Number Theory and Modern Cryptography (CSL413)
Ritu Thombre (BT17CSE084)
Lab Assignment 3

Contents

List of Figures

1 Brief Introduction to the paper

Multi-writer Multi-reader Boolean Keyword Searchable Encryption scheme is implemented using the example of a hospital. Following is the brief description of the scheme:

1. MWMRBKSE is based on Key-Policy Attribute Based Encryption (KP-ABE), where each word in the ciphertext (which is an encrypted message from data writer) is considered as an attribute, and a boolean search query from data reader is converted to a boolean key policy. If attributes of some ciphertext are satisfying the key policy of data reader, then data reader can decrypt the corresponding ciphertext.

2. Servers: ETA(Enterprise Trusted Authority) and SS(Storage Server) play the role of servers.

   - ETA sets up (public parameter pp, master public key mpk, master secret key msk) and publishes (pp,mpk). ETA initializes reader list and writer list and stores them on the storage server (SS). Whenever a new reader/writer wants to register, ETA issues them a pair (server key, private key) and adds the corresponding reader/writer to the reader/writer list and updates the reader/writer list in the storage server (SS). Whenever a reader/writer wants to deregister, ETA removes them from the reader/writer list and updates the reader/writer list in the storage server (SS).
   - SS stores the list of currently authorized readers and writers. Whenever a data writers creates a new ciphertext, SS stores that ciphertext. Whenever a data reader wants to search something, he/she creates a boolean Query Q and issues it to SS. SS applies query Q on the currently stored ciphertexts and returns the corresponding result.

3. Clients: Role of clients is played by data readers (DR) and data writers (DW)

   - Data Writers: Role of data writers is played by doctors who want to save patient data in the server. This patient data is encrypted and stored in the storage server. Eg. "Mrs Smith with age 31 was diagnosed with COVID-19 and treated by Dr. Sanchez" is encrypted and stored along with the attributes (Patient name: "Mrs Smith",Patient age: 31, Diagnosed with :"COVID-19", Doctor name: "Dr. Sanchez").
   - Data Reader: Role of data readers is played by doctors who want to read patient history, data analysts who want to analyse hospital data, patients who want to search doctor information, etc. Data reader issues his/her query to SS. SS creates token (which is basically a key-policy) based on the query and searches if currently stored ciphertext attributes satisfy the token key-policy and returns the corresponding result. Eg. if a reader wants to search "COVID patients treated by Dr. Sanchez", which has corresponding query ("COVID" and "Dr. Sanchez"). Since, attributes (Patient name: "Mrs Smith",Patient age: 31, Diagnosed with :"COVID-19", Doctor name: "Dr. Sanchez") satisfy the query "COVID patients treated by Dr. Sanchez", SS will return "Mrs Smith with age 31 was diagnosed with COVID-19 and treated by Dr. Sanchez".

2 Algorithms

MWMRBKSE has nine algorithms as follows :

1. Setup: Setup function takes in input keyword space and outputs public parameter (pp), master public key (mpk) and master secret key (msk).

2. Writer Registration: Writer registration takes in the input of (writer ID,mpk,pp) and issues a pair (writer secret key,writer server key) to the writer and adds writer to the writer list.

3. Reader Registration: Reader registration takes in the input of (reader ID,mpk,msk,pp) and issues a pair (reader secret key,reader server key) to the reader and adds reader to the reader list.

4. Writer deregistration: Writer deregistration takes in the input of writer ID and writer list, and removes the writer with writer ID from the writer list.

5. Reader deregistration: Reader deregistration takes in the input of reader ID and reader list, and removes the reader with reader ID from the reader list.

6. Encryption: Encrypt function takes in the (writer secret key,mpk,message) and creates ciphertext CC,and send (CC,writer ID,message) to the the storage server.

7. C_accept: C_accept function takes in the input (CC,writer ID,message) and check if the writer with writer ID is present in the writer list. If writer is authorized, SS will accept the CC and stores it, else SS will reject the ciphretext.

8. Token_Gen: Token generation function takes in the input of (pp,mpk,reader ID,query Q) and outputs a token TQ = (D0i,D1i,D2i,D3i,I,reader ID) and send it to the server

9. Search: Search algorithm takes in the token TQ = (D0i,D1i,D2i,D3i,I,reader ID) and checks if the reader with reader ID is present in the reader list. If the reader is authorized, then search algorithm will check for all the ciphertext that satisfy query policy from the reader and returns the corresponding ciphertexts, else it rejects the reader.

Out of these 9 algorithms, we have implemented first 7 algorithms along with the LSSS access structure which converts input boolean query to a LSSS matrix.

3 Code Implementation: Setup

3.1 Header Files

Header files, global variable and macros are shown in the Fig. 1.

```cpp
#include "pbc.h"
#include<gmp.h>
#include "pbc_test.h"
#include<time.h>
using namespace std;
#include<bits/stdc++.h>
#include<algorithm>
#include<iostream>
#include<string.h>
#include<vector>
#define KEYWORD_SIZE 10

unordered_map<string,mpz_t> keywords;

```

Figure 1: Headers

1. Setup(λ, $\mathcal{MS}$)

1 From input message space $\mathcal{MS}$, the algorithm identifies the potential keywords and define a keyword space $\mathcal{KS}$ containing n keyword fields.

2 Runs a generator algorithm $\mathbb{G}$ to obtain (p_1, p_2, p_3, p_4, G, GT, e). Here $G = G_{p_1} \times G_{p_2} \times G_{p_3} \times G_{p_4}$, G and GT are cyclic groups of order $N = p_1 p_2 p_3 p_4$. The algorithm then computes a master public key mpk and a master secret key msk as follows:

- Selects $\alpha, \beta_1, \beta_2, \ldots, \beta_n \in Z_N, g, u, u_1, u_2, \ldots, u_n \in G_{p_1}, X_3 \in G_{p_3}, X_4, h_1, h_2, \ldots, h_n \in G_{p_4}$ at random.
- Computes $H_i = (u_i \cdot h_i)$, $u'_i = u^{\beta_i}$ for $1 \leq i \leq n$.
- Sets pp=($G, GT, e, g, n, \mathcal{KS}$), mpk=(N, g, g^α, H_i, X_4) and msk=(α, u'_i, X_3) for $1 \leq i \leq n$.

3 Sets up two empty lists $RL = \{\perp, \perp\}$ and $WL = \{\perp, \perp\}$ and stores them onto SS.

Figure 2: Setup Algorithm

3.2 Initialization of groups and pairings

As shown in Fig. 2, we have to generate 4 primes $p1, p2, p3, p4$ and multiply them together to get N which will server as order of the groups $G1, G2, G3, G4, G$. Random numbers $r1, r2, r3, r4$ are generated by using mpz_urandomm(mpz_t r,gmp_randstate state,mpz_t size), where $size$ is chosen randomly and and r gets value from $(0, size - 1)$. Primes $p1, p2, p3, p4$ are generated from random numbers $r1, r2, r3, r4$ using mpz_nextprime(mpz_t prime,mpz_t random). N is obtained by multiplying $p1, p2, p3, p4$ using mpz_mul(mpz_t N,mpz_t p1,mpz_t p2)

This paper uses $Type - A1$ pairing. So, the $p1, p2, p3, p4$ are used to generate pairings $G1, G2, G3, G4, G$ using pbc_param_init_a1_gen(pbc_param_t param,mpz_t prime) and pairing_init_pbc_param(pbc_pairing pairing,pbc_param_t param).

