ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

# Privacy Preserving Group Ranking

Marc Ilunga Tshibumbu Mukendi

School of Computer and Communication Sciences

Semester Project

June 2016

**Responsible**
Prof. Serge Vaudenay
EPFL / LASEC

**Supervisor**
Handan ~~Kilin~~
EPFL / LASEC

LASEC

## 0.1 Motivation

Group ranking is a process used to find ~~a~~ the best candidate from a group. The process involves a party called the ***initiator*** and other~~s~~ parties called ***participants***. The initiator ~~usually~~ has a vector of preferences on some attributes and the participants have vectors of values, each corresponding to the attributes sought after by the initiator. At the end of the process, one or more candidates with attributes best matching the initiator preferences will be selected. This has many applications such as online marketing, personal interest matching, job recruitment and selecting candidates for medical experiment.

However, this process is more and more used in virtual environment in today's society and as a consequence privacy concerns emerge. A trivial approach to Group ranking would leak private informations about all the candidates even if some of the will not be picked at the end of the process. As an example~~,~~ online forms are used to carry such ranking. A naive implementation of the ranking would require that the candidates provide private information (i.e by answering questions on the form). Since some group ranking imply ranking candidates on based on sensitive information, such implementation of group ranking poses a problem to privacy.

Given the necessity of group ranking in a wide range of real world~~'s~~ application and the need to preserve candidates privacy, ~~is there a way to perform group ranking while preserving privacy of all the parties?~~ ~~How do we prevent candidates to learn informations about other candidates? How do we prevent candidates to learn information about the initiator and thus cheating in the process? This is the subject of this paper.~~

~~In this report we present and explain a protocol~~

~~that performs privacy preserving group ranking and we give a Java implementation of the protocol.~~

## 0.2 The protocol

In this section we give a detailed explanation of the protocol.

### 0.2.1 Framework

The protocol is executed cooperatively by $n + 1$ parties. An initiator ~~plus~~ n ~~Participant~~ P0, P1... Pn. The protocol assumes that the participants are willing to accept the initiator invitation and to submit their private information if selected eventually. The questionnaire given by the initiator is represented as a m-dimensional vector. The initiator holds a m-dimensional vector ~~v0~~ indicating the preferred values of each of the question of the questionnaire, another m-dimensional vector represents the weight associated to each questions. Furthermore, the questionnaire comprises: "Equal" questions meaning that the initiator is looking for a specific attribute. "Greater than" question means that the initiator is looking for values exceeding some threshold. We assume without loss of generality that the first t questions are"greater than" questions. Finally the answer of each candidates is also represented by a m-dimensional Vector. A complete description of the protocol framework is given below:

$P_0$ generates a group $\mathbb{G}_q \leftarrow \mathcal{G}(1^{\mathcal{K}})$, picks a generator $g$ and publishes them. $P_0$ also publishes a vector of attribute names and an integer $k$, where $1 \leq k \leq n$.
Private Input: $\mathbf{v}_0$ and $\mathbf{w}$ from $P_0$; $\mathbf{v}_j$ from participant $P_j$, $1 \leq j \leq n$.

**Secure gain computation:**

1) $P_0$ chooses a random $h$-bit integer $\rho$.
2) Every participant $P_j$ generates $\mathbf{w}'_j = [\mathbf{vg}_j^{\mathsf{T}}, (\mathbf{ve}_j * \mathbf{ve}_j)^{\mathsf{T}}, \mathbf{ve}_j^{\mathsf{T}}, 1]^{\mathsf{T}}$. As in the dot product protocol of Sec. IV-A, $P_j$ computes $QX$, $\mathbf{c}'$, $\mathbf{g}$ and sends them to $P_0$.
3) Upon receiving $(Q_j X_j, \mathbf{c}'_j, \mathbf{g}_j)$ from a participant $P_j$, $P_0$ chooses $\rho_j \leftarrow_R \{0, 1, \cdots, \rho\}$ and constructs $\mathbf{v}'_j = [\rho \mathbf{wg}^{\mathsf{T}}, -\rho \mathbf{we}^{\mathsf{T}}, 2\rho(\mathbf{we} * \mathbf{ve}_0)^{\mathsf{T}}, \rho_j]^{\mathsf{T}}$. $P_0$ computes $a_j = z_j - \mathbf{c}'_j \cdot \mathbf{v}'_j$, $h_j = \mathbf{g}_j^{\mathsf{T}} \cdot \mathbf{v}'_j$ and sends them back to $P_j$.
4) Upon receiving $(a_j, h_j)$ from $P_0$, $P_j$ calculates $\beta_j = (a_j + h_j \cdot R_2/R_3)/b$ and converts it to an unsigned integer (see Sec. III-A).

**Unlinkable gain comparison:**

5) $\forall 1 \leq j \leq n$, $P_j$ picks private key $x_j \leftarrow_R \mathbb{Z}_q$ and publishes $y_j = g^{x_j}$. $P_j$ proves the knowledge of $x_j$ to the rest of parties (Sec. IV-E).
6) Each participant $P_j$ represents her $\beta_j$ in binary bits $[\beta_j]_B = [\beta_j^l, \beta_j^{l-1}, \cdots, \beta_j^1]$ , encrypts and publishes them as $E(\beta_j)_B = [E(\beta_j^l), \cdots, E(\beta_j^1)]$. Here, the encryption is done using joint key $y = \prod_{j=1}^{n} y_j$.
7) Each participant $P_j$ gets encrypted data $\{E(\beta_i)_B\}_{i=1, i \neq j}^{n}$ from others. For each encrypted data $E(\beta_i)_B$, $P_j$ does following calculation for $1 \leq t \leq l$:
   - $E(\gamma_i^t) = E(\beta_j^t + \beta_i^t - 2\beta_j^t \beta_i^t)$, where $\gamma_i^t = \beta_j^t \oplus \beta_i^t$;
   - $E(\omega_i^t) = E((l - t + 1) - \sum_{v=t+1}^{l} (\gamma_i^t - \gamma_i^v) - \gamma_i^t)$;
   - $E(\tau_i^t) = E(\omega_i^t + \beta_i^t)$.

   $P_j$ sends all the ciphertexts $\mathcal{E}_j = \{e : e \in E(\tau_i) \wedge 1 \leq i \leq n \wedge i \neq j\}$ to $P_1$, where $E(\tau_i) = \{E(\tau_i^t)\}_{t=1}^{l}$.
8) After receiving the ciphertext sets from all the rest parties, $P_1$ constructs a vector $V = [\mathcal{E}_1, \mathcal{E}_2, \cdots, \mathcal{E}_n]$. Starting at $P_1$, each participant $P_j$ applies following steps to each element $\mathcal{E}_i$ $(i \neq j)$ in $V$:
   - For each ciphertext $(c_t, c'_t) \in \mathcal{E}_i$, replaces $c_t$ by $\bar{c}_t = c_t/(c'_t)^{x_j}$. Picks $r \leftarrow_R \mathbb{Z}_q$ and updates the ciphertext with $((\bar{c}_t)^r, (c'_t)^r)$.
   - Permutes the ciphertexts in each set $\mathcal{E}_i$.

   $P_j$ then sends the permuted vector $V$ to $P_{j+1}$. If $P_j$ is the last one, $P_n$, she sends the element of the vector back to the corresponding participant.

**Ranking Submission:**

Upon receiving the final result $\tilde{E}_j$ from $P_n$, participant $P_j$ $(1 \leq j \leq n)$ decrypts each element ciphertext $(c, c')$ by using $g^m = c/c'^{x_j}$ and checks $g^m = 1$. Let $d$ be the number of zeroes and then the ranking of $P_j$ is $d_j = d + 1$. If $d_j \leq k$, $P_j$ submits $\mathbf{v}_j$ to $P_0$ as well as the ranking.

