de taille k entre low_size et up_size
  */
    long low = 2, up = 2;
    for(int i = 1; i < low_size-1 ; i++) {
        low *= 2;
    }
    for(int i = 1; i < up_size ; i++) {
        up *= 2;
    }
    up = up - 1;

    long p = rand_long(low, up);
    int i = 0;
    while(i != 1) {
        p = rand_long(low, up);
        i = is_prime_miller(p, k);
    }
    return p;
}
```

2-Implémentation du protocole RSA

Pour l'implémentation du protocole, on crée un fichier protocoleRSA.c munit d'un main (protocoleRSA_main.c) ainsi que son header.

Génération d'une paire (clé publique, clé secrète)

Pour générer un couple (clé secrète sKey=(u,n) , clé publique pKey=(s,n)), on génère à l'aides des fonctions de primalite.c, des nombres premiers p et q distincts et on réalise les opérations suivantes :

- 1. Calculer n=p×q et t=(p−1)×(q−1)
- 1. Générer aléatoirement des entiers s inférieur à t jusqu'à en trouver un tel que PGCD(s, t) = 1
- 1. Déterminer u tel que s×u mod t=1

Pour déterminer PGCD(s,t), on peut utiliser une version récursive de l'algorithme d'Euclide étendu donnée dans le sujet.

1. On créé alors une fonction permettant de générer sKey et pKey

```
void generate_key_values(long p, long q, long* n, long *s, long *u) {
    /*
    Permet de generer la cle publique pkey=(s,n) et la cle secrete skey=(u,n) a
    partir des nombres premiers p et q en suivant le protocole RSA
    */
    *n = p*q;
    int t = (p-1) * (q-1);
    long v;

    while(extended_gcd(*s, t, u, &v) != 1) {
      *s = rand_long(1, t);
    }
}
```

Chiffrement et déchiffrement de messages

Pour l'envoi d'un message m d'une personne A à une personne B, on chiffre m à l'aide de la clé publique de B en calculant le chiffrement c=m^s mod n. À la réception, le message initial est retrouvé avec la clé secrète de B en calculant m^u mod n. On utilise la fonction modpow de primalite.c pour coder le chiffrement et le déchiffrement.

2.

```
long* encrypt(char* chaine, long s, long n) {
  /*Chiffre la chaine de caractere avec la cle publique pKey=(s,n)
  */
    int size = (strlen(chaine));
    long *encrypted = (long*) malloc(sizeof(long)*size);
    if(encrypted==NULL){
      return NULL;
    }
    for(int i = 0; i < size; i++) {
        encrypted[i] = modpow((long)chaine[i], s, n);
    }
    return encrypted;
}
```

3.

```
char* decrypt(long* crypted, int size, long u, long n) {
  /*Dechiffre un message a l'aide de la cle secrete sKey=(u,n)
  */
    char* decrypted = (char*) malloc(sizeof(char)*(size+1));
    for(int i = 0; i < size; i++) {
        decrypted[i] = (char)modpow(crypted[i], u, n);
    }
    decrypted[size] = '\0';
    return decrypted;
}
```

On teste enfin notre protocole avec la fonction print_long_vector et le main donné dans le sujet.

Déclaration sécurisée

On veut simuler les déclarations de votes et de candidature des citoyens à l'aide du protocole RSA.

3-Manipulations de structures sécurisées

Ici, le chiffrement et le déchiffrement sont utilisés respectivement pour signer une déclaration et vérifier son authenticité avec une signature. On créé un fichier declarations.c ainsi qu'un main et son header afin de simuler ces déclarations.

Manipulation de clés

On définit une structure Key pour les clés et on les initialise en suivant le protocole avec init_pair_keys. La fonction key_to_str crée une représentation en chaîne de caractère de la clé donnée en paramètre et inversement, str_to_key, crée une clé à l'aide d'une chaîne de caractères.

```
typedef struct key_{
  long val;
  long n;
}Key;
```

```
void init_key(Key* key, long val, long n) {
  /*Initialise une cle deja allouee
  */
  key->val = val;
  key->n = n;
}

void init_pair_keys(Key* pKey, Key* sKey, long low_size, long up_size) {
    /*initialise les valeurs d'une structure Key en utilisant le protocole RSA */
  long p = random_prime_number(low_size, up_size, N);
  long q = random_prime_number(low_size, up_size, N);

  while(p == q) {
    q = random_prime_number(low_size, up_size, N);
  }

  long n, s, u;
  generate_key_values(p, q, &n, &s, &u);
  if(u < 0) {
    long t = (p - 1) * (q - 1);
    u += t;
  }
    init_key(sKey, u, n);
  init_key(pKey, s, n);
}

char* key_to_str(Key* key) {
    /* passe une variable de type Key à sa
   représentation sous forme de chaîne de caractères */
  char s[256];
    if(key == NULL) {
        return NULL;
    }
    sprintf(s, "(%lx, %lx)", key->val, key->n);
  return strdup(s);
}

Key* str_to_key(char* str) {
    /* passe d'une représentation sous forme de chaîne de
    caractères à sa variable de type Key */
    long val, n;
    Key* nKey = (Key*) malloc(sizeof(Key*));
    char buffer[256];
    for(int i = 0; i < strlen(str); i++) {
        buffer[i] = str[i];
    }

    buffer[strlen(str) + 1] = '\0';
    sscanf(buffer, "(%lx, %lx)", &val, &n);
    init_key(nKey, val, n);
    return nKey;
}
```

Signature

On implémente désormais une structure Signature qui correspond à un tableau de long et sa longueur afin d'attester l'authenticité d'une déclaration.

```
typedef struct signature_ {
    long* content;
    int size;
} Signature;
```

La fonction init_signature permet d'allouer et d'initialisé une signature avec un tableau de long alloué et initialisé.

La fonction sign crée une signature à partir du message et de la clé secrète de l'émetteur.

