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

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 others parties called ***participants***. The initiator usually has a vector of preferences on some attributes and 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 (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 ρ.
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 β_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 $\tilde{c}_t = c_t/(c'_t)^{x_j}$. Picks $r \leftarrow_R \mathbb{Z}_q$ and updates the ciphertext with $((\tilde{c}_t)^r, (c'_t)^r)$.
   - 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.~~

```
1  package lasecbachelorprject.epfl.ch.privacypreservinghousing.
       crypto;
2
3  import java.math.BigInteger;
4  import java.security.SecureRandom;
5
6  public class GroupGenerator {
7
8      private static int minBitLength;
9
10     private static int certainty;
11
12     private static BigInteger ONE = BigInteger.ONE;
13     private static BigInteger generator, prime, group;
14     private static SecureRandom secureRandom;
15
16     public GroupGenerator(int minBitLength, int certainty,
       SecureRandom secureRandom, boolean lengthCheck){
17         if(minBitLength < 1024 && lengthCheck )
18                 throw  new IllegalArgumentException("Prime
       should have at least 512 bits");
19         this.minBitLength = minBitLength;
20         this.certainty = certainty;
21         this.secureRandom = secureRandom;
22         getSafePrime();
23
24     }
25
26
27
28     private static void getSafePrime(){
29
30         BigInteger a;
31         do {
32             a = new BigInteger(minBitLength,secureRandom);
33
34             a = a.add(ONE);
35
36             group = new BigInteger(minBitLength,certainty,
       secureRandom);
37
38             prime = a.multiply(group).add(ONE);
39         }
40         while(!prime.isProbablePrime(certainty));
41
42
43
```

```
44          boolean isGen;
45          do{
46              isGen = true;
47              generator = new BigInteger(prime.bitLength(),
    secureRandom);
48              generator = generator.mod(prime.subtract(BigInteger.
    ONE)).add(BigInteger.ONE);
49              generator = generator.modPow(a, prime);
50              if(generator.equals(ONE)){
51                  isGen = false;
52              }
53          }while (!(isGen && !generator.equals(BigInteger.ZERO)));
54      }
55
56
57      public static BigInteger getPrime(){
58          if(prime == null)
59          {
60              getSafePrime();
61          }
62          return prime;
63      }
64      public  static BigInteger getGenerator(){
65          return  generator;
66      }
67
68      public  static BigInteger getGroup(){
69          return group;
70      }
71
72
73
74  }
```

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 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.

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

```
2
3  import java.math.BigInteger;
4  import java.security.SecureRandom;
5
6  import lasecbachelorprject.epfl.ch.privacypreservinghousing.
       crypto.SecureDotProductParty;
7  import lasecbachelorprject.epfl.ch.privacypreservinghousing.
       helpers.Poll;
8
9  import static java.lang.Math.ceil;
10 import static java.lang.Math.log;
11 import static java.lang.System.arraycopy;
12
13
14 public class Owner {
15     //TODO: Normal users with inheritance
16     public SecureDotProductParty me;
17     private BigInteger rho;
18     private int h;
19     private BigInteger[] criterionVector;
20     public Poll myPoll;
21
22
23
24     private BigInteger[] myAttVector; // 10 for test
25     private BigInteger[] myWeightVector;
26     private BigInteger[] wPrimeVector;
27     public int l;
28     private int k = 2 ; //for now
29     private int t = 5; // for now
30     private SecureRandom random;
31     private final int greaterLength;
32     private final int equalLength;
33
34     public Owner(BigInteger criterionEqualVector [],BigInteger[]
        criterionGreaterVector, BigInteger[] weightEqualVector,
       BigInteger[] weightGreaterVector){
35         greaterLength = criterionGreaterVector.length;
36         equalLength = criterionEqualVector.length;
37
38         l = (int) ceil(log(equalLength + criterionGreaterVector.
       length)) +
39                 criterionEqualVector[0].bitLength() +
40                 2*weightEqualVector[0].bitLength() + 2 ;
41         random = new SecureRandom();
42
43         do{
44             h = random.nextInt() % 20;
45         }while (h <= 0);
46         //TODO: Decomment following line
47         rho = (new BigInteger(h, random)).abs();
48
49         generateOwnerVprime(criterionEqualVector,
       weightGreaterVector,weightEqualVector);
```

```
50
51          me = new SecureDotProductParty(rho);
52          me.setMyvector(wPrimeVector);
53
54
55
56      }
57
58      private void generateOwnerVprime(BigInteger[]
        criterionEqVector, BigInteger[] weightEqualVector, BigInteger
        [] weightGreaterVector) {
59          /*Vector of size gr + 2 *eq*/
60          wPrimeVector = new BigInteger[greaterLength + 2*
        equalLength];
61
62          /*[vgT,..]*/
63          arraycopy(weightGreaterVector,0, wPrimeVector,0,
        greaterLength);
64
65          BigInteger v[] = new BigInteger[equalLength];
66          /* [vgT,(ve*ve)T,..]*/
67          for (int i = 0; i < equalLength ; i++) {
68              v[i] = weightEqualVector[i].negate();
69          }
70          arraycopy(v,0,wPrimeVector,greaterLength,equalLength);
71
72          for (int i = 0; i <equalLength; i++) {
73              v[i] = weightEqualVector[i].multiply(
        criterionEqVector[i]);
74              v[i] = v[i].add(v[i]);
75          }
76          arraycopy(v,0,wPrimeVector,greaterLength+equalLength,
        equalLength);
77          for (int i = 0; i < wPrimeVector.length ; i++) {
78              wPrimeVector[i] = wPrimeVector[i].multiply(rho);
79          }
80
81          myAttVector = new BigInteger[wPrimeVector.length];
82          for (int i = 0; i <myAttVector.length; i++) {
83              myAttVector[i] = wPrimeVector[i];
84          }
85      }
86
87      public void initiatePoll(int Participants){
88
89
90          myPoll = new Poll(Participants);
91
92      }
93
94
95      public BigInteger[] getMyAttVector() {
96          return myAttVector.clone();
97      }
```

```
98
99
100
101
102 }
```

