Cryptography and Network Security Lab

Digital Assignment-2

Sajag Agrawal

21BCT0438

Q1 Aes

Code

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
using namespace std;
#define AES_BLOCK_SIZE 16
#define AES_KEY_SIZE 32
#define NUM_ROUNDS 14

typedef uint8_t byte;

// AES S-box and inverse S-box
static const byte sbox[256] = {
   0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
   0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
   0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
   0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
   0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
   0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
   0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
   0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
   0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
```

```
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
};

// AES S-box and inverse S-box
static const byte rsbox[256] = {
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,
    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,
    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,
    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,
    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,
    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,
    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,
    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,
    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d
};
```

```
// AES round constants
static const byte Rcon[11] = {
  0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36
};

byte roundKey[AES_KEY_SIZE];

// Function declarations
byte GMul(byte a, byte b);
void InvShiftRows(byte* state);
void InvSubBytes(byte* state);
void InvMixColumns(byte* state);

void KeyExpansion(const byte* key);
void SubBytes(byte* state);
void ShiftRows(byte* state);
void MixColumns(byte* state);
void AddRoundKey(byte* state, const byte* roundKey);
void AES_Encrypt(const byte* plaintext, byte* ciphertext);
void AES_Decrypt(const byte* ciphertext, byte* plaintext);

// Galois multiplication
byte GMul(byte a, byte b) {
  byte p = 0;
  byte counter;
  byte hi_bit_set;
  for (counter = 0; counter < 8; counter++) {
    if (b & 1) {
      p ^= a;
```

```
        }
        hi_bit_set = (a & 0x80);
        a <<= 1;
        if (hi_bit_set) {
            a ^= 0x1b; /* x^8 + x^4 + x^3 + x + 1 */
        }
        b >>= 1;
    }
    return p;
}

// Key Expansion function
void KeyExpansion(const byte* key) {
    memcpy(roundKey, key, AES_KEY_SIZE);

    int bytesGenerated = AES_KEY_SIZE;
    int rconIteration = 1;
    byte temp[4];

    while (bytesGenerated < (AES_BLOCK_SIZE * (NUM_ROUNDS + 1))) {
        memcpy(temp, roundKey + bytesGenerated - 4, 4);

        if (bytesGenerated % AES_KEY_SIZE == 0) {
            // Rotate left
            uint8_t t = temp[0];
            temp[0] = temp[1];
            temp[1] = temp[2];
            temp[2] = temp[3];
            temp[3] = t;
```

```
            // SubWord: apply S-box
            for (int i = 0; i < 4; ++i) {
                temp[i] = sbox[temp[i]];
            }
// XOR with Rcon
            temp[0] ^= Rcon[rconIteration++];
        } else if (AES_KEY_SIZE > 24 && bytesGenerated % AES_KEY_SIZE == 16) {
            // SubWord: apply S-box
            for (int i = 0; i < 4; ++i) {
                temp[i] = sbox[temp[i]];
            }
        }

        for (int i = 0; i < 4; ++i) {
            roundKey[bytesGenerated] = roundKey[bytesGenerated - AES_KEY_SIZE] ^ temp[i];
            bytesGenerated++;
        }
    }
}

// SubBytes transformation
void SubBytes(byte* state) {
    for (int i = 0; i < AES_BLOCK_SIZE; ++i) {
        state[i] = sbox[state[i]];
    }
}

// ShiftRows transformation
```

```
void ShiftRows(byte* state) {
    byte tmp;

    // Second row
    tmp = state[1];
    state[1] = state[5];
    state[5] = state[9];
    state[9] = state[13];
    state[13] = tmp;

    // Third row
    tmp = state[2];
    state[2] = state[10];
    state[10] = tmp;
    tmp = state[6];
    state[6] = state[14];
    state[14] = tmp;
// Fourth row
    tmp = state[15];
    state[15] = state[11];
    state[11] = state[7];
    state[7] = state[3];
    state[3] = tmp;
}

// MixColumns transformation
void MixColumns(byte* state) {
    byte tmp[4];
```

```
    for (int i = 0; i < AES_BLOCK_SIZE; i += 4) {
        tmp[0] = GMul(0x02, state[i]) ^ GMul(0x03, state[i + 1]) ^ state[i + 2] ^ state[i + 3];
        tmp[1] = state[i] ^ GMul(0x02, state[i + 1]) ^ GMul(0x03, state[i + 2]) ^ state[i + 3];
        tmp[2] = state[i] ^ state[i + 1] ^ GMul(0x02, state[i + 2]) ^ GMul(0x03, state[i + 3]);
        tmp[3] = GMul(0x03, state[i]) ^ state[i + 1] ^ state[i + 2] ^ GMul(0x02, state[i + 3]);

        for (int j = 0; j < 4; ++j) {
            state[i + j] = tmp[j];
        }
    }
}

// AddRoundKey transformation
void AddRoundKey(byte* state, const byte* roundKey) {
    for (int i = 0; i < AES_BLOCK_SIZE; ++i) {
        state[i] ^= roundKey[i];
    }
}

