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

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

Abrahim Ladha
Georgia Institute of Technology
abrahimladha@protonmail.ch

August 30, 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 also 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 Mulitparty Computation*. In a given system of n players, each player P_i has a secret input x_i. The players want to compute some $f(x_1, x_2, ..., x_n)$ 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 $f(x_1, x_2, ..., x_n) = \bigcap_{i=1}^{n} x_i$.

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 comprised, no information of the other robots is compromised. Even if $n - 1$ robots are compromised, no information is revealed about the nth robot.

Hamming Distance is a measurement between two bitstrings. It represents the number of substituitions 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 $\Sigma_{i=1}^{n-1} a_i x^i$ where a_i is an element of some finite field and $\langle a_1, a_2, ..., a_n \rangle$. 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. (CITE)

3 Proposal

We propose a different solution than in (cite polynomial guy). Given n sets, $x_1, ..., x_n$, 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 is then added to the sets to compute the intersection of. If is it 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 CITE and CITE. 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

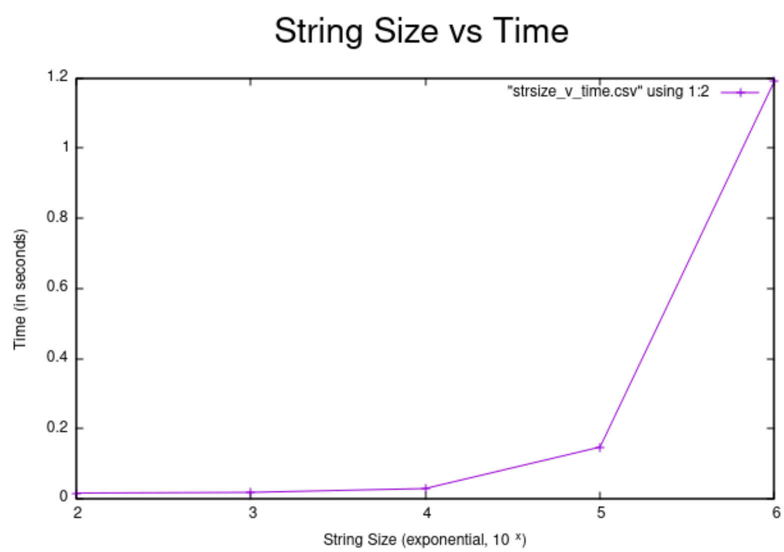

Figure 1: Time taken based on String length. x-scale is exponential to demonstrate that this is infact $O(n)$ and not $O(1)$.

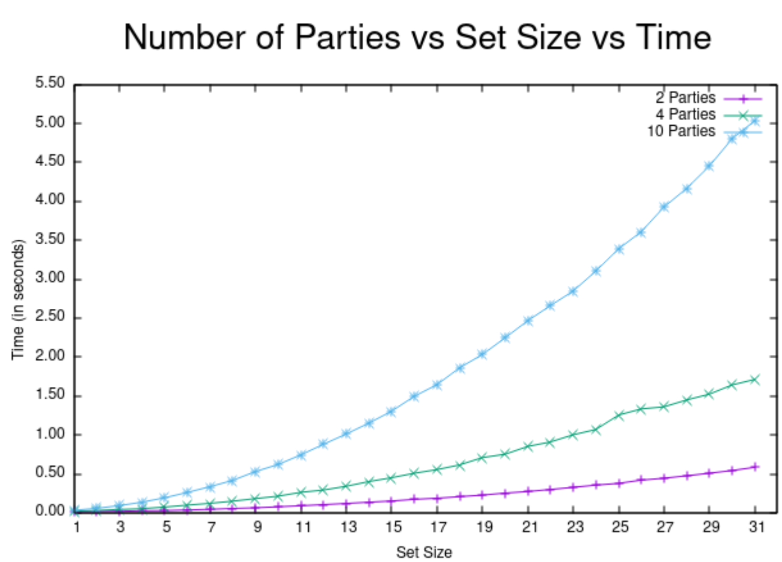

Figure 2: Time in seconds it takes to compute intersection of sets of different sizes. Shown here for 2, 4, and 10 parties

