

# A Secure Set Intersection Protocol using Hamming Distance as a Comparator.

Dr. Amar Rasheed  
Armstrong State University  
amar.rasheed@armstrong.edu

Abraham Ladha  
Georgia Institute of Technology  
abrahimladha@protonmail.ch

August 31, 2016

# 1 Introduction

We propose a method of securely computing set intersections among parties using a hamming distance protocols as a comparator function.

## 2 Background

What exactly is cryptography? Cryptography is the practice and study of secure communications in the presence of adversaries. It is using Mathematics to secure information. Cryptography is very new and yet very old. Julius Caesar used to encrypt his messages he deemed of military significance using the *Caesar Cipher*, which shifted every letter over by three. The field in which Dr. Rasheed and I studied is called *Secure Multiparty Computation*. In a given system of  $n$  players, each player  $P_i$  has a secret input  $x_i$ . The players want to compute some  $f(x_1, x_2, \dots, x_n)$  while revealing no information about their inputs. A real world example would be if you have three co-workers, and they want to find out who has the highest salary without revealing their salaries to each other. This means we have  $n = 3$  players, and  $f(x_1, x_2, x_3) = \max(x_1, x_2, x_3)$ . A good MPC protocol satisfies two properties:

1. Input Privacy - No information about the players inputs should be able to be inferred during the execution of the protocol. The only information that should be inferred is whatever could have been seen by seeing the output of the function alone.
2. Correctness - No player or players who may deviate from the protocol should be able to force honest parties to output an incorrect result.

The problem we are trying to solve in this research is to construct a MPC protocol such that each player  $P_i$  has an input  $x_i$  that is some finite set. and  $f(x_1, x_2, \dots, x_n) = \bigcap_{i=1}^n x_i$ .

This has direct application. Consider the case where  $n$  robots wish to communicate. There are eleven frequency channels on which they may decide to send information. Not every robot can always access every channel, but they need to find out which channels they can communicate on. Each players input would be the set of frequencies that they are eligible to communicate on. In the case that a robot is compromised, no information of the other robots is compromised. Even if  $n - 1$  robots are compromised, no information is revealed about the  $n$ th robot.

Hamming Distance is a measurement between two bitstrings. It represents the number of substitutions required to make two numbers identical. For example, if  $X = 1001$  and  $Y = 1101$ , then  $d_h(X, Y) = 1$ . Not related to this research, hamming distances have a lot of unique properties. They form a metric space on  $\{0, 1\}^n$  (also known as a hamming space). A hamming space can be represented as a  $Q_n$  graph with where the hamming distance between any two elements is the shortest walk between their representative two vertices.

They have many applications in coding theory, graph theory, cryptography, and information theory.

Oblivious Transfer (OT) is a protocol type in which a sender transfers one of many pieces of information, but is oblivious as to what piece has been transferred.

Threshold homomorphic cryptosystems are systems that in order to decrypt an encrypted message, require the work of several parties is required. If there are  $n$  parties, and atleast  $t$  of them must aid in the decryption, then we call this a  $(t, n)$ -threshold scheme. As a more layman example, consider having  $n$  parties, and some  $n-1$  degree polynomial  $\sum_{i=1}^{n-1} a_i x^i$  where  $a_i$  is an element of some finite field and  $\langle a_1, a_2, \dots, a_n \rangle$ . is our message to decrypt (our shared secret). We give each player some unique point that is a solution to our polynomial. Clearly if they work together, by methods of Lagrange interpolation they can solve for the coefficients of our polynomial since they have  $n$  points points and the polynomial is defined as having degree  $n-1$ . However this requires all  $n$  parties. Even if  $n-1$  parties converse, there will be many possible solutions in our field. In the real numbers, given some polynomial of degree  $n-1$ , and if you don't have  $n$  or more points, then there exist infinite solutions for your coefficients of your polynomial. This can be considered a  $(n, n)$ -threshold scheme, and its called Shamir's Secret sharing scheme. [2]

