Rapport

Projet Cryptographie

ARI Moussa

BATTON Hugo

BELARIBI Sofiane

KOUALET Arnaud-Jordan

LEFEBVRE Guillaume

MARIE-NELLY Raphael

Une image contenant graphiques vectoriels

Description générée automatiquement

# 1 - Présentation du projet

Le projet est constitué de 3 parties implémentant en Java des problèmes de cryptographie. Afin de résoudre ces problèmes, nous avons implémenté et utiliser le cryptosystème de Paillier car ces propriétés homomorphies, son caractère probabiliste et sa sécurité sémantique sont très utiles.

La classe Paillier est constituée des méthodes suivantes :

- keyGen() : renvoie pk et sk

- encrypt(pk, m) : renvoie em une encryption du message m selon la clef publique pk

- encryptPlus(pk, m) : renvoie M et r, avec r un entier aléatoire utilisé pour l’encyption

- decrypt(pk, sk, em) : renvoie m le résultat du décryptage de em

- decryptPlus(pk, sk, em) : renvoie m et r

# 2 - Protocole Multiplication

## Contexte

Le protocole de multiplication implémenté permet à Alice d’obtenir l’encryption du produit de deux messages x et y dont elle a uniquement les encryptions X=Encrypt(pkB, x), Y=Encrypt(pkB, y).

Pour ce faire, elle communique sur le réseau avec Bob dont elle connait la clef publique pkB.

## Implémentation

La classe Multiproof a pour attribut une instance de la classe Paillier, une méthode multiproof(XA, YA, pkB, skB). Cette dernière déroule le protocole instauré entre Alice et Bob en 2 phases et renvoie XYA = enccypt(pk, xA\*yA) si le protocole est validé, 0 sinon.

Phase 1 : Calcul de l’encryption du produit

Phase 2 : Preuve du calcul réalisé par Bob pour Alice.

Une nomenclature des variables permet de connaitre à qui elles appartiennent ; elles sont nommées du suffixe A pour Alice et B pour Bob. De plus, les variables encryptées commencent par une majuscule.

**Phase 1**

Alice génère sA et tA aléatoirement afin de masquer X et Y qui doivent rester inconnus de Bob, puis envoie UA=XA \* SA et VA = YA \* TA.

Grace à l’homomorphisme additif de Paillier, Bob peut les décrypter et obtenir uB = xA+sA et vB = yB+tB. Il renvoie alors à Alice l’encryption WB = encrypt(pkB, uB\*vB)

Alice peut alors en déduire facile XYA ( Encrypt(pk, xA\*yA)) en utilisant les propriétés homorphiques.

**Phase 2**

Le but de cette phase est que Bob prouve qu’il a bien réalisé un produit et qu’il l’a envoyer à Alice.

Pour ce faire, ils échangent certaines valeurs encryptées et d’autres générés aléatoirement par Alice. Bob va devoir effectuer devoir décrypter des message comprenant les termes UA et VA. Il utilisera la méthode DecryptPlus(…) afin d’envoyer à Alice les messages décrypté et les r.

Pour finir, Alive pourra tester 3 assertions. Si l’une d’entre elles est fausse, cela signifie que Bob à trahi le protocole.

La classe Multiproof contient également une méthode main() à exécute pour lancer le protocole.

Dans cette méthode, on définit aléatoirement x et y puis les clefs pk, sk de Bob pour finalement encrypter x et y.

Ensuite on fait appel à la méthode Multiproof (…).

Une vérification du produit est réalisée à la fin pour confirmer l’algorithme.

# 3 - Calcul 3 DNF

## Contexte

Dans cette partie nous avons implémenté le protocole *EvalDNF* permettant d’obtenir le résultat d’une fonction 3DNF. Ce protocole fait intervenir deux participants :

* Alice qui possède une fonction la fonction 3DNF secrète f construite à partir de T variables booléennes
* Bob possède T variables booléennes secrètes x1 … xt et il souhaite obtenir f(x1…xt ) sans rien révéler sur ses variables et sans rien apprendre sur la fonction d’évaluation d’Alice.

## Implémentation

Ce protocole a été implémenté dans la classe DNF.java, le protocole est déroulé dans le constructeur de cette classe et peut être scindé en trois étapes :

* Phase 1 : Initialisation des variables booléennes, de la formule 3DNF et envoi des encryptions de ces variables à Alice
* Phase 2 : Évaluation de chaque clause de la formule par Alice grâce aux encryptions des variables et envoi du résultat à Bob
* Phase 3 : Évaluation de la formule 3DNF par Bob grâce aux évaluations de chaque clause de la formule envoyée par Alice.

### Phase 1

Dans la classe 3DNF nous commençons d’abord par une première phase d’initialisation qui permet de créer une instance de Paillier pour Alice et bob et générer les clés publiques et privées.

Pour bob, Les différentes variables booléennes sont créées au sein d’une liste de BigInteger avec 0 représentant un false et 1 représentant un true. Ensuite Bob encrypte ces variables grâce à sa clé privée afin de les envoyer à Alice.

Du coté de Alice nous déclarons la fonction 3-DNF, pour cela nous déclarons une liste *IndiceDNF* qui contiendra la liste des clauses. Chaque clause est une liste de couple de valeur, la première indique l’indice de la variable courante dans la liste des variable encryptées transmises par Bob et la deuxième valeur est un booléen indiquant si dans la clause c’est la négation ou non de la variable qui est utilisée.

### Phase 2

Afin d’obtenir l’évaluation d’une clause, Alice utilise l’addition homomorphie afin d’obtenir l’encryptions de la somme des 3 variables. Etant donné que ces variables sont soit 0 ou 1, la somme donne au maximum 3 si c’est une conjonction de vrai et la valeur est 0 ou 1 ou 2 si elle fausse. Comment alors différencier une clause évaluée à vrai vs faux ?

Pour cela nous rajoutons à l’encryptions de cette somme la valeur -3 encore une fois grâce à l’addition homomorphique. Nous aurons alors l’encryptions de 0 si la clause est vraie et une autre valeur si elle est fausse. Cela nous permet par la suite à Bob de déterminer l’évaluation d’une clause.

Le but de ce protocole est aussi de garantir la confidentialité sur la formule de la 3DNF détenu par Alice, si nous envoyons juste la liste des encryptions de l’évaluation de chaque clause de façon ordonnée alors Bob peut par brute force exploiter cette liste pour essayer de deviner la formule.

Afin de palier à cela, dans un premier temps nous mélangeons la liste des encryptions des clauses avant de l’envoyer à Bob car l’ordre des clauses n’est pas important pour évaluer toute la formule.

Ensuite nous rajoutons dans la liste un nombre aléatoire de clause dont la valeur est fausse car cela n’entrave pas l’intégrité de la formule qui est une conjonction de clause.

Grace à deux étapes supplémentaires, nous garantissons à Alice que Bob ne peut pas exploiter la liste envoyée afin de tirer des informations sur la formule 3-DNF.

### Phase 3

Enfin cette liste de clause est envoyée à Bob, il lui suffit alors de parcourir cette liste, de décrypter l’évaluation de chaque clause grâce à la clé publique d’Alice et de comparer cette évaluation à la valeur « 0 ». Si au moins une clause est égale à 0 alors le résultat de la formule est déclaré comme vrai car la formule est une conjonction de clause.

# 4 - Bracelet electronique

TODO