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

```
#ifndef BASIC_H
#define BASIC_H
#include <string>
#include <gmpxx.h>
#include <gmp.h>
#include <math.h>
#include <iostream>
#include <vector>
using namespace std;
vector<string> bitstring_generator(unsigned long n);
vector<unsigned long> string_to_vector(string s);
vector<string> bitstrings_of_length(vector<string> strings, int n);
unsigned long vector_sum(vector<unsigned long> v);
double vector_sumd(vector<double> v);
unsigned long basic_scheme(vector<unsigned long> X, vector<unsigned long> Y);
#endif
```

basic.cpp

```
#include <string>
#include <gmpxx.h>
#include <gmp.h>
#include <math.h>
#include <iostream>
#include <vector>
using namespace std;
/*
 * returns a vector of all bitstrings up to a
 * certain length
*/
vector<string> bitstring_generator(unsigned long n){
    vector<string> bitstrings(pow(2,n));
    bitstrings.push_back("0");
    bitstrings.push_back("1");
    for(int j = 1; j < n; j++){
        int x = bitstrings.size();
        for(int i = 0; i < x; i++){
            string temp1 = bitstrings[i] + "0";
            string temp2 = bitstrings[i] + "1";
            if((find(bitstrings.begin(), bitstrings.end(),temp1)) ==
            bitstrings.end()){
                bitstrings.push_back(bitstrings[i] + "0");
            }
            if((find(bitstrings.begin(), bitstrings.end(),temp2)) ==
            bitstrings.end()){
                bitstrings.push_back(bitstrings[i] + "1");
            }
        }
    }
    return bitstrings;
}
/*
 * given some bitstring, it returns its
```

```
 * vector equivalent
 */
vector<unsigned long> string_to_vector(string s){
     vector<unsigned long> Y(s.length());
    for(int i = 0; i < Y.size(); i++){
        Y[i] = s[i] - '0';
    }
    return Y;
}
/*
 * filters a vector of bitstrings to only use those
 * of a certain length
 */
vector<string> bitstrings_of_length(vector<string> strings, int n){
     vector<string> bitstrings;
    for(int i = 0; i < strings.size(); i++){
        if(strings[i].size() == n)
            bitstrings.push_back(strings[i]);
    }
    return bitstrings;
}
/*
 * computes sum of a vector of ulongs
 */
unsigned long vector_sum(vector<unsigned long> v){
    unsigned long sum;
    for(int i = 0; i < v.size(); i++)
        sum += v[i];
    return sum;
}
/*
 * computes sum of a vector of doubles
 */
double vector_sumd(vector<double> v){
    double sum;
    for(int i = 0; i < v.size(); i++)
        sum += v[i];
    return sum;
}
/*
 * given two vectors, returns their hamming distance
 */
unsigned long basic_scheme(vector<unsigned long> X, vector<unsigned long> Y){
    //tiny error check
    if(X.size() != Y.size()){
        cout << "vector sizes do not match" << endl;
        cout << X.size()  << "\t" << Y.size() << endl;
        return 0;
    }
    unsigned long domain = X.size() + 1;
    vector<unsigned long> r(X.size());
    unsigned long seed = (unsigned long)time(NULL);
    gmp_randstate_t rstate;
    gmp_randinit_mt(rstate);
    gmp_randseed_ui(rstate,seed);
    mpz_class temp, n;
    n = X.size() + 1; //could be bigger
    //make r a vector of uniformly randoms
    for(int i = 0; i < X.size(); i++){
        mpz_urandomm(temp.get_mpz_t(),rstate,n.get_mpz_t());
        r[i] = temp.get_ui();
    }
    //compute their sum
    unsigned long R = vector_sum(r);
    vector<unsigned long> t(X.size());
    vector<unsigned long> tuple(2);
    for(int i = 0; i < X.size(); i++){
        tuple[0] = (r[i] + X[i]) % domain;
```

```
        tuple[1] = (r[i] + (1 - X[i])) % domain;
        t[i] = tuple[Y[i]]; //choosing
    }
    //comute their sum
    unsigned long T = vector_sum(t);
    //subtract off the random values
    int total = T - R;
    //some modular issues
    while(total < 0){
        total += (X.size() + 1);
    }
    return total;
}
```