### 3 Proposal

We propose a different solution than in [3]. Given  $n$  sets,  $x_1, \dots, x_n$ , we compute the intersections of two sets at a time. We compare each element of one set to every element of the other set. If the hamming distance of two elements compared is zero, this implies that they are identical, so they are added to a new temporary set. This temporary set is then added to the sets to compute the intersection of. If it is the case that some  $x_i \cap x_j = \emptyset$  for  $i < j$ , then we simply take  $x_i$  as it has higher precedence. For the secure hamming distance method, we borrow two protocols from [1]. known as the basic scheme and the fully secure scheme. They are both secure against different types of adversaries. The basic scheme is secure against semi-honest (passive) adversaries. This means we can assume adversaries cooperate to gather information and do not deviate from the protocol specification. The fully secure scheme is secure against malicious (active) adversaries. In this case they may deviate from the protocol specification and attempt to cheat, as the design of the protocol ensures its a futile endeavor.

### 4 Results

Due to some issues, The fully secure scheme does not run with out running out of memory. This will continue to be worked on as time progresses. The basic scheme is fully implemented and we have recorded and presented some

information based off its efficiency

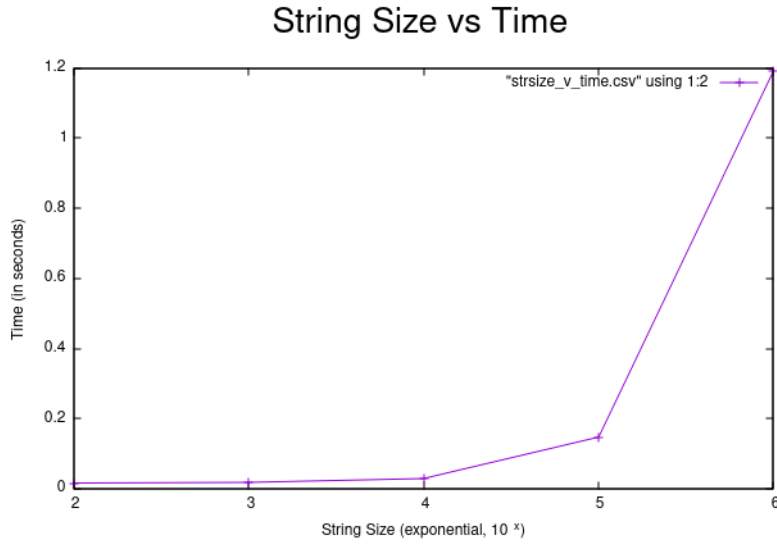


Figure 1: Time taken based on String length. x-scale is exponential to demonstrate that this is infact  $O(n)$  and not  $O(1)$ .

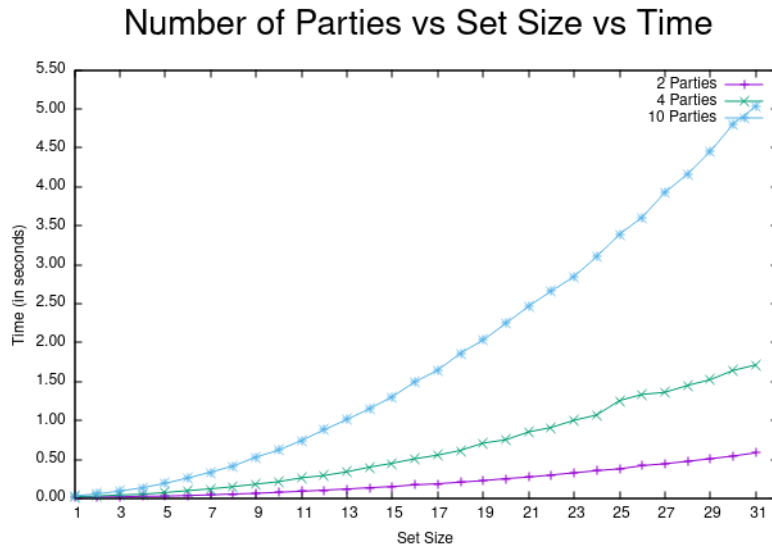


Figure 2: Time in seconds it takes to compute intersection of sets of different sizes. Shown here for 2, 4, and 10 parties