Code for pairing generation is shown in Fig. 3 and Fig. 4.

```
int main(int argc, char **argv)
//int main()
{
    mpz_t r1,r2,r3,r4,p1,p2,p3,p4;
    mpz_t size_p;

    mpz_init(p1);
    mpz_init(p2);
    mpz_init(p3);
    mpz_init(p4);

    mpz_init(r1);
    mpz_init(r2);
    mpz_init(r3);
    mpz_init(r4);

    mpz_init(size_p);

    srand(time(0));
    mpz_set_si(size_p,rand());
    gmp_randstate_t state;
    gmp_randinit_mt(state);

    // generate random numbers
    mpz_urandomm(r1,state,size_p);
    mpz_urandomm(r2,state,size_p);
    mpz_urandomm(r3,state,size_p);
    mpz_urandomm(r4,state,size_p);

    // generate prime numbers
    mpz_nextprime(p1,r1);
    mpz_nextprime(p2,r2);
    mpz_nextprime(p3,r3);
    mpz_nextprime(p4,r4);

    mpz_t N;
    mpz_init(N);
    mpz_mul(N,p1,p2);
    mpz_mul(N,N,p3);
    mpz_mul(N,N,p4);

```

Figure 3: Prime Generation

```
//group G1 of order p1
pairing_t pairing1;
pbc_param_t param1;
pbc_param_init_a1_gen(param1,p1);
pairing_init_pbc_param(pairing1,param1);
//group G2 of order p2
pairing_t pairing2;
pbc_param_t param2;
pbc_param_init_a1_gen(param2,p2);
pairing_init_pbc_param(pairing2,param2);
//group G3 of order p3
pairing_t pairing3;
pbc_param_t param3;
pbc_param_init_a1_gen(param3,p3);
pairing_init_pbc_param(pairing3,param3);
//group G4 of order p4
pairing_t pairing4;
pbc_param_t param4;
pbc_param_init_a1_gen(param4,p4);
pairing_init_pbc_param(pairing4,param4);
//group GT of order N=p1*p2*p3*p4
pairing_t pairing_final;
pbc_param_t param_final;
pbc_param_init_a1_gen(param_final,N);
pairing_init_pbc_param(pairing_final,param_final);

```

Figure 4: Pairing Generation

3.3 pp,mpk,msk

All the operations shown in Fig. 2 for generating pp,mpk,msk are done using gmp functions (exponentiation and multiplication). If pairing of some element is unspecified (such as H_i), it is assumed to be belonging to group G of order N. Code for these operations is shown in Fig. 5. [1]

Keyword space is also created. Keyword space contains predefined 10 attributes, which are strings. Keyword space is implemented using unordered_map, which maps these string attributes to mpz elements. Keyword space is shown in Fig. 6.

```
// find apha and beta[i] values using ZN
mpz_t alpha;
mpz_init(alpha);
mpz_urandomm(alpha,state,N);
mpz_t beta[keywords.size()];
for (int i=0;i<keywords.size();i++)
{
    mpz_init(beta[i]);
    mpz_urandomm(beta[i],state,N);
}
// find g,u and u[i] values
mpz_t g;
mpz_init(g);
mpz_urandomm(g,state,p1);
mpz_t single_u;
mpz_init(single_u);
mpz_urandomm(single_u,state,p1);
mpz_t u[keywords.size()];
for (int i=0;i<keywords.size();i++)
{
    mpz_init(u[i]);
    mpz_urandomm(u[i],state,p1);
}
// find X3
mpz_t X3;
mpz_init(X3);
mpz_urandomm(X3,state,p3);
// find X4 and h[i] values
mpz_t X4;
mpz_init(X4);
mpz_urandomm(X4,state,p4);
mpz_t h[keywords.size()];
for (int i=0;i<keywords.size();i++)
{
    mpz_init(h[i]);
    mpz_urandomm(h[i],state,p4);
}
// calculate u_prime[i] and H[i]
mpz_t H[keywords.size()],u_prime[keywords.size()];
for (int i=0;i<keywords.size();i++)
{
    mpz_init(H[i]);
    mpz_init(u_prime[i]);
    mpz_mul(H[i],u[i],h[i]);
    mpz_mod(H[i],H[i],N);
    mpz_powm(u_prime[i],single_u,beta[i],N);
}
```

Figure 5: Code calculations required for setup

Now for generating pp,mpk,msk. Structures are defined for pp,mpk,msk. Once pp,mpk,msk are created, they are initialized and then assigned the values that are calculated in Fig. 5. G and GT are not used in pp.[2] Code for pp,mpk,msk is shown in Fig.7, Fig.8, Fig.9.

[1] keyword.size() is the size of KS(keyword space) and is defined to be 10

[2] PBC pairings only have $G1$,$G2$ and GT in them, but the paper has shown $G1$,$G2$,$G3$,$G4$,GT. To avoid the cross-pairing operation, we implemented all the operation using GMP, and only used PBC for applying pairing.

```cpp
    vector<string> temp;

    temp.push_back("hospital1");
    temp.push_back("hospital2");
    temp.push_back("doctor1");
    temp.push_back("doctor2");
    temp.push_back("patient1");
    temp.push_back("patient2");
    temp.push_back("disease1");
    temp.push_back("disease2");
    temp.push_back("age1");
    temp.push_back("age2");

    for(int i=0;i<temp.size();i++)
    {
        mpz_t t;
        mpz_init(t);
        mpz_set_si(t,101+i);
        mpz_init(keywords[temp[i]]);
        mpz_set(keywords[temp[i]],t);
    }

```

Figure 6: Keyword space

```cpp
// definition of public parameters
typedef struct public_parameter_def
{
    mpz_t N;
    mpz_t g;
    int n;
    unordered_map<string,mpz_t> KS;
} public_parameter;

// initialization of public parameters
public_parameter initialize_pp(public_parameter pp)
{
    mpz_init(pp.N);
    mpz_init(pp.g);
    return pp;
}

// assignment of public parameters
public_parameter assign_pp(public_parameter pp,mpz_t N,mpz_t g,unordered_map<string,mpz_t> keywords)
{
    mpz_set(pp.N,N);
    mpz_set(pp.g,g);
    pp.n = keywords.size();
    for (auto i: keywords)
    {
        mpz_init(pp.KS[i.first]);
        mpz_set(pp.KS[i.first],keywords[i.first]);
    }
    return pp;
}
```

Figure 7: pp structure, initialization and assignment Code

Now, using the structures Fig.7, Fig.8, Fig.9 and setup calculations from Fig.5, we are ready to create pp,mpk,msk as shown in Fig.10.

Setup algorithms also initializes empty reader and writer list. Since every reader and writer has their own secret and server keys, we have used unordered_map that maps reader/writer ID to their corresponding server keys,secret keys. reader/writer list are also implemented using unordered_map that maps reader/writer ID to the tuple (reader/writer ID, server key).

```
// definition of master public key
typedef struct mpk_def
{
    mpz_t N;
    mpz_t g;
    mpz_t g_to_the_alpha;
    mpz_t H[KEYWORD_SIZE];
    mpz_t X4;
}master_public_key;

// initialization of master public key
master_public_key initialize_mpk(master_public_key mpk)
{
    mpz_init(mpk.N);
    mpz_init(mpk.g);
    mpz_init(mpk.g_to_the_alpha);
    for(int i=0;i<KEYWORD_SIZE;i++)
        mpz_init(mpk.H[i]);
    mpz_init(mpk.X4);
    return mpk;
}

// assignment of master public key
master_public_key assign_mpk(master_public_key mpk,mpz_t N,mpz_t g,mpz_t g_to_the_alpha,mpz_t H[],mpz_t X4)
{
    mpz_set(mpk.N,N);
    mpz_set(mpk.g,g);
    mpz_set(mpk.g_to_the_alpha,g_to_the_alpha);
    for(int i=0;i<KEYWORD_SIZE;i++)
        mpz_set(mpk.H[i],H[i]);
    mpz_set(mpk.X4,X4);
    return mpk;
}
```

Figure 8: mpk structure, initialization and assignment Code

```
// definition of master secret key
typedef struct msk_def
{
    mpz_t alpha;
    mpz_t u_prime[KEYWORD_SIZE];
    mpz_t u[KEYWORD_SIZE];
    mpz_t X3;
}master_secret_key;

// initialization of master secret key
master_secret_key initialize_msk(master_secret_key msk)
{
    mpz_init(msk.alpha);
    for(int i=0;i<KEYWORD_SIZE;i++)
    {
        mpz_init(msk.u_prime[i]);
        mpz_init(msk.u[i]);
    }
    mpz_init(msk.X3);
    return msk;
}

// assignment of master secret key
master_secret_key assign_msk(master_secret_key msk,mpz_t alpha,mpz_t u_prime[],mpz_t u[],mpz_t X3)
{
    mpz_set(msk.alpha,alpha);
    for(int i=0;i<KEYWORD_SIZE;i++)
    {
        mpz_set(msk.u_prime[i],u_prime[i]);
        mpz_set(msk.u[i],u[i]);
    }
    mpz_set(msk.X3,X3);
    return msk;
}
```