```
Signature* init_signature(long* content, int size) {
  /*Rempli une signature avec un tableau de long deja alloue
  */
    Signature* s = (Signature*) malloc(sizeof(Signature*));
    if(s == NULL) {
        return NULL;
    }
    s->content=content;
    s->size = size;

    return s;
}

Signature* sign(char* mess, Key* sKey) {
  /*Cree une signature a partir d'un message mess
  */
    Signature* s = init_signature(encrypt(mess, sKey->val, sKey->n), strlen(mess));
    return s;
}
```

On nous fournit dans le sujet 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.

Déclarations signées

On peut désormais créer des déclarations signées. On définit la structure Protected suivante:

```
typedef struct protected_ {
  Key* pKey;
  char* mess;
  Signature* sgn;
} Protected;
```

On crée les fonctions suivantes pour pouvoir manipuler ces déclarations signées:

```
Protected* init_protected(Key* pKey, char* mess, Signature* sgn) {
    /*Alloue et initialise une structure protected
    */
    Protected* pr = (Protected*) malloc(sizeof(Protected));
    if (pr == NULL) {
        printf("Erreur d'allocation !");
        exit(EXIT_FAILURE);
    }
    pr->pKey = pKey;
    pr->mess = strdup(mess);
    pr->sgn = sgn;
    return pr;
}

char* protected_to_str(Protected* p) {
    /*Passe d'une structure protected a sa representation sous forme
    de chaine de caractere*/
    char s[256]="";
    char* key = key_to_str(p->pKey);
    char* sgn = signature_to_str(p->sgn);
    sprintf(s,"(%s, %s, %s)", key, p->mess, sgn);
    free(key);
    free(sgn);
    return strdup(s);
```

```
}

Protected* str_to_protected(char* s) {
    /*Passe d'une chaine de caractere a une structure protected
    */
    char key_str[256];
    char mess_str[256];
    char sgn_str[256];
    Key* key;
    Signature* sgn;
    Protected* newPr;

    sscanf(s, "(%s, %s, %s)", key_str, mess_str, sgn_str);
    key = str_to_key(key_str);
    sgn = str_to_signature(sgn_str);
    newPr = init_protected(key, mess_str, sgn);
    return newPr;
}

int verify(Protected *pr) {
  /*Verifie que la siganture contenue dans une structure
   protected correspond bien a la personne et au message*/
  char* b = decrypt( pr->sgn->content, pr->sgn->size, pr->pKey->val, pr->pKey->n );
  int i = strcmp( pr->mess,b );
  free(b);
  if ( i == 0 ){
     return 1 ;
   }
  return 0;
}
```

Dans declarations_main.c, on s’inspire du main donné dans le sujet pour tester nos structures.

```
int main(void){

  srand(time(NULL));

  //Testing Init Keys
  Key* pKey = malloc(sizeof(Key));
  Key* sKey = malloc(sizeof(Key));
  init_pair_keys(pKey, sKey, 3 ,7);
  printf("Test Init Keys :\n");
  printf("pKey: (%lx,%lx)\n", pKey->val, pKey->n);
  printf("sKey: (%lx,%lx)\n", sKey->val, sKey->n);

  //Testing Key Serialization
  char* chaine = key_to_str(pKey);
  printf("\nTest key serialization:\n");
  printf("key_to_str: %s \n", chaine);
  Key* k = str_to_key(chaine);
  printf("str_to_key: (%lx,%lx)\n", k->val, k->n);

  //Testing Signature
  //Candidate Keys
  Key* pKeyC = malloc(sizeof(Key));
  Key* sKeyC = malloc(sizeof(Key));
  init_pair_keys(pKeyC, sKeyC, 3, 7);
  //Declaration
  char* tmp  = key_to_str(pKey) ;
  char* mess = key_to_str(pKeyC);
  printf("\nTest Signature\n");
  printf("%s vote pour %s\n", tmp, mess);
  Signature* sgn = sign(mess, sKey);
  printf("Signature: ");
  print_long_vector(sgn->content,sgn->size);


  //Testing protected
  char *key, *sign ;

  Protected* pr = init_protected(pKey, mess, sgn);
  //verif
  if( verify ( pr ) ) {
    printf ( "\nSignature valide\n" ) ;
  }else{
    printf ( "\nSignature non valide\n" ) ;
  }
  free(chaine);
  free(pr);
  chaine = protected_to_str ( pr ) ;
  printf ( "\nprotected_to_str : %s \n" , chaine ) ;
  pr = str_to_protected ( chaine ) ;
  key = key_to_str ( pr->pKey );
  sign = signature_to_str ( pr->sgn ) ;
```

```
  printf ( "str_to_protected: %s ,%s ,%s \n\n" , key ,pr->mess , sign ) ;

  generate_random_data(5, 2);

  free ( k ) ;
  free ( pr->mess );
  free ( pr->sgn->content );
  free ( pr->sgn );
  free ( pr->pKey );
  free ( sgn->content ) ;
  free ( pr ) ;
  free ( key ) ;
  free ( tmp ) ;
  free ( sgn ) ;
  free ( mess ) ;
  free ( sign ) ;
  free ( pKey ) ;
  free ( sKey ) ;
  free ( pKeyC ) ;
  free ( sKeyC ) ;
  free ( chaine ) ;
  return 0;
}
```

4-Création de données pour simuler le processus de vote

On génère des données pour simuler le scrutin : on crée avec la fonction generate_random_data, 3 fichiers, un fichier contenant les clés de tous les citoyens, un fichier indiquant les candidats et un fichier contenant les déclarations signées.

```
void generate_random_data(int nv, int nc) {
  /*Cree 3 fichiers, contenant respectivement les cles,
  les candidats et toutes les declarations signees*/
  int a;
  Key* pTable[nv];
  Key* sTable[nv];
  Key* candidates[nc];
  Key *pKey, *sKey;
  Signature *sgn;
  Protected* pr;
  char *b1, *b2;

  //Keys
  FILE *f1 = fopen("keys.txt", "w");
  if (f1 == NULL) {
        printf("Erreur d'ouverture fichier \n");
        exit(EXIT_FAILURE);
    }

  for(int i = 0; i < nv; i++) {
    pKey = (Key*) malloc(sizeof(Key));
    sKey = (Key*) malloc(sizeof(Key));

    if (!pKey || !sKey) {
      fprintf(stderr, "Erreur d'allocation\n");
      return;
    }
    init_pair_keys(pKey, sKey, 7, 10);
    pTable[i] = pKey;
  sTable[i] = sKey;
  b1 = key_to_str(pKey);
  b2 = key_to_str(sKey);
  fprintf(f1, "(%lx,%lx) (%lx,%lx)\n", pKey->val, pKey->n, sKey->val, sKey->n);
  free(b1);
    free(b2);
}
fclose(f1);