Since It takes  $O(n)$  time to process a string, and to process sets is a simple combinatorial algorithm running in  $O(n \log n)$ , the time for this algorithm to run in  $O(n^2 \log n^2)$

## 5 Appendices

Here we present the actual code. Please keep in mind files for the fully secure scheme are not implemented correctly but are presented for completion's sake.

basic.h

```

1 #ifndef BASIC_H
2 #define BASIC_H
3 #include <string>
4 #include <gmpxx.h>
5 #include <gmp.h>
6 #include <math.h>
7 #include <iostream>
8 #include <vector>
9 using namespace std;
10 vector<string> bitstring_generator(unsigned long n);
11 vector<unsigned long> string_to_vector(string s);
12 vector<string> bitstrings_of_length(vector<string> strings, int n);
13 unsigned long vector_sum(vector<unsigned long> v);
14 double vector_sumd(vector<double> v);
15 unsigned long basic_scheme(vector<unsigned long> X, vector<unsigned long> Y);
16 #endif

```

basic.cpp

```

1 #include <string>
2 #include <gmpxx.h>
3 #include <gmp.h>
4 #include <math.h>
5 #include <iostream>
6 #include <vector>
7 using namespace std;
8 /*
9  * returns a vector of all bitstrings up to a
10  * certain length
11  */
12 vector<string> bitstring_generator(unsigned long n){
13     vector<string> bitstrings(pow(2,n));
14     bitstrings.push_back("0");
15     bitstrings.push_back("1");
16     for(int j = 1; j < n; j++){
17         int x = bitstrings.size();
18         for(int i = 0; i < x; i++){
19             string temp1 = bitstrings[i] + "0";
20             string temp2 = bitstrings[i] + "1";
21             if((find(bitstrings.begin(), bitstrings.end(), temp1)) ==
22                bitstrings.end()){
23                 bitstrings.push_back(bitstrings[i] + "0");
24             }
25             if((find(bitstrings.begin(), bitstrings.end(), temp2)) ==
26                bitstrings.end()){
27                 bitstrings.push_back(bitstrings[i] + "1");
28             }
29         }
30     }
31     return bitstrings;
32 }
33 /*
34  * given some bitstring, it returns its

```

```

35  * vector equivalent
36  */
37  vector<unsigned long> string_to_vector(string s){
38      vector<unsigned long> Y(s.length());
39      for(int i = 0; i < Y.size(); i++){
40          Y[i] = s[i] - '0';
41      }
42      return Y;
43  }
44  /*
45  * filters a vector of bitstrings to only use those
46  * of a certain length
47  */
48  vector<string> bitstrings_of_length(vector<string> strings, int n){
49      vector<string> bitstrings;
50      for(int i = 0; i < strings.size(); i++){
51          if(strings[i].size() == n)
52              bitstrings.push_back(strings[i]);
53      }
54      return bitstrings;
55  }
56  /*
57  * computes sum of a vector of ulongs
58  */
59  unsigned long vector_sum(vector<unsigned long> v){
60      unsigned long sum;
61      for(int i = 0; i < v.size(); i++)
62          sum += v[i];
63      return sum;
64  }
65  /*
66  * computes sum of a vector of doubles
67  */
68  double vector_sumd(vector<double> v){
69      double sum;
70      for(int i = 0; i < v.size(); i++)
71          sum += v[i];
72      return sum;
73  }
74  /*
75  * given two vectors, returns their hamming distance
76  */
77  unsigned long basic_scheme(vector<unsigned long> X, vector<unsigned long> Y){
78      //tiny error check
79      if(X.size() != Y.size()){
80          cout << "vector sizes do not match" << endl;
81          cout << X.size() << "\t" << Y.size() << endl;
82          return 0;
83      }
84      unsigned long domain = X.size() + 1;
85      vector<unsigned long> r(X.size());
86      unsigned long seed = (unsigned long)time(NULL);
87      gmp_randstate_t rstate;
88      gmp_randinit_mt(rstate);
89      gmp_randseed_ui(rstate, seed);
90      mpz_class temp, n;
91      n = X.size() + 1; //could be bigger
92      //make r a vector of uniformly randoms
93      for(int i = 0; i < X.size(); i++){
94          mpz_urandomm(temp.get_mpz_t(), rstate, n.get_mpz_t());
95          r[i] = temp.get_ui();
96      }
97      //compute their sum
98      unsigned long R = vector_sum(r);
99      vector<unsigned long> t(X.size());
100     vector<unsigned long> tuple(2);
101     for(int i = 0; i < X.size(); i++){
102         tuple[0] = (r[i] + X[i]) % domain;