Participant.java for participants.

```
1  package lasecbachelorprject.epfl.ch.privacypreservinghousing.
       user;
2
3
4  import java.math.BigInteger;
5  import java.security.SecureRandom;
6  import java.util.ArrayList;
7  import java.util.HashMap;
8  import java.util.List;
9  import java.util.Map;
10
11 import lasecbachelorprject.epfl.ch.privacypreservinghousing.
       Activities.Application;
12 import lasecbachelorprject.epfl.ch.privacypreservinghousing.
       crypto.ElGamal;
13 import lasecbachelorprject.epfl.ch.privacypreservinghousing.
       crypto.EncryptedBinaryComparator;
14 import lasecbachelorprject.epfl.ch.privacypreservinghousing.
       crypto.SecureDotProductParty;
15 import lasecbachelorprject.epfl.ch.privacypreservinghousing.
       crypto.ZeroKnowledgeProver;
16 import lasecbachelorprject.epfl.ch.privacypreservinghousing.
       crypto.ZeroKnowledgeVerifier;
17 import lasecbachelorprject.epfl.ch.privacypreservinghousing.
       helpers.DataBase;
18
19 import static lasecbachelorprject.epfl.ch.
       privacypreservinghousing.crypto.ElGamal.homomorphicEncryption
       ;
20 import static lasecbachelorprject.epfl.ch.
       privacypreservinghousing.crypto.ElGamal.
       multHomomorphicEncryption;
21 import static lasecbachelorprject.epfl.ch.
       privacypreservinghousing.crypto.EncryptedBinaryComparator.
       createBinaryArray;
22 import static lasecbachelorprject.epfl.ch.
       privacypreservinghousing.crypto.EncryptedBinaryComparator.
       finalDecryption;
23 import static lasecbachelorprject.epfl.ch.
       privacypreservinghousing.crypto.SecureDotProductParty.
       copyBigIntArrayToIntArray;
24
25 public class Participant extends Person implements User {
26
27
28     private final BigInteger prime;
29     private final BigInteger generator;
```

```
30      private final int k;
31      public SecureDotProductParty secureDotProduct;
32      public BigInteger[] wPrimeVector;
33      private BigInteger privateKey;
34      private BigInteger publicKey;
35      public ZeroKnowledgeProver prover;
36      public ZeroKnowledgeVerifier verifier;
37      private SecureRandom random;
38      private BigInteger group;
39      int replyEqSize;
40      int replyGrSize;
41      private BigInteger gain;
42      private BigInteger[][] cypher;
43      private int myCandidateNumber;
44      private static Map<Integer, BigInteger[][]> othersGain;
45      private static List<BigInteger[][]> encryptedComparisonList;
46      private List<List<BigInteger[][]>> list;
47      private int l;
48      private EncryptedBinaryComparator comparisonTool =
    EncryptedBinaryComparator.getGainComparisonTool();
49      private BigInteger commonKey;
50      private List<BigInteger[][]> finalComp;
51      private int ranking;
52
53
54      public Participant(BigInteger replyGr[], BigInteger[]
    replyEq,int l, int k) {
55          replyEqSize = replyEq.length;
56          replyGrSize = replyGr.length;
57          wPrimeVector = new BigInteger[replyGrSize + 2 *
    replyEqSize];
58          generateWPrimeVector(replyGr, replyEq);
59          secureDotProduct = new SecureDotProductParty();
60          secureDotProduct.setMyvector(wPrimeVector);
61          group = Application.group;
62          prime = Application.prime;
63          generator = Application.generator;
64          random = new SecureRandom();
65
66          prover = new ZeroKnowledgeProver(prime, group, generator
    );
67          verifier = new ZeroKnowledgeVerifier(prime, group,
    generator);
68          othersGain = new HashMap<>();
69          encryptedComparisonList = new ArrayList<>();
70          this.l = l;
71          this.k = k;
72      }
73
74      private void generateWPrimeVector(BigInteger replyGr[],
    BigInteger[] replyEq) {
75          copyBigIntArrayToIntArray(replyGr, 0, wPrimeVector, 0,
    replyGrSize);
76
```

```
77          BigInteger[] v = new BigInteger[replyEqSize];
78
79          for (int i = 0; i < replyEqSize; i++) {
80              v[i] = replyEq[i].pow(2);
81          }
82          copyBigIntArrayToIntArray(v, 0, wPrimeVector,
    replyGrSize, replyEqSize);
83          copyBigIntArrayToIntArray(replyEq, 0, wPrimeVector,
    replyEqSize + replyEqSize, replyEqSize);
84      }
85
86      public BigInteger[] getReplyVector() {
87          return wPrimeVector;
88      }
89
90
91      public void convertGain(int l) {
92          BigInteger two = BigInteger.ONE.add(BigInteger.ONE);
93          gain = secureDotProduct.getBeta().add(two.pow(l));
94          wPrimeVector = createBinaryArray(gain,l);
95      }
96
97      public void generatePrivateKey() {
98          privateKey = (new BigInteger(prime.bitLength(), random))
    .mod(prime);
99          publicKey = generator.modPow(privateKey, prime);
100         DataBase.getDataBase().publishElGamalPublicKey(this,
    publicKey);
101         prover.setX(privateKey);
102     }
103
104
105     public void setKeyToVerify(BigInteger key) {
106         verifier.setY(key);
107     }
108
109
110     public void encryptWithCommonkey() {
111         ElGamal.setCommonKey(DataBase.getEncryptionKey());
112         cypher = ElGamal.encryptMany(wPrimeVector);
113
114     }
115
116
117     public void setMyCandidateNumber(int number) {
118         myCandidateNumber = number;
119     }
120
121     public void sendEncryptedComparisonToDB() {
122         DataBase.pushComparisonVector(this, comparisonTool.
    getEncryptedComparisons());
123     }
124
125     public void getEpsilonVector(List<List<BigInteger[][]>> list
```

```
    ) {
126         this.list = comparisonTool.chainedDecryption(list,
    myCandidateNumber, privateKey, prime,group);
127 
128     }
129 
130     public void sendBackDecryptedListToDB() {
131         DataBase.pushPartialListDecryption(list);
132     }
133 
134 
135     public void participantGain(Integer participantIndex,
    BigInteger[][] gain) {
136         othersGain.put(participantIndex, gain);
137     }
138 
139     public void compareWithParticipants() {
140         //List of encrypted bit by bit comparisons
141         encryptedComparisonList = new ArrayList<>();
142         //Procedure for candidate I
143         for (Integer index : othersGain.keySet()) {
144 
145             //Computation of the gama's factors
146             List<BigInteger[]> gamas = new ArrayList<>(l);
147             //table of beta's
148             BigInteger[][] betaI = othersGain.get(index);
149             BigInteger[] tmp;
150             BigInteger[] tmp2;
151             for (int t = 0; t < l; t++) {
152                 tmp = homomorphicEncryption(betaI[t], cypher[t])
    ;
153                 tmp2 = multHomomorphicEncryption(betaI[t],
    wPrimeVector[t].multiply(BigInteger.valueOf(-2)));
154                 gamas.add(t, homomorphicEncryption(tmp, tmp2));
155             }
156             /*
157              * Optimisation begin with t = l so singe loop
158              * Pull the creation outside the loops
159              */
160             BigInteger[] val;//(l -t +1 )
161             BigInteger[][] sum = new BigInteger[l][2];
162             BigInteger[][] negGamaT = new BigInteger[l][2];
163             BigInteger[][] omegas = new BigInteger[l][2];
164             BigInteger[][] taus = new BigInteger[l][2];
165             for (int t = 0; t < l; t++) {
166                 val = ElGamal.encrypt(BigInteger.valueOf(l - t))
    ;
167                 tmp = ElGamal.getNegativeciphers(gamas.get(t));
168                 negGamaT[t] = multHomomorphicEncryption(tmp,
    BigInteger.valueOf(l - t));
169                 sum[t] = homomorphicEncryption(gamas.subList(t +
     1, l));
170                 omegas[t] = homomorphicEncryption(val, sum[t],
    negGamaT[t]);// E( l-t+1 + sum of (gamav - gmai))
```

```
171                     taus[t] = homomorphicEncryption(omegas[t], betaI
    [t]);
172                 }
173                 encryptedComparisonList.add(taus);
174             }
175             DataBase.pushComparisonVector(this,
    encryptedComparisonList);
176         }
177
178
179
180
181         public void receiveFinalComp(List<BigInteger[][]> comp) {
182             this.finalComp = comp;
183
184         }
185
186         private void computeSelfRanking(){
187             ranking = finalDecryption(finalComp);
188         }
189
190         private void rankingSubmission(){
191             if (ranking <= k){
192                 DataBase.submitsResults(this,ranking, wPrimeVector);
193             }
194         }
195
196 }
```