//Candidates
FILE *f2 = fopen("candidates.txt", "w");
if (f2 == NULL) {
  printf("Erreur d'ouverture fichier \n");
  exit(EXIT_FAILURE);
}

for(int i = 0; i < nc; i++) {
  a = (int) rand() % nv;
  candidates[i] = pTable[a];
  b1 = key_to_str(candidates[i]);
  fprintf(f2, "%s\n", b1);
  free(b1);
}
printf("here\n");
```

```
  fclose(f2);

  //Declarations
  FILE *f3 = fopen("declarations.txt","w");
  if (f3 == NULL) {
    printf("Erreur d'ouverture fichier \n");
    exit(EXIT_FAILURE);
  }

  for(int i = 0; i < nv; i++) {
    a = (int) rand() % nc;
    b1 = key_to_str(candidates[a]);
    sgn = sign(b1, sTable[i]);
    pr = init_protected(pTable[i], b1, sgn);
    b2 = protected_to_str(pr);
    fprintf(f3, "%s\n", b2);
    free(sgn->content);
    free(sgn);
    free(b1);
    free(b2);
    free(pr);
  }

  fclose(f3);
  for(int i = 0; i < nv; i++) {
    free(pTable[i]);
    free(sTable[i]);
  }
  return;
}
```

Base de déclarations centralisée

On s'intéresse maintenant à un système de votes centralisés : on utilisera tout d'abord la lecture et le stockage de données sous formes de liste chaînées (depuis les fichiers .txt candidates.txt, keys.txt et declarations.txt). On travaille à partir d'ici dans les fichiers decentralise.c, decentralise.h et decentralise_main.c.

5-Lecture et stockage des données dans des listes chaînées

Liste chaînée de clés

On utilise la structure suivante pour les listes chaînées de clés:

```
typedef struct cellKey {
    Key* data;
    struct cellKey* next;
} CellKey;
```

On implémente les fonctions suivante pour les manipuler:

```
CellKey* create_cell_key(Key* key) {
  /*Alloue et initialise une cellule de liste chainee
  */
    CellKey* c = (CellKey*) malloc(sizeof(CellKey*));
    struct cellKey* nxt = (struct cellKey*) malloc(sizeof(struct cellKey*));
    if(c == NULL) {
        return NULL;
    }
    c->data = key;
    c->next = nxt;
    return c;
}

CellKey* add_cell_key(CellKey* liste, Key* nKey) {
    /*Ajoute une cle en tete de liste
    */
    CellKey* new = create_cell_key(nKey);
    new->next = liste;
    return new;
}

CellKey* read_public_keys(char* file) {
    /*Retourne une liste chainee contenant toutes les cles publiques du fichier
```

```
    */
    FILE* f = fopen(file, "r");
    if(f == NULL) {
      printf("Erreur d'ouverture fichier \n");
      return NULL;
    }
    CellKey *ck = NULL;
    char buffer[256];
    char s[100];
    Key *key ;

    while(fgets(buffer,256,f) != NULL) {
      sscanf(buffer,"%s ",s);
      key = str_to_key(buffer);
      add_cell_key(ck,key);
    }
    fclose(f);
    return ck;
}

void print_list_keys(CellKey* LCK) {
  /*Permet d'afficher une liste chainee de cles
  */
    if(LCK == NULL) {
        printf("la liste est vide");
        exit(EXIT_FAILURE);
    }
    CellKey* courant = LCK;
    while(courant != NULL) {
        char* printKey = key_to_str(courant->data);
        printf("%p - %s\n", courant, printKey);
        courant = courant->next;
    }
}

void delete_cell_key(CellKey* c) {
  /*Supprime une cellule de liste chainee de cles
  */
    if(c == NULL) {
        printf("la liste est vide");
        exit(EXIT_FAILURE);
    }
    free(c->data);
    c->data = NULL;
    c = c->next;
}

void delete_list_keys(CellKey* liste) {
  /*Supprime une liste chainee de cles
  */
    if(liste == NULL) {
        printf("la liste est vide");
        exit(EXIT_FAILURE);
    }
    CellKey* courant = liste;
    while(courant != NULL) {
        delete_cell_key(liste);
        courant = courant->next;
    }
}
```

Liste chaînée de déclarations signées

On utilise la structure suivante pour les listes chaînées de déclarations signées:

```
typedef struct cellProtected {
    Protected* data;
    struct cellProtected* next;
} CellProtected;
```

De même, on les manipule à l’aide des fonctions suivantes :

```
CellProtected* create_cell_protected(Protected* pr) {
  /*Alloue et initialise une cellule de liste chainee de protected
  */
    CellProtected* p = (CellProtected*) malloc(sizeof(CellProtected*));
    struct cellProtected* nxt = (struct cellProtected*) malloc(sizeof(struct cellProtected*));
    if(p == NULL) {
        return NULL;
    }
    p->data = pr;
```

```
    p->next = nxt;
    return p;
}

void add_cell_protected(CellProtected** cp, Protected* nPr) {
  /*Ajoute une declaration signee en tete de liste
  */
  CellProtected *new = create_cell_protected(nPr);
  new->next = *cp;
  *cp = new;
}

CellProtected* read_protected(char* file) {
  /*Lit le fichier file et cree une liste de toutes les declarations signees du fichier
    */
    FILE *f = fopen(file, "r");
    if (f == NULL) {
        printf("Erreur ouverture fichier\n");
        exit(EXIT_FAILURE);
    }
    CellProtected *cp = NULL;
    char buffer[256];
    Protected *pr;

    while (fgets(buffer, 256, f)) {
       pr = str_to_protected(buffer);
       add_cell_protected(&cp,pr);
    }
    fclose(f);
    return cp;
}

void print_list_protected(CellProtected* cp){
  /*Fonction d'affichage d'une liste de declarations signees
  */
  CellProtected *tmp = cp;
  if (!cp){ //si liste vide
      printf("liste vide !\n");
  }
  while (tmp) { //on parcours et on affiche
    char *str_pr = protected_to_str(tmp->data);
    printf("%s\n", str_pr);
    tmp = tmp->next;
    free(str_pr);
  }
}

void delete_cell_protected(CellProtected* p) {
  /*Supprime une cellule de liste chainee de protected
  */
    if(p == NULL) {
        printf("la liste est vide");
        exit(EXIT_FAILURE);
    }
    free(p->data);
    p = p->next;
}

void delete_list_protected(CellProtected* liste) {
  /*Supprime une liste chainee de declaratiosn signees
  */
    if(liste == NULL) {
        printf("la liste est vide");
        exit(EXIT_FAILURE);
    }
    CellProtected* courant = liste;
    while(courant != NULL) {
        delete_cell_protected(liste);
        courant = courant->next;
    }
}
```

6-Détermination du gagnant de l’élection

Une fois les données stockées dans les listes, on peut les vérifier et éliminer toute tentative de fraude:

```
void fraude(CellProtected** d) {
    /*Supprime toute déclaration frauduleuse*/
    if(d == NULL) {
      return;
    }
```

```
    else {
      CellProtected* tmp = (CellProtected*) malloc(sizeof(CellProtected*));
      tmp = *d;
      while(*d) {
          if(!verify((*d)->data)) {
            *d = (*d)->next;
            delete_cell_protected(*d);
          }
      }
      CellProtected* suiv = (*d)->next;
      while(suiv) {
          if(!verify(suiv->data)){
            suiv = suiv->next;
            delete_cell_protected(suiv->next);
            tmp->next = suiv;
          }
          else {
            tmp = tmp->next;
            suiv = suiv->next;
          }
      }
  }
}
```

Pour déterminer le gagnant, on utilisera deux tables de hachage, la première comptant le nombres de votes et vérifiant si le candidat est bien dans la liste et le deuxième, vérifiant si les votes sont uniques et soumis par des votants inscrits sur la liste électorale. Pour cela, on utilise ces structures:

```
typedef struct hashcell{
  Key* key;
  int val;
} HashCell;

typedef struct hashtable{
  HashCell** tab;
  int size;
} HashTable;
```

Les fonctions permettant de se servir des tables de hachage et de calculer le vainqueur sont les suivantes :

```
HashCell* create_hashcell(Key* key) {
    /*Alloue et initialise une cellule de la table de hachage
    */
    HashCell *h = (HashCell*)malloc(sizeof(HashCell));
    if(h == NULL) {
      printf("Erreur d'allocation");
      exit(EXIT_FAILURE);
    }
    h->key = key;
    h->val = 0;
    return h;
}

int hash_function(Key* key, int size) {
    /*Retourne la positon d'un element de la table de hachage
    */
    return key->val % size;
}

int find_position(HashTable* t, Key* key) {
  if(!key){printf("key NULL\n");return -1;}
  if(!t){printf("table NULL\n");return -1;}

  // on calcule f(val)
  unsigned int pos = hash_function(key, t->size);
  if (t->tab[pos] == NULL ||( t->tab[pos]->key->val==key->val && t->tab[pos]->key->n==key->n))  {
    // si la position est libre ou si la clé se situe déjà à f(val) on renvoie f(val)

    return pos;
  }

  unsigned int pos2;
  for(int i = 1; i<(t->size);i++){
    pos2 =(pos+i)%(t->size);
    // on parcours les cases suivantes jusqu'à trouver une case libre ou si la clé
    //est déjà présente
    if( t->tab[pos2] == NULL){
      //printf("Clé non présente dans le tableau\n");
      return pos2;
    }
```

```
    if(t->tab[pos2]->key==key){
      return pos2;
    }
  }

  return pos;
}

HashTable* create_hashtable(CellKey* keys, int size) {
    /*Cree et initialise une table de hachage de taille size*/
    HashTable *hash = (HashTable*) malloc(sizeof(HashTable));
    hash->tab = (HashCell**) malloc(sizeof(HashCell*) * size);
    if (hash->tab == NULL) {
      printf("Erreur d'allocation !");
      exit(EXIT_FAILURE);
    }
    hash->size = size ;
    CellKey *tmp = keys;
    for (int i = 0; i < hash->size; i++) {
      hash->tab[i] = NULL;
    }

    while (tmp) {
      int pos =  find_position(hash, tmp->data);
      hash->tab[pos] = create_hashcell(tmp->data);
      tmp = tmp->next;
    }
    return hash;
}

void delete_hashtable(HashTable* h) {
  /*Supprime une table de hachage*/
    for(int i = 0; i < h->size; i++){
      free(h->tab[i]);
    }
    free(h->tab);
    free(h);
}

Key* compute_winner(CellProtected* decl, CellKey* candidates, CellKey* voters, int sizeC, int sizeV) {
  /*Calcule le vainqueur de l'election etant donnes une liste de declarations
  avec signatures valides, une liste de candidats, et une liste de personnes autorisees a
  voter*/
  HashTable *hv = create_hashtable(voters, sizeV);
  HashTable *hc = create_hashtable(candidates, sizeC);

  Key *keyV=(Key*)malloc(sizeof(Key*));
  Key *keyC=(Key*)malloc(sizeof(Key*));
  int posV, posC;

  // On parcours les déclarations
  while (decl != NULL) {
    keyV = (decl->data)->pKey;
    posV = find_position(hv, keyV);

    // On vérifie que les électeurs existent et qu'ils n'ont pas votés
    if (((hv->tab)[posV] != NULL) && (((hv->tab)[posV])->val == 0)) {
      keyC = str_to_key((decl->data)->mess);
      posC = find_position(hc, keyC);

      // on vérifie que le candidat existe
      if ((hc->tab)[posC] != NULL) {
        // On ajoute un vote et on incrémente la valeur de l'électeur pour qu'il ne vote pas de nouveau
        ((hc->tab)[posC])->val += 1;
        ((hv->tab)[posV])->val = 1;
      }

      free(keyC);
    }

    decl = decl->next;
  }


  // on cherche le gagnant
  int max = 0;
  int position;
  Key *keyTmp=(Key*)malloc(sizeof(Key*));
  Key *keygagnant=(Key*)malloc(sizeof(Key*));

  while (candidates != NULL) {
    // on calcule la position des candidats dans la table
    position = find_position(hc, candidates->data);

