TD10 - Structure de données

Code	LU2IN006
Type de cours	Travaux dirigés
Complété ?	☐
Jour du cours	@17/04/2023

PINHO FERNANDES Enzo - L2 Mono-Info S4 - GR6

TD10 : Blockchain et structures de données sécurisées

▼ Exercice 1 : Vérification de transactions bancaires

📖 **Enoncé :**

Exercice 1 – Vérification de transactions bancaires

Le transfert bancaire est une opération financière qui permet de déplacer de l'argent d'un compte bancaire à un autre. Un des enjeux de sécurité est de faire en sorte que personne ne puisse ajouter une transaction depuis le compte de quelqu'un d'autre. De ce fait, chaque transaction doit être accompagnée d'une signature électronique pour attester de sa validité. Cette signature doit dépendre à la fois de l'identité de l'utilisateur et du contenu de la transaction, et doit pouvoir être vérifiée facilement par n'importe quel utilisateur. Pour mettre cela en place, chaque utilisateur possède une identité numérique définie par un couple de clés :

- Une clé secrète/privée, qui ne doit être connue que par lui et qui lui sert à signer ses transactions.
- Un clé publique permettant aux autres utilisateurs de vérifier ses signatures.

Comme dans le protocole RSA, on suppose ici que chaque clé est composée de deux grands entiers : (n, e) pour la clé secrète et (n, d) pour la clé publique.

Pour signer une transaction, un utilisateur doit procéder comme suit :

- Créer une empreinte de la transaction à l'aide d'une fonction de hachage cryptographique.
- Créer une signature en chiffrant cette empreinte à l'aide d'une fonction `int* chiffrer(char* hash, int n, int e)`.
- Envoyer la transaction accompagnée de sa signature aux autres utilisateurs.

Pour vérifier son identité, les autres utilisateurs devront ensuite :

- Créer une empreinte de la transaction reçue à l'aide de la même fonction de hachage.
- Déchiffrer la signature à l'aide d'une fonction `char* dechiffrer(int* hash, int n, int d)` et comparer le résultat obtenu avec l'empreinte de la transaction reçue.

Q 1.1 En sachant que chaque utilisateur est identifié par sa clé publique (n, d), proposez une structure `User` permettant de représenter un utilisateur.

Q 1.2 Proposez une structure `Transaction` représentant une transaction entre deux utilisateurs.

Q 1.3 Supposons que les empreintes sont créées par la fonction `char* hashFunction(Transaction* t)`. Écrivez une fonction qui utilise la clé privée d'un utilisateur pour signer une transaction puis écrivez une fonction qui vérifie une transaction signée.

Réponses :

▼ Exercice 1 :

```
typedef struct _User {
  char *name;
  int n;
  int d;
} User;
```

▼ Exercice 2 :

```
typedef struct _Transaction{
  char *from; // On aurait pu identifier par les utilisateurs.
  char *to;   // On a supposé que le nom est unique...
  int amount;
  int *signature;
  char *date;
} Transaction;
```

▼ Exercice 3 :

```
void sign(Transaction *t, int n, int e){
  char *h = hashFunction(t);
  int *c = chiffrer(h, n, e);
  t->signature = c;
}

// On aurait pu mettre User au lieu de n et d.
int verif(Transaction *t, int n, int d){
  char *h = hashFunction(t);
  char *m = dechiffrer(t->signature, n, d);
  return (strcmp(h, m) == 0);
}
```

▼ Exercice 2 : Registre de transactions et arbre de hachage

📖 **Enoncé :**

Exercice 2 – Registre de transactions et arbre de hashage

Dans cet exercice, on s'intéresse à l'implémentation d'un registre de transactions, permettant de vérifier rapidement si une transaction a été falsifiée ou non.