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

```
1  package lasecbachelorprject.epfl.ch.privacypreservinghousing.
       crypto;
2
3
4  import java.math.BigInteger;
5  import java.security.SecureRandom;
6
7  /*
8   * Class that a represent parties in secure dot product protocol
9   */
10
11 public class SecureDotProductParty  {
12
13     private BigInteger [][] Q;
14     private BigInteger  [][] X;
15     private BigInteger  factors[];
16     private BigInteger  qTimesX[][];
17     private BigInteger [] cPrime;
18     private BigInteger [] g;
19
20     private int  dDimension;
21     private SecureRandom secureRandom;
```

```
22      private int  sDimension;
23
24      private BigInteger  b;
25      private BigInteger [] c;
26      private int  rThRow;
27      private BigInteger  R1;
28      private BigInteger  R2;
29      private BigInteger  R3;
30
31      private BigInteger [] y;
32      private BigInteger  z;
33      private BigInteger  a;
34      private BigInteger  h;
35      private BigInteger  rho;
36      private BigInteger  rhoMax;
37      private BigInteger  beta;
38
39      public BigInteger [] getPrimeVector() {
40          return primeVector.clone();
41      }
42
43      private BigInteger [] primeVector;
44
45      public BigInteger  dotProduct;
46      public BigInteger gain;
47
48      //TODO: Copy the values so that the vector can't be modified
        in the outside
49      public SecureDotProductParty(BigInteger rho){
50
51          secureRandom = new SecureRandom();
52          this.rhoMax = new BigInteger(String.valueOf(rho));
53
54      }
55
56      public SecureDotProductParty(){
57          secureRandom = new SecureRandom();
58      }
59
60
61      //initiate The dot product for myVector
62      public void  initiateDotProduct(){
63
64
65
66          //TODO : Constant matrix dimension
67          sDimension = 100; //1 + secureRandom.nextInt(10);
68          rThRow = secureRandom.nextInt(sDimension);
69
70          //Genrate Q  and compute b. TODO: Skip rth row in the
      for an assigne later
71          Q = new BigInteger [sDimension][sDimension];
72          for (int i = 0; i < sDimension ; i++) {
73              for (int j = 0; j <sDimension ; j++) {
```

```
74                  //TODO: Correct BOUnd
75                      Q[i][j] = BigInteger.ONE;//new BigInteger(
    String.valueOf(secureRandom.nextInt(100)));
76
77              }
78          }
79
80
81          //TODO: b!!!
82          b = BigInteger.ZERO;
83          for (int i = 0; i < sDimension; i++) {
84              b = b.add(Q[i][rThRow]);// safeAdd(b,Q[i][rThRow]);
85          }
86          //Generate X
87          X = new BigInteger [sDimension][dDimension];
88
89          for (int i = 0; i <sDimension ; i++) {
90              for (int j = 0; j <dDimension ; j++) {
91                  if(i != rThRow){
92                      //TODO: Ccrrect next INT LIMIT
93                      X[i][j] = BigInteger.valueOf(secureRandom.
    nextInt(100)+1);
94                  }
95                  else{
96                      X[i][j] = primeVector[j];
97                      }
98                  }
99
100             }
101
102
103
104
105         //scalar factor to compute the "c" vector.
106         //factors[i] = sum(Qji).
107         factors = new BigInteger [sDimension];
108
109         factors[rThRow] = BigInteger.ZERO;
110         for (int i = 0; i <sDimension ; i++) {
111             factors[i] = BigInteger.ZERO;
112             if(i != rThRow) {
113                 for (int j = 0; j < sDimension; j++) {
114
115                     factors[i] =  factors[i].add(Q[j][i]);
116                 }
117             }
118         }
119
120         //Generate c
121         c = new BigInteger [dDimension];
122
123         for (int i = 0; i < dDimension ; i++) {
124             c[i] = BigInteger.ZERO;
125             for (int j = 0; j < sDimension ; j++) {
```

```
126                 c[i] = c[i].add(X[j][i].multiply(factors[j]));
127             }
128         }
129 
130         //Generate f
131         BigInteger [] f = new BigInteger [dDimension];
132         for (int i = 0; i <dDimension ; i++) {
133             f[i] = BigInteger.valueOf(secureRandom.nextInt(100)
    + 1);
134         }
135 
136         //TODO: Change the 1
137         R1 = BigInteger.valueOf(secureRandom.nextInt(100) +1);
138         R2 = BigInteger.valueOf(secureRandom.nextInt(100) +1);
139         R3 = BigInteger.valueOf(secureRandom.nextInt(100) +1);
140 
141         //Compute Q*X
142          qTimesX = new BigInteger [sDimension][dDimension];
143         for (int i = 0; i <sDimension ; i++) {
144             for (int j = 0; j <dDimension ; j++) {
145                 qTimesX[i][j] = BigInteger.ZERO;
146                 for (int k = 0; k <sDimension ; k++) {
147                     qTimesX[i][j] = qTimesX[i][j].add(Q[i][k].
    multiply(X[k][j]));
148                 }
149             }
150         }
151 
152         //c'
153         cPrime = new BigInteger [dDimension];
154         BigInteger  R1TimesR2 = R1.multiply(R2);
155         for (int i = 0; i <dDimension ; i++) {
156             cPrime[i] =  c[i].add(R1TimesR2.multiply(f[i]));
157         }
158 
159         BigInteger  R1TimesR3 = R1.multiply(R3);
160 
161         //Generate g
162         g = new BigInteger [dDimension];
163         for (int i = 0; i <dDimension ; i++) {
164             g[i] = R1TimesR3.multiply(f[i]);
165         }
166 
167     }
168 
169 
170 
171     public void sendInitialDataToOtherParty(
    SecureDotProductParty party){
172         party.receiveQTimesX(qTimesX,cPrime,g);
173     }
174 
175     public void sendAH(SecureDotProductParty party){
176         party.receiveAH(a, h);
```

```
177     }
178
179     private void receiveAH(BigInteger  a, BigInteger  h) {
180         this.a = a;
181         this.h = h;
182         beta = (a.add((h.multiply(R2)).divide(R3))).divide(b);
183     }
184
185     public void sendBeta(SecureDotProductParty party){
186         party.receiveBeta(beta);
187     }
188
189     public void sendAlpha(SecureDotProductParty party){
190         party.receiveAlpha(rho);
191     }
192     private void receiveBeta(BigInteger  beta) {
193         this.beta = beta;
194     }
195
196     public BigInteger  getBeta(){
197             return beta;
198     }
199     public BigInteger  getAlpha(){return rho;}
200
201
202
203     private void receiveQTimesX(BigInteger [][] qTimesX,
    BigInteger [] cPrime, BigInteger [] g) {
204        /* if(this.cPrime.length != primeVector.length || this.g.
    length != primeVector.length){
205             throw new IllegalArgumentException("The dot product
    can't be computed because of dimensions mismatch. Expected
    vector size: "+
206                                                 myvector.length
    + " received cPrime Size: " + (this.cPrime.length - 1) +"
    received g size: "+ (this.g.length -1));
207         }*/
208         this.qTimesX = qTimesX;
209         this.cPrime = cPrime;
210         this.g = g;
211         BigInteger [] rhoVector;
212
213
214         rho = new BigInteger(rhoMax.bitLength() -1, secureRandom
    );
215         rhoVector = primeVector.clone();
216         rhoVector [rhoVector.length - 1 ] = rho;
217
218         /*double[][] qTimesX = (double[][]) get("QtimesX");
219         double [] cPrime = (double[]) get("cPrime");
220         double [] g = (double[]) get("G");*/
221         y = computeY(qTimesX, rhoVector);
222         z = vectorElementsSum(y);
223         a = z.subtract(normalDotProduct(cPrime,rhoVector));
```

```
224         h = normalDotProduct(g,rhoVector);
225 
226         /*this.put("Y", y);
227         this.put("Z", z);
228         this.put("A", a);
229         this.put("H",h);*/
230 
231     }
232 
233 
234 
235 
236     private BigInteger [] computeY(BigInteger [][] qTimesX,
    BigInteger [] myVectorPrime) {
237         int dim = qTimesX.length;
238         BigInteger  y[] = new BigInteger [dim];
239         for (int i = 0; i <dim ; i++) {
240             y[i] = normalDotProduct(qTimesX[i], myVectorPrime);
241         }
242         return y;
243     }
244 
245     public static BigInteger  normalDotProduct(BigInteger [] v1,
     BigInteger [] v2 ){
246         BigInteger  res = BigInteger.ZERO;
247         for (int i = 0; i < v1.length ; i++) {
248             res = res.add(v1[i].multiply(v2[i]));
249         }
250         return res;
251     }
252 
253     private static BigInteger [] vectorAddition(BigInteger [] v1
    , BigInteger  v2[]){
254         int dim = v1.length;
255         BigInteger  [] v = new BigInteger [dim];
256         for (int i = 0; i < dim; i++) {
257             v[i] = v1[i].add(v2[i]);
258         }
259         return v;
260     }
261 
262     public static BigInteger[] vectorsElemMult(BigInteger[] v1,
    BigInteger[] v2){
263         BigInteger[] res = new BigInteger[v1.length];
264         for (int i = 0; i <v1.length ; i++) {
265             res[i] = v1[i].multiply(v2[i]);
266         }
267         return res;
268     }
269 
270     public static BigInteger[] vectorsElemMultMod(BigInteger[]
    v1, BigInteger[] v2,BigInteger prime){
271         BigInteger[] res = new BigInteger[v1.length];
272         for (int i = 0; i <v1.length ; i++) {
```

```
273             res[i] = (v1[i].multiply(v2[i])).mod(prime);
274         }
275         return res;
276     }
277 
278 
279 
280 
281     public static BigInteger  vectorElementsSum(BigInteger [] v1
    ){
282         int dim = v1.length;
283         BigInteger  res = BigInteger.ZERO;
284         for (int i = 0; i <dim; i++) {
285             res = res.add(v1[i]);
286         }
287         return res;
288     }
289 
290     public static BigInteger [] vectorScalarMult(BigInteger []
    vector, BigInteger  scalar){
291         BigInteger[] res = new BigInteger[vector.length];
292         for (int i = 0; i < vector.length ; i++) {
293             res[i] = vector[i].multiply(scalar);
294         }
295         return res;
296     }
297 
298     public static BigInteger[] vectorScalarMultMod(BigInteger[]
    vector, BigInteger scalar, BigInteger prime){
299         BigInteger[] res = new BigInteger[vector.length];
300         for (int i = 0; i < vector.length; i++) {
301             res[i] = (vector[i].multiply(scalar)).mod(prime);
302         }
303         return res;
304     }
305 
306     public static BigInteger[] vectorsElemExpo(BigInteger[] v,
    int expo){
307         for (int i = 0; i < v.length; i++) {
308             v[i] = v[i].pow(expo);
309         }
310         return v;
311     }
312 
313     public static BigInteger[] vectorsElemModExpo(BigInteger[] v
    , BigInteger expo, BigInteger prime){
314         BigInteger[] res = new BigInteger[v.length];
315         for (int i = 0; i < v.length; i++) {
316             res[i] = v[i].modPow(expo,prime);
317         }
318         return res;
319     }
320 
321 
```

```
322
323
324     public void setAlphaMax(BigInteger  alphaMax) {
325         this.rhoMax = alphaMax;
326     }
327
328
329     private void receiveAlpha(BigInteger  alpha) {
330         this.rho = alpha;
331     }
332
333
334     public void getPartialGain(int  l) {
335         BigInteger two = BigInteger.ONE.add(BigInteger.ONE);
336         gain = (new BigInteger(String.valueOf(dotProduct))).add(
    two.pow(l-1));
337     }
338
339     public void setMyvector(BigInteger [] vector) {
340         dDimension = vector.length + 1;
341         primeVector = new BigInteger [dDimension];
342         System.arraycopy(vector,0,primeVector,0,vector.length);
343         primeVector[vector.length] = BigInteger.ONE;
344
345     }
346
347     public static void copyBigIntArrayToIntArray(BigInteger[]
    src, int  srcPos,BigInteger [] des, int  destPos, int  number
    ){
348         for (int i = srcPos,j= destPos; i <srcPos+number ; i++,
    j++) {
349             des[j] = src[i];
350         }
351     }
352
353     public BigInteger  getRhoForParticipant() {
354         return rho;
355     }
356
357
358     static final long  safeAdd(long  left, long  right) {
359         if (right > 0 ? left > Long.MAX_VALUE - right
360                 : left < Long.MIN_VALUE - right) {
361             throw new ArithmeticException("Longoverflow");
362         }
363         return left + right;
364     }
365
366     static final long  safeSubtract(long  left, long  right) {
367         if (right > 0 ? left < Long.MIN_VALUE + right
368                 : left > Long.MAX_VALUE + right) {
369             throw new ArithmeticException("Longoverflow");
370         }
371         return left - right;
```

```
372     }
373
374     static final long safeMultiply(long left, long right) {
375         if (right > 0 ? left > Long.MAX_VALUE/right
376                 || left < Long.MIN_VALUE/right
377                 : (right < -1 ? left > Long.MIN_VALUE/right
378                 || left < Long.MAX_VALUE/right
379                 : right == -1
380                 && left == Long.MIN_VALUE) ) {
381             throw new ArithmeticException("Integer overflow");
382         }
383         return left * right;
384     }
385
386     static final long safeDivide(long left, long right) {
387         if ((left == Long.MIN_VALUE) && (right == -1)) {
388             throw new ArithmeticException("Integer overflow");
389         }
390         return left / right;
391     }
392
393     static final long safeNegate(long a) {
394         if (a == Long.MIN_VALUE) {
395             throw new ArithmeticException("Integer overflow");
396         }
397         return -a;
398     }
399     static final long safeAbs(long a) {
400         if (a == Integer.MIN_VALUE) {
401             throw new ArithmeticException("Integer overflow");
402         }
403         return Math.abs(a);
404     }
405
406
407
408 }
```