    // si le candidat à plus de vote que celui en tête précedemment
  if(hc->tab[position]){
    if (((hc->tab)[position])->val > max) {
```

```
      max = ((hc->tab)[position])->val;
      keyTmp = ((hc->tab)[position])->key;
      init_key(keygagnant, keyTmp->val, keyTmp->n);
    }
  }

  candidates = candidates->next;
  }

  delete_hashtable(hv);
  delete_hashtable(hc);

  return keygagnant;
}
```

On teste nos fonctions dans decentralise_main.c:

```
int NOMBRE_CANDIDATS=5;
int NOMBRE_DECLARATIONS=1000;

int main(void) {
  generate_random_data(5,2);
    CellKey* listeVoters = read_public_keys("keys.txt");
  printf("fichiers ouverts1\n");
  CellKey* listeCandidates = read_public_keys("candidates.txt")  ;
  printf("fichiers ouverts2\n");
  CellProtected* listeDeclaration = read_protected("declarations.txt") ;
  printf("fichiers ouverts3\n");

  Key* winner = compute_winner(listeDeclaration, listeCandidates, listeVoters, NOMBRE_CANDIDATS, NOMBRE_DECLARATIONS);
  printf("compute_winner OK\n");

  char* keys = key_to_str(winner);
  printf("L'élu est %s\n", keys);

  free(keys);
  //printf("delete 1\n");
  delete_cell_key(listeVoters);
  //printf("delete2\n");
  delete_cell_key(listeCandidates);
  //printf("delete3\n");
  /*
   * delete_celles_protected creee rarement des erreurs de segmentations :
   * peut etre on a free un element du protected dans hv quand on les supprime apres avoir vote
   * solution copier sur je sais plus qui qui modifie juste la valeur de val et les supprimes pas
   * peut etre que c est la key audessus
   * sinon chez pas
   */
  delete_cell_protected(listeDeclaration);
    return 0 ;
}
```

Blocs et persistance de données

On arrive finalement à la partie principale du projet : la création d’un système électoral décentralisé utilisant un blockchain. Un blockchain est une base de donnée décentralisée et sécurisé. Chaque citoyen possède une copie de la base, et cette base est mise à jour avec chaque vote. La fraude est rendue difficile avec une fonction de hachage (mécanisme de consensus dit *proof of work*) et donc par la demande de la valeur hachée de la donnée data du bloc précédent à chaque vote. Dans cette partie, on écrit dans le fichier block.c, ainsi que son header block.h.

7-Structure d’un block et persistance

On s’intéresse dans cet exercice à la gestion de blocs:

```
typedef struct block{
    Key* author;
    CellProtected* votes;
    unsigned char* hash;
    unsigned char* previous_hash;
    int nonce;
}Block;
```

Lecture et écriture de blocs

```
void write_block(char *nomfic, Block* block){
  /*Permet d'ecrire un bloc dans un fichier*/
  FILE *f = fopen(nomfic, "w+");
  if (f == NULL) {
    printf("ERREUR D'OUVERTURE DE FICHIER");
    exit(EXIT_FAILURE);
  }
  char *strkey = key_to_str(block->author);
  fprintf(f, "%s\t%s\t%s\t%d\n", strkey,block->hash,block->previous_hash, block->nonce);
  free(strkey);
  CellProtected *tmp = block->votes;
  while (tmp) {
    char *strprotected = protected_to_str(tmp->data);
    fprintf(f, "%s\n", strprotected);
    free(strprotected);
    tmp = tmp->next;
  }
  fclose(f);
}