Figure 9: msk structure, initialization and assignment Code

Storage server is also initialized. Storage server stores the ciphertext along with the message for each data writer. Storage server is implemented using unordered_multimap which maps data writer ID, to the ciphertext structure (Fig. 20), which contains the actual message and the corresponding ciphertext from the data writer. Initialization of reader/writer list,server keys, secret keys and storage server in shown in Fig. 10.

```
public_parameter pp;
pp = initialize_pp(pp);
pp = assign_pp(pp,N,g,keywords);
gmp_printf("N: %Zd\n",N);
gmp_printf("g: %Zd\n",g);
cout<<"\nPublic Paramater created successfully\n\n";
mpz_t g_to_the_alpha;
mpz_init(g_to_the_alpha);
mpz_powm(g_to_the_alpha,g,alpha,p1);
master_public_key mpk;
mpk = initialize_mpk(mpk);
mpk = assign_mpk(mpk,N,g,g_to_the_alpha,H,X4);
gmp_printf("N: %Zd\n",N);
gmp_printf("g: %Zd\n",g);
gmp_printf("g^alpha: %Zd\n",g_to_the_alpha);
cout<<"H values:\n";
for(int i=0;i<10;i++)
    gmp_printf("%Zd ",H[i]);
gmp_printf("\nX4: %Zd \n",X4);
cout<<"\nMaster public key created successfully\n\n";
master_secret_key msk;
msk = initialize_msk(msk);
msk = assign_msk(msk,alpha,u_prime,u,X3);
gmp_printf("alpha: %Zd \n",alpha);
gmp_printf("X3: %Zd \n",X3);
cout<<"u_prime values :\n";
for(int i=0;i<10;i++)
    gmp_printf("%Zd ",u_prime[i]);
cout<<"\n\nMaster secret key created successfully\n";
unordered_map<int,mpz_t> writer_list;
unordered_map<int,mpz_t> writer_server_keys;
unordered_map<int,mpz_t> writer_private_keys;
unordered_map<int,reader_secret_key> reader_private_keys;
unordered_map<int,reader_server_key> reader_server_keys,reader_list;
//storage servers
unordered_multimap<int,ciphertext> SS;
int current_wid = 0,current_rid=0;
```

Figure 10: Setup Code

4 Code Implementation: Writer Registration and Deregistration

2. W_Registration(pp, mpk, WID)

1 Selects $x_{WID} \in Z_N$ at random and sets a secret key $U_{WID} = x_{WID}$.
2 From (mpk, U_{WID}), it computes $y_{WID} = g^{-x_{WID}}$ and sets server side key $S_{WID} = (WID, y_{WID})$.
3 Updates the local copy of WL as $WL = (WL) \cup \{WID, S_{WID}\}$.
4 Replaces WL stored at SS with the modified WL.

Figure 11: Writer registration algorithm

8. W_Deregistration(WID, WL)

1 It revokes a writer WID by updating local copy of WL as WL = WL - $\{WID, S_{WID}\}$.
2 Replaces WL at server with the modified WL.

Figure 12: Writer deregistration algorithm

Writer registration algorithm is shown in Fig.11. Writer ID which is unique to every writer is taken as an input. All the operations are carried out using gmp functions (exponentiation and inverse). Once the writer server keys and writer secret keys are created, they are added to writer_server_keys and writer_private_keys. Writer is also added to the writer_list.

Writer deregistration algorithm is shown in Fig.12. Writer ID which is to be deleted is taken as an input. Writer ID is then used to remove the corresponding writer from writer_server_keys and writer_private_keys and writer_list.

Code for writer registration and deregistration is shown in Fig.13

```
// WRITER registration
void register_writer(unordered_map<int,mpz_t> &writer_list,unordered_map<int,mpz_t> &writer_server_keys,unordered_map<int,mpz_t> &writer_private_keys,
    int wid,public_parameter pp,master_public_key mpk, gmp_randstate_t state)
{
    // writer secret key
    mpz_t x_wid;
    mpz_init(x_wid);
    mpz_urandomm(x_wid,state,mpk.N);
    mpz_init(writer_private_keys[wid]);
    mpz_set(writer_private_keys[wid],x_wid);
    // server-writer key
    mpz_t y_wid;
    mpz_init(y_wid);
    mpz_powm(y_wid,mpk.g,x_wid,mpk.N);
    mpz_invert(y_wid,y_wid,mpk.N);
    mpz_init(writer_server_keys[wid]);
    mpz_set(writer_server_keys[wid],y_wid);
    // update writer list
    mpz_init(writer_list[wid]);
    mpz_set(writer_list[wid],y_wid);

}

// WRITER deregistration
void deregister_writer(unordered_map<int,mpz_t> &writer_list,unordered_map<int,mpz_t> &writer_server_keys,unordered_map<int,mpz_t> &writer_private_keys,int wid_to_delete)
{
    writer_list.erase(wid_to_delete);
    writer_private_keys.erase(wid_to_delete);
    writer_server_keys.erase(wid_to_delete);

}
```

Figure 13: Code for Writer registration and deregistration

5 Code Implementation: Reader Registration

3. R_Registration(pp, mpk, msk, RID)

1. Selects $x_{RID}, x'_{RID} \in Z_N$ at random.
2. Computes $y_{RID} = \alpha / x_{RID}$ and $u'_{i\_RID} = (u'_i)^{1/x'_{RID}}$ for $1 \leq i \leq n$.
3. Sets a read secret key $U_{RID} = (x_{RID}, u_i, u'_{i\_RID})$ and a server side key as $S_{RID} = (RID, y_{RID}, x'_{RID})$.
4. Updates the local copy of RL as $RL = (RL) \cup \{RID, S_{RID}\}$.
5. Replaces RL stored at SS with the modified RL.

Figure 14: Reader registration algorithm

Reader registration algorithm is shown in Fig.14. Reader ID which is unique to every reader is taken as an input. N^{th} root calculation is done using gmp function mpz_root. Division operation was performed using pbc function pbc_div. [3]

As reader server keys and reader private keys have multiple parameters in them, we created structures for them which are shown in Fig. 15

Structures of reader keys are assigned their corresponding values. Once the reader server keys and reader secret keys are created, they are added to reader_server_keys and reader_private_keys. Reader is also added to the reader_list. Code for reader registration is shown in Fig. 16

[3] In GMP division, we had to choose from remainder division or quotient division. Since, it was not specified in the paper if remainder or quotient is used, we used pbc division function.

```c
// definition of READER SECRET KEY
typedef struct reader_secret_key_def
{
    mpz_t x_rid;
    mpz_t u[KEYWORD_SIZE];
    mpz_t u_prime_rid[KEYWORD_SIZE];
}reader_secret_key;

// initialization of READER SECRET KEY
reader_secret_key initialize_reader_secret_key(reader_secret_key u_rid)
{
    mpz_init(u_rid.x_rid);
    for(int i=0;i<KEYWORD_SIZE;i++)
    {
        mpz_init(u_rid.u[i]);
        mpz_init(u_rid.u_prime_rid[i]);
    }
    return u_rid;
}

// definition of READER SERVER KEY
typedef struct reader_server_key_def
{
    int rid;
    mpz_t y_rid;
    mpz_t x_prime_rid;
}reader_server_key;

// initialization of READER SERVER KEY
reader_server_key initialize_reader_server_key(reader_server_key s_rid)
{
    mpz_init(s_rid.y_rid);
    mpz_init(s_rid.x_prime_rid);
    return s_rid;
}
```