Figure 1: Framework

**Explanation of the protocol** In our implementation, the same group will be used for every iteration of the protocol. The implementation uses a prime-order group of at least 1024 bits, ~~thus the computation of discrete log is made difficult.~~

```
package lasecbachelorprject.epfl.ch.privacypreservinghousing.
    crypto;

import java.math.BigInteger;
import java.security.SecureRandom;

public class GroupGenerator {

    private static int minBitLength;

    private static int certainty;

    private static BigInteger ONE = BigInteger.ONE;
    private static BigInteger generator, prime, group;
    private static SecureRandom secureRandom;

    public GroupGenerator(int minBitLength, int certainty,
    SecureRandom secureRandom, boolean lengthCheck){
        if(minBitLength < 1024 && lengthCheck )
                throw  new IllegalArgumentException("Prime
    should have at least 512 bits");
        this.minBitLength = minBitLength;
        this.certainty = certainty;
        this.secureRandom = secureRandom;
        getSafePrime();

    }



    private static void getSafePrime(){

        BigInteger a;
        do {
            a = new BigInteger(minBitLength,secureRandom);

            a = a.add(ONE);

            group = new BigInteger(minBitLength,certainty,
    secureRandom);

            prime = a.multiply(group).add(ONE);
        }
        while(!prime.isProbablePrime(certainty));



```

```
        boolean isGen;
        do{
            isGen = true;
            generator = new BigInteger(prime.bitLength(),
    secureRandom);
            generator = generator.mod(prime.subtract(BigInteger.
    ONE)).add(BigInteger.ONE);
            generator = generator.modPow(a, prime);
            if(generator.equals(ONE)){
                isGen = false;
            }
        }while (!(isGen && !generator.equals(BigInteger.ZERO)));
    }


    public static BigInteger getPrime(){
        if(prime == null)
        {
            getSafePrime();
        }
        return prime;
    }
    public  static BigInteger getGenerator(){
        return  generator;
    }

    public  static BigInteger getGroup(){
        return group;
    }



}
```

**Secure gain computation** First we define what is the gain in the context of this protocol.

Given a criterion vector v0 = [v01, v02,..., v0m] and the weight vector w=[w1,w2,...,wm]. The partial gain value of Pj is pj = $\sum_{k=t+1}^{m} wkv_k^j$ - $\sum_{k=1}^{t}(w_k(v_k^j)^2 - 2w_k v_k^j v_k^0)$ In terms of dot products, the partial gain is given by $wg \cdot vg_j - we \cdot (ve_j * ve_j) + 2(we * ve_0) \cdot ve_j$ The "*" operation is an element-wise multiplication of two vectors. ve0 is the equal to part of the criterion vector and vg0 is the greater than part. we, wg, vej and vgj are defined in this similar fashion.

Steps 1 through 4 are implemented by the classes: Owner.java for the initiator.

```
package lasecbachelorprject.epfl.ch.privacypreservinghousing.
    user;
```

```
import java.math.BigInteger;
import java.security.SecureRandom;

import lasecbachelorprject.epfl.ch.privacypreservinghousing.
    crypto.SecureDotProductParty;
import lasecbachelorprject.epfl.ch.privacypreservinghousing.
    helpers.Poll;

import static java.lang.Math.ceil;
import static java.lang.Math.log;
import static java.lang.System.arraycopy;


public class Owner {
    //TODO: Normal users with inheritance
    public SecureDotProductParty me;
    private BigInteger rho;
    private int h;
    private BigInteger[] criterionVector;
    public Poll myPoll;



    private BigInteger[] myAttVector; // 10 for test
    private BigInteger[] myWeightVector;
    private BigInteger[] wPrimeVector;
    public int l;
    private int k = 2 ; //for now
    private int t = 5; // for now
    private SecureRandom random;
    private final int greaterLength;
    private final int equalLength;

    public Owner(BigInteger criterionEqualVector [],BigInteger[]
     criterionGreaterVector, BigInteger[] weightEqualVector,
    BigInteger[] weightGreaterVector){
        greaterLength = criterionGreaterVector.length;
        equalLength = criterionEqualVector.length;

        l = (int) ceil(log(equalLength + criterionGreaterVector.
    length)) +
                criterionEqualVector[0].bitLength() +
                2*weightEqualVector[0].bitLength() + 2 ;
        random = new SecureRandom();

        do{
            h = random.nextInt() % 20;
        }while (h <= 0);
        //TODO: Decomment following line
        rho = (new BigInteger(h, random)).abs();

        generateOwnerVprime(criterionEqualVector,
    weightGreaterVector,weightEqualVector);
```

```
        me = new SecureDotProductParty(rho);
        me.setMyvector(wPrimeVector);



    }

    private void generateOwnerVprime(BigInteger[] criterionEqVector, BigInteger[] weightEqualVector, BigInteger[] weightGreaterVector) {
        /*Vector of size gr + 2 *eq*/
        wPrimeVector = new BigInteger[greaterLength + 2*equalLength];

        /*[vgT,..]*/
        arraycopy(weightGreaterVector,0, wPrimeVector,0, greaterLength);

        BigInteger v[] = new BigInteger[equalLength];
        /* [vgT,(ve*ve)T,..]*/
        for (int i = 0; i < equalLength ; i++) {
            v[i] = weightEqualVector[i].negate();
        }
        arraycopy(v,0,wPrimeVector,greaterLength,equalLength);

        for (int i = 0; i <equalLength; i++) {
            v[i] = weightEqualVector[i].multiply(criterionEqVector[i]);
            v[i] = v[i].add(v[i]);
        }
        arraycopy(v,0,wPrimeVector,greaterLength+equalLength,equalLength);
        for (int i = 0; i < wPrimeVector.length ; i++) {
            wPrimeVector[i] = wPrimeVector[i].multiply(rho);
        }

        myAttVector = new BigInteger[wPrimeVector.length];
        for (int i = 0; i <myAttVector.length; i++) {
            myAttVector[i] = wPrimeVector[i];
        }
    }

    public void initiatePoll(int Participants){


        myPoll = new Poll(Participants);

    }


    public BigInteger[] getMyAttVector() {
        return myAttVector.clone();
    }
```

```
}
```