```

```

103         tuple[1] = (r[i] + (1 - X[i])) % domain;
104         t[i] = tuple[Y[i]]; //choosing
105     }
106     //compute their sum
107     unsigned long T = vector_sum(t);
108     //subtract off the random values
109     int total = T - R;
110     //some modular issues
111     while(total < 0){
112         total += (X.size() + 1);
113     }
114     return total;
115 }

```

#### main.cpp

```

1 #include <string>
2 #include <gmpxx.h>
3 #include <gmp.h>
4 #include <math.h>
5 #include <iostream>
6 #include <vector>
7 #include <fstream>
8 #include "basic.h"
9 using namespace std;
10 int main(void){
11     int n = 10; //how many parties?
12     ifstream in("inputn.txt"); //input file
13     ofstream out("output.txt"); //output file
14     vector<vector<string>> input(n);
15     string line;
16     int i = 0;
17     while(getline(in,line)){
18         if(line == "----"){ //our delimiter for sets
19             i++;
20         }
21         else{
22             input[i].push_back(line);
23         }
24     }
25     vector<string> output;
26     for(int k = 0; k < input.size() - 1; k++){
27         for(int i = 0; i < input[0].size(); i++){
28             for(int j = 0; j < input[1].size(); j++){
29                 vector<unsigned long> temp1 = string_to_vector(input[0][i]);
30                 vector<unsigned long> temp2 = string_to_vector(input[1][j]);
31                 if(basic_scheme(temp1,temp2) == 0){
32                     output.push_back(input[1][j]);
33                 }
34             }
35         }
36         if(output.size() != 0){
37             input.push_back(output);
38         }
39         else{
40             input.push_back(input[0]);
41         }
42         input.erase(input.begin(),input.begin()+1);
43         output.erase(output.begin(),output.end());
44     }
45     for(int k = 0; k < input[1].size(); k++){
46         out << input[1][k] << endl; //second set
47     }
48 }