Figure 15: Structure for reader server key and reader private key

```c
// READER registration
void register_reader(unordered_map<int,reader_secret_key> &reader_private_keys,unordered_map<int,reader_server_key> &reader_server_keys,unordered_map<int,reader_server_key> &reader_list,
    int rid,public_parameter pp,master_public_key mpk,master_secret_key msk,gmp_randstate_t state,pairing_t pairing_final)
{
    mpz_t x_rid,x_prime_rid;
    mpz_init(x_rid);
    mpz_init(x_prime_rid);
    mpz_urandomm(x_rid,state,mpk.N);
    mpz_urandomm(x_prime_rid,state,mpk.N);
    element_t y,a,b;
    element_init_Zr(y, pairing_final);
    element_init_Zr(a, pairing_final);
    element_init_Zr(b, pairing_final);
    element_set_mpz(a,msk.alpha);
    element_set_mpz(b,x_rid);
    element_div(y,a,b);
    mpz_t y_rid;
    mpz_init(y_rid);
    element_to_mpz(y_rid,y);
    mpz_t u_prime_rid[KEYWORD_SIZE];

    unsigned long int x_prime_rid_th_root=mpz_get_ui(x_prime_rid);
    for(int i=0;i<KEYWORD_SIZE;i++)
    {
        mpz_init(u_prime_rid[i]);
        mpz_root(u_prime_rid[i],msk.u_prime[i],x_prime_rid_th_root);
    }
    reader_secret_key u_rid = initialize_reader_secret_key(u_rid);
    mpz_set(u_rid.x_rid,x_rid);
    for(int i=0;i<KEYWORD_SIZE;i++)
    {
        mpz_set(u_rid.u[i],msk.u[i]);
        mpz_set(u_rid.u_prime_rid[i],u_prime_rid[i]);
    }
    reader_private_keys[rid] = u_rid;


    reader_server_key s_rid = initialize_reader_server_key(s_rid);
    s_rid.rid = rid;
    mpz_set(s_rid.y_rid,y_rid);
    mpz_set(s_rid.x_prime_rid,x_prime_rid);
    reader_server_keys[rid] = s_rid;
    reader_list[rid] = s_rid;
}
```


Figure 16: Code for reader registration

6 Code Implementation: Reader Deregistration

9. R_Deregistration(RID, RL)

1. It revokes a writer RID by updating local copy of RL as RL = RL - $\{RID, S_{RID}\}$.
2. Replaces RL at server with the modified RL.

Figure 17: Reader deregistration algorithm

Reader deregistration algorithm is shown in Fig.12. Reader ID which is to be deleted is taken as an input. Reader ID is then used to remove the corresponding reader from reader_server_keys and reader_private_keys and reader_list.

Code for writer registration and deregistration is shown in Fig. 18

```cpp
// READER deregistration
void deregister_reader(unordered_map<int,reader_secret_key> &reader_private_keys,unordered_map<int,reader_server_key> &reader_server_keys,
    unordered_map<int,reader_server_key> &reader_list,int rid_to_delete)
{

    reader_list.erase(rid_to_delete);
    reader_private_keys.erase(rid_to_delete);
    reader_server_keys.erase(rid_to_delete);

}
```

Figure 18: Code for Reader deregistration

7 Code Implementation: Encryption

4. Encryption(mpk, W, U_{WID}, M')

1 Selects $s \in Z_N$, h, Z_0, Z_{1i}, $Z'_{1i} \in G_{p_4}$ at random.
2 From input mpk and $W = \{w_1, w_2, \ldots, w_n\}$, it computes ciphertext components CC =(C_0, C'_0, C_i, C'_i) as follows
$C_0 = e(g, g^{\alpha})^s$, $C'_0 = (gh)^s \cdot Z_0$, $C'_i = (u'_i)^s \cdot Z'_{1i}$, $C_i = ((H_i)^{w_i})^s \cdot Z_{1i} \cdot g^{U_{WID}}$ for $1 \leq i \leq n$.
3 Sets a ciphertext $C = (CC, WID, M')$ and sends it to SS.

5. C_Accept(C, WL)

1 For input $WID \in C$, the algorithm checks WL.
2 If WID is found in WL then the algorithm
- Replaces C_i=$C_i \cdot y_{WID}$=$((H_i)^{w_i})^s \cdot Z_{1i}$
- Stores C on storage space.

3 Else
- Rejects input ciphertext C.

Figure 19: Encryption and C_accept algorithms

Encryption and C_accept algorithms are shown in Fig.19. Encryption function takes in the input (mpk, M', W, U_{WID}) and outputs the ciphertext $CC = (C0, C0', C_i, C_i')$. M' is the message from the data writer. mpk is the master public key created as shown in Fig. 8. U_{WID} is the secret key of the data writer with writer id WID, which is allotted to the writer as shown in Fig. 11. $W = (w_1, w_2, ..., w_n)$ are the mpz numbers corresponding to the attributes extracted from the data writer input message M'. These attribute are extracted using the keyword space maintained as shown in Fig. 6.

Once the ciphertext CC is calculated, encryption function send it to the C_accept function, which checks if the ciphertext is from an authorized data writer. If ciphertext is from an authorized data writer, then some additional calculations are performed and CC is stored in the storage server along with the message M' and the data writer id WID, else CC is rejected.

As ciphertext CC, contains multiple elements in it, we created a structure for the ciphertext, which is shown in Fig. 20. Once ciphertext structure is created, we performed all the operations shown in Fig.19 using GMP and PBC functions, as shown in Fig. 21.

We needed to apply pairing to calculate C_0, using g and g^α, which are present in the *mpk*. But, g and g^α are mpz numbers and we needed them to be points on the elliptic curve in order to apply pairing. So we converted g and g^α to the points on elliptic curve using element_from_hash [4] by passing the order of group G_{p1} (to which g and g^α belonged) as a size of g and g^α.

```
// encrypt
typedef struct ciphertext_type
{
    element_t C0;
    mpz_t C0_prime;
    mpz_t C[KEYWORD_SIZE],C_prime[KEYWORD_SIZE];
    string message;
    int word_size;

}ciphertext;
ciphertext initialize_cc(ciphertext cc,pairing_t pairing1)
{
    element_init_GT(cc.C0,pairing1);
    mpz_init(cc.C0_prime);
    for(int i=0;i<KEYWORD_SIZE;i++)
    {
        mpz_init(cc.C[i]);
        mpz_init(cc.C_prime[i]);
    }
    return cc;
}
```

Figure 20: Ciphertext structure

[4] element_from_hash(x,size of x)=(x,y) which is a point on the elliptic curve

```cpp
void c_accept(ciphertext cc,unordered_multimap<int,ciphertext> &SS,int wid_writer,unordered_map<int,mpz_t> writer_list,mpz_t N)
{
    int word_size = cc.word_size;
    for(int i=0;i<word_size;i++)
    {
        mpz_mul(cc.C[i],cc.C[i],writer_list[wid_writer]);
        mpz_mod(cc.C[i],cc.C[i],N);
    }
    SS.insert({wid_writer,cc});
}
void encrypt(const master_public_key mpk,master_secret_key msk,mpz_t W[],string M,int wid_writer,unordered_multimap<int,ciphertext> &SS,
    int word_size,gmp_randstate_t state,pairing_t pairing1,pairing_t pairing2,pairing_t pairing3,pairing_t pairing4,pairing_t pairing_final,
    mpz_t p1,mpz_t p2,mpz_t p3,mpz_t p4,mpz_t N,unordered_map<int,mpz_t> writer_private_keys,unordered_map<int,mpz_t> writer_list)
{
    mpz_t s;
    mpz_init(s);
    mpz_urandomm(s,state,N);
    mpz_t h,Z0,Z1[word_size],Z1_prime[word_size];

    mpz_init(h);
    mpz_urandomm(h,state,p4);

    mpz_init(Z0);
    mpz_urandomm(Z0,state,p4);

    for(int i=0;i<word_size;i++)
    {
        mpz_init(Z1[i]);
        mpz_urandomm(Z1[i],state,p4);
        mpz_init(Z1_prime[i]);
        mpz_urandomm(Z1_prime[i],state,p4);
    }
    //cout<<"Created required variables\n";
    element_t g,g_to_the_alpha;

    element_init_G1(g, pairing1);
    element_init_G2(g_to_the_alpha, pairing1);

    mpz_t temp_g,temp_g_to_the_alpha;
    mpz_init(temp_g);
    mpz_init(temp_g_to_the_alpha);
    mpz_set(temp_g,mpk.g);
    mpz_set(temp_g_to_the_alpha,mpk.g_to_the_alpha);
    element_from_hash(g,temp_g,mpz_get_ui(p1));
    element_from_hash(g_to_the_alpha,temp_g_to_the_alpha,mpz_get_ui(p1));
    element_t C0;
    element_init_GT(C0, pairing1);

    pairing_apply(C0, g, g_to_the_alpha, pairing1);
    element_pow_mpz(C0,C0,s);
    //cout<<"Created C0\n";

    mpz_t C0_prime;
    mpz_init(C0_prime);
    mpz_mul(C0_prime,mpk.g,h);
    mpz_mod(C0_prime,C0_prime,N);
    mpz_powm(C0_prime,C0_prime,s,N);
    mpz_mul(C0_prime,C0_prime,Z0);
    mpz_mod(C0_prime,C0_prime,N);
    //cout<<"Created C0_prime"<<endl;
    mpz_t C_prime[word_size];
    for(int i=0;i<word_size;i++)
    {
        mpz_init(C_prime[i]);
        mpz_powm(C_prime[i],msk.u_prime[i],s,N);
        mpz_mul(C_prime[i],C_prime[i],Z1_prime[i]);
        mpz_mod(C_prime[i],C_prime[i],N);
    }
    //cout<<"Created C_prime"<<endl;
    mpz_t C[word_size];
    for(int i=0;i<word_size;i++)
    {
        mpz_init(C[i]);
        mpz_powm(C[i],mpk.H[i],W[i],N);
        mpz_powm(C[i],C[i],s,N);
        mpz_mul(C[i],C[i],Z1[i]);
        mpz_mod(C[i],C[i],N);
        mpz_t g_to_the_xwid;
        mpz_init(g_to_the_xwid);
        mpz_powm(g_to_the_xwid,mpk.g,writer_private_keys[wid_writer],N);
        mpz_mul(C[i],C[i],g_to_the_xwid);
        mpz_mod(C[i],C[i],N);
    }
    //cout<<"Created C"<<endl;
    ciphertext cc = initialize_cc(cc,pairing1);
    element_set(cc.C0,C0);
    mpz_set(cc.C0_prime,C0_prime);
    for(int i=0;i<word_size;i++)
    {
        mpz_set(cc.C[i],C[i]);
        mpz_set(cc.C_prime[i],C_prime[i]);
    }
    cc.word_size = word_size;
    cc.message = M;
    //cout<<"Created Ciphertext"<<endl;
    c_accept(cc,SS,wid_writer,writer_list,N);
}
```