Participant.java for participants.

```
package lasecbachelorprject.epfl.ch.privacypreservinghousing.
    user;


import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import lasecbachelorprject.epfl.ch.privacypreservinghousing.
    Activities.Application;
import lasecbachelorprject.epfl.ch.privacypreservinghousing.
    crypto.ElGamal;
import lasecbachelorprject.epfl.ch.privacypreservinghousing.
    crypto.EncryptedBinaryComparator;
import lasecbachelorprject.epfl.ch.privacypreservinghousing.
    crypto.SecureDotProductParty;
import lasecbachelorprject.epfl.ch.privacypreservinghousing.
    crypto.ZeroKnowledgeProver;
import lasecbachelorprject.epfl.ch.privacypreservinghousing.
    crypto.ZeroKnowledgeVerifier;
import lasecbachelorprject.epfl.ch.privacypreservinghousing.
    helpers.DataBase;

import static lasecbachelorprject.epfl.ch.
    privacypreservinghousing.crypto.ElGamal.homomorphicEncryption
    ;
import static lasecbachelorprject.epfl.ch.
    privacypreservinghousing.crypto.ElGamal.
    multHomomorphicEncryption;
import static lasecbachelorprject.epfl.ch.
    privacypreservinghousing.crypto.EncryptedBinaryComparator.
    createBinaryArray;
import static lasecbachelorprject.epfl.ch.
    privacypreservinghousing.crypto.EncryptedBinaryComparator.
    finalDecryption;
import static lasecbachelorprject.epfl.ch.
    privacypreservinghousing.crypto.SecureDotProductParty.
    copyBigIntArrayToIntArray;

public class Participant extends Person implements User {


    private final BigInteger prime;
    private final BigInteger generator;
```

```
    private final int k;
    public SecureDotProductParty secureDotProduct;
    public BigInteger[] wPrimeVector;
    private BigInteger privateKey;
    private BigInteger publicKey;
    public ZeroKnowledgeProver prover;
    public ZeroKnowledgeVerifier verifier;
    private SecureRandom random;
    private BigInteger group;
    int replyEqSize;
    int replyGrSize;
    private BigInteger gain;
    private BigInteger[][] cypher;
    private int myCandidateNumber;
    private static Map<Integer, BigInteger[][]> othersGain;
    private static List<BigInteger[][]> encryptedComparisonList;
    private List<List<BigInteger[][]>> list;
    private int l;
    private EncryptedBinaryComparator comparisonTool = EncryptedBinaryComparator.getGainComparisonTool();
    private BigInteger commonKey;
    private List<BigInteger[][]> finalComp;
    private int ranking;


    public Participant(BigInteger replyGr[], BigInteger[] replyEq, int l, int k) {
        replyEqSize = replyEq.length;
        replyGrSize = replyGr.length;
        wPrimeVector = new BigInteger[replyGrSize + 2 * replyEqSize];
        generateWPrimeVector(replyGr, replyEq);
        secureDotProduct = new SecureDotProductParty();
        secureDotProduct.setMyvector(wPrimeVector);
        group = Application.group;
        prime = Application.prime;
        generator = Application.generator;
        random = new SecureRandom();

        prover = new ZeroKnowledgeProver(prime, group, generator);
        verifier = new ZeroKnowledgeVerifier(prime, group, generator);
        othersGain = new HashMap<>();
        encryptedComparisonList = new ArrayList<>();
        this.l = l;
        this.k = k;
    }

    private void generateWPrimeVector(BigInteger replyGr[], BigInteger[] replyEq) {
        copyBigIntArrayToIntArray(replyGr, 0, wPrimeVector, 0, replyGrSize);

```

```
        BigInteger[] v = new BigInteger[replyEqSize];

        for (int i = 0; i < replyEqSize; i++) {
            v[i] = replyEq[i].pow(2);
        }
        copyBigIntArrayToIntArray(v, 0, wPrimeVector, replyGrSize, replyEqSize);
        copyBigIntArrayToIntArray(replyEq, 0, wPrimeVector, replyEqSize + replyEqSize, replyEqSize);
    }

    public BigInteger[] getReplyVector() {
        return wPrimeVector;
    }


    public void convertGain(int l) {
        BigInteger two = BigInteger.ONE.add(BigInteger.ONE);
        gain = secureDotProduct.getBeta().add(two.pow(l));
        wPrimeVector = createBinaryArray(gain,l);
    }

    public void generatePrivateKey() {
        privateKey = (new BigInteger(prime.bitLength(), random)).mod(prime);
        publicKey = generator.modPow(privateKey, prime);
        DataBase.getDataBase().publishElGamalPublicKey(this, publicKey);
        prover.setX(privateKey);
    }


    public void setKeyToVerify(BigInteger key) {
        verifier.setY(key);
    }


    public void encryptWithCommonkey() {
        ElGamal.setCommonKey(DataBase.getEncryptionKey());
        cypher = ElGamal.encryptMany(wPrimeVector);

    }


    public void setMyCandidateNumber(int number) {
        myCandidateNumber = number;
    }

    public void sendEncryptedComparisonToDB() {
        DataBase.pushComparisonVector(this, comparisonTool.getEncryptedComparisons());
    }

    public void getEpsilonVector(List<List<BigInteger[][]>> list
```

```java
) {
        this.list = comparisonTool.chainedDecryption(list, myCandidateNumber, privateKey, prime,group);

    }

    public void sendBackDecryptedListToDB() {
        DataBase.pushPartialListDecryption(list);
    }


    public void participantGain(Integer participantIndex, BigInteger[][] gain) {
        othersGain.put(participantIndex, gain);
    }

    public void compareWithParticipants() {
        //List of encrypted bit by bit comparisons
        encryptedComparisonList = new ArrayList<>();
        //Procedure for candidate I
        for (Integer index : othersGain.keySet()) {

            //Computation of the gama's factors
            List<BigInteger[]> gamas = new ArrayList<>(l);
            //table of beta's
            BigInteger[][] betaI = othersGain.get(index);
            BigInteger[] tmp;
            BigInteger[] tmp2;
            for (int t = 0; t < l; t++) {
                tmp = homomorphicEncryption(betaI[t], cypher[t]);
                tmp2 = multHomomorphicEncryption(betaI[t], wPrimeVector[t].multiply(BigInteger.valueOf(-2)));
                gamas.add(t, homomorphicEncryption(tmp, tmp2));
            }
            /*
             * Optimisation begin with t = l so singe loop
             * Pull the creation outside the loops
             */
            BigInteger[] val;//(l -t +1 )
            BigInteger[][] sum = new BigInteger[l][2];
            BigInteger[][] negGamaT = new BigInteger[l][2];
            BigInteger[][] omegas = new BigInteger[l][2];
            BigInteger[][] taus = new BigInteger[l][2];
            for (int t = 0; t < l; t++) {
                val = ElGamal.encrypt(BigInteger.valueOf(l - t));
                tmp = ElGamal.getNegativeciphers(gamas.get(t));
                negGamaT[t] = multHomomorphicEncryption(tmp, BigInteger.valueOf(l - t));
                sum[t] = homomorphicEncryption(gamas.subList(t + 1, l));
                omegas[t] = homomorphicEncryption(val, sum[t], negGamaT[t]);// E( l-t+1 + sum of (gamav - gmai))
```

```
                taus[t] = homomorphicEncryption(omegas[t], betaI[t]);
            }
            encryptedComparisonList.add(taus);
        }
        DataBase.pushComparisonVector(this, encryptedComparisonList);
    }




    public void receiveFinalComp(List<BigInteger[][]> comp) {
        this.finalComp = comp;

    }

    private void computeSelfRanking(){
        ranking = finalDecryption(finalComp);
    }

    private void rankingSubmission(){
        if (ranking <= k){
            DataBase.submitsResults(this,ranking, wPrimeVector);
        }
    }

}
```