```

#### elgamal\_v.cpp

```

1 | #include <string>
2 | #include <gmpxx.h>
3 | #include <gmp.h>
4 | #include <math.h>
5 | #include <iostream>
6 | #include <map>
7 | using namespace std;
8 | /*
9 |  * zero knowledge proof for elgamal
10 |  */
11 | bool elgamal_ff(mpz_class q, mpz_class h, mpz_class g, mpz_class b1,
12 |               mpz_class b2, mpz_class x1, mpz_class x2){
13 |     map <string, mpz_class> buffer;
14 |     mpz_class l, rand;
15 |     mpz_ui_pow_ui(l.get_mpz_t(), 2, 199);
16 |     unsigned long seed = (unsigned long)time(NULL);
17 |     gmp_randstate_t rstate;
18 |     gmp_randinit_mt(rstate);
19 |     gmp_randseed_ui(rstate, seed);
20 |     mpz_class u1, u2, div1, div2;
21 |
22 |     mpz_powm(div1.get_mpz_t(), g.get_mpz_t(), x1.get_mpz_t(), q.get_mpz_t());
23 |     u1 = b2 / div1;
24 |     //u1 = b2/(g^x1)
25 |
26 |     mpz_powm(div2.get_mpz_t(), g.get_mpz_t(), x2.get_mpz_t(), q.get_mpz_t());
27 |     u2 = b2 / div2;
28 |     //u1 = b2/(g^x1)
29 |
30 |     mpz_class v1, v2, c2, t1, t2;
31 |     mpz_urandomb(v1.get_mpz_t(), rstate, 310);
32 |     mpz_urandomb(v2.get_mpz_t(), rstate, 310);
33 |     mpz_urandomb(c2.get_mpz_t(), rstate, 310);
34 |     v1 %= q;
35 |     v2 %= q;
36 |     c2 %= q;
37 |     //random v1, v2, c2, in Zq
38 |
39 |     mpz_powm(t1.get_mpz_t(), h.get_mpz_t(), v1.get_mpz_t(), q.get_mpz_t());
40 |     //t1 = h^v1
41 |
42 |     mpz_class temp1, temp2;
43 |     mpz_powm(temp1.get_mpz_t(), u2.get_mpz_t(), c2.get_mpz_t(), q.get_mpz_t());
44 |     mpz_powm(temp2.get_mpz_t(), h.get_mpz_t(), v2.get_mpz_t(), q.get_mpz_t());
45 |     t2 = temp1 * temp2;
46 |     //t2 = u2^c2 * h^v2
47 |
48 |     mpz_class c;
49 |     mpz_urandomb(c.get_mpz_t(), rstate, 310);
50 |     c %= q;
51 |     mpz_class c1, r1, r2;
52 |     c1 = c - c2;
53 |     r1 = v1 - c1;
54 |     r2 = v2;
55 |
56 |     mpz_class temp3, temp4, temp5, temp6;
57 |     mpz_powm(temp5.get_mpz_t(), u2.get_mpz_t(), c2.get_mpz_t(), q.get_mpz_t());
58 |     mpz_powm(temp6.get_mpz_t(), h.get_mpz_t(), v2.get_mpz_t(), q.get_mpz_t());
59 |     mpz_powm(temp3.get_mpz_t(), u1.get_mpz_t(), c1.get_mpz_t(), q.get_mpz_t());
60 |     mpz_powm(temp4.get_mpz_t(), h.get_mpz_t(), r2.get_mpz_t(), q.get_mpz_t());
61 |
62 |     return ((c == c1 + c2) && (t2 == temp5 * temp6) && (t1 == temp3 * temp4))
63 |         ;
}

```

fully\_secure.h

```

1 | #ifndef FULLY_SECURE_H

```



```

2 #define FULLY_SECURE_H
3 #include <string>
4 #include <gmpxx.h>
5 #include <gmp.h>
6 #include <math.h>
7 #include <iostream>
8 #include <vector>
9 #include <fstream>
10 using namespace std;
11 mpz_class vector_sum(vector<mpz_class> v);
12 mpz_class fully_secure(vector<unsigned long> X, vector<unsigned long> Y);
13 #endif

```

### fully\_secure.cpp

```

1 #include <gmp.h>
2 #include <gmpxx.h>
3 #include <math.h>
4 #include <iostream>
5 #include <map>
6 #include <string>
7 #include <vector>
8 #include <fstream>
9 #include "elgamal.h"
10 #include "elgamal_v.h"
11 //somehow include elgamal as a commitment library
12 using namespace std;
13 mpz_class vector_sum(vector<mpz_class> v){
14     mpz_class sum;
15     for(int i = 0; i < v.size(); i++){
16         sum += v[i];
17     }
18     return sum;
19 }
20 mpz_class uniformly_random(mpz_class q){
21     unsigned long seed = (unsigned long)time(NULL);
22     gmp_randstate_t rstate;
23     gmp_randinit_mt(rstate);
24     gmp_randseed_ui(rstate,seed);
25     mpz_class g;
26     mpz_urandomm(g.get_mpz_t(),rstate,q.get_mpz_t());
27     return g;
28 }
29 mpz_class fully_secure(vector<unsigned long> X, vector<unsigned long> Y){
30     if(X.size() != Y.size()){
31         cout << "vector sizes do not match" << endl;
32         cout << X.size() << "\t" << Y.size() << endl;
33         return 0;
34     }
35
36     map<string,mpz_class> buffer;
37
38
39     /*
40      * I used random.org to pick a random mersenne prime, which is in base 10
41      * format, cited from http://bigprimes.net/pages/archive/mersenne/M21
42      * .txt
43
44      */
45     ifstream input_q("q.txt");
46     string q_str;
47     getline(input_q,q_str);
48     mpz_class q(q_str,10);
49     /*
50      *Random mersenne chosen by random.org
51      *http://bigprimes.net/pages/archive/mersenne/M21.txt
52      */