Block* read_block(char* nomfic){
  /*Permet de lire un bloc depuis un fichier*/
    FILE* f=fopen(nomfic,"r");
    if (!f){
        printf("erreur d'ouverture du fichier %s\n", nomfic);
        return NULL;
    }
    Block* b=(Block*) malloc (sizeof(Block));
    if(!b){
        printf("erreur d'allocation du block\n");
        fclose(f);
        return NULL;
    }
    char buf[256];
    fgets(buf, 256, f);
    char tmp[256], tmp1[256], tmp2[256], tmp3[256];
    if(sscanf(buf, "%s", tmp) == 1) {
        b->author=str_to_key(tmp);
    }
    fgets(buf, 256, f);
    if(sscanf(buf, "%s", tmp) == 1) {
        printf("hash : %s\n", tmp);
        b->hash=(unsigned char*)strdup(tmp);
        printf("copy : %s\n", b->hash);
    }
    fgets(buf, 256, f);
    if(sscanf(buf, "%s", tmp) == 1) {
        b->previous_hash=(unsigned char*)strdup(tmp);
    }
    int i;
    fgets(buf, 256, f);
    if(sscanf(buf, "%d", &i) == 1) {
        b->nonce=i;
    }
    CellProtected* declarations=(CellProtected*) malloc (sizeof(CellProtected));
    while(fgets(buf, 256, f)){
        if (sscanf(buf, "%s %s %s", tmp1, tmp2, tmp3)==3){
            Protected* pr=str_to_protected(buf);
            add_cell_protected(&declarations, pr);
        }
        else{
            printf("erreur de lecture\n");
            fclose(f);
            delete_list_protected(declarations);
            free(b->hash);
            free(b->previous_hash);
            free(b->author);
            free(b);
        }
    }
    b->votes=declarations;
    fclose(f);
    return b;
}
```

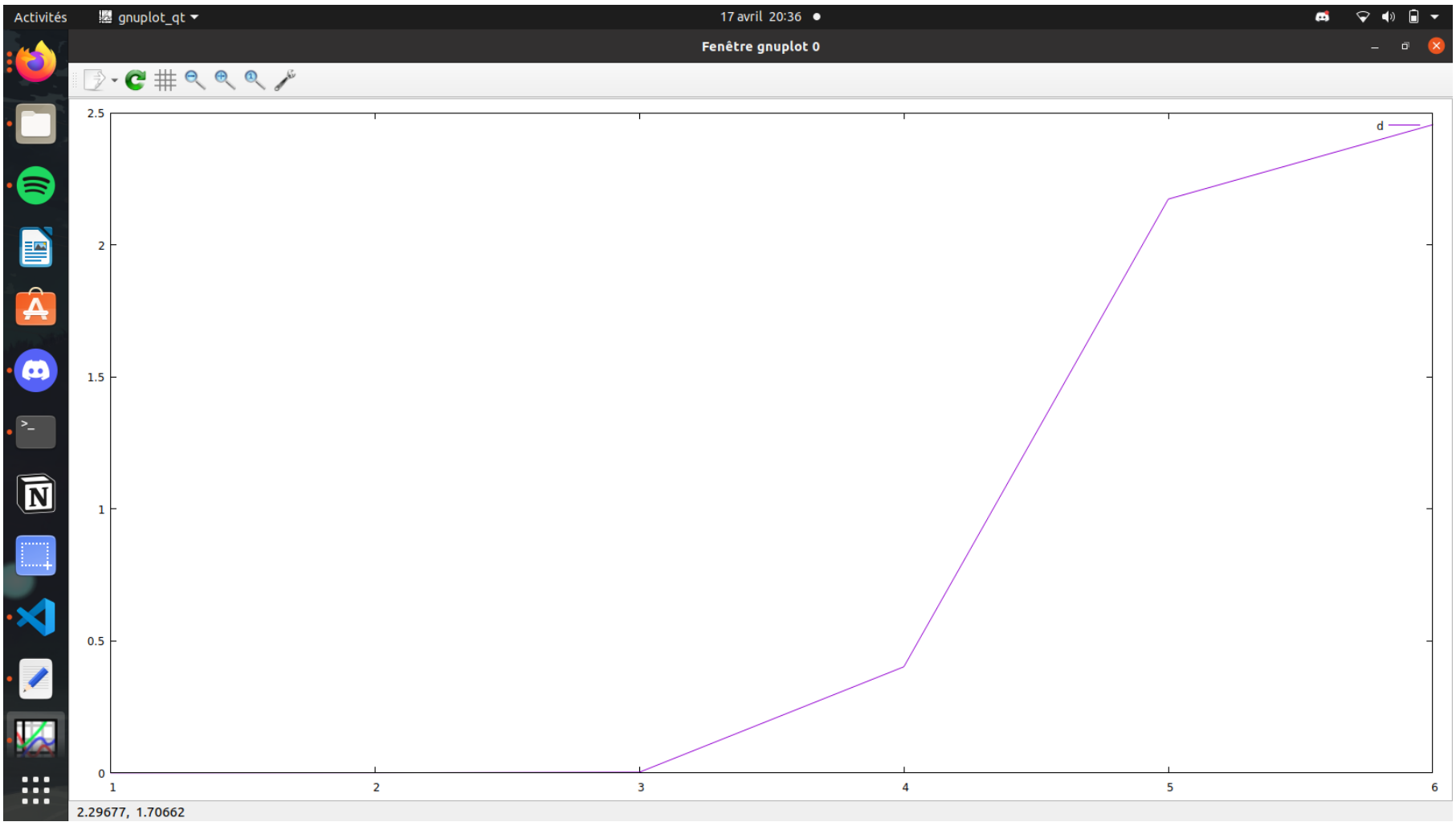
Création de blocs valides

Pour créer des blocks, on utilise le *proof of work* afin qu’ils soient valides.

Les fonctions créées qui permettent de manipuler et de vérifier ces blocks sont les suivantes:

```
char* block_to_str(Block* block){
  /*Genere une chaine de caracteres representant un bloc*/
  char str[11];
  char *auteur =key_to_str(block->author);
  char *finalchar=(char*)calloc(sizeof(char),10000);
  if(!finalchar) printf("Erreur d'allocation \n");
  finalchar[0] = '\0';
  strcat(finalchar,auteur);
  strcat(finalchar,"\n");

  if(block->previous_hash){
  strcat(finalchar,block->previous_hash);
  }
  else{
    strcat(finalchar,"(null)");
  }

  strcat(finalchar,"\n");
  CellProtected * tmp = block->votes;
  char *votes;

  while(tmp){
    votes=protected_to_str(tmp->data);
    strcat(finalchar,votes);
    strcat(finalchar,"\n");
    free(votes);
    tmp=tmp->next;

  }

  sprintf(str, "%d", block->nonce);
  strcat(finalchar,str);
  strcat(finalchar,"\n");

  free(auteur);

  return finalchar;

}

unsigned char* hash_char(char* chaine){
  /*Retourne la valeur hachee d'une chaine de caracteres obtenur par l'lgorithme SHA256 */
    return strdup(SHA256(chaine, strlen(chaine), 0));
}

void compute_proof_of_work(Block *B, int d){
  /*Rend un bloc valide*/
    B->nonce = 0;
    while(1){
        char* blockstr = block_to_str(B);
        unsigned char* hashed = hash_char(blockstr);
        int i=0;
        while(i < d){
            if(hashed[i]=='0'){
                i++;
            }
            else{
                (B->nonce)++;
                break;
            }
        }
        if (i==d) break;
    }
}//Courbe du temps

int verify_block(Block* B, int d){
  /*Verifie qu'un bloc est valide*/
    int i=0;
    while (i<d){
        if (B->hash[i]!='0'){
            return 0;
        }
        i++;
    }
    return 1;
}
```

8. En étudiant le temps moyen de compute_proof_of_work, on obtient la courbe suivante:



On en déduit que la valeur de d à partir de laquelle ce temps dépasse une seconde est 3 .

```
void delete_block(Block* b){
  /*Supprime un bloc sans liberer la memoire associee au champs author*/
    free(b->hash);
    free(b->previous_hash);
    CellProtected *tmp;
    while(b->votes){
        tmp = b->votes->next;
        free(b->votes);
        b->votes = tmp;
    }
    free(b);
}

void free_block(Block *block) {
    delete_block(block);
    free(block);
```

8-Structure arborescente

Dans le cas où plusieurs blocs contient la valeur hachée d’un même bloc, on obtient une structure arborescente. On doit alors faire confiance à la chaîne la plus longue à partir de la racine de l’arbre.

Manipulation d’un arbre de blocs

La représentation d’un arbre de blocs se fait avec la structure suivante:

```
typedef struct block_tree_cell{
  Block* block;
  struct block_tree_cell* father;
  struct block_tree_cell* firstChild;
  struct block_tree_cell* nextBro;
  int height;
}CellTree;
```

On peut représenter un arbre avec un nombre arbitraire de fils et gérer cette structure avec les fonctions :