Figure 21: Code for Encryption and C_accept

8 Demonstration

We have created a demonstration of our code which shows (pp,mpk,msk) generation and allows user to choose from reader/writer registration and deregistration, allows users to print the list of readers/writers (Fig. 22), allows an authorized writer to add message and the corresponding ciphertext (calculation shown in Fig. 21) to the storage server and print all the messages in the storage server (Fig. 23).

To ensure that every reader and writer gets a unique ID, current_rid and current_wid are initialized to 0. Whenever a new reader/writer wants to register, current_rid and current_wid is incremented and allotted as the corresponding reader/writer ID. Code for demo is shown in Fig. 24. Output for the demo code is shown in Fig. 25

```
// print READERS
void print_readers(unordered_map<int,reader_server_key> reader_list)
{
    if(reader_list.empty()==true)
        cout<<"No readers found\n";
    else
    {
        for(auto i: reader_list)
        {
            cout<<"Reader ID :"<<i.first<<endl;
            gmp_printf("y_rid : %Zd\n",i.second.y_rid);
            gmp_printf("x_prime_rid : %Zd\n",i.second.x_prime_rid);
            cout<<"--------------------------------------\n";
        }
    }
}

// print WRITERS
void print_writers(unordered_map<int,mpz_t> writer_list)
{
    if(writer_list.empty()==true)
        cout<<"No writers found\n";
    else
    {
        for(auto i: writer_list)
        {
            cout<<"Writer ID :"<<i.first<<endl;
            gmp_printf("y_wid : %Zd\n",i.second);
            cout<<"--------------------------------------\n";
        }
    }
}
```

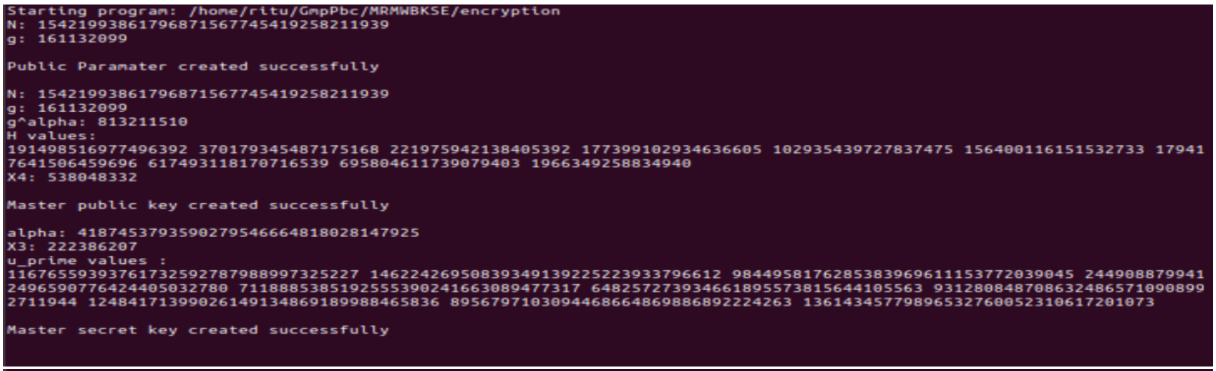
Figure 22: Code for printing readers and writers

```
//print messsages
void print_messages(unordered_multimap<int,ciphertext> &SS)
{
    if(SS.empty()==true)
        cout<<"No messages found\n";
    else
    {
        for(auto i: SS)
        {
            cout<<"Writer ID :"<<i.first<<endl;
            cout<<"Message :"<<i.second.message<<endl;
            element_printf("C0 : %B\n",i.second.C0);
            gmp_printf("C0_prime : %Zd\n",i.second.C0_prime);
            cout<<"C values:\n";
            for(int j=0;j<i.second.word_size;j++)
                gmp_printf("C[%d] : %Zd\n",j,i.second.C[j]);
            cout<<"C_prime values:\n";
            for(int j=0;j<i.second.word_size;j++)
                gmp_printf("C_prime[%d] : %Zd\n",j,i.second.C_prime[j]);
            cout<<"--------------------------------------\n";
        }
    }
}
```