Unlinkeable Gain comparison NEED TO DESCRIBE PRECISELY UNLINKEABLE!!!! In this phase each participant $P_i, 1 \leq ~~j~~ i \leq n$ blindly compares his gain ~~to~~ others. ~~The gains are converted~~ to binary-array β ~~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.

```
1 package lasecbachelorprject.epfl.ch.privacypreservinghousing.
      crypto;
2
3 import java.math.BigInteger;
4 import java.security.SecureRandom;
```

```
5
6  public class ZeroKnowledgeProver {
7
8      private BigInteger x;
9      private BigInteger h;
10     private BigInteger z;
11     private BigInteger prime;
12     private BigInteger group;
13     private BigInteger generator;
14     private SecureRandom secureRandom;
15     public BigInteger r;
16
17     //TODO: Group should be the published key
18     public ZeroKnowledgeProver(BigInteger prime, BigInteger
       group, BigInteger generator){
19         this.prime = prime;
20         this.group = group;
21         this.generator = generator;
22         secureRandom = new SecureRandom();
23     }
24
25     public void initiateProof(){
26         generateH();
27     }
28     private void generateH(){
29         r = new BigInteger(group.bitLength(),secureRandom);
30         r = r.mod(group);
31         h = generator.modPow(r, prime);
32     }
33
34     public void sendH(ZeroKnowledgeVerifier verifier){
35         verifier.receiveH(h);
36     }
37
38     private void computeZ(BigInteger c){
39         z = (r.add(x.multiply(c))).mod(group);
40     }
41
42     public BigInteger sendZ(BigInteger c){
43         computeZ(c);
44         return z;
45     }
46
47
48     public void receiveC(BigInteger c) {
49         computeZ(c);
50     }
51
52     public void setX(BigInteger x){
53         this.x = x;
54     }
55 }
```

```
1  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