// AES Encryption function
void AES_Encrypt(const byte* plaintext, byte* ciphertext) {
    byte state[AES_BLOCK_SIZE];
    int round;

    memcpy(state, plaintext, AES_BLOCK_SIZE);

    AddRoundKey(state, roundKey);

    for (round = 1; round < NUM_ROUNDS; ++round) {
```

```
        SubBytes(state);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(state, roundKey + round * AES_BLOCK_SIZE);
    }

SubBytes(state);
    ShiftRows(state);
    AddRoundKey(state, roundKey + NUM_ROUNDS * AES_BLOCK_SIZE);

    memcpy(ciphertext, state, AES_BLOCK_SIZE);
}
// AES Decryption function
void AES_Decrypt(const byte* ciphertext, byte* plaintext) {
    byte state[AES_BLOCK_SIZE];
    int round;

    memcpy(state, ciphertext, AES_BLOCK_SIZE);

    AddRoundKey(state, roundKey + NUM_ROUNDS * AES_BLOCK_SIZE);

    for (round = NUM_ROUNDS - 1; round > 0; --round) {
        InvShiftRows(state);
        InvSubBytes(state);
        AddRoundKey(state, roundKey + round * AES_BLOCK_SIZE);
        InvMixColumns(state);
    }

    InvShiftRows(state);
```

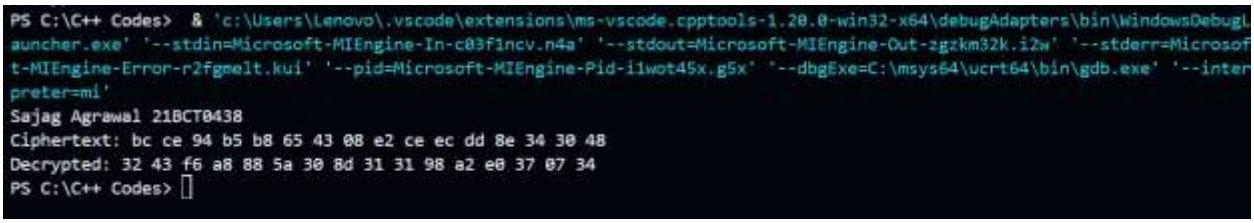
```
    InvSubBytes(state);
    AddRoundKey(state, roundKey);

    memcpy(plaintext, state, AES_BLOCK_SIZE);
}
// Inverse SubBytes transformation
void InvSubBytes(byte* state) {
    for (int i = 0; i < AES_BLOCK_SIZE; ++i) {
        state[i] = rsbox[state[i]];
    }
}

// Inverse ShiftRows transformation
void InvShiftRows(byte* state) {
    byte tmp;

    // Second row
    tmp = state[1];
    state[1] = state[13];
    state[13] = state[9];
    state[9] = state[5];
    state[5] = tmp;

    // Third row
    tmp = state[2];
    state[2] = state[10];
    state[10] = tmp;
    tmp = state[6];
    state[6] = state[14];
```

```
    state[14] = tmp;
// Fourth row
    tmp = state[3];
    state[3] = state[7];
    state[7] = state[11];
    state[11] = state[15];
    state[15] = tmp;
}

// Inverse MixColumns transformation
void InvMixColumns(byte* state) {
    byte tmp[4];

    for (int i = 0; i < AES_BLOCK_SIZE; i += 4) {
        tmp[0] = GMul(0x0E, state[i]) ^ GMul(0x0B, state[i + 1]) ^ GMul(0x0D, state[i + 2]) ^ GMul(0x09, state[i + 3]);
        tmp[1] = GMul(0x09, state[i]) ^ GMul(0x0E, state[i + 1]) ^ GMul(0x0B, state[i + 2]) ^ GMul(0x0D, state[i + 3]);
        tmp[2] = GMul(0x0D, state[i]) ^ GMul(0x09, state[i + 1]) ^ GMul(0x0E, state[i + 2]) ^ GMul(0x0B, state[i + 3]);
        tmp[3] = GMul(0x0B, state[i]) ^ GMul(0x0D, state[i + 1]) ^ GMul(0x09, state[i + 2]) ^ GMul(0x0E, state[i + 3]);

        for (int j = 0; j < 4; ++j) {
            state[i + j] = tmp[j];
        }
    }
}

int main() {
```

```
  printf("Sajag Agrawal 21BCT0438\n");
  byte key[AES_KEY_SIZE] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f
  };

  byte plaintext[AES_BLOCK_SIZE] = {
    0x32, 0x43, 0xf6, 0xa8, 0x88, 0x5a, 0x30, 0x8d,
    0x31, 0x31, 0x98, 0xa2, 0xe0, 0x37, 0x07, 0x34
  };

  byte ciphertext[AES_BLOCK_SIZE];
  byte decrypted[AES_BLOCK_SIZE];

  KeyExpansion(key);

