ANALYSIS OF RSA ALGORITHM

(Rivest-Shamir-Adleman algorithm)

PROJECT MEMBERS:

1. Shiva Tejaswi Rampally (1213135428)
2. Ravi Teja Upmaka (1213086704)
3. Sai Pranav Rangabhatla (1213140784)

1. INTRODUCTION

RSA algorithm was developed by Ron Rivest, Adi Shamir, and Len Adleman back in 1977. RSA follows Public Key Cryptography where the encryption is done using set of two keys namely public key and private key. In this cryptography, the data is encrypted using public key which is widely known by all the parties and the decryption can be done by the receiver's private key which is known only by the receiver. In general, Public key cryptography provides encryption and authenticity. RSA algorithm follows public key cryptography which provides above traits of PKC. RSA algorithm provides both encryption and authentication.

RSA is widely used for encryption of digital data. RSA provides both secrecy and digital signatures i.e., electronically signing the data and used in various systems like Electronic Credit Card payment systems. RSA algorithm requires generation of large prime numbers in the beginning of the algorithm. We have used GNU Multiple Precision Arithmetic Library to handle large prime numbers and their operations. We have used Miller Rabin Primality Testing for testing whether the large random number generated is prime or not. Although RSA being considered one of the secure algorithms, plain RSA is susceptible to various attacks. Therefore, some techniques have been proposed in various standards which would alter the plain RSA algorithm to make it secure against the attacks. One technique implemented in this project is the implementation of RSA OAEP algorithm which imposes padding onto the input message thereby making the RSA algorithm non-deterministic. This scheme would therefore be secure against attacks like Adaptive Chosen Ciphertext Attack and Chosen Plaintext Attack. Studies shows that numerous attacks have been found to be a threat to RSA algorithm. These attacks merely illustrate the problems of wrong usage of algorithms rather than being a damaging attack to the algorithm. Along with the implementation of algorithm, some of the attacks have been studied in this project.

As stated above in this project we have implemented the RSA OAEP algorithm which is secure against some of the attacks which are possible against plain RSA algorithm. We have implemented the RSA OAEP encryption scheme from the RFC 8017 by carefully following all the guidelines listed in this standard. We have also learnt about different attacks which are possible on plain RSA and the possible techniques which could be incorporated in the algorithm to keep the RSA algorithm secure against these attacks.

2. IMPLEMENTAION DESCRIPTION

RSA- ALGORITHM

The algorithm is used for communicating between two parties say A and B.

1. Large prime numbers P and Q are chosen. These large prime numbers are generally random (The random numbers are checked using primality test whether they are prime or not).
2. N = P*Q is calculated (Generally N is 1024 bits long).
3. $\phi(N)=(P-1)(Q-1)$ is calculated.
4. After calculating $\phi(N)$, value 'e' is calculated such that e and $\phi(N)$ are co-primes which means that GCD (e, $\phi(N)$) =1.
5. After calculating e, d is calculated such that (d*e) mod $\phi(N)$ =1.
6. Now we have N, d, e with us and the pair (d,N) is considered as the private key and (e,N) is considered as the public key.
7. After the key generation, when A wants to send a message to B, A encrypts the message (Assume M) using B's public key e. and the message can be formed as $C= M^e \bmod N$ where C is the cipher text formed. Since the Algorithm follows PKC, the encrypted message can only be decrypted using the private key of B which is d.
8. When B wants to decrypt the message he/she will do $M= C^d= M^{ed} \bmod N = M \bmod N$ which gives the message M.