main.cpp

```
#include <string>
#include <gmpxx.h>
#include <gmp.h>
#include <math.h>
#include <iostream>
#include <vector>
#include <fstream>
#include "basic.h"
using namespace std;
int main(void){
    int n = 10; //how many parties?
    ifstream in("inputn.txt"); //input file
    ofstream out("output.txt"); //output file
    vector<vector<string>> input(n);
    string line;
    int i = 0;
    while(getline(in,line)){
        if(line == "----"){ //our delimiter for sets
            i++;
        }
        else{
            input[i].push_back(line);
        }
    }
    vector<string> output;
    for(int k = 0; k < input.size() - 1; k++){
        for(int i =  0; i < input[0].size(); i++){
            for(int j = 0; j < input[1].size(); j++){
                vector<unsigned long> temp1 = string_to_vector(input[0][i]);
                vector<unsigned long> temp2 = string_to_vector(input[1][j]);
                if(basic_scheme(temp1,temp2) == 0){
                    output.push_back(input[1][j]);
                }
            }
        }
        if(output.size() != 0){
            input.push_back(output);
        }
        else{
            input.push_back(input[0]);
        }
        input.erase(input.begin(),input.begin()+1);
        output.erase(output.begin(),output.end());
    }
    for(int k = 0; k < input[1].size(); k++){
        out << input[1][k] << endl; //second set
    }
}
```

elgamal_v.cpp

```
#include <string>
#include <gmpxx.h>
#include <gmp.h>
#include <math.h>
#include <iostream>
#include <map>
using namespace std;
/*
 * zero knowledge proof for elgamal
 */
bool elgamal_ff(mpz_class q, mpz_class h, mpz_class g, mpz_class b1,
        mpz_class b2, mpz_class x1, mpz_class x2){
    map <string, mpz_class> buffer;
    mpz_class l, rand;
    mpz_ui_pow_ui(l.get_mpz_t(),2,199);
    unsigned long seed = (unsigned long)time(NULL);
    gmp_randstate_t rstate;
    gmp_randinit_mt(rstate);
    gmp_randseed_ui(rstate,seed);
    mpz_class u1, u2, div1, div2;

    mpz_powm(div1.get_mpz_t(),g.get_mpz_t(),x1.get_mpz_t(),q.get_mpz_t());
    u1 = b2 / div1;
    //u1 = b2/(g^x1)

    mpz_powm(div2.get_mpz_t(),g.get_mpz_t(),x2.get_mpz_t(),q.get_mpz_t());
    u2 = b2 / div2;
    //u1 = b2/(g^x1)

    mpz_class v1, v2, c2, t1, t2;
    mpz_urandomb(v1.get_mpz_t(),rstate,310);
    mpz_urandomb(v2.get_mpz_t(),rstate,310);
    mpz_urandomb(c2.get_mpz_t(),rstate,310);
    v1 %= q;
    v2 %= q;
    c2 %= q;
    //random v1, v2, c2, in Zq

    mpz_powm(t1.get_mpz_t(),h.get_mpz_t(),v1.get_mpz_t(),q.get_mpz_t());
    //t1 = h^v1

    mpz_class temp1, temp2;
    mpz_powm(temp1.get_mpz_t(),u2.get_mpz_t(),c2.get_mpz_t(),q.get_mpz_t());
    mpz_powm(temp2.get_mpz_t(),h.get_mpz_t(),v2.get_mpz_t(),q.get_mpz_t());
    t2 = temp1 * temp2;
    //t2 = u2^c2 * h^v2

    mpz_class c;
    mpz_urandomb(c.get_mpz_t(),rstate,310);
    c %= q;
    mpz_class c1, r1, r2;
    c1 = c - c2;
    r1 = v1 - c1;
    r2 = v2;

    mpz_class temp3,temp4,temp5, temp6;
    mpz_powm(temp5.get_mpz_t(),u2.get_mpz_t(),c2.get_mpz_t(),q.get_mpz_t());
    mpz_powm(temp6.get_mpz_t(),h.get_mpz_t(),v2.get_mpz_t(),q.get_mpz_t());
    mpz_powm(temp3.get_mpz_t(),u1.get_mpz_t(),c1.get_mpz_t(),q.get_mpz_t());
    mpz_powm(temp4.get_mpz_t(),h.get_mpz_t(),r2.get_mpz_t(),q.get_mpz_t());

    return ((c == c1 + c2) && (t2 == temp5 * temp6) && (t1 == temp3 * temp4))
        ;
}
```