SecureDotProductParty.java is used to implement the secure dot product protocol proposed by Ioanids and used in this protocol

```
package lasecbachelorprject.epfl.ch.privacypreservinghousing.crypto;


import java.math.BigInteger;
import java.security.SecureRandom;

/*
 * Class that a represent parties in secure dot product protocol
 */

public class SecureDotProductParty {

    private BigInteger [][] Q;
    private BigInteger  [][] X;
    private BigInteger  factors[];
    private BigInteger  qTimesX[][];
    private BigInteger [] cPrime;
    private BigInteger [] g;

    private int  dDimension;
    private SecureRandom secureRandom;
```

```java
    private int  sDimension;

    private BigInteger  b;
    private BigInteger [] c;
    private int  rThRow;
    private BigInteger  R1;
    private BigInteger  R2;
    private BigInteger  R3;

    private BigInteger [] y;
    private BigInteger  z;
    private BigInteger  a;
    private BigInteger  h;
    private BigInteger  rho;
    private BigInteger  rhoMax;
    private BigInteger  beta;

    public BigInteger [] getPrimeVector() {
        return primeVector.clone();
    }

    private BigInteger [] primeVector;

    public BigInteger  dotProduct;
    public BigInteger gain;

    //TODO: Copy the values so that the vector can't be modified
     in the outside
    public SecureDotProductParty(BigInteger rho){

        secureRandom = new SecureRandom();
        this.rhoMax = new BigInteger(String.valueOf(rho));

    }

    public SecureDotProductParty(){
        secureRandom = new SecureRandom();
    }


    //initiate The dot product for myVector
    public void  initiateDotProduct(){



        //TODO : Constant matrix dimension
        sDimension = 100; //1 + secureRandom.nextInt(10);
        rThRow = secureRandom.nextInt(sDimension);

        //Genrate Q  and compute b. TODO: Skip rth row in the
    for an assigne later
        Q = new BigInteger [sDimension][sDimension];
        for (int i = 0; i < sDimension ; i++) {
            for (int j = 0; j <sDimension ; j++) {
```

```
                //TODO: Correct BOUnd
                    Q[i][j] = BigInteger.ONE;//new BigInteger(
    String.valueOf(secureRandom.nextInt(100)));

            }
        }


        //TODO: b!!!
        b = BigInteger.ZERO;
        for (int i = 0; i < sDimension; i++) {
            b = b.add(Q[i][rThRow]);// safeAdd(b,Q[i][rThRow]);
        }
        //Generate X
        X = new BigInteger [sDimension][dDimension];

        for (int i = 0; i <sDimension ; i++) {
            for (int j = 0; j <dDimension ; j++) {
                if(i != rThRow){
                    //TODO: Ccrrect next INT LIMIT
                    X[i][j] = BigInteger.valueOf(secureRandom.
    nextInt(100)+1);
                }
                else{
                    X[i][j] = primeVector[j];
                    }
                }

            }




        //scalar factor to compute the "c" vector.
        //factors[i] = sum(Qji).
        factors = new BigInteger [sDimension];

        factors[rThRow] = BigInteger.ZERO;
        for (int i = 0; i <sDimension ; i++) {
            factors[i] = BigInteger.ZERO;
            if(i != rThRow) {
                for (int j = 0; j < sDimension; j++) {

                    factors[i] = factors[i].add(Q[j][i]);
                }
            }
        }

        //Generate c
        c = new BigInteger [dDimension];

        for (int i = 0; i < dDimension ; i++) {
            c[i] = BigInteger.ZERO;
            for (int j = 0; j < sDimension ; j++) {
```

```java
                c[i] = c[i].add(X[j][i].multiply(factors[j]));
            }
        }

        //Generate f
        BigInteger [] f = new BigInteger [dDimension];
        for (int i = 0; i <dDimension ; i++) {
            f[i] = BigInteger.valueOf(secureRandom.nextInt(100) + 1);
        }

        //TODO: Change the 1
        R1 = BigInteger.valueOf(secureRandom.nextInt(100) +1);
        R2 = BigInteger.valueOf(secureRandom.nextInt(100) +1);
        R3 = BigInteger.valueOf(secureRandom.nextInt(100) +1);

        //Compute Q*X
         qTimesX = new BigInteger [sDimension][dDimension];
        for (int i = 0; i <sDimension ; i++) {
            for (int j = 0; j <dDimension ; j++) {
                qTimesX[i][j] = BigInteger.ZERO;
                for (int k = 0; k <sDimension ; k++) {
                    qTimesX[i][j] = qTimesX[i][j].add(Q[i][k].multiply(X[k][j]));
                }
            }
        }

        //c'
        cPrime = new BigInteger [dDimension];
        BigInteger  R1TimesR2 = R1.multiply(R2);
        for (int i = 0; i <dDimension ; i++) {
            cPrime[i] =  c[i].add(R1TimesR2.multiply(f[i]));
        }

        BigInteger  R1TimesR3 = R1.multiply(R3);

        //Generate g
        g = new BigInteger [dDimension];
        for (int i = 0; i <dDimension ; i++) {
            g[i] = R1TimesR3.multiply(f[i]);
        }

    }



    public void sendInitialDataToOtherParty(SecureDotProductParty party){
        party.receiveQTimesX(qTimesX,cPrime,g);
    }

    public void sendAH(SecureDotProductParty party){
        party.receiveAH(a, h);
```

```
    }

    private void receiveAH(BigInteger  a, BigInteger  h) {
        this.a = a;
        this.h = h;
        beta = (a.add((h.multiply(R2)).divide(R3))).divide(b);
    }

    public void sendBeta(SecureDotProductParty party){
        party.receiveBeta(beta);
    }

    public void sendAlpha(SecureDotProductParty party){
        party.receiveAlpha(rho);
    }
    private void receiveBeta(BigInteger  beta) {
        this.beta = beta;
    }

    public BigInteger  getBeta(){
            return beta;
    }
    public BigInteger  getAlpha(){return rho;}



    private void receiveQTimesX(BigInteger [][] qTimesX,
    BigInteger [] cPrime, BigInteger [] g) {
       /* if(this.cPrime.length != primeVector.length || this.g.
    length != primeVector.length){
            throw new IllegalArgumentException("The dot product
    can't be computed because of dimensions mismatch. Expected
    vector size: "+
                                                  myvector.length
    + " received cPrime Size: " + (this.cPrime.length - 1) +"
    received g size: "+ (this.g.length -1));
        }*/
        this.qTimesX = qTimesX;
        this.cPrime = cPrime;
        this.g = g;
        BigInteger [] rhoVector;


        rho = new BigInteger(rhoMax.bitLength() -1, secureRandom
    );
        rhoVector = primeVector.clone();
        rhoVector [rhoVector.length - 1 ] = rho;

        /*double [][] qTimesX = (double [][]) get("QtimesX");
        double [] cPrime = (double []) get("cPrime");
        double [] g = (double []) get("G");*/
        y = computeY(qTimesX, rhoVector);
        z = vectorElementsSum(y);
        a = z.subtract(normalDotProduct(cPrime,rhoVector));
```

```
        h = normalDotProduct(g,rhoVector);

        /*this.put("Y", y);
        this.put("Z", z);
        this.put("A", a);
        this.put("H",h);*/

    }




    private BigInteger [] computeY(BigInteger [][] qTimesX,
    BigInteger [] myVectorPrime) {
        int dim = qTimesX.length;
        BigInteger  y[] = new BigInteger [dim];
        for (int i = 0; i <dim ; i++) {
            y[i] = normalDotProduct(qTimesX[i], myVectorPrime);
        }
        return y;
    }

    public static BigInteger  normalDotProduct(BigInteger [] v1,
     BigInteger [] v2 ){
        BigInteger  res = BigInteger.ZERO;
        for (int i = 0; i < v1.length ; i++) {
            res = res.add(v1[i].multiply(v2[i]));
        }
        return res;
    }

    private static BigInteger [] vectorAddition(BigInteger [] v1
    , BigInteger  v2[]){
        int dim = v1.length;
        BigInteger  [] v = new BigInteger [dim];
        for (int i = 0; i < dim; i++) {
            v[i] = v1[i].add(v2[i]);
        }
        return v;
    }

    public static BigInteger[] vectorsElemMult(BigInteger[] v1,
    BigInteger[] v2){
        BigInteger[] res = new BigInteger[v1.length];
        for (int i = 0; i <v1.length ; i++) {
            res[i] = v1[i].multiply(v2[i]);
        }
        return res;
    }

    public static BigInteger[] vectorsElemMultMod(BigInteger[]
    v1, BigInteger[] v2,BigInteger prime){
        BigInteger[] res = new BigInteger[v1.length];
        for (int i = 0; i <v1.length ; i++) {
```

```
                res[i] = (v1[i].multiply(v2[i])).mod(prime);
            }
            return res;
        }




        public static BigInteger  vectorElementsSum(BigInteger [] v1){
            int dim = v1.length;
            BigInteger  res = BigInteger.ZERO;
            for (int i = 0; i <dim; i++) {
                res = res.add(v1[i]);
            }
            return res;
        }

        public static BigInteger [] vectorScalarMult(BigInteger [] vector, BigInteger  scalar){
            BigInteger[] res = new BigInteger[vector.length];
            for (int i = 0; i < vector.length ; i++) {
                res[i] = vector[i].multiply(scalar);
            }
            return res;
        }

        public static BigInteger[] vectorScalarMultMod(BigInteger[] vector, BigInteger scalar, BigInteger prime){
            BigInteger[] res = new BigInteger[vector.length];
            for (int i = 0; i < vector.length; i++) {
                res[i] = (vector[i].multiply(scalar)).mod(prime);
            }
            return res;
        }

        public static BigInteger[] vectorsElemExpo(BigInteger[] v, int expo){
            for (int i = 0; i < v.length; i++) {
                v[i] = v[i].pow(expo);
            }
            return v;
        }

        public static BigInteger[] vectorsElemModExpo(BigInteger[] v, BigInteger expo, BigInteger prime){
            BigInteger[] res = new BigInteger[v.length];
            for (int i = 0; i < v.length; i++) {
                res[i] = v[i].modPow(expo,prime);
            }
            return res;
        }