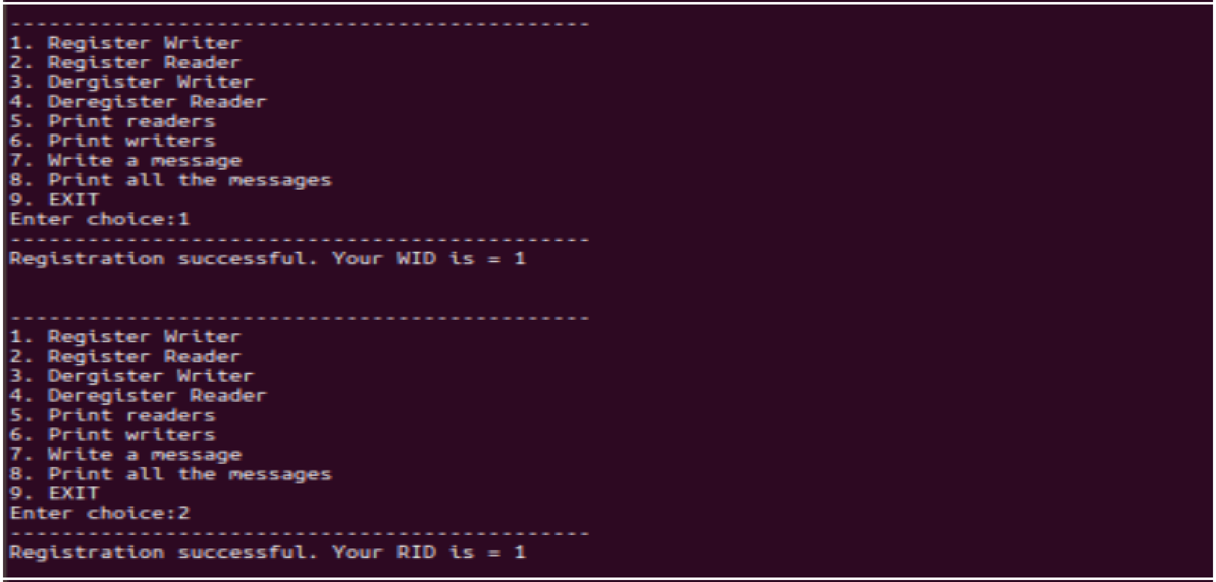
Figure 23: Code for printing messages and ciphertext stored in the storage server

```cpp
	while(true)
	{
		cout<<"\n\n-------------------------------------------\n";
		cout<<"1. Register Writer\n";
		cout<<"2. Register Reader\n";
		cout<<"3. Dergister Writer\n";
		cout<<"4. Deregister Reader\n";
		cout<<"5. Print readers\n";
		cout<<"6. Print writers\n";
		cout<<"7. Write a message\n";
		cout<<"8. Print all the messages\n";
		cout<<"9. EXIT\n";
		cout<<"Enter choice:";
		cin>>choice;
		cout<<"-------------------------------------------\n";
		if(choice == 1)
		{
			current_wid++;
			register_writer(writer_list,writer_server_keys,writer_private_keys,current_wid,pp,mpk,state);
			cout<<"Registration successful. Your WID is = "<<current_wid<<"\n";
		}
		if(choice == 2)
		{
			current_rid++;
			register_reader(reader_private_keys,reader_server_keys,reader_list,current_rid,pp,mpk,msk,state,pairing_final);
			cout<<"Registration successful. Your RID is = "<<current_rid<<"\n";
		}
		if(choice == 3)
		{
			cout<<"Enter your writer ID:";
			int wid_to_delete;
			cin>>wid_to_delete;
			if(writer_list.find(wid_to_delete) != writer_list.end())
			{
				deregister_writer(writer_list,writer_server_keys,writer_private_keys,wid_to_delete);
				cout<<"Writer "<<wid_to_delete<<" successfully deleted\n";
			}
			else
				cout<<"Error : Writer ID not found\n";
		}
		if(choice == 4)
		{
			cout<<"Enter your reader ID:";
			int rid_to_delete;
			cin>>rid_to_delete;
			if(reader_list.find(rid_to_delete) != reader_list.end())
			{
				deregister_reader(reader_private_keys,reader_server_keys,reader_list,rid_to_delete);
				cout<<"Reader "<<rid_to_delete<<" successfully deleted\n";
			}
			else
				cout<<"Error : Reader ID not found\n";
		}
		if(choice == 5)
			print_readers(reader_list);
		if(choice == 6)
			print_writers(writer_list);
		if(choice == 7)
		{
			int wid_writer;
			char a[200];
			cout<<"Enter your WID :";
			cin>>wid_writer;
			if(writer_list.find(wid_writer) == writer_list.end())
				cout<<"Unauthorized Writer!! Please register\n";
			else
			{
				cout<<"Enter message: ";
				scanf("\n");
				scanf("%[^\n]s",a);
				vector<string> words;
				string M = convert_to_string(a);
				string M_copy = M;
				string token;
				while(token != M_copy)
				{
					token = M_copy.substr(0,M_copy.find_first_of(" "));
					M_copy = M_copy.substr(M_copy.find_first_of(" ")+1);
					words.push_back(token);
				}
				int count_attributes=0;
				for(auto i: keywords)
				{
					for(int j=0;j<words.size();j++)
					{
						if(i.first == words[j])
							count_attributes++;
					}
				}
				mpz_t W[count_attributes];
				for(int i=0;i<count_attributes;i++)
					mpz_init(W[i]);
				int curr=0;
				for(auto i: keywords)
				{
					for(int j=0;j<words.size();j++)
					{
						if(i.first == words[j])
						{
							mpz_set(W[curr],i.second);
							curr++;
						}
					}
				}
				//for(int i=0;i<count_attributes;i++)
				//	gmp_printf("%Zd\n",W[i]);
				if(count_attributes>0)
					encrypt(mpk,msk,W,M,wid_writer,SS,count_attributes,state,pairing1,pairing2,pairing3,pairing4,pairing_final,p1,p2,p3,p4,N,writer_private_keys,writer_list);
				else
					cout<<"Enter valid message\n";
			}
		}
		if(choice == 8)
			print_messages(SS);
		if(choice == 9)
			break;
	}
	pairing_clear(pairing1);
	pairing_clear(pairing2);
	pairing_clear(pairing3);
	pairing_clear(pairing4);
	pairing_clear(pairing_final);
	return 0;
}
```

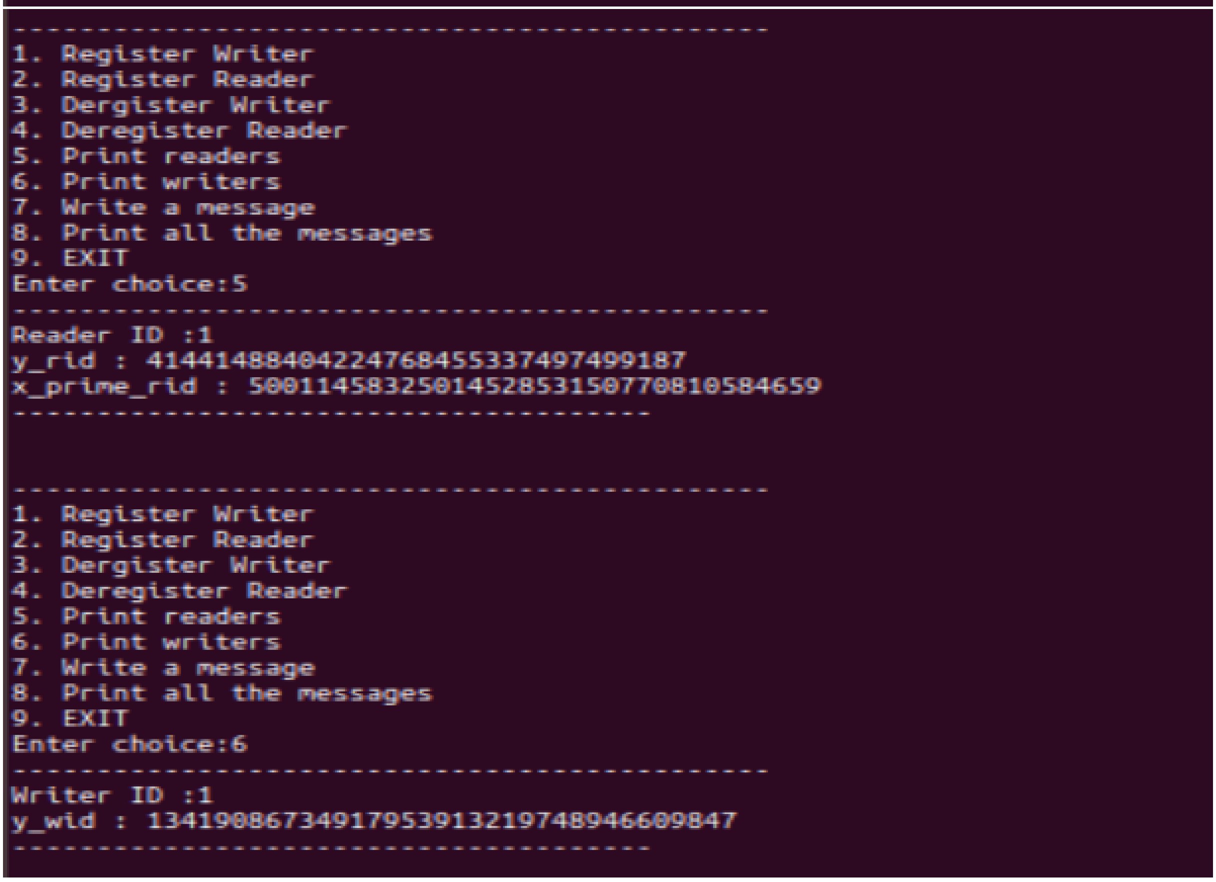

Figure 24: Code for demonstration



```
Starting program: /home/ritu/GmpPbc/MRMWBKSE/encryption
N: 15421993861796871567745419258211939
g: 161132099

Public Paramater created successfully

N: 15421993861796871567745419258211939
g: 161132099
g^alpha: 813211510
H values:
191498516977496392 370179345487175168 221975942138405392 177399102934636605 102935439727837475 156400116151532733 17941
7641506459696 617493118170716539 695804611739079403 1966349258834940
X4: 538048332

Master public key created successfully

alpha: 4187453793590279546664818028147925
X3: 222386207
u_prime values :
11676559393761732592787988997325227 14622426950839349139225223933796612 9844958176285383969611153772039045 244908879941
2496590776424405032780 7118885385192555390241663089477317 648257273934661895573815644105563 931280848708632486571090899
2711944 12484171399026149134869189988465836 8956797103094468664869886892224263 13614345779896532760052310617201073

Master secret key created successfully
```