fully_secure.h

```
#ifndef FULLY_SECURE_H
```

```
#define FULLY_SECURE_H
#include <string>
#include <gmpxx.h>
#include <gmp.h>
#include <math.h>
#include <iostream>
#include <vector>
#include <fstream>
using namespace std;
mpz_class vector_sum(vector<mpz_class> v);
mpz_class fully_secure(vector<unsigned long> X, vector<unsigned long> Y);
#endif
```

fully_secure.cpp

```
#include <gmp.h>
#include <gmpxx.h>
#include <math.h>
#include <iostream>
#include <map>
#include <string>
#include <vector>
#include <fstream>
#include "elgamal.h"
#include "elgamal_v.h"
//somehow include elgamal as a commitment library
using namespace std;
mpz_class vector_sum(vector<mpz_class> v){
    mpz_class sum;
    for(int i = 0; i < v.size(); i++){
        sum += v[i];
    }
    return sum;
}
mpz_class uniformly_random(mpz_class q){
    unsigned long seed = (unsigned long)time(NULL);
    gmp_randstate_t rstate;
    gmp_randinit_mt(rstate);
    gmp_randseed_ui(rstate,seed);
    mpz_class g;
    mpz_urandomm(g.get_mpz_t(),rstate,q.get_mpz_t());
    return g;
}
mpz_class fully_secure(vector<unsigned long> X, vector<unsigned long> Y){
    if(X.size() != Y.size()){
        cout << "vector sizes do not match" << endl;
        cout << X.size() << "\t" << Y.size() << endl;
        return 0;
    }

    map<string,mpz_class> buffer;


    /*
     * I used random.org to pick a random mersenne prime, which is in base 10
          format, cited from http://bigprimes.net/pages/archive/mersenne/M21
         .txt
     */
    ifstream input_q("q.txt");
    string q_str;
    getline(input_q,q_str);
    mpz_class q(q_str,10);
    /*
     *Random mersenne chosen by random.org
     *http://bigprimes.net/pages/archive/mersenne/M21.txt
     */