```

```java


    public void setAlphaMax(BigInteger  alphaMax) {
        this.rhoMax = alphaMax;
    }


    private void receiveAlpha(BigInteger  alpha) {
        this.rho = alpha;
    }


    public void getPartialGain(int  l) {
        BigInteger two = BigInteger.ONE.add(BigInteger.ONE);
        gain = (new BigInteger(String.valueOf(dotProduct))).add(
    two.pow(l-1));
    }

    public void setMyvector(BigInteger [] vector) {
        dDimension = vector.length + 1;
        primeVector = new BigInteger [dDimension];
        System.arraycopy(vector,0,primeVector,0,vector.length);
        primeVector[vector.length] = BigInteger.ONE;

    }

    public static void copyBigIntArrayToIntArray(BigInteger[]
    src, int  srcPos,BigInteger [] des, int  destPos, int  number
     ){
        for (int i = srcPos,j= destPos; i <srcPos+number ; i++,
    j++) {
            des[j] = src[i];
        }
    }

    public BigInteger  getRhoForParticipant() {
        return rho;
    }


    static final long  safeAdd(long  left, long  right) {
        if (right > 0 ? left > Long.MAX_VALUE - right
                : left < Long.MIN_VALUE - right) {
            throw new ArithmeticException("Longoverflow");
        }
        return left + right;
    }

    static final long  safeSubtract(long  left, long  right) {
        if (right > 0 ? left < Long.MIN_VALUE + right
                : left > Long.MAX_VALUE + right) {
            throw new ArithmeticException("Longoverflow");
        }
        return left - right;
```

```
    }

    static final long safeMultiply(long left, long right) {
        if (right > 0 ? left > Long.MAX_VALUE/right
                || left < Long.MIN_VALUE/right
                : (right < -1 ? left > Long.MIN_VALUE/right
                || left < Long.MAX_VALUE/right
                : right == -1
               && left == Long.MIN_VALUE) ) {
            throw new ArithmeticException("Integer overflow");
        }
        return left * right;
    }

    static final long safeDivide(long left, long right) {
        if ((left == Long.MIN_VALUE) && (right == -1)) {
            throw new ArithmeticException("Integer overflow");
        }
        return left / right;
    }

    static final long safeNegate(long a) {
        if (a == Long.MIN_VALUE) {
            throw new ArithmeticException("Integer overflow");
        }
        return -a;
    }
    static final long safeAbs(long a) {
        if (a == Integer.MIN_VALUE) {
            throw new ArithmeticException("Integer overflow");
        }
        return Math.abs(a);
    }



}
```

**Unlinkeable Gain comparison** NEED TO DESCRIBE PRECISELY UNLINKEABLE!!!! In this phase each participant $P_i, 1 \leq \cancel{j} \leq n$ blindly compares his gain ~~to~~ others. ~~The gains are converted~~ to binary-array $\beta$ ~~and each bits is encrypted~~ with ~~a~~ modified version of ElGamal cryptosystem ~~beforehand~~. For the common key y, each participant picks a private key xj, publishes the corresponding public key and proves the knowledge of that key to other participant. The proof of knowledge is done by classes ZeroknowledgeProver.java and ZeroKnowledgeVerifier.java.

```
package lasecbachelorprject.epfl.ch.privacypreservinghousing.
    crypto;

import java.math.BigInteger;
import java.security.SecureRandom;
```

```java

public class ZeroKnowledgeProver {

    private BigInteger x;
    private BigInteger h;
    private BigInteger z;
    private BigInteger prime;
    private BigInteger group;
    private BigInteger generator;
    private SecureRandom secureRandom;
    public BigInteger r;

    //TODO: Group should be the published key
    public ZeroKnowledgeProver(BigInteger prime, BigInteger group, BigInteger generator){
        this.prime = prime;
        this.group = group;
        this.generator = generator;
        secureRandom = new SecureRandom();
    }

    public void initiateProof(){
        generateH();
    }
    private void generateH(){
        r = new BigInteger(group.bitLength(),secureRandom);
        r = r.mod(group);
        h = generator.modPow(r, prime);
    }

    public void sendH(ZeroKnowledgeVerifier verifier){
        verifier.receiveH(h);
    }

    private void computeZ(BigInteger c){
        z = (r.add(x.multiply(c))).mod(group);
    }

    public BigInteger sendZ(BigInteger c){
        computeZ(c);
        return z;
    }


    public void receiveC(BigInteger c) {
        computeZ(c);
    }

    public void setX(BigInteger x){
        this.x = x;
    }
}
```

```java
package lasecbachelorprject.epfl.ch.privacypreservinghousing.
```

```
    crypto;

import java.math.BigInteger;
import java.security.SecureRandom;

public class ZeroKnowledgeVerifier {
    private BigInteger y;
    private BigInteger h;
    private BigInteger z;
    private BigInteger prime, group, generator, qMinusOne;
    private BigInteger c;
    private SecureRandom secureRandom;

    public ZeroKnowledgeVerifier(BigInteger prime, BigInteger group, BigInteger generator){
        this.prime = prime;
        this.generator = generator;
        this.group = group;
        this.qMinusOne = group.subtract(BigInteger.ONE);
        secureRandom = new SecureRandom();
    }

    public BigInteger sendC(){
        c = new BigInteger(group.bitLength(), secureRandom);
        return  c = c.mod(group);

    }



    public void receiveH(BigInteger h) {
        this.h = h;
    }

    public void receiveZ(BigInteger z) {
        this.z = z;
    }

    public void setY(BigInteger y) {
        this.y = y;
    }


    public boolean verifyWithZ(BigInteger z, BigInteger sharedC) {
        BigInteger gPowZ = generator.modPow(z,prime);
        BigInteger hTimesYPowC = (h.multiply(y.modPow(sharedC, prime))).mod(prime);
        return  gPowZ.equals(hTimesYPowC);
    }
}
```

The actual comparison is executed at step 7. The comparison on Encrypted bits, using additive homomorphism of the modified ElGamal cryptosystem that we shortly describe here.