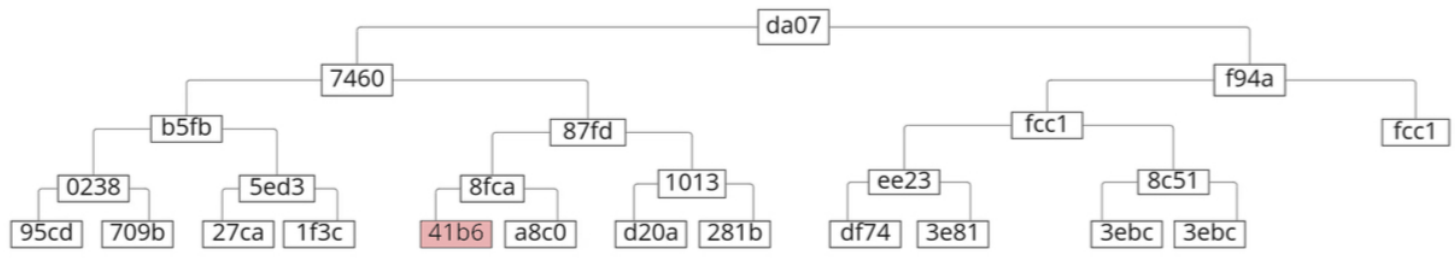
Q 2.1 On souhaite stocker les transactions sous forme de liste chaînée. Proposez une structure permettant de représenter un registre de transactions.

Q 2.2 Écrivez une fonction d'insertion d'une nouvelle transaction dans le registre.

Q 2.3 Pour réaliser une empreinte d'un registre, on pourrait appliquer une fonction de hashage sur la concaténation des hash de ses transactions. Dans ce cas, que doit-on faire pour prouver qu'une transaction donnée n'a pas été modifiée ?

Q 2.4 Une autre façon de réaliser une empreinte d'un registre est d'organiser les hash des transactions sous forme d'un arbre de Merkle. Rappelez la définition de cette structure de données.

Q 2.5 Sachant que le registre est organisé sous forme d'arbre de Merkle, comment peut-on vérifier qu'une transaction donnée n'a pas été modifiée ? Appliquez cette méthode pour la transaction dont le hash est colorié en rouge sur l'arbre suivant.



Q 2.6 Proposez une structure C permettant de représenter un arbre de Merkle. Donnez une fonction permettant de créer un noeud à partir d'un hash.

Q 2.7 Écrivez une fonction qui construit un arbre de Merkle correspondant à un tableau de n hash donné en entrée, et qui retourne la racine de l'arbre. Pour simplifier l'écriture, on supposera que l'on possède une implémentation de file de chaînes de caractères.

Q 2.8 Écrivez une fonction qui renvoie un ensemble de nœuds suffisant à prouver l'authenticité d'une feuille de l'arbre. Cette fonction devra aussi retourner, pour chacun de ces noeuds, s'il s'agit d'un fils gauche (-1) ou d'un fils droit (1). On pourra supposer avoir à disposition une implémentation de file de noeuds et de file d'entiers.

Q 2.9 Écrivez une fonction qui vérifie l'authenticité d'une feuille donnée de l'arbre, en utilisant la fonction précédente.

Réponses :

Exercice 1 :

```
typedef struct _Registre{
  Transaction *t;
  struct _Registre *next;
} Registre; // Ledger
```

Exercice 2 :

```
Registre* insererEnTete(Registre *l, User *from, User *to, int amount, char *date, int *signature){
  Transaction *t = (Transaction*)malloc(sizeof(Transaction));
  t->from = from->name; // (Ou juste from et to)
  t->to = to->name;
  t->amount = amount;
  t->date = date;
  t->signature = signature;

  Registre l2 = (Registre*)malloc(sizeof(Registre));
  l2->t = t;
  l2->next = l;
  return l2;
}
```

Exercice 3 :

- $Registre(Transaction_1, Transaction_2, \dots, Transaction_n)$
  - $h = hash(Registre) = hash(hash(Transaction_1) \,||\, hash(Transaction_2) \,||\, ... \,||\, hash(Transaction_n))$

- On calcule $hash(Transaction_i)$ pour $i \in \{1, 2, \dots, n\}$
- On calcule $hash(hash(Transaction_1) \,||\, \dots \,||\, hash(Transaction_n))$
- On vérifie que le résultat est h.
  - $O(1)$