```

```
    unsigned long seed = (unsigned long)time(NULL);
    gmp_randstate_t rstate;
    gmp_randinit_mt(rstate);
    gmp_randseed_ui(rstate,seed);
    mpz_class g;
    mpz_urandomm(g.get_mpz_t(),rstate,q.get_mpz_t());

    vector<vector<mpz_class>> com_X(X.size());
    vector<vector<mpz_class>> com_Y(Y.size(),vector<mpz_class>(2,0));

    //get an elgamal tuple
    vector<mpz_class> keys = keygen(g,q);
    mpz_class h,x;
    h = keys[0];
    x = keys[1]; //secret
    cout << "asd" << endl;
    mpz_class temp_random;
    mpz_class temp;
    for(int i = 0; i < X.size(); i++){
        //mpz_class temp;
        //temp = Y[i];
        mpz_set_ui(temp.get_mpz_t(),Y[i]);
        com_Y[i] = encryption(temp,h,q,g);
        //buffer["com_Y0" + to_string(i)] = com_Y[0][i];
        //buffer["com_Y1" + to_string(i)] = com_Y[1][i];
        //uncommenting these really likes to make GNU-MP run out of virtual
            memory. no idea why.

        //generate a commitment for each xi, yi and add it to com_X and com_Y

        cout << "a";
        //mpz_urandomm(temp_random.get_mpz_t(),rstate,q.get_mpz_t());
        temp_random = uniformly_random(q);
        cout << "b";
        mpz_class ytemp1,ytemp2;
        cout << "c";
        ytemp1 = Y[i];
        cout << "d" << i;
        ytemp2 = 1 - Y[i];
        //cout << elgamal_v(q,h,g,temp_random,ytemp1,ytemp2) << endl;
        //call elgamals verifier function on yi
    }
    vector<mpz_class> r(X.size());
    for(int i = 0; i < X.size(); i++){
        mpz_urandomm(r[i].get_mpz_t(),rstate,q.get_mpz_t());
    }
    cout << "peasche!" << endl;
    mpz_class R = vector_sum(r);

    mpz_class alpha,beta,tau,rho;
    mpz_urandomm(alpha.get_mpz_t(),rstate,q.get_mpz_t());
    mpz_urandomm(beta.get_mpz_t(),rstate,q.get_mpz_t());
    mpz_urandomm(tau.get_mpz_t(),rstate,q.get_mpz_t());
    mpz_urandomm(rho.get_mpz_t(),rstate,q.get_mpz_t());

    cout << "ayylmao" << endl;
    vector<vector<mpz_class>> ab(X.size(),vector<mpz_class>(2,0));
    vector<vector<mpz_class>> A(ab.size(),vector<mpz_class>(2,0));
    vector<vector<mpz_class>> B(ab.size(),vector<mpz_class>(2,0));
    for(int i = 0; i < ab.size(); i++){
        //mpz_urandomm(r[i].get_mpz_t(),rstate,q.get_mpz_t());
        ab[i][0] = (r[i] + X[i]) % q;
        ab[i][1] = (r[i] + (1 - X[i])) % q;
        A[i] = encryption(ab[i][0],alpha,q,g);
        B[i] = encryption(ab[i][1],beta,q,g);
        //buffer["A0" + to_string(i)] = A[0][i];
        //buffer["A1" + to_string(i)] = A[1][i];
        //buffer["B0" + to_string(i)] = B[0][i];
```

```
        //buffer["B1" + to_string(i)] = B[1][i];
    }
    for(int i = 0; i < ab.size(); i++){
        mpz_class temp_random;
        mpz_urandomm(temp_random.get_mpz_t(),rstate,q.get_mpz_t());
        cout << "ab" << i << endl;
        cout << elgamal_v(q,h,g,temp_random,ab[0][i],ab[1][i]) << endl;
        //call elgamal verifier method showing |bi = ai| = 1
    }
    cout << "widdle farther" << endl;
    vector<mpz_class> t(ab.size());
    vector<vector<mpz_class>> C(ab.size());
    for(int i = 0; i < ab.size(); i++){
        t[i] = ab[i][Y[i]];
        C[i] = encryption(t[i],tau,q,g);
        //buffer["C0"+ to_string(i)] = C[0][i];
        //buffer["C1"+ to_string(i)] = C[1][i];
    }
    mpz_class T = vector_sum(t);

    //option 1
    vector<mpz_class> Ct(2); //our elgamal tuple
        Ct[0] = 1;
        Ct[1] = 1;
    for(int i = 0; i < ab.size(); i++){
        Ct[0] *= C[0][i];
        Ct[1] *= C[1][i]; //since multiplying integers is the same as and-ing bitstrings
    }
    cout << "woah were far" << endl;
    //buffer T
    buffer["T"] = T;
    //call proof again

    vector<mpz_class> Cl(2);
    Cl[0] = 1;
    Cl[1] = 1;
    for(int i = 0; i < ab.size(); i++){
        Cl[0] *= buffer["C0" + to_string(i)];
        Cl[1] *= buffer["C1" + to_string(i)];
    }
    bool a = Ct[0] == Cl[0];
    bool b = Ct[1] == Cl[1];
    if(a && b) cout << "yes" << endl;
    else cout << "no" << endl;
    //compute Ct again but with C from buffer and verify
    return T - R;
}
```

elgamal.h

```
#ifndef ELGAMAL_H
#define ELGAMAL_H
#include <string>
#include <gmpxx.h>
#include <gmp.h>
#include <math.h>
#include <iostream>
#include <map>
#include <vector>
#include <fstream>
using namespace std;
vector<mpz_class> keygen(mpz_class g, mpz_class q);
vector<mpz_class> encryption(mpz_class m, mpz_class h, mpz_class q, mpz_class g);
mpz_class decryption(vector<mpz_class> c, mpz_class x, mpz_class q);
#endif
```