**Modified Elgamal** Group Gq is a prime order multiplicative group for which the DDH problem is hard and g is a generator in Gq. Then, the ModfiElGamal cryptosystem is described as: Key generation: a private key is a random element x in Zq and the corresponding public key is y = $g^x$. Encryption: a ciphertext of a message M is of form E(M)=($g^m y^r$, $g^r$), where r R Zq. ¿¿¿DESCRIPTION OF DECRYPTION ¿¿¿SHOW HOMOMORPHIC ADDITIVITY AND MULTIPLICATIVE IF ONE PLAIN TEXT IS KNOW. ¿¿¿SHOW THAT SINCE WE ARE ONLY WORKING WITH 0 and 1 FOR THE GAIN THERE IS NO PROBLEM WITH DECRYPTION

```
package lasecbachelorprject.epfl.ch.privacypreservinghousing.
    crypto;

import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import lasecbachelorprject.epfl.ch.privacypreservinghousing.user
    .Participant;

public class ElGamal {


    private static BigInteger prime,group,generator, privateKey,
     myPublicKey;
    private static final BigInteger ONE = BigInteger.ONE;
    private static ElGamal cryptoSystem;
    private static SecureRandom secureRandom;
    private static BigInteger commonKey;
    //TODO: Remove after test
    private static Map<Participant, BigInteger> mySecretKey;
    private static Map<Participant, BigInteger> myPukey;

    //TODO: fix group, generator. Add method for secret key

    private ElGamal(BigInteger prime, BigInteger group,
    BigInteger generator){
        mySecretKey = new HashMap<>();
        myPukey= new HashMap<>();
        this.prime = prime;
        this.group = group;
        this.generator = generator;
        secureRandom = new SecureRandom();
```

```
    }

    public static ElGamal getElGamal(BigInteger prime, BigInteger group, BigInteger generator){
        if (cryptoSystem == null){
            cryptoSystem = new ElGamal(prime, group, generator);
        }
        return cryptoSystem;
    }



    public static BigInteger[] encrypt(BigInteger bit){
        if(bit == null) {
            throw new IllegalArgumentException("Message to encrypt is null");
        }
        BigInteger r = new BigInteger(group.bitLength(), secureRandom);
        r = r.mod(group);
        BigInteger gPowMessage = generator.modPow(bit,prime);
        BigInteger yPowR = commonKey.modPow(r, prime);
        BigInteger[] cipher = new BigInteger[2];
        cipher[0] = gPowMessage.multiply(yPowR).mod(prime);
        cipher[1] = generator.modPow(r, prime);
        return cipher;
    }

    public static BigInteger[][] encryptMany(BigInteger[] bits){
        BigInteger[][] res = new BigInteger[bits.length][2];
        for (int i = 0; i <bits.length ; i++) {
            res[i] = encrypt(bits[i]);
        }
        return res;
    }

    public static  void setPrivateKey(BigInteger privateKey){
        if(privateKey == null){
            throw  new IllegalArgumentException("Null Argument to Set private key");
        }

        ElGamal.privateKey = privateKey;
        myPublicKey = generator.modPow(privateKey,prime);
    }

    public static void setMySKey(Participant p, BigInteger key){
        mySecretKey.put(p,key);
        myPukey.put(p,generator.modPow(key,prime));

    }

    public static BigInteger getMyPuKey(Participant p){
        return myPukey.get(p);
```

```
    }

    public static void setCommonKey(BigInteger commonKey){
        ElGamal.commonKey = commonKey;
    }


    public static BigInteger decrypt(BigInteger[] cypher){
        if(cypher == null || cypher.length != 2|| cypher[0] ==
null|| cypher[1] == null ){
            throw new IllegalArgumentException("null or bad
length cypher");
        }

        BigInteger msg = (cypher[0].multiply(cypher[1].modPow(
privateKey.negate(),prime))).mod(prime);

        return msg.equals(BigInteger.ONE) ? BigInteger.ZERO :
BigInteger.ONE;
    }

    public static BigInteger decryptMe(Participant p, BigInteger
[] cypher){
        if(cypher == null || cypher.length != 2|| cypher[0] ==
null|| cypher[1] == null ){
            throw new IllegalArgumentException("null or bad
length cypher");
        }
        setPrivateKey(mySecretKey.get(p));
        BigInteger msg = (cypher[0].multiply(cypher[1].modPow(
privateKey.negate(),prime))).mod(prime);

        return msg.equals(BigInteger.ONE) ? BigInteger.ZERO :
BigInteger.ONE;
    }



    public static List<BigInteger> decryptMany(List<BigInteger
[]> table){
        ArrayList<BigInteger> res = new ArrayList<>(table.size()
);
        for (BigInteger[] t: table) {
            res.add(decrypt(t));
        }
        return res;
    }

    public static Collection<BigInteger> decryptManyME(
Participant p,Collection<BigInteger[]> table){
        ArrayList<BigInteger> res = new ArrayList<>(table.size()
);
        for (BigInteger[] t: table) {
            res.add(decryptMe(p,t));
```

```
        }
        return res;
    }

    public static BigInteger getmyPublicKey(){
        return  myPublicKey;
    }


    public static BigInteger[] homomorphicEncryption(Collection<
    BigInteger[]> ciphers){
        BigInteger[] res = new  BigInteger[]{BigInteger.ONE,
    BigInteger.ONE};
        for (BigInteger[] c: ciphers) {
            res = SecureDotProductParty.vectorsElemMultMod(res,c
    ,prime);
        }
        return res;
    }

    public static BigInteger[] getNegativeciphers(BigInteger[]
    cypher){
        return SecureDotProductParty.vectorsElemModExpo(cypher,
    BigInteger.valueOf(Long.parseLong("-1")),prime);
    }

    public static BigInteger[] homomorphicEncryption(BigInteger
    [] c1, BigInteger[] c2){
        return SecureDotProductParty.vectorsElemMultMod(c1,c2,
    prime);
    }

    public static BigInteger[] homomorphicEncryption(BigInteger
    []... ciphers){
        BigInteger[] res = new BigInteger []{BigInteger.ONE,
    BigInteger.ONE};
        for(BigInteger[] c : ciphers){
            res = homomorphicEncryption(res,c);
        }
        return res;
    }

    public static BigInteger[] multHomomorphicEncryption(
    BigInteger[]cipher, BigInteger otherWord){
       return SecureDotProductParty.vectorsElemModExpo(cipher,
    otherWord,prime);
    }

    public static BigInteger getPrime(){
        return prime;
    }

    public static BigInteger getGenerator(){
        return getGenerator();
```

```
    }

    public static BigInteger getGroup(){
        return group;
    }

}
```

The comparison with an other participant's(WLOG $P_j$) gain is done as follow: First stage the participant of concern let say $P_i$ executes a bit-wise xor with his gain vector $\beta_i$ gain yielding a new vector $\gamma$.
Next stage, the participant generates a new vector $\omega$ where each elements is computed by the formula $\omega^t = (l-t+1) - (l-t+1)\cdot\gamma^t + \sum_{v=t+1}^{l}(\gamma^v)$
Last stage: The last generated vector is $\tau$ where $\tau^t = \omega^t + \beta_i^t$ At the end of the process the $\tau$ vector will have at most one 0 and if it contains on then $P_i$ has smaller gain than $P_j$
Now we show that this comparison is correct.

**Proof** Let a = $a_{n-1}...a_1a_0$ and b = $b_{n-1}...b_1b_0$ be two n bits unsigned numbers in binary form and we want to compare a to b. Let suppose W.L.O.G that the first k-1 most significant bits of a and b are the same, $0 \leq k-1 \leq n$. The $\gamma$ vector of stage one will have k-1 zeros in the last k-1 elements. This implies that the corresponding values of the $\omega$ vector will have values of at least 1 so will be the values of the corresponding $\tau$ vector. The value of the kth element in each vectors depends on the two following cases.
*Case 1*: a is greater than b. In this case $a^k = 1$ and $b^k = 0$. This implies that $\gamma^k = 1$. This also means that $\omega^k = 0$ and $\tau^t = 0 + a^k k = 1$.
*Case 2*: a is smaller than b. We have $a^k = 0$ and $b^k = 1$. This implies that $\gamma^k = 1$. This also means that $\omega^k = 0$ and $\tau^k = 0 + a^k = 0$.