```
CellTree* create_node(Block* b){
  /*Permet de creer et d'initialiser un noeud*/
  CellTree *abr=(CellTree*)malloc(sizeof(CellTree));
  if(abr==NULL){
```

```
    printf("Erreur d'allocation \n");
    exit(EXIT_FAILURE);
  }
  abr->block=b;
  abr->father=NULL;
  abr->firstChild=NULL;
  abr->nextBro=NULL;
  abr->height=0;
  return abr;
}

int update_height(CellTree* father, CellTree* child){
  /*Permet de mettre a jour la hauteur du noeud father quand l'un de ses fils a ete modifie*/
  if(father->height==maxf(father->height,child->height+1)){
    return 0;
  }
  return 1;
}

void add_child(CellTree* father, CellTree* child){
  /*Ajoute un fils a un noeud, en mettant a jour la hauteur de tous les ascendants*/
  child->nextBro=father->firstChild;
  father->firstChild=child;
  child->father=father;
  CellTree* pere=father;
  CellTree* enfant=child;
  while(pere){
    if(update_height(pere,enfant)){
      pere->height=enfant->height+1;
    }
    enfant=pere;
    pere=enfant->father;
  }
}

void print_tree(CellTree* abr){
  /*Affiche un arbre*/
  if(abr==NULL){
    return;
  }
  else{
    char* blockstr = block_to_str(abr->block);
    printf("====================================\n");
    printf("bloc: %s, hauteur: %d\n", blockstr, abr->height);
    printf("====================================\n");
    free(blockstr);
    if(abr->firstChild){
      print_tree(abr->firstChild);
    }
    if(abr->nextBro){
      print_tree(abr->nextBro);
    }
  }
}

void delete_node(CellTree* node){
  /*Supprime un noeud*/
  free_block(node->block);
  free(node);
}

void delete_tree(CellTree* abr){
  /*Supprime un arbre*/
  if(abr==NULL){
    return;
  }
  delete_tree(abr->firstChild);
  delete_tree(abr->nextBro);
  delete_node(abr);
}
```

Détermination du dernier bloc

On veut déterminer efficacement le dernier bloc de la chaîne la plus longue afin de créer de nouveaux blocs.

```
CellTree* highest_child(CellTree* abr){
  /*Renvoie le noeud fils avec la plus grande hauteur*/
  CellTree* child=abr->firstChild;
  CellTree *highest=child;
  int h=child->height;
  while(child){
    if(child->height>h){
      h=child->height;
```

```
      highest=child;
    }
    child=child->nextBro;
  }
  return highest;
}

CellTree* last_node(CellTree* abr){
  /*Permet de retourner le dernier bloc de cette plus longue chaine*/
  if(abr->height>0){
    return last_node(highest_child(abr));
  }
  return abr;
}
```

Extraction des déclarations de vote

Chaque bloc contient une liste chaînée de déclarations signées (champs votes). Il suffit de fusionner ces listes et d'appeler compute_winner.

8. Pour obtenir une liste chaînée de déclarations signées de longueur n telle quelle, la complexite de fusion est en O(n). Pour avoir une complexité en O(1), il faudrait changer notre structure en liste circulaire doublement chaînée.

```
CellProtected* fusion(CellProtected* fus,CellProtected* pr1, CellProtected* pr2){
  /*Fusionne deux listes chainees de declarations signees*/
  //CellProtected *fus=(CellProtected*)malloc(sizeof(CellProtected*));
  //CellProtected *pr=(CellProtected*)malloc(sizeof(CellProtected*));
  CellProtected* pr=pr1;
  if(fus->data==NULL && pr){
    fus=create_cell_protected(pr->data);
    pr=pr->next;
  }
  while(pr){
    add_cell_protected(&fus,pr->data);
    pr=pr->next;
  }
  pr=pr2;
  if(fus->data==NULL && pr){
    fus=create_cell_protected(pr->data);
    pr=pr->next;
  }
  while(pr){
    add_cell_protected(&fus,pr->data);
    pr=pr->next;
  }
  return fus;
}//Complexite et modifs*

CellProtected *max_chaines(CellTree* abr,CellProtected *votes){
  /*Retourne la liste obtenue par fusion des listes chainees de declarations
  contenues dans les blocs de la plus longue chaine*/
  if(abr->height>0){
    CellTree* highest=highest_child(abr);
    CellProtected* decla=fusion(votes,highest->block->votes,abr->block->votes);
    return max_chaines(highest,decla);
  }
  return votes;
}
```

9-Simulation du processus de vote

On simule le fonctionnement d'une blockchain en utilisant le répertoire Blockchain, représentant la blockchain qu'un citoyen a construit en local et les fichiers Pending_block, fichier contenant le dernier bloc créé et Pending_votes.txt, contenant les votes en attente.

Vote et création de blocs valides

On crée des fonctions permettant la soumission de votes avec tous les mécanismes présentés depuis le début du projet :

```
void submit_vote(Protected *p){
  /*Permet à un citoyen de soumettre un vote*/
  FILE *f=fopen("Pending_votes.txt","a");
  char* s=protected_to_str(p);
  fprintf(f,"%s\n",s);
  free(s);
  fclose(f);
}
```

```
void create_block(CellTree* tree, Key* author, int d){
  /*cree un block valide contenant les votes en attente et ecrit le bloc obtenu dans un fichier 'Pending_block'*/
  Block *block=(Block*)malloc(sizeof(Block*));
  block->votes=read_protected("Pending_votes.txt");
  block->author=author;
  block->hash=hash_char("");

  CellTree *lastnode=last_node(tree);
  block->previous_hash=strdup(lastnode->block->hash);
  compute_proof_of_work(block,d);

  CellTree *child=create_node(block);
  add_child(lastnode,child);
  remove("Pending_votes.txt");
  write_block("Pending_block.txt",block);
}

void add_block(int d, char* name){
  /*Verifie que le bloc representé par le fichier 'Pending_block' est valide*/
  Block *block = read_block("Pending_block.txt");
  printf("=== Dans add_block ===\n");
  printf("hash = %s, previous hash = %s\n", block->hash, block->previous_hash);
  if (verify_block(block, d)) {
    FILE *f = fopen(name, "w");
    write_block(name, block);
    char rep[50] = "Blockchain/";
    rename(name, strcat(rep, name));
    fclose(f);
  }
  free_block(block);
}
```

Lecture de l’arbre et calcul du gagnant

On va maintenant implémenter une fonction qui crée l’arbre de blocs contenus dans le répertoire Blockchain et une fonction calculant le vainqueur à l’aide des déclarations de la blockchain :