elgamal.cpp

```
#include <iostream>
#include <gmp.h>
#include <gmpxx.h>
#include <math.h>
#include <string>
#include <vector>
#include <map>
using namespace std;
//pick q and g, G is always Zq
vector<mpz_class> keygen(mpz_class g, mpz_class q){

    unsigned long seed = (unsigned long)time(NULL);
    gmp_randstate_t rstate;
    gmp_randinit_mt(rstate);
    gmp_randseed_ui(rstate,seed);
    mpz_class x;
    mpz_urandomm(x.get_mpz_t(),rstate,q.get_mpz_t());
    //randomly pick x from G
    mpz_class h;
    mpz_powm(h.get_mpz_t(),g.get_mpz_t(),x.get_mpz_t(),q.get_mpz_t());
    //h = g^x
    vector<mpz_class> keys(2);
    keys[0] = h;
    keys[1] = x;
    return keys;
    //keys[0] is public, keep keys[1] secret
    //publish h as pubkey
    //x is private key and is kept secret
}
vector<mpz_class> encryption(mpz_class m, mpz_class h, mpz_class q, mpz_class g){
    unsigned long seed = (unsigned long)time(NULL);
    gmp_randstate_t rstate;
    gmp_randinit_mt(rstate);
    gmp_randseed_ui(rstate,seed);
    mpz_class y;
    mpz_urandomm(y.get_mpz_t(),rstate,q.get_mpz_t());
    //randomly pick y from G
    mpz_class c1;
    mpz_powm(c1.get_mpz_t(),g.get_mpz_t(),y.get_mpz_t(),q.get_mpz_t());
    //c1 = g^y

    mpz_class s;
    mpz_powm(s.get_mpz_t(),h.get_mpz_t(),y.get_mpz_t(),q.get_mpz_t());
    //s = h^y
    mpz_class c2;
    c2 = m*s;
    //c2 = m * s

    //(c1,c2) is sent as a tuple
    vector<mpz_class> c(2);
    c[0] = c1;
    c[1] = c2;

    return c;
}
mpz_class decryption(vector<mpz_class> c, mpz_class x, mpz_class q){
    mpz_class c1 = c[0];
    mpz_class c2 = c[1];

    mpz_class s;
    mpz_powm(s.get_mpz_t(),c1.get_mpz_t(),x.get_mpz_t(),q.get_mpz_t());
    //s = c1^x

    mpz_class inv_s;
    inv_s = q - s;
```

```
    //get inverse of s from s?
    mpz_class m;
    m = c2 * inv_s;

    return m;
    //m = c2^s^(-1) thats modular multiplicative inverse of s
    //s^-1 = (c1^-1)^x
    //return m
}
```

crypto.cpp

```
#include <string>
#include <gmpxx.h>
#include <gmp.h>
#include <math.h>
#include <iostream>
#include <vector>
#include <map>
#include <fstream>
using namespace std;
//predetermine N as some large ass prime of mpz_class
vector<mpz_class> keygen(){
    vector<mpz_class> c;
    //should return a fully secure e and d as a tuple in c
    //multiplicative inverse of each other in  Zn
}
void OT(/*y a bit*/mpz_class m0,mpz_class m1, unsigned long y){
    //gen buffer

    mpz_class N; //temp
    unsigned long seed = (unsigned long)time(NULL);
    gmp_randstate_t rstate;
    gmp_randinit_mt(rstate);
    gmp_randseed_ui(rstate,seed);

    mpz_class x0, x1;
    mpz_urandomm(x0.get_mpz_t(),rstate,N.get_mpz_t());
    mpz_urandomm(x1.get_mpz_t(),rstate,N.get_mpz_t());
    mpz_class e, d;
    //e, x0, x1 to buffer

    mpz_class k;
    mpz_urandomm(k.get_mpz_t(),rstate,N.get_mpz_t());
    unsigned long b = y;
    mpz_class v;
    mpz_class temp;
    mpz_powm(temp.get_mpz_t(),k.get_mpz_t(),e.get_mpz_t(),N.get_mpz_t());
    v = (((b == 0)? x0 : x1) + temp )%N;
    //push v, also buffer xinputs
    mpz_class k0, k1,temp2,temp3;
    mpz_powm(temp2.get_mpz_t(),x0.get_mpz_t(),d.get_mpz_t(),N.get_mpz_t());
    mpz_powm(temp3.get_mpz_t(),x1.get_mpz_t(),d.get_mpz_t(),N.get_mpz_t());
    k0 = (v - temp2)%N;
    k1 = (v - temp3)%N;
    mpz_class n0, n1;
    n0 = k0 + m0;
    n1 = k1 + m1;
    //buffer n0,n1;

    cout << ((b == 0)? n0 : n1 ) - k;
    //n0,n1 from buffer
}
```

References