Lastly, $\omega^j \geq 1, j < k$. This comes form the fact that $(l-t+1) - (l-t+1)\cdot\gamma^j \geq 0$ and also the sum is at least 1 due to the $\gamma^k$. This shows that the comparison method will give at then at most one zero and if a ¡ b then the comparison will yield one zero and none otherwise.(i.e a $\geq$ b).

These operation are done with the class EncryptedBinaryComparator.java

```
package lasecbachelorprject.epfl.ch.privacypreservinghousing.
    crypto;

import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
```

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import lasecbachelorprject.epfl.ch.privacypreservinghousing.user
    .Participant;

import static lasecbachelorprject.epfl.ch.
    privacypreservinghousing.crypto.ElGamal.decrypt;
import static lasecbachelorprject.epfl.ch.
    privacypreservinghousing.crypto.ElGamal.decryptMany;
import static lasecbachelorprject.epfl.ch.
    privacypreservinghousing.crypto.ElGamal.homomorphicEncryption
    ;
import static lasecbachelorprject.epfl.ch.
    privacypreservinghousing.crypto.ElGamal.
    multHomomorphicEncryption;


public class EncryptedBinaryComparator {

    private static BigInteger[][] myGain;
    private static Map<Integer, BigInteger[][]> othersGain;
    private static List<BigInteger[][]> encryptedComparisonList;
    private static BigInteger[] plainGain;
    private static BigInteger[] othersPlain;
    private static int l;
    private static EncryptedBinaryComparator gainComparisonTool;
    private static Map<Participant,BigInteger[][]> selfGain;
    private static Map<Participant,Map<Integer, BigInteger[][]>>
     selfOthersGain;
    private static Map<Participant,List<BigInteger[][]>>
    selfCryptComp;
    private static Map<Participant,BigInteger[]> selfPlainGain;
    public  static List<BigInteger[]> gamas;
    public static BigInteger[][] betaI;
    public  static BigInteger[] tmp;
    public static BigInteger[] tmp2;//(l -t +1 )
    public static BigInteger[] val;
    public static BigInteger[][] sum;
    public static BigInteger[][] negGamaT;
    public static BigInteger[][] omegas;
    public static BigInteger[][] taus;

    private EncryptedBinaryComparator(){
        selfCryptComp = new HashMap<>();
        selfGain = new HashMap<>();
        selfPlainGain = new HashMap<>();
        selfOthersGain = new HashMap<>();
        othersGain = new HashMap<>();
    }

    public static EncryptedBinaryComparator
    getGainComparisonTool(){
```

```
        if(gainComparisonTool == null){
            gainComparisonTool = new EncryptedBinaryComparator()
;
        }
        return gainComparisonTool;
    }

    public static void setMyGain(BigInteger [][] encryptedGain,
BigInteger[] plainGain){
        if(encryptedGain == null)
            throw new NullPointerException("Null encryptedGain")
;
        l = encryptedGain.length;
        myGain = encryptedGain.clone();
        EncryptedBinaryComparator.plainGain = plainGain.clone();
    }
    public  static void setL(int l){
        EncryptedBinaryComparator.l = l;
    }
    public static void setMyGainMe(Participant p, BigInteger
[][] encryptedGain, BigInteger[] plainGain){
        if(encryptedGain == null)
            throw new NullPointerException("Null encryptedGain")
;
        selfGain.put(p,encryptedGain);
        selfPlainGain.put(p,plainGain);


    }



    public static void setOthersGain(Integer participantIndex,
BigInteger[][] gain){
        othersGain.put(participantIndex,gain);
    }

    public static void setOthersGainMe(Participant p,Integer
participantIndex, BigInteger[][] gain){
        if(selfOthersGain.get(p) == null){

        }
        othersGain.put(participantIndex,gain);
    }
    /**
     * Compares own gain with other's
     */
    public static void compareWithParticipants(){
        //List of encrypted bit by bit comparisons
        encryptedComparisonList = new ArrayList<>();
        //Procedure for candidate I
        for (Integer index: othersGain.keySet()) {

            //Computation of the gama's factors
```

```
            gamas = new ArrayList<>(l);
            //table of beta's
            betaI = othersGain.get(index).clone();
            for (int t = 0; t < l  ; t++) {
                tmp = homomorphicEncryption(betaI[t],myGain[t]);
                tmp2 = multHomomorphicEncryption(betaI[t],
    plainGain[t].multiply(BigInteger.valueOf(-2)));
                gamas.add(t,homomorphicEncryption(tmp, tmp2));
                //Un-comment and bring fields back in the method
                BigInteger expe1 = plainGain[t].add(decrypt(
    betaI[t])).mod(BigInteger.valueOf(2));
                BigInteger expe2 = plainGain[t].add(othersPlain[
    t]).mod(BigInteger.valueOf(2));
                BigInteger res = decrypt(gamas.get(t));

                if(!(othersPlain[t].equals(decrypt(betaI[t]))))
                    throw new IllegalStateException();
                if(!othersPlain[t].equals(decrypt(othersGain.get
    (index)[t])))
                    throw new IllegalStateException();
                if(!plainGain[t].equals(decrypt(myGain[t])))
                    throw new IllegalStateException();
                if(!expe1.equals(expe2))
                    throw new IllegalStateException();
                if(!expe2.equals(res))
                    throw new IllegalStateException();

            }
            /*
             * Optimisation begin with t = l so singe loop
             * Pull the creation outside the loops
             */
            sum = new BigInteger[l][2];
            negGamaT = new BigInteger[l][2];
            omegas = new BigInteger[l][2];
            taus = new BigInteger[l][2];
            List<BigInteger> plainGama = decryptMany(gamas);
            for (int t = l-1; t >= 0; t--) {
                val = ElGamal.encrypt(BigInteger.valueOf(l-t));
                tmp = ElGamal.getNegativeciphers(gamas.get(t));
                negGamaT[t] = multHomomorphicEncryption(tmp,
    BigInteger.valueOf(l-t));
                sum[t] = homomorphicEncryption(gamas.subList(t
    +1,l));
                omegas[t] = homomorphicEncryption(val, sum[t],
    negGamaT[t]);// E( l-t+1 + sum of (gamav - gmai) - gamai)
                taus[t] = homomorphicEncryption(omegas[t],
    myGain[t]);
                BigInteger expSum = BigInteger.valueOf(l-t);
                BigInteger expeNegGama = plainGama.get(t).
    multiply(expSum).negate();
                BigInteger expGamSum = SecureDotProductParty.
    vectorElementsSum((BigInteger[])plainGama.subList(t+1,l).
    toArray(new BigInteger[l-t-1]));
```

```
                BigInteger expeOmeg = expSum.add(expeNegGama).add(expGamSum);
                BigInteger expTau = expeOmeg.add(plainGain[t]);
                if(!compEncrypted(plainGama.get(t),(decrypt(gamas.get(t)))))
                    throw new IllegalStateException();
                if(!compEncrypted(expSum,decrypt(val)))
                    throw  new IllegalStateException();
                if(!compEncrypted(expeNegGama,decrypt(negGamaT[t])))
                    throw new IllegalStateException();
                if(!compEncrypted(expGamSum,decrypt(sum[t])))
                    throw new IllegalStateException();
                if(!compEncrypted(expeOmeg,decrypt(omegas[t])))
                    throw new IllegalStateException();
                if(!compEncrypted(expTau,decrypt(taus[t])))
                    throw new IllegalStateException();