```
1  package lasecbachelorprject.epfl.ch.privacypreservinghousing.
       crypto;
2
3  import java.math.BigInteger;
4  import java.security.SecureRandom;
5  import java.util.ArrayList;
6  import java.util.Collection;
7  import java.util.HashMap;
8  import java.util.List;
9  import java.util.Map;
10
11 import lasecbachelorprject.epfl.ch.privacypreservinghousing.user
       .Participant;
12
13 public class ElGamal {
14
15
16     private static BigInteger prime,group,generator, privateKey,
        myPublicKey;
17     private static final BigInteger ONE = BigInteger.ONE;
18     private static ElGamal cryptoSystem;
19     private static SecureRandom secureRandom;
20     private static BigInteger commonKey;
21     //TODO: Remove after test
22     private static Map<Participant, BigInteger> mySecretKey;
23     private static Map<Participant, BigInteger> myPukey;
24
25     //TODO: fix group, generator. Add method for secret key
26
27     private ElGamal(BigInteger prime, BigInteger group,
       BigInteger generator){
28         mySecretKey = new HashMap<>();
29         myPukey= new HashMap<>();
30         this.prime = prime;
31         this.group = group;
32         this.generator = generator;
33         secureRandom = new SecureRandom();
```

```
34      }
35
36      public static ElGamal getElGamal(BigInteger prime,
        BigInteger group, BigInteger generator){
37          if (cryptoSystem == null){
38              cryptoSystem = new ElGamal(prime, group, generator);
39          }
40          return cryptoSystem;
41      }
42
43
44
45      public static BigInteger[] encrypt(BigInteger bit){
46          if(bit == null) {
47              throw new IllegalArgumentException("Message to
        encrypt is null");
48          }
49          BigInteger r = new BigInteger(group.bitLength(),
        secureRandom);
50          r = r.mod(group);
51          BigInteger gPowMessage = generator.modPow(bit,prime);
52          BigInteger yPowR = commonKey.modPow(r, prime);
53          BigInteger[] cipher = new BigInteger[2];
54          cipher[0] = gPowMessage.multiply(yPowR).mod(prime);
55          cipher[1] = generator.modPow(r, prime);
56          return cipher;
57      }
58
59      public static BigInteger[][] encryptMany(BigInteger[] bits){
60          BigInteger[][] res = new BigInteger[bits.length][2];
61          for (int i = 0; i <bits.length ; i++) {
62              res[i] = encrypt(bits[i]);
63          }
64          return res;
65      }
66
67      public static  void setPrivateKey(BigInteger privateKey){
68          if(privateKey == null){
69              throw  new IllegalArgumentException("Null Argument
        to Set private key");
70          }
71
72          ElGamal.privateKey = privateKey;
73          myPublicKey = generator.modPow(privateKey,prime);
74      }
75
76      public static void setMySKey(Participant p, BigInteger key){
77          mySecretKey.put(p,key);
78          myPukey.put(p,generator.modPow(key,prime));
79
80      }
81
82      public static BigInteger getMyPuKey(Participant p){
83          return myPukey.get(p);
```