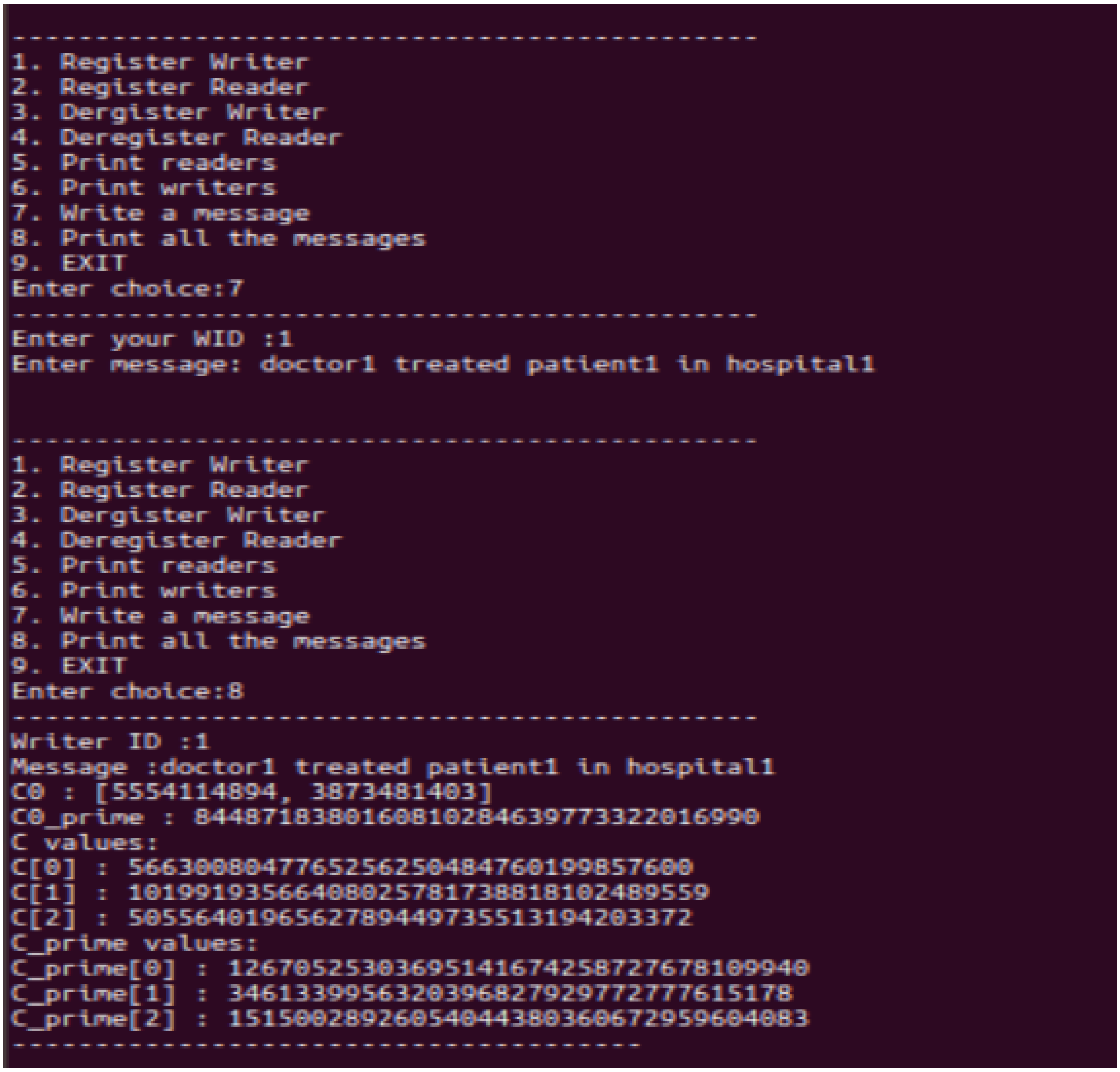


```
--------------------------------------------
1. Register Writer
2. Register Reader
3. Dergister Writer
4. Deregister Reader
5. Print readers
6. Print writers
7. Write a message
8. Print all the messages
9. EXIT
Enter choice:1
--------------------------------------------
Registration successful. Your WID is = 1


--------------------------------------------
1. Register Writer
2. Register Reader
3. Dergister Writer
4. Deregister Reader
5. Print readers
6. Print writers
7. Write a message
8. Print all the messages
9. EXIT
Enter choice:2
--------------------------------------------
Registration successful. Your RID is = 1
```



```
--------------------------------------------
1. Register Writer
2. Register Reader
3. Dergister Writer
4. Deregister Reader
5. Print readers
6. Print writers
7. Write a message
8. Print all the messages
9. EXIT
Enter choice:5
--------------------------------------------
Reader ID :1
y_rid : 41441488404224768455337497499187
x_prime_rid : 5001145832501452853150770810584659
------------------------------------


--------------------------------------------
1. Register Writer
2. Register Reader
3. Dergister Writer
4. Deregister Reader
5. Print readers
6. Print writers
7. Write a message
8. Print all the messages
9. EXIT
Enter choice:6
--------------------------------------------
Writer ID :1
y_wid : 13419086734917953913219748946609847
------------------------------------
```

```
-----------------------------------------------
1. Register Writer
2. Register Reader
3. Dergister Writer
4. Deregister Reader
5. Print readers
6. Print writers
7. Write a message
8. Print all the messages
9. EXIT
Enter choice:3
-----------------------------------------------
Enter your writer ID:2
Error : Writer ID not found


-----------------------------------------------
1. Register Writer
2. Register Reader
3. Dergister Writer
4. Deregister Reader
5. Print readers
6. Print writers
7. Write a message
8. Print all the messages
9. EXIT
Enter choice:4
-----------------------------------------------
Enter your reader ID:1
Reader 1 successfully deleted


-----------------------------------------------
1. Register Writer
2. Register Reader
3. Dergister Writer
4. Deregister Reader
5. Print readers
6. Print writers
7. Write a message
8. Print all the messages
9. EXIT
Enter choice:5
-----------------------------------------------
No readers found
```



```
-------------------------------------------------
1. Register Writer
2. Register Reader
3. Dergister Writer
4. Deregister Reader
5. Print readers
6. Print writers
7. Write a message
8. Print all the messages
9. EXIT
Enter choice:7
-------------------------------------------------
Enter your WID :1
Enter message: doctor1 treated patient1 in hospital1


-------------------------------------------------
1. Register Writer
2. Register Reader
3. Dergister Writer
4. Deregister Reader
5. Print readers
6. Print writers
7. Write a message
8. Print all the messages
9. EXIT
Enter choice:8
-------------------------------------------------
Writer ID :1
Message :doctor1 treated patient1 in hospital1
C0 : [5554114894, 3873481403]
C0_prime : 844871838016081028463977332201699O
C values:
C[0] : 56630080477652562504847601998576O0
C[1] : 1019919356640802578173881810248955 9
C[2] : 5055640196562789449735513194203372
C_prime values:
C_prime[0] : 126705253036951416742587276781099 40
C_prime[1] : 3461339956320396827929772777615178
C_prime[2] : 1515002892605404438036067295960408 3
------------------------------------
```

Figure 25: Output of the demo code

9 LSSS Access structure

In MWMRBKSE, each query Q is considered as a policy (Boolean formula). Formally, a Boolean formula is represented as an LSSS access structure which consists of a secret sharing matrix a. A is an × m share generating matrix where the number of rows, i.e., is equal to the number of keywords in Q. We assume that no keyword is repeated in query Q.

Token Generation algorithm (Fig. 26) takes in the boolean query from the user in an LSSS format, based on the algorithm shown in Fig.27. Input query is in the post-fix form [5].

We converted the post-fix input query to a binary tree, with node values as (AND,OR,SET(attributes)). Every node also has a vector which will be assigned values when tree is passed to LSSS algorithm. LSSS matrix will be generated using the vectors of the nodes which have attribute values.

Code for LSSS matrix generation is shown in the Fig. 28. Output for this code with an execution is shown in the Fig. 29.

6. TokGen(pp, mpk, U_{RID}, Q)

1 For the input Boolean query $Q = (\mathbb{A}, \rho, \mathbb{T})$, it first selects a random vector v such that $\mathbb{A}_i \cdot v = x_{RID}$ for $1 \leq i \leq \ell$.