            }
            encryptedComparisonList.add(taus);
        }
    }


    /**
     * Method that's compare two ElGamal encrypted numbers.
     * Computes the XOR bit-by-bit and finally.
     * Computation is done so that the first right most bit is set to Zero
     * Since the encryption Scheme is Homomorphic we can compute : M1+M2 also M1*M2
     * M1, M1 in F2
     * @param cipher1
     * @param cipher2
     */
    public static List<BigInteger[]> compareNumbers(BigInteger[] plain1, BigInteger[][] cipher1, BigInteger[][] cipher2){
        if (cipher1.length != cipher2.length){
            throw new IllegalArgumentException("Ciphers must have the same size");
        }
        checkInput(cipher1);
        checkInput(cipher2);
        int l = cipher1.length;
        //List of encrypted bit by bit comparisons



            //Computation of the gama's factors
            List<BigInteger[]> gamas = new ArrayList<>(l);
            //table of beta's
            BigInteger[] tmp;
            BigInteger [] tmp2;
            for (int t = 0; t <l ; t++) {
```

```
                tmp = homomorphicEncryption(cipher1[t],cipher2[t]);
                tmp2 = multHomomorphicEncryption(cipher2[t], plain1[t].multiply(BigInteger.valueOf(-2)));
                gamas.add(t,homomorphicEncryption(tmp,tmp2));
            }
            /*
             * Optimisation begin with t = l so singe loop
             * Pull the creation outside the loops
             */
            BigInteger[] val;//(l -t +1 )
            List<BigInteger[]> sum = new ArrayList<>(l);
            List<BigInteger[]> negGamaT = new ArrayList<>(l);
            List<BigInteger[]> omegas = new ArrayList<>(l);
            List<BigInteger[]> taus = new ArrayList<>(l);
            for (int t = l -1; t >= 0; t--) {
                val = ElGamal.encrypt(BigInteger.valueOf(l-t));
                tmp = ElGamal.getNegativeciphers(gamas.get(t));
                negGamaT.add(t,multHomomorphicEncryption(tmp, BigInteger.valueOf(l-t)));
                sum.add(t,homomorphicEncryption(gamas.subList(t+1,l)));
                omegas.add(t,homomorphicEncryption(val,sum.get(t),negGamaT.get(t)));// E( l-t+1 + sum of (gamav - gmai))
                taus.add(t,homomorphicEncryption(omegas.get(t), cipher1[t]));
            }
            return taus;


    }


    /**
     * Check the encrypted input
     * Mostly for size
     * @param cipher1
     */
    private static void checkInput(BigInteger[][] cipher1) {
        for (BigInteger[] c: cipher1) {
            if(c.length!= 2 || c[0] == null || c[1] == null){
                throw new IllegalArgumentException("Bad cipher as argument either size or one of cipher compoment is Null");
            }
        }
    }

    public static BigInteger[][] getEncryptedCompWithCandidate(Integer candidateIndex){
        return encryptedComparisonList.get(candidateIndex);
    }

    public static List<BigInteger[][]> getEncryptedComparisons()
    {
```

```
        return new ArrayList<>(encryptedComparisonList);
    }


    /**
     * Method for chained decryption(i.e Phase 8)
     * @param list V vector in the paoer
     * @param myIndex
     * @param privateKey
     * @param prime
     * @return
     */
    public static List<List<BigInteger[][]>> chainedDecryption(
List<List<BigInteger[][]>> list, int myIndex, BigInteger
privateKey, BigInteger prime, BigInteger group) {
        BigInteger r;
        for (int i = 0; i < list.size() ; i++) {
            if(i != myIndex){
                for (int j = 0; j < list.size() ; j++) {
                    r = new BigInteger(group.bitLength(),new
SecureRandom());
                    r = r.mod(group);
                    for (int k = 0; k <l ; k++) {
                        BigInteger ct = list.get(i).get(j)[k
][0];
                        BigInteger ct_prime = list.get(i).get(j)
[k][1];
                        ct = (ct.multiply((ct_prime.modPow(
privateKey,prime)).modInverse(prime))).modPow(r,prime);
                        list.get(i).get(j)[k][0] = ct;
                        list.get(i).get(j)[k][1] = ct_prime.
modPow(r,prime);
                    }
                    Collections.shuffle(Arrays.asList(list.get(i
).get(j)));
                }
            }
        }
        return list;
    }


    /**
     * Returns the binary representation of a number as an array
.
     * i.e if g = abcd with a,b,c,d in {0,1}
     * then the array is [d,c,b,a] and indexOf(d) = 0;
     * @param gain the number to convert
     * @return
     */
    public static BigInteger[] createBinaryArray(BigInteger gain
, int length){
        String t = (gain.toString(2));
        int l = t.length();
```

```
        BigInteger[] binaryVector = new BigInteger[length];
        for (int i = 0; i < length ; i++) {
            if(i < l) {
                binaryVector[i] = new BigInteger(String.valueOf(
    t.charAt(l - i - 1)));
            }
            else{
                binaryVector[i] = BigInteger.ZERO;
            }
        }
        return  binaryVector;
    }

    public static int finalDecryption(List<BigInteger[][]> comp)
    {
        ArrayList<BigInteger> res;
        int nb = 0;
        for (BigInteger[][] b: comp) {
            res = (ArrayList)decryptMany(Arrays.asList(b));
            for (BigInteger c: res) {
                    if(c.equals(BigInteger.ZERO)){
                        nb++;
                    }
                }
            }
        return nb + 1 ;
    }


    public static BigInteger[] getOthersPlain() {
        return othersPlain;
    }

    public static void setOthersPlain(BigInteger[] othersPlain)
    {
        EncryptedBinaryComparator.othersPlain = othersPlain;
    }

    private static boolean compEncrypted(BigInteger expected,
    BigInteger res){
        return (!expected.equals(BigInteger.ZERO) && res.equals(
    BigInteger.ONE) ||
                expected.equals(BigInteger.ZERO) && res.equals(
    BigInteger.ZERO));
    }

    public static int findDiffindex(BigInteger n1, BigInteger n2
    ){
        int length = Math.max(n1.bitLength(),n2.bitLength());
        BigInteger[] v1 = createBinaryArray(n1,length);
        BigInteger[] v2 = createBinaryArray(n2,length);
        int index = v1.length - 1;
        boolean found = false;
        while(!found){
```

```
            if(!v1[index].equals(v2[index])){
                found = true;
            }
            else{
                index--;
            }
        }
        return index;
    }
}
```

Once the comparison with all of the others gain is done, the participant proceed to decrypt the comparison in a chained fashion. And at last each Participant count the number of zeros yielded by comparing his gain with others and submit his ranking and his gain to the initiator if the number of zero is less than k, meaning that the participants is amongst the kth the best participants.

## 0.3 Simulation and results

¿¿¿SHOW CODE FOR A SMALL APPLICATION
¿¿¿PLOT DIFFERENT RUNNING TIMES AS FUNCTION OF ONE PARAMETER: number Participant, security parameter of ELGAMAL

## 0.4 Conclusion

¿¿¿¿DESCRIPTION OF WHAT CAN BE DONE IF OWNER IS DISHONEST
¿¿¿¿DESCRIPTION OF WHAT CAN BE DONE IF SOME PARTICIPANTS ARE DISHONEST AND COLLARBORATES
¿¿¿¿MENTION THE FACT THAT THERE COULD BE A POSSIBLE ATTACK ON THE DOT PRODUCT PROTOCOL SO THAT AT LEAT ONE PRIVATE ATTRIBUTE CAN BE LEARNED. QUOTE PAPER PROOVING IMPOSSIBILITY OF DISTRIBUTED SECURE DOT PRODUCT
¿¿¿¿THANK THE LAB FOR THE PROJECT AND HANDAN FOR A GOOD SUPERVISION.

## 0.5 Bibliography