```
84      }
85
86      public static void setCommonKey(BigInteger commonKey){
87          ElGamal.commonKey = commonKey;
88      }
89
90
91      public static BigInteger decrypt(BigInteger[] cypher){
92          if(cypher == null || cypher.length != 2|| cypher[0] ==
        null|| cypher[1] == null ){
93              throw new IllegalArgumentException("null or bad
        length cypher");
94          }
95
96          BigInteger msg = (cypher[0].multiply(cypher[1].modPow(
        privateKey.negate(),prime))).mod(prime);
97
98          return msg.equals(BigInteger.ONE) ? BigInteger.ZERO :
        BigInteger.ONE;
99      }
100
101     public static BigInteger decryptMe(Participant p, BigInteger
        [] cypher){
102         if(cypher == null || cypher.length != 2|| cypher[0] ==
        null|| cypher[1] == null ){
103             throw new IllegalArgumentException("null or bad
        length cypher");
104         }
105         setPrivateKey(mySecretKey.get(p));
106         BigInteger msg = (cypher[0].multiply(cypher[1].modPow(
        privateKey.negate(),prime))).mod(prime);
107
108         return msg.equals(BigInteger.ONE) ? BigInteger.ZERO :
        BigInteger.ONE;
109     }
110
111
112
113     public static List<BigInteger> decryptMany(List<BigInteger
        []> table){
114         ArrayList<BigInteger> res = new ArrayList<>(table.size()
        );
115         for (BigInteger[] t: table) {
116             res.add(decrypt(t));
117         }
118         return res;
119     }
120
121     public static Collection<BigInteger> decryptManyME(
        Participant p,Collection<BigInteger[]> table){
122         ArrayList<BigInteger> res = new ArrayList<>(table.size()
        );
123         for (BigInteger[] t: table) {
124             res.add(decryptMe(p,t));
```

```
125         }
126         return res;
127     }
128 
129     public static BigInteger getmyPublicKey(){
130         return  myPublicKey;
131     }
132 
133 
134     public static BigInteger[] homomorphicEncryption(Collection<
    BigInteger[]> ciphers){
135         BigInteger[] res = new  BigInteger[]{BigInteger.ONE,
    BigInteger.ONE};
136         for (BigInteger[] c: ciphers) {
137             res = SecureDotProductParty.vectorsElemMultMod(res,c
    ,prime);
138         }
139         return res;
140     }
141 
142     public static BigInteger[] getNegativeciphers(BigInteger[]
    cypher){
143         return SecureDotProductParty.vectorsElemModExpo(cypher,
    BigInteger.valueOf(Long.parseLong("-1")),prime);
144     }
145 
146     public static BigInteger[] homomorphicEncryption(BigInteger
    [] c1, BigInteger[] c2){
147         return SecureDotProductParty.vectorsElemMultMod(c1,c2,
    prime);
148     }
149 
150     public static BigInteger[] homomorphicEncryption(BigInteger
    []... ciphers){
151         BigInteger[] res = new BigInteger []{BigInteger.ONE,
    BigInteger.ONE};
152         for(BigInteger[] c : ciphers){
153             res = homomorphicEncryption(res,c);
154         }
155         return res;
156     }
157 
158     public static BigInteger[] multHomomorphicEncryption(
    BigInteger[]cipher, BigInteger otherWord){
159        return SecureDotProductParty.vectorsElemModExpo(cipher,
    otherWord,prime);
160     }
161 
162     public static BigInteger getPrime(){
163         return prime;
164     }
165 
166     public static BigInteger getGenerator(){
167         return getGenerator();
```

```
168     }
169
170     public static BigInteger getGroup(){
171         return group;
172     }
173
174 }
```

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 β_i gain yielding a new vector γ.
Next stage, the participant generates a new vector ω 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 τ where $\tau^t = \omega^t + \beta_i^t$ At the end of the process the τ 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 γ vector of stage one will have k-1 zeros in the last k-1 elements. This implies that the corresponding values of the ω vector will have values of at least 1 so will be the values of the corresponding τ 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 γ^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

```
1 package lasecbachelorprject.epfl.ch.privacypreservinghousing.
      crypto;
2
3 import java.math.BigInteger;
4 import java.security.SecureRandom;
5 import java.util.ArrayList;
6 import java.util.Arrays;
7 import java.util.Collections;
```

```
8  import java.util.HashMap;
9  import java.util.List;
10 import java.util.Map;
11
12 import lasecbachelorprject.epfl.ch.privacypreservinghousing.user
       .Participant;
13
14 import static lasecbachelorprject.epfl.ch.
       privacypreservinghousing.crypto.ElGamal.decrypt;
15 import static lasecbachelorprject.epfl.ch.
       privacypreservinghousing.crypto.ElGamal.decryptMany;
16 import static lasecbachelorprject.epfl.ch.
       privacypreservinghousing.crypto.ElGamal.homomorphicEncryption
       ;
17 import static lasecbachelorprject.epfl.ch.
       privacypreservinghousing.crypto.ElGamal.
       multHomomorphicEncryption;
18
19
20 public class EncryptedBinaryComparator {
21
22     private static BigInteger[][] myGain;
23     private static Map<Integer, BigInteger[][]> othersGain;
24     private static List<BigInteger[][]> encryptedComparisonList;
25     private static BigInteger[] plainGain;
26     private static BigInteger[] othersPlain;
27     private static int l;
28     private static EncryptedBinaryComparator gainComparisonTool;
29     private static Map<Participant,BigInteger[][]> selfGain;
30     private static Map<Participant,Map<Integer, BigInteger[][]>>
        selfOthersGain;
31     private static Map<Participant,List<BigInteger[][]>>
       selfCryptComp;
32     private static Map<Participant,BigInteger[]> selfPlainGain;
33     public  static List<BigInteger[]> gamas;
34     public static BigInteger[][] betaI;
35     public  static BigInteger[] tmp;
36     public static BigInteger[] tmp2;//(l -t +1 )
37     public static BigInteger[] val;
38     public static BigInteger[][] sum;
39     public static BigInteger[][] negGamaT;
40     public static BigInteger[][] omegas;
41     public static BigInteger[][] taus;
42
43     private EncryptedBinaryComparator(){
44         selfCryptComp = new HashMap<>();
45         selfGain = new HashMap<>();
46         selfPlainGain = new HashMap<>();
47         selfOthersGain = new HashMap<>();
48         othersGain = new HashMap<>();
49     }
50
51     public static EncryptedBinaryComparator
       getGainComparisonTool(){
```