```

```

52 unsigned long seed = (unsigned long)time(NULL);
53 gmp_randstate_t rstate;
54 gmp_randinit_mt(rstate);
55 gmp_randseed_ui(rstate, seed);
56 mpz_class g;
57 mpz_urandomm(g.get_mpz_t(), rstate, q.get_mpz_t());
58
59 vector<vector<mpz_class>> com_X(X.size());
60 vector<vector<mpz_class>> com_Y(Y.size(), vector<mpz_class>(2, 0));
61
62 //get an elgamal tuple
63 vector<mpz_class> keys = keygen(g, q);
64 mpz_class h, x;
65 h = keys[0];
66 x = keys[1]; //secret
67 cout << "asd" << endl;
68 mpz_class temp_random;
69 mpz_class temp;
70 for(int i = 0; i < X.size(); i++){
71     //mpz_class temp;
72     //temp = Y[i];
73     mpz_set_ui(temp.get_mpz_t(), Y[i]);
74     com_Y[i] = encryption(temp, h, q, g);
75     //buffer["com_Y0" + to_string(i)] = com_Y[0][i];
76     //buffer["com_Y1" + to_string(i)] = com_Y[1][i];
77     //uncommenting these really likes to make GNU-MP run out of virtual
78     //memory. no idea why.
79
80     //generate a commitment for each xi, yi and add it to com_X and com_Y
81
82     cout << "a";
83     //mpz_urandomm(temp_random.get_mpz_t(), rstate, q.get_mpz_t());
84     temp_random = uniformly_random(q);
85     cout << "b";
86     mpz_class ytemp1, ytemp2;
87     cout << "c";
88     ytemp1 = Y[i];
89     cout << "d" << i;
90     ytemp2 = 1 - Y[i];
91     //cout << elgamal_v(q, h, g, temp_random, ytemp1, ytemp2) << endl;
92     //call elgamals verifier function on yi
93 }
94 vector<mpz_class> r(X.size());
95 for(int i = 0; i < X.size(); i++){
96     mpz_urandomm(r[i].get_mpz_t(), rstate, q.get_mpz_t());
97 }
98 cout << "peasche!" << endl;
99 mpz_class R = vector_sum(r);
100
101 mpz_class alpha, beta, tau, rho;
102 mpz_urandomm(alpha.get_mpz_t(), rstate, q.get_mpz_t());
103 mpz_urandomm(beta.get_mpz_t(), rstate, q.get_mpz_t());
104 mpz_urandomm(tau.get_mpz_t(), rstate, q.get_mpz_t());
105 mpz_urandomm(rho.get_mpz_t(), rstate, q.get_mpz_t());
106
107 cout << "ayylmao" << endl;
108 vector<vector<mpz_class>> ab(X.size(), vector<mpz_class>(2, 0));
109 vector<vector<mpz_class>> A(ab.size(), vector<mpz_class>(2, 0));
110 vector<vector<mpz_class>> B(ab.size(), vector<mpz_class>(2, 0));
111 for(int i = 0; i < ab.size(); i++){
112     //mpz_urandomm(r[i].get_mpz_t(), rstate, q.get_mpz_t());
113     ab[i][0] = (r[i] + X[i]) % q;
114     ab[i][1] = (r[i] + (1 - X[i])) % q;
115     A[i] = encryption(ab[i][0], alpha, q, g);
116     B[i] = encryption(ab[i][1], beta, q, g);
117     //buffer["A0" + to_string(i)] = A[0][i];
118     //buffer["A1" + to_string(i)] = A[1][i];
119     //buffer["B0" + to_string(i)] = B[0][i];