  AES_Encrypt(plaintext, ciphertext);

  printf("Ciphertext: ");
  for (int i = 0; i < AES_BLOCK_SIZE; ++i) {
    printf("%02x ", ciphertext[i]);
  }
  printf("\n");

AES_Decrypt(ciphertext, decrypted);

  printf("Decrypted: ");
```

```
    for (int i = 0; i < AES_BLOCK_SIZE; ++i) {
        printf("%02x ", decrypted[i]);
    }
    printf("\n");


    return 0;
}
```

Output-



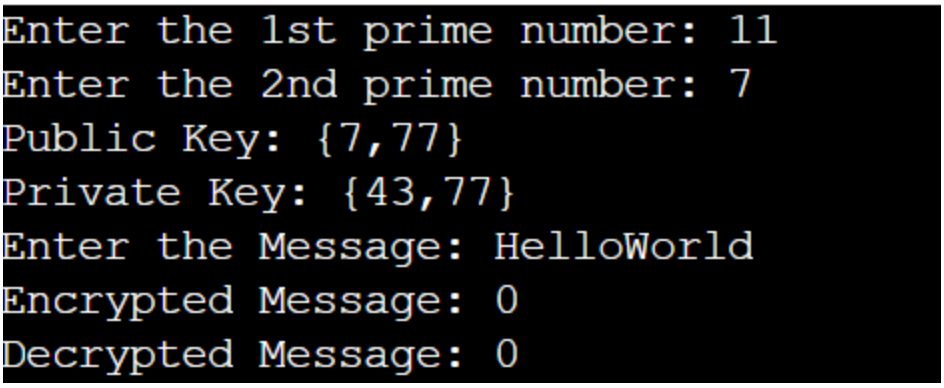
Q2 RSA

Code-import java.util.*;

```
class Main {
    public static long gcd(long a, long b){
        if (a == 0){
            return b;
        }
        return gcd(b%a,a);
    }
    public static long modinverse(long a, long b){
        for (int i = 1; i < b; i++){
            if ((a * i)%b == 1){
                return i;
            }
        }
        return 1;
```

```
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the 1st prime number:");
        long p = sc.nextLong();
        long e;
        System.out.println("Enter the 2nd prime number:");
        long q = sc.nextLong();
        long n = p * q;
        long z = (p - 1) * (q - 1);
        for (e = 2; e < z;e++){
            if (gcd(e,z) == 1){
                break;
            }
        }
        System.out.println("Public Key : {" + e + "," + n + "}");
        long d = modinverse(e,z);
        System.out.println("Private Key : {" + d + "," + n + "}");
        System.out.println("Enter the Message: ");
        long message = sc.nextLong();
        double cipher_text = Math.pow(message,e)%n;
        double plain_text = Math.pow(cipher_text,d);
        System.out.println("Encrypted Message : " + cipher_text);
        System.out.println("Decrypted Message : " + plain_text);
    }
}
```

Output-



Q3 Diffie Hellman Exchange

Code

```
#include<bits/stdc++.h>
using namespace std;
int private_key_1=0;
int private_key_2=0;
void keyexchange(int p,int g){
 double a,b;
 cout<<"Enter private number 1: "<<endl;
 cin>>a;
 cout<<"Enter private number 2: "<<endl;
 cin>>b;
 cout<<"Public values are computing..."<<endl;
 int sub1=pow(g, a);
 int x;
 x=sub1%p;
 int sub2=pow(g,b);
 int y=sub2%p;
 cout<<"Exchanging the public values "<<endl;
//  int temp;
//  temp=x;
```

```
//  x=y;
//  y=temp;
 cout<<"Symmetric keys are computing "<<endl;
 int ka,kb;
 int sub3=pow(y,a);
 ka=sub3%p;
 int sub4=pow(x,b);
 kb=sub4%p;
 cout<<"Secret key 1 is: "<<ka<<endl;
 cout<<"Secret key 2 is: "<<kb<<endl;



}
int main(){
   int p,g;
   cout<<"Enter value of p: "<<endl;
   cin>>p;
   cout<<"Enter value of g: "<<endl;
   cin>>g;
   keyexchange(p,g);
   return 0;
}
```

Output

```
PS C:\C++ Codes>  & 'c:\Users\Lenovo\.vscode\extensions\ms-vscode.cpptools-1.20.0-win32-x64\debugAdapters\bin\WindowsDebugL
auncher.exe' '--stdin=Microsoft-MIEngine-In-ulft3rgx.1he' '--stdout=Microsoft-MIEngine-Out-sg5jcc1f.a0c' '--stderr=Microsof
t-MIEngine-Error-hibltalr.1dz' '--pid=Microsoft-MIEngine-Pid-kt4r5kv0.wgr' '--dbgExe=C:\msys64\ucrt64\bin\gdb.exe' '--inter
preter=mi'
Enter value of p:
7
Enter value of g:
5
Enter private number 1:
3
Enter private number 2:
4
Public values are computing...
Exchanging the public values
Symmetric keys are computing
Secret key 1 is: 1
Secret key 2 is: 1
```