```
CellTree* read_tree(int nv, int nc){
  /*Construit l'arbre correspondant aux blocs contenus dans le répertoire
  'Blockchain'*/
  CellTree **T = (CellTree**)malloc(sizeof(CellTree*) * (nv + nc));
    DIR *rep = opendir("./Blockchain/");
    int i = 0;
    if (rep != NULL) {
      struct dirent *dir;
      while ((dir = readdir(rep))) {
        if (strcmp(dir->d_name, ".") != 0 && strcmp(dir->d_name, "..") != 0) {
          printf("Chemin du fichier : ./Blockchain/%s\n", dir->d_name);
          char dir_name[50] = "./Blockchain/";
          Block *block = read_block(strcat(dir_name, dir->d_name));
          CellTree *ct = create_node(block);
          T[i] = ct;
          i++;
        }
      }
      closedir(rep);
    }
    int size = i;
    for (int i = 0; i < size; i++) {
      for (int j = 0; j < size; j++) {
        if (T[j]->block->previous_hash == T[i]->block->hash) {
          add_child(T[i], T[j]);
        }
      }
    }
    for (int i = 0; i < size; i++) {
      if (T[i]->father == NULL) {
        CellTree *racine = T[i];
        free(T);
        return racine;
      }
    }
  return NULL;
}

Key* compute_winner_BT(CellTree* tree, CellKey* candidates, CellKey* voters, int sizeC, int sizeV){
  /*Determine le gagnant de l'election en se basant sur la plus longue chaine de
  l'arbre*/
  CellProtected *votes = (CellProtected *)malloc(sizeof(Protected));
  CellProtected *decl = max_chaines(tree, votes);
  fraude(&votes);
  Key *gkey = compute_winner(decl, candidates, voters, sizeC, sizeV);
```

```
  free(votes);
  return gkey;
}
```

Enfin, notre simulation de processus électoral est complète. On la met en application dans block_main.c:

```
int main() {
    const char *s = "Rosetta code";
    unsigned char *d = SHA256(s, strlen(s),0);
    int i;
    for(i = 0; i < SHA256_DIGEST_LENGTH ; i++) {
        printf("%02x", d[i]);
    }
    putchar('\n') ;

  unsigned char* a = hash_char("Rosetta code");

  for (i = 0; i < SHA256_DIGEST_LENGTH; i ++)
  printf("%02x", a[i]);
  putchar('\n');
  free(a);

  Block *block = read_block("block.txt");
  write_block("Block.txt", block);
  char *blockstr = block_to_str(block);
  printf("block_to_str: \n%s\n", blockstr);
  free(blockstr);
  compute_proof_of_work(block, 3);
  printf("%d\n", verify_block(block, 3));
  //plot "proof_of_work.txt" using 1:2 title 'd' with lines
  delete_block(block);

    return 0;
}
```

Conclusion

Pour conclure notre rapport, nous aimerions mettre en avant le fait que nous avons eu du mal à résoudre quelques Segmentation Fault et à créer des mains et jeux de tests efficace et pertinents. Nous avons aussi notamment des warnings dans block.c que nous n’avons pas pu enlever par manque de temps. Cependant, dans l’ensemble, malgré le fait que l’on a eu du mal à gérer notre temps, nous avons bien compris le sujet et les enjeux du projet.

Utilisation du blockchain dans le cadre d’une élection

L’utilisation d’une blockchain dans le cadre d’un processus de vote n’est pas infallible même si elle possède des avantages: les sources de fraude peuvent venir de chaînes pas assez longues, ou de variations et de diminutions dans la quantité de vote à un temps donné, notamment vers les dernières minutes.

Elle est très efficace dans le cadre de la cryptomonnaie, où les transactions de font de manières constante et où le flux ne s’arrête pas, mais ici, il y’a trop de failles à exploiter.

Makefile

```
CFLAGS = -Wall
CC = gcc

PROGRAMS = run_signature_main run_main run_rsa_main run_decentralise_main run_block_main

all: $(PROGRAMS)

run_block_main: protocoleRSA.o primalite.o declarations.o decentralise.o block.o block_main.o
$(CC) -o $@ $(CFLAGS) $^ -lm

run_decentralise_main: protocoleRSA.o primalite.o declarations.o decentralise.o decentralise_main.o
$(CC) -o $@ $(CFLAGS) $^ -lm

run_signature_main: protocoleRSA.o primalite.o declarations.o declarations_main.o
$(CC) -o $@ $(CFLAGS) $^ -lm

run_rsa_main: protocoleRSA.o primalite.o protocoleRSA_main.o
$(CC) -o $@ $(CFLAGS) $^ -lm
```

```
run_main: primalite.o primalite_main.o
$(CC) -o $@ $(CFLAGS) $^ -lm

primalite_main.o: primalite_main.c
$(CC) -c $(CFLAGS) primalite_main.c

protocoleRSA_main.o: protocoleRSA_main.c
$(CC) -c $(CFLAGS) protocoleRSA_main.c

declarations_main.o: declarations_main.c
$(CC) -c $(CFLAGS) declarations_main.c

decentralise_main.o: decentralise_main.c
$(CC) -c $(CFLAGS) decentralise_main.c

block_main.o: block_main.c
$(CC) -c $(CFLAGS) block_main.c

procotoleRSA.o: protocoleRSA.c
$(CC) -c $(CFLAGS) protocoleRSA.c

primalite.o: primalite.c
$(CC) -c $(CFLAGS) primalite.c

declarations.o: declarations.c
$(CC) -c $(CFLAGS) declarations.c

decentralise.o: decentralise.c
$(CC) -c $(CFLAGS) decentralise.c

block.o: block.c
$(CC) -c $(CFLAGS) block.c

clean:
rm -f *.o *~ $(PROGRAMS)
rm -f *.txt
```