```

```

119         //buffer["B1" + to_string(i)] = B[1][i];
120     }
121     for(int i = 0; i < ab.size(); i++){
122         mpz_class temp_random;
123         mpz_urandomm(temp_random.get_mpz_t(), rstate, q.get_mpz_t());
124         cout << "ab" << i << endl;
125         cout << elgamal_v(q, h, g, temp_random, ab[0][i], ab[1][i]) << endl;
126         //call elgamal verifier method showing |bi = ai| = 1
127     }
128     cout << "widdle farther" << endl;
129     vector<mpz_class> t(ab.size());
130     vector<vector<mpz_class>> C(ab.size());
131     for(int i = 0; i < ab.size(); i++){
132         t[i] = ab[i][Y[i]];
133         C[i] = encryption(t[i], tau, q, g);
134         //buffer["C0" + to_string(i)] = C[0][i];
135         //buffer["C1" + to_string(i)] = C[1][i];
136     }
137     mpz_class T = vector_sum(t);
138
139     //option 1
140     vector<mpz_class> Ct(2); //our elgamal tuple
141     Ct[0] = 1;
142     Ct[1] = 1;
143     for(int i = 0; i < ab.size(); i++){
144         Ct[0] *= C[0][i];
145         Ct[1] *= C[1][i]; //since multiplying integers is the same as and-ing
                           //bitstrings
146     }
147     cout << "woah were far" << endl;
148     //buffer T
149     buffer["T"] = T;
150     //call proof again
151
152     vector<mpz_class> Cl(2);
153     Cl[0] = 1;
154     Cl[1] = 1;
155     for(int i = 0; i < ab.size(); i++){
156         Cl[0] *= buffer["C0" + to_string(i)];
157         Cl[1] *= buffer["C1" + to_string(i)];
158     }
159     bool a = Ct[0] == Cl[0];
160     bool b = Ct[1] == Cl[1];
161     if(a && b) cout << "yes" << endl;
162     else cout << "no" << endl;
163     //compute Ct again but with C from buffer and verify
164     return T - R;
165 }

```

## elgamal.h

```

1 #ifndef ELGAMAL_H
2 #define ELGAMAL_H
3 #include <string>
4 #include <gmpxx.h>
5 #include <gmp.h>
6 #include <math.h>
7 #include <iostream>
8 #include <map>
9 #include <vector>
10 #include <fstream>
11 using namespace std;
12 vector<mpz_class> keygen(mpz_class g, mpz_class q);
13 vector<mpz_class> encryption(mpz_class m, mpz_class h, mpz_class q, mpz_class
    g);
14 mpz_class decryption(vector<mpz_class> c, mpz_class x, mpz_class q);
15 #endif

```

# elgamal.cpp

```

1 #include <iostream>
2 #include <gmp.h>
3 #include <gmpxx.h>
4 #include <math.h>
5 #include <string>
6 #include <vector>
7 #include <map>
8 using namespace std;
9 //pick q and g, G is always Zq
10 vector<mpz_class> keygen(mpz_class g, mpz_class q){
11
12     unsigned long seed = (unsigned long)time(NULL);
13     gmp_randstate_t rstate;
14     gmp_randinit_mt(rstate);
15     gmp_randseed_ui(rstate,seed);
16     mpz_class x;
17     mpz_urandomm(x.get_mpz_t(),rstate,q.get_mpz_t());
18     //randomly pick x from G
19     mpz_class h;
20     mpz_powm(h.get_mpz_t(),g.get_mpz_t(),x.get_mpz_t(),q.get_mpz_t());
21     //h = g^x
22     vector<mpz_class> keys(2);
23     keys[0] = h;
24     keys[1] = x;
25     return keys;
26     //keys[0] is public, keep keys[1] secret
27     //publish h as pubkey
28     //x is private key and is kept secret
29 }
30 vector<mpz_class> encryption(mpz_class m, mpz_class h, mpz_class q, mpz_class
    g){
31     unsigned long seed = (unsigned long)time(NULL);
32     gmp_randstate_t rstate;
33     gmp_randinit_mt(rstate);
34     gmp_randseed_ui(rstate,seed);
35     mpz_class y;
36     mpz_urandomm(y.get_mpz_t(),rstate,q.get_mpz_t());
37     //randomly pick y from G
38     mpz_class c1;
39     mpz_powm(c1.get_mpz_t(),g.get_mpz_t(),y.get_mpz_t(),q.get_mpz_t());
40     //c1 = g^y
41
42     mpz_class s;
43     mpz_powm(s.get_mpz_t(),h.get_mpz_t(),y.get_mpz_t(),q.get_mpz_t());
44     //s = h^y
45     mpz_class c2;
46     c2 = m*s;
47     //c2 = m * s
48
49     //(c1,c2) is sent as a tuple
50     vector<mpz_class> c(2);
51     c[0] = c1;
52     c[1] = c2;
53
54     return c;
55 }
56 mpz_class decryption(vector<mpz_class> c, mpz_class x, mpz_class q){
57     mpz_class c1 = c[0];
58     mpz_class c2 = c[1];
59
60     mpz_class s;
61     mpz_powm(s.get_mpz_t(),c1.get_mpz_t(),x.get_mpz_t(),q.get_mpz_t());
62     //s = c1^x
63
64     mpz_class inv_s;
65     inv_s = q - s;