2 It selects $r_i \in Z_N$, $V_{0i}, V_{1i}, V_{2i}, V_{3i} \in G_{p_3}$ at random for $1 \leq i \leq \ell$..

3 Then computes $D_{0i} = g^{\mathbb{A}_i v} \cdot V_{0i}$, $D_{1i} = (u_{\rho(i)})^{r_i t_{\rho(i)}} \cdot V_{1i}$, $D_{2i} = (u'_{i\_RID})^{r_i} \cdot V_{2i}$, $D_{3i} = g^{r_i} \cdot V_{3i}$.

4 From $(\mathbb{A}, \rho)$, DR computes a set of minimum subset I' (that can satisfy a query Q) as $I' = \{(I_1, \sigma_1), (I_2, \sigma_2), \ldots\}$ where $I_k \subseteq \{1, 2, \ldots, \ell\}$ and $\sigma_k = \{\sigma_{jk} | 1 \leq j \leq |I_k|\}$ satisfying Eq. (1).

5 Finally, DR possessing RID outputs a token for query Q as $T_Q = (D_{0i}, D_{1i}, D_{2i}, D_{3i}, I', RID)$ for $1 \leq i \leq \ell$.

Figure 26: Algorithm for Token Generation based on the input query

[5]post-fix(A and B or C)=A B and C or. Basically, post-fix expression is obtained when a binary tree is traversed in a post-order manner

```
Subroutine vector(T: tree);

Input: tree T, a Boolean formula
in the form of a binary tree
(each node has 0 or 2 children)
with AND and OR as nodes

Output: Returns a vector v(x) for each node
x of T, with all vectors of same length

Notes: In implementation,
T is a record containing T.node
(possible values are: AND, OR),
T.left, T.right, T.L, T.vector; so v(x)=T.vector
{
L=0  /* length of vector
maxL=0  /*maximal length of vector
IF T is a leaf  /*T.left=NIL; x=T;
THEN RETURN T.vector= empty
/* v(x)=T.vector
ELSE
CHILDV(T, maxL);
Padding(T, maxL)
}

Function Padding(T, maxL) {
IF L< maxL
{ add maxL-L 0's at the end of v(x); L=maxL};
IF x is not a leaf
{
Padding(T.left, maxL);
Padding(T.right, maxL)
}
}
```

```
Function CHILDV(T, maxL)  {
IF T.node=OR
{
ORCHILD(T.left, T.vector, maxL);
ORCHILD(T.right, T.vector, maxL)
}
IF T.node=AND
{
ANDLEFTCHILD(T.left, T.vector, maxL);
ANDRIGHTCHILD(T.left, T.vector, maxL)
}
}

Function ORCHILD(X:tree, Y, maxL)  {
X.vector=Y /* returns same vector before
padding for the root node
CHILDV(X, maxL)
}

Function ANDCHILDLEFT(X:tree, Y, maxL) {
X.L= X.L+1  /* length of v(x) increases by 1
IF X.L> maxL
{
maxL=X.L  /* increases maxL if needed
X.vector = Y | 1  /* adds 1 at
the end of vector and returns it
}
CHILDV(X, maxL)
}

Function ANDCHILDRIGHT(X;tree, Y, maxL) {
X.L= X.L+1;
IF X.L> maxL
maxL=X.L;
X.vector=(0^{X.L-1},-1)/*X.L-1 0's before -1
CHILDV(X, maxL)
}
```

Figure 27: Algorithm for generating LSSS matrix based on input boolean query tree

```
QueryTree* childv(QueryTree*root,int* maxL);
QueryTree* orchild(QueryTree* node,vector<int> vector_from_parent,int *maxL);
QueryTree* leftandchild(QueryTree* node,vector<int> vector_from_parent,int *maxL);
QueryTree* rightandchild(QueryTree* node,vector<int> vector_from_parent,int *maxL);

//LSSS
QueryTree* rightandchild(QueryTree* node,vector<int> vector_from_parent,int *maxL)
{
    int length = vector_from_parent.size()+1;
    //int length = (node->vec).size()+1;
    (node->vec).clear();
    for(int i=0;i<length-1;i++)
        (node->vec).push_back(0);
    (node->vec).push_back(-1);
    if((node->vec).size() > *maxL)
        *maxL = (node->vec).size();
    node = childv(node, maxL);
    return node;
}

QueryTree* leftandchild(QueryTree* node,vector<int> vector_from_parent,int *maxL)
{
    (node->vec).clear();
    for(int i=0;i<vector_from_parent.size();i++)
        (node->vec).push_back(vector_from_parent[i]);
    (node->vec).push_back(1);
    if((node->vec).size() > *maxL)
        *maxL = (node->vec).size();
    node = childv(node, maxL);
    return node;
}
QueryTree* orchild(QueryTree* node,vector<int> vector_from_parent,int *maxL)
{
    (node->vec).clear();
    for(int i=0;i<vector_from_parent.size();i++)
        (node->vec).push_back(vector_from_parent[i]);
    node = childv(node, maxL);
    return node;
}
QueryTree* childv(QueryTree* root,int* maxL)
{
    if(root->value == "OR" || root->value == "or")
    {
        root->left = orchild(root->left,root->vec,maxL);
        root->right = orchild(root->right,root->vec,maxL);
    }
    if(root->value == "AND" || root->value == "and")
    {
        root->left = leftandchild(root->left,root->vec,maxL);
        root->right = rightandchild(root->right,root->vec,maxL);
    }
    return root;
}
QueryTree* padding(QueryTree*root,int* maxL)
{
    if((root->vec).size() < *maxL)
    {
        int pads = *maxL - (root->vec).size();
        for(int i=0;i<pads;i++)
            (root->vec).push_back(0);
    }
    if(root->is_leaf == 0)
    {
        root->left = padding(root->left,maxL);
        root->right = padding(root->right,maxL);
    }
    return root;
}
QueryTree* LSSS(QueryTree *root)
{
    if(root != NULL)
    {
        int maxL = 0;
        if(root->is_leaf == 1)
            return root;
        else
        {
            root = childv(root,&maxL);
            cout<<maxL;
            root = padding(root,&maxL);
        }
    }
}
```

```cpp
int main()
{
    char a[200];
    cout<<"Enter query: ";
    scanf("\n");
    scanf("%[^\n]s",a);
    vector<string> words;
    char *pch;
    pch = strtok(a," ");
    while(pch != NULL)
    {
        words.push_back(convert_to_String(pch));
        pch = strtok(NULL," ");
    }
    cout<<"Following are the words in the query:\n";
    for(int i=0;i<words.size();i++)
        cout<<words[i]<<endl;
    cout<<"----------------------------------------------\n";
    QueryTree* r = constructTree(words);
    r = LSSS(r);
    map<string,vector<int>> LSSS;
    cout<<"\nFollowing is the level order traversal of query:\n";
    printLevelOrder(r);
    cout<<"----------------------------------------------";
    cout<<"\nFollowing is the in order traversal of query:\n";
    inOrder(r,LSSS);
    cout<<"\n----------------------------------------------\n";
    cout<<"\nFollowing is the LSSS matrix:\n";
    for(auto i: LSSS)
    {
        cout<<"Attribute: "<<i.first<<"\t";
        cout<<"Vector: [";
        for(int j=0;j<i.second.size();j++)
            cout<<i.second[j]<<" ";
        cout<<"]"<<endl;
    }
    cout<<"\n----------------------------------------------\n";
    return 0;
}
```

Figure 28: Code for generating LSSS matrix from input post-fix boolean query

```
ritu@ritu-VirtualBox:~/GmpPbc/MRMWBKSE$ ./a.out
Enter query: doctor1 doctor2 and city1 and
Created tree successfully

Following is the level order traversal of query:
{ and 0  0   }
{ and 1  0   }{ city1 -1  0   }
{ doctor1 1  1   }{ doctor2 0  -1   }
-----------------------------------------------
Following is the in order traversal of query:
{ doctor1 1  1   }{ and 1  0   }{ doctor2 0  -1   }{ and 0  0   }{ city1 -1  0   }
-----------------------------------------------

Following is the LSSS matrix:
Attribute: city1        Vector: [-1 0 ]
Attribute: doctor1      Vector: [1 1 ]
Attribute: doctor2      Vector: [0 -1 ]

-----------------------------------------------
```

Figure 29: Output of LSSS matrix from input post-fix boolean query