```
52          if(gainComparisonTool == null){
53              gainComparisonTool = new EncryptedBinaryComparator()
    ;
54          }
55          return gainComparisonTool;
56      }
57
58      public static void setMyGain(BigInteger [][] encryptedGain,
    BigInteger[] plainGain){
59          if(encryptedGain == null)
60              throw new NullPointerException("Null encryptedGain")
    ;
61          l = encryptedGain.length;
62          myGain = encryptedGain.clone();
63          EncryptedBinaryComparator.plainGain = plainGain.clone();
64      }
65      public  static void setL(int l){
66          EncryptedBinaryComparator.l = l;
67      }
68      public static void setMyGainMe(Participant p, BigInteger
    [][] encryptedGain, BigInteger[] plainGain){
69          if(encryptedGain == null)
70              throw new NullPointerException("Null encryptedGain")
    ;
71          selfGain.put(p,encryptedGain);
72          selfPlainGain.put(p,plainGain);
73
74
75      }
76
77
78
79      public static void setOthersGain(Integer participantIndex,
    BigInteger[][] gain){
80          othersGain.put(participantIndex,gain);
81      }
82
83      public static void setOthersGainMe(Participant p,Integer
    participantIndex, BigInteger[][] gain){
84          if(selfOthersGain.get(p) == null){
85
86          }
87          othersGain.put(participantIndex,gain);
88      }
89      /**
90       * Compares own gain with other's
91       */
92      public static void compareWithParticipants(){
93          //List of encrypted bit by bit comparisons
94          encryptedComparisonList = new ArrayList<>();
95          //Procedure for candidate I
96          for (Integer index: othersGain.keySet()) {
97
98              //Computation of the gama's factors
```

```
99              gamas = new ArrayList<>(l);
100             //table of beta's
101             betaI = othersGain.get(index).clone();
102             for (int t = 0; t < l  ; t++) {
103                 tmp = homomorphicEncryption(betaI[t],myGain[t]);
104                 tmp2 = multHomomorphicEncryption(betaI[t],
    plainGain[t].multiply(BigInteger.valueOf(-2)));
105                 gamas.add(t,homomorphicEncryption(tmp, tmp2));
106                 //Un-comment and bring fields back in the method
107                 BigInteger expe1 = plainGain[t].add(decrypt(
    betaI[t])).mod(BigInteger.valueOf(2));
108                 BigInteger expe2 = plainGain[t].add(othersPlain[
    t]).mod(BigInteger.valueOf(2));
109                 BigInteger res = decrypt(gamas.get(t));
110
111                 if(!(othersPlain[t].equals(decrypt(betaI[t]))))
112                     throw new IllegalStateException();
113                 if(!othersPlain[t].equals(decrypt(othersGain.get
    (index)[t])))
114                     throw new IllegalStateException();
115                 if(!plainGain[t].equals(decrypt(myGain[t])))
116                     throw new IllegalStateException();
117                 if(!expe1.equals(expe2))
118                     throw new IllegalStateException();
119                 if(!expe2.equals(res))
120                     throw new IllegalStateException();
121
122             }
123             /*
124              * Optimisation begin with t = l so singe loop
125              * Pull the creation outside the loops
126              */
127             sum = new BigInteger[l][2];
128             negGamaT = new BigInteger[l][2];
129             omegas = new BigInteger[l][2];
130             taus = new BigInteger[l][2];
131             List<BigInteger> plainGama = decryptMany(gamas);
132             for (int t = l-1; t >= 0; t--) {
133                 val = ElGamal.encrypt(BigInteger.valueOf(l-t));
134                 tmp = ElGamal.getNegativeciphers(gamas.get(t));
135                 negGamaT[t] = multHomomorphicEncryption(tmp,
    BigInteger.valueOf(l-t));
136                 sum[t] = homomorphicEncryption(gamas.subList(t
    +1,l));
137                 omegas[t] = homomorphicEncryption(val, sum[t],
    negGamaT[t]);// E( l-t+1 + sum of (gamav - gmai) - gamai)
138                 taus[t] = homomorphicEncryption(omegas[t],
    myGain[t]);
139                 BigInteger expSum = BigInteger.valueOf(l-t);
140                 BigInteger expeNegGama = plainGama.get(t).
    multiply(expSum).negate();
141                 BigInteger expGamSum = SecureDotProductParty.
    vectorElementsSum((BigInteger[])plainGama.subList(t+1,l).
    toArray(new BigInteger[l-t-1]));
```

```
142                 BigInteger expeOmeg = expSum.add(expeNegGama).
    add(expGamSum);
143                 BigInteger expTau = expeOmeg.add(plainGain[t]);
144                 if(!compEncrypted(plainGama.get(t),(decrypt(
    gamas.get(t)))))
145                     throw new IllegalStateException();
146                 if(!compEncrypted(expSum,decrypt(val)))
147                     throw  new IllegalStateException();
148                 if(!compEncrypted(expeNegGama,decrypt(negGamaT[t
    ])))
149                     throw new IllegalStateException();
150                 if(!compEncrypted(expGamSum,decrypt(sum[t])))
151                     throw new IllegalStateException();
152                 if(!compEncrypted(expeOmeg,decrypt(omegas[t])))
153                     throw new IllegalStateException();
154                 if(!compEncrypted(expTau,decrypt(taus[t])))
155                     throw new IllegalStateException();
156 
157             }
158             encryptedComparisonList.add(taus);
159         }
160     }
161 
162 
163     /**
164      * Method that's compare two ElGamal encrypted numbers.
165      * Computes the XOR bit-by-bit and finally.
166      * Computation is done so that the first right most bit is
    set to Zero
167      * Since the encryption Scheme is Homomorphic we can compute
    : M1+M2 also M1*M2
168      * M1, M1 in F2
169      * @param cipher1
170      * @param cipher2
171      */
172     public static List<BigInteger[]> compareNumbers(BigInteger[]
    plain1, BigInteger[][] cipher1, BigInteger[][] cipher2){
173         if (cipher1.length != cipher2.length){
174             throw new IllegalArgumentException("Ciphers must
    have the same size");
175         }
176         checkInput(cipher1);
177         checkInput(cipher2);
178         int l = cipher1.length;
179         //List of encrypted bit by bit comparisons
180 
181 
182 
183             //Computation of the gama's factors
184             List<BigInteger[]> gamas = new ArrayList<>(l);
185             //table of beta's
186             BigInteger[] tmp;
187             BigInteger [] tmp2;
188             for (int t = 0; t <l ; t++) {
```