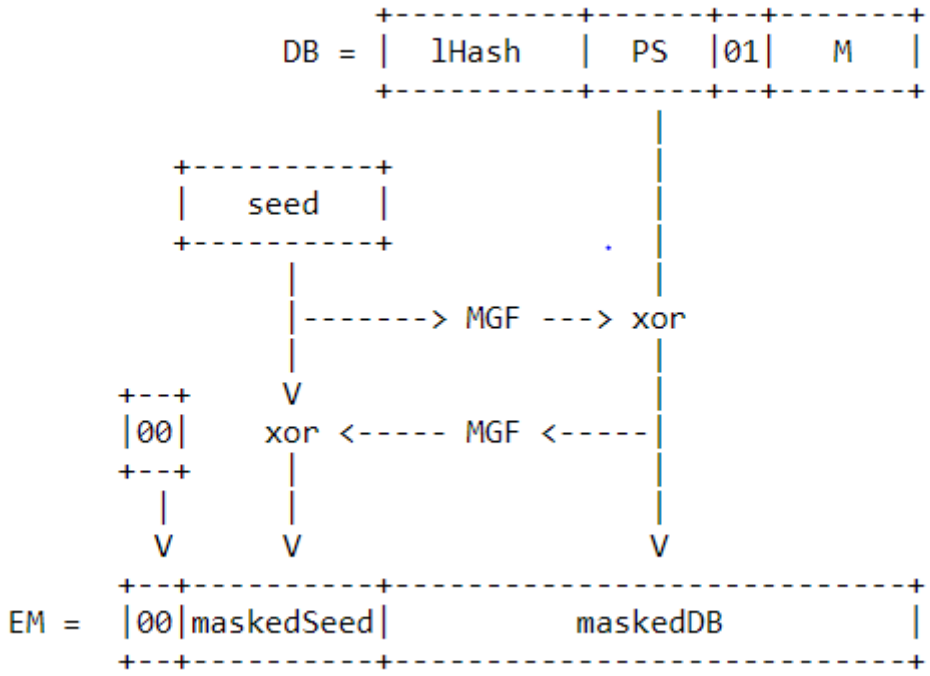
Generation of large prime numbers p and q

The first task involved in the project is the generation of large prime numbers p and q. We were unaware of a data type to store large integers. We then learnt the fact that GMP library can be used to store large integers. We have initially generated large random numbers using "mpz_urandom" function which takes as input the number of bits of the random number to be generated. We then stored the large random numbers generated in the mpz_t data type variables. We then do a primality test by calling the Miller Rabin Primality Test function by passing the random number generated above as a parameter to the method. The Miller Rabin Primality Test checks whether the large random number generated is probable prime or composite. If it is composite, we generate another random number and do the same process of checking the primality of the number. We continue this process until we obtain a large prime number. We implemented the Miller Rabin Primality Test using the standard FIPS 186-3.

RSA OAEP

We have learnt that plain RSA is susceptible to certain attacks like Adaptive Chosen Ciphertext Attack and Chosen Plaintext Attacks. Chosen Plaintext attacks occur if the encryption algorithm is deterministic in nature. We made sure that the algorithm is non-deterministic by generating an element of randomness for every plaintext. For this purpose, we have implemented the RSA OAEP algorithm from the RFC 8017(PKCS v2.2) which encodes the given plain text by introducing an element of randomness called the seed and thus making it non-deterministic. We then pass the encoded message to the encryption algorithm instead of the original plaintext. Chosen Ciphertext Attack is described later in the report under the Attacks against RSA section of the report.

The following is the way we have implemented the RSA OAEP algorithm



The plaintext M and the label are taken as input to the RSA OAEP encryption algorithm. The message should be of length less than (Modulus n- 2*hlen -2) where hlen is the length of the output of the hash algorithm SHA256.

We have implemented the RSA OAEP according to the RFC 8017 standard. The concept of Mask Generation Function was used in implementing RSA OAEP encryption algorithm. We learnt how to generate outputs of desired length from any inputs using the concept of Mask Generation Functions. The Mask Generation Function has been implemented as per the standard (RFC 8017).

The non-deterministic factor of the OAEP encoding scheme comes from the randomness of the seed generated which ensures that on input of the same message multiple times a random encoded message is obtained which makes it difficult for the adversary to guess the plain text based on the cipher text. This ensure that the scheme is CCA secure.

According to the RFC, the task was to generate a random octet string seed of length hlen. We have generated the seed randomly using c++ rand() modulo 128 which returns characters between ASCII range 0 to 127 for every iteration. The Masked Seed is being appended to the encoded message so that the receiver can decode the encoded message using the masked seed. So, some or other way the receiver need to have the knowledge about the random seed that was used while encoding the message.

The encoded message is in the octet representation. We have converted the encoded message to integer representation using OS2IP primitive before passing it to the encryption algorithm. We then convert the ciphertext obtained from the encryption algorithm to octet representation using I2OSP primitive before decoding the decrypted ciphertext. We have implemented the primitives as per the standard (RFC 8017).

Learning Outcomes:

- We have learnt how to use GNU Multiple Precision Arithmetic Library in handling large integers and performing operations. We have also used gmp_random functions to get a random value of desired length.
- Learnt how to implement RSA OAEP encryption scheme and its role in preventing some of the attacks.
- For messages of different length, the encoded message generated using the OAEP padding scheme is of same length i.e., modulus_n (denoted as 'k' in the standard) which results in same encryption and decryption time for any message.