Exercice 4 :

- Un arbre de Merkle est un arbre binaire tel que $\forall$ noeud n, si n n'es pas une feuille, `n.value == hash(n.l_child.value || n.r_child.value)`

Exercice 5 :

- Pour vérifier que `Transaction_i` est valide :
  - On calcule `h = hash(hash(Transaction_i) || hash(frere(Transaction_i)))`
  - On vérifie que `h == parent(Transaction_i)`
  - On lance la vérification sur le parent.
- On s'arrête à la racine.
  - $O(log(n))$

Exercice 6 :

```
typdef struct _MT{
  char *h;
  struct _MT *parent;
  struct _MT *l_child;
  struct _MT *r_child;
} MT;

MT* creerNode(char *h){
  MT *m = (MT*)malloc(sizeof(MT));
  m->h = strdup(h);
  m->parent = NULL;
  m->l_child = NULL;
  m->r_child = NULL;
  return m;
}
```

▼ Exercice 7 :

```
MT* buildMTlvl(MT *nodes, int n){
  MT *next_lvl = (MT *)malloc(sizeof(n/2 * sizeof(MT));
  for(int i = 0; i < n/2; i++){
    MT l_child = nodes[2*i];
    MT r_child = nodes[2*i+1];
    next_lvl[i].l_child = l_child;
    next_lvl[i].r_child = r_child;
    next_lvl[i].parent = NULL;
    next_lvl[i].h = hashFunction(l_child.h || r_child.h); // Fct à base de strconcat.
  }

  return next_lvl;
}

// Pas propre en terme fuite mémoire.
MT buildMT(char **hashes, int n){
  MT *nodes = (MT*)malloc(sizeof(MT)*n);
  for(int i = 0; i < n; i++){
    nodes[i] = creerNode(hashes[i]);
  }

  while(n > 1){
    nodes = buildMTlvl(nodes, n)
    n = n/2;
  }

  return nodes;
}
```

▼ Exercice 8 :

▼ Exercice 9 :

▼ Exercice 3 : Blockchain

📖 **Enoncé :**

Exercice 3 – Blockchain

Dans cet exercice, on s'intéresse à la création de blocs dans une blockchain avec un mécanisme de consensus dit par *proof of work* (preuve de travail). Pour cela, on utilise la structure de bloc suivante :

```
typedef struct bloc{
   char* previous_hash:     // hash du bloc precedent
   char* root;              // racine de l'arbre de Merkle des transactions
   int d;                   // difficult'e de creation
   int nonce;               // preuve de travail
   int date;                // date de creation du bloc
   LC* data;                // des transactions
} Bloc;
```

Plus précisément, pour créer un bloc contenant des transactions bancaires, on va demander au créateur du bloc d'inverser partiellement une fonction de hachage cryptographique : le créateur doit trouver un entier `nonce` tel que la valeur hachée des métadonnées du bloc (`previous hash`, `merkleTreeRoot`) concataténée à `nonce` donne une valeur de hachage qui commence par `d` zéros. Avec une fonction de hachage cryptographique, cette inversion partielle ne peut être réalisée que par brute force (et le temps de calcul nécessaire croît exponentiellement avec `d`). De ce fait, la valeur `nonce` constitue une sorte de "preuve de travail" au sens où elle permet de vérifier facilement que le créateur du bloc a fait beaucoup de calculs pour réaliser cette inversion partielle. Un bloc ne sera considéré comme valide que s'il est accompagné d'une preuve de travail.

Q 3.1 En supposant que la fonction `char* hashBlock(Bloc* b)` retourne la valeur hachée de.la chaîne de caractères composée des champs `previous_hash`, `root` et `nonce`, écrivez une fonction `void compute_proof_of_work(Bloc* B)` qui met à jour l'entier `nonce` du bloc jusqu'à ce que la valeur de hachage obtenue commence par `d` zéros.

Q 3.2 Écrivez une fonction qui vérifie si un bloc est valide.

💡 **Réponses :**

▼ **Exercice 1 :**

▼ **Exercice 2 :**