```

```

66 //get inverse of s from s?
67 mpz_class m;
68 m = c2 * inv_s;
69
70 return m;
71 //m = c2*s^(-1) thats modular multiplicative inverse of s
72 //s^-1 = (c1^-1)^x
73 //return m
74 }

```

# crypto.cpp

```

1 #include <string>
2 #include <gmpxx.h>
3 #include <gmp.h>
4 #include <math.h>
5 #include <iostream>
6 #include <vector>
7 #include <map>
8 #include <fstream>
9 using namespace std;
10 //predetermine N as some large ass prime of mpz_class
11 vector<mpz_class> keygen(){
12     vector<mpz_class> c;
13     //should return a fully secure e and d as a tuple in c
14     //multiplicative inverse of each other in Zn
15 }
16 void OT(*y a bit*/mpz_class m0,mpz_class m1, unsigned long y){
17     //gen buffer
18
19     mpz_class N; //temp
20     unsigned long seed = (unsigned long)time(NULL);
21     gmp_randstate_t rstate;
22     gmp_randinit_mt(rstate);
23     gmp_randseed_ui(rstate,seed);
24
25     mpz_class x0, x1;
26     mpz_urandomm(x0.get_mpz_t(),rstate,N.get_mpz_t());
27     mpz_urandomm(x1.get_mpz_t(),rstate,N.get_mpz_t());
28     mpz_class e, d;
29     //e, x0, x1 to buffer
30
31     mpz_class k;
32     mpz_urandomm(k.get_mpz_t(),rstate,N.get_mpz_t());
33     unsigned long b = y;
34     mpz_class v;
35     mpz_class temp;
36     mpz_powm(temp.get_mpz_t(),k.get_mpz_t(),e.get_mpz_t(),N.get_mpz_t());
37     v = (((b == 0)? x0 : x1) + temp)%N;
38     //push v, also buffer xinputs
39     mpz_class k0, k1,temp2,temp3;
40     mpz_powm(temp2.get_mpz_t(),x0.get_mpz_t(),d.get_mpz_t(),N.get_mpz_t());
41     mpz_powm(temp3.get_mpz_t(),x1.get_mpz_t(),d.get_mpz_t(),N.get_mpz_t());
42     k0 = (v - temp2)%N;
43     k1 = (v - temp3)%N;
44     mpz_class n0, n1;
45     n0 = k0 + m0;
46     n1 = k1 + m1;
47     //buffer n0,n1;
48
49     cout << ((b == 0)? n0 : n1 ) - k;
50     //n0,n1 from buffer
51 }

```

## References

- [1] Julien Bringer, Herve Chabann, Alain Patey *SHADE: Secure HAMming Distance computation from oblivious transfer*. 2012.
- [2] Adi Shamir *How to Share a Secret*. 1979.
- [3] Benny Pinkas, Thomas Schneider, Michael Zohner *Faster Private Set Intersection based on OT Extension* 2014.
- [4] Mehmet S. Kiraz,, Berry Schoenmakers, Jos Villegas *Efficient Committed Oblivious Transfer of Bit Strings*. 2007.