```
189                 tmp = homomorphicEncryption(cipher1[t],cipher2[t
    ]);
190                 tmp2 = multHomomorphicEncryption(cipher2[t],
    plain1[t].multiply(BigInteger.valueOf(-2)));
191                 gamas.add(t,homomorphicEncryption(tmp,tmp2));
192             }
193             /*
194              * Optimisation begin with t = l so singe loop
195              * Pull the creation outside the loops
196              */
197             BigInteger[] val;//(l -t +1 )
198             List<BigInteger[]> sum = new ArrayList<>(l);
199             List<BigInteger[]> negGamaT = new ArrayList<>(l);
200             List<BigInteger[]> omegas = new ArrayList<>(l);
201             List<BigInteger[]> taus = new ArrayList<>(l);
202             for (int t = l -1; t >= 0; t--) {
203                 val = ElGamal.encrypt(BigInteger.valueOf(l-t));
204                 tmp = ElGamal.getNegativeciphers(gamas.get(t));
205                 negGamaT.add(t,multHomomorphicEncryption(tmp,
    BigInteger.valueOf(l-t)));
206                 sum.add(t,homomorphicEncryption(gamas.subList(t
    +1,l)));
207                 omegas.add(t,homomorphicEncryption(val,sum.get(t
    ),negGamaT.get(t)));// E( l-t+1 + sum of (gamav - gmai))
208                 taus.add(t,homomorphicEncryption(omegas.get(t),
    cipher1[t]));
209             }
210             return  taus;
211
212
213     }
214
215
216     /**
217      * Check the encrypted input
218      * Mostly for size
219      * @param cipher1
220      */
221     private static void checkInput(BigInteger[][] cipher1) {
222         for (BigInteger[] c: cipher1) {
223             if(c.length!= 2 || c[0] == null || c[1] == null){
224                 throw new IllegalArgumentException("Bad cipher
    as argument either size or one of cipher compoment is Null");
225             }
226         }
227     }
228
229     public static BigInteger[][] getEncryptedCompWithCandidate(
    Integer candidateIndex){
230         return encryptedComparisonList.get(candidateIndex);
231     }
232
233     public static List<BigInteger[][]> getEncryptedComparisons()
    {
```

```
234         return new ArrayList<>(encryptedComparisonList);
235     }
236 
237 
238     /**
239      * Method for chained decryption(i.e Phase 8)
240      * @param list V vector in the paoer
241      * @param myIndex
242      * @param privateKey
243      * @param prime
244      * @return
245      */
246     public static List<List<BigInteger[][]>> chainedDecryption(
    List<List<BigInteger[][]>> list, int myIndex, BigInteger
    privateKey, BigInteger prime, BigInteger group) {
247         BigInteger r;
248         for (int i = 0; i < list.size() ; i++) {
249             if(i != myIndex){
250                 for (int j = 0; j < list.size() ; j++) {
251                     r = new BigInteger(group.bitLength(),new
    SecureRandom());
252                     r = r.mod(group);
253                     for (int k = 0; k <l ; k++) {
254                         BigInteger ct = list.get(i).get(j)[k
    ][0];
255                         BigInteger ct_prime = list.get(i).get(j)
    [k][1];
256                         ct = (ct.multiply((ct_prime.modPow(
    privateKey,prime)).modInverse(prime))).modPow(r,prime);
257                         list.get(i).get(j)[k][0] = ct;
258                         list.get(i).get(j)[k][1] = ct_prime.
    modPow(r,prime);
259                     }
260                     Collections.shuffle(Arrays.asList(list.get(i
    ).get(j)));
261                 }
262             }
263         }
264         return list;
265     }
266 
267 
268     /**
269      * Returns the binary representation of a number as an array
    .
270      * i.e if g = abcd with a,b,c,d in {0,1}
271      * then the array is [d,c,b,a] and indexOf(d) = 0;
272      * @param gain the number to convert
273      * @return
274      */
275     public static BigInteger[] createBinaryArray(BigInteger gain
    , int length){
276         String t = (gain.toString(2));
277         int l = t.length();
```

```
278         BigInteger[] binaryVector = new BigInteger[length];
279         for (int i = 0; i < length ; i++) {
280             if(i < l) {
281                 binaryVector[i] = new BigInteger(String.valueOf(
    t.charAt(l - i - 1)));
282             }
283             else{
284                 binaryVector[i] = BigInteger.ZERO;
285             }
286         }
287         return  binaryVector;
288     }
289
290     public static int finalDecryption(List<BigInteger[][]> comp)
    {
291         ArrayList<BigInteger> res;
292         int nb = 0;
293         for (BigInteger[][] b: comp) {
294             res = (ArrayList)decryptMany(Arrays.asList(b));
295             for (BigInteger c: res) {
296                     if(c.equals(BigInteger.ZERO)){
297                         nb++;
298                     }
299                 }
300             }
301         return nb + 1 ;
302     }
303
304
305     public static BigInteger[] getOthersPlain() {
306         return othersPlain;
307     }
308
309     public static void setOthersPlain(BigInteger[] othersPlain)
    {
310         EncryptedBinaryComparator.othersPlain = othersPlain;
311     }
312
313     private static boolean compEncrypted(BigInteger expected,
    BigInteger res){
314         return (!expected.equals(BigInteger.ZERO) && res.equals(
    BigInteger.ONE) ||
315                 expected.equals(BigInteger.ZERO) && res.equals(
    BigInteger.ZERO));
316     }
317
318     public static int findDiffindex(BigInteger n1, BigInteger n2
    ){
319         int length = Math.max(n1.bitLength(),n2.bitLength());
320         BigInteger[] v1 = createBinaryArray(n1,length);
321         BigInteger[] v2 = createBinaryArray(n2,length);
322         int index = v1.length - 1;
323         boolean found = false;
324         while(!found){
```

```
325                 if(!v1[index].equals(v2[index])){
326                     found = true;
327                 }
328                 else{
329                     index--;
330                 }
331             }
332             return index;
333         }
334 }
```

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