3. CRYPTO LEARNING

ATTACKS ON RSA ALGORITHM

Brute Force Attack:

- This attack is only possible for small values of p and q.
- RSA algorithm can be broken in the following manner:
    1. Adversary has public key e, n.
    2. He then computes the floor of Sqrt(n) and stores it in a variable 'x'.
    3. He computes x-1 and stores it in 'y'.
    4. He then computes n over y and checks for the remainder. If remainder is 0, then n/y returns one of the prime numbers p or q. Store n/y in p and go to step 6.
    5. If remainder is not 0, then subtract 2 from y and update y = y-2. Repeat steps 4 and 5.
    6. Q = n/p
- Thus the adversary is able to compute the prime numbers p and q from the public key e,n.

Learning Outcomes:

- To ensure that RSA is secure to brute force attack, we have to make sure that we generate large prime numbers p and q. This is because if p and q are large prime numbers, then the attacker trying to implement brute force attack gets stuck in the loop for a very long time because he has to try out every possible odd number.
- We have overcome this attack by ensuring that the prime numbers generated are bits of length 1024 (128 bytes), thus making it difficult for the adversary to computer prime numbers p and q from e and n.

Small Public Exponent E:

- RSA is prone to this attack if the algorithm uses a small public key exponent e and small messages. This is because the cipher text generated using the formula c = m^e mod n. If the values m and e are small, then the result is strictly less than modulus n which would enable the adversary to easily decrypt the cipher text by taking the eth root of the cipher text over the integers.

Learning Outcomes:

- The universal values for public key exponent 'e' are commonly 3, 17 and 65537. In order to move away from generality, we have chosen a large random public key exponent 'e', thus securing RSA against "small public key exponent E" attack as stated above.

Common Modulus Attack:

- This attack is possible in the following scenario: Consider an organization where leader wants to send the same message to multiple people having different values of public key exponent 'e' and same modulus N i.e., leader decides to generate separate <e, n> and <d, n> tag pair for every party that he wants to send a message(m) to, but all the parties have the same p, q and n.
- Suppose the leader wants to send the same message(m) to two parties P1 and P2. Public key pair of P1 is <e1, n> and P2 is <e2, n>, such that gcd(e1, e2) = 1.

  The leader encrypts message m for P1 as: E = m^e1 mod n
  The leader encrypts message m for P2 as: F = m^e2 mod n
- Since the gcd(e1, e2) = 1, according to extended Euclidean algorithm x * e1 + y * e2 = 1.
- The Adversary has values E, F and computes the values x and y from the above equation.
- The Adversary computes the message m by using the following equation

  E^x * F^y mod n = message (m)
  (m^e1)^x * (m^e2)^y mod n
  m^(x * e1 + y * e2) from the extended Euclidean algorithm
  m^1
  m

Learning Outcomes

- In order to ensure that RSA is secure against common modulus attack, we have to keep gcd(e1, e2) > 1 where e1, e2 are public key exponents of different parties.
- Every party should have unique values for p, q, e and d.
- Another way to secure this attack is to pad the given message using RSA-OAEP scheme which we have implemented.

Chosen Plain Text Attack:

- Chosen Plain Text Attack is possible if RSA encryption is a deterministic encryption algorithm i.e., the algorithm has no random component.
- The Adversary can access the encryption oracle and get the cipher texts for various chosen plain texts. This way the adversary can find out the patterns using various plain text, cipher text pairs.
- In this way he will be able to guess the plain text given a new cipher text.

Learning Outcomes:

- From the above the attack, we learnt that the RSA scheme is semantically secure if the adversary cannot distinguish two cipher texts from each other even if he has the knowledge the corresponding plain texts.
- So, we need to ensure that the RSA scheme is non-deterministic in nature.
- This can be achieved either by padding a message or by introducing an element of randomness before encryption like RSA OAEP.

Chose Cipher Text Attack:

- Chosen Cipher Text Attack is possible when the attacker sends various cipher texts to the receiver to decrypt them and then uses these decryptions to recover the plain text.
- The Attack is implemented in the following manner:
    1. Suppose the sender 'A' wants to send a message 'm' to the receiver 'B', he encrypts the message 'm' to produce the cipher text c = m^e mod n.
    2. The Adversary can intercept the cipher text 'c' and chooses a random 'r' and calculates c1 = c * r^e mod n.
    3. The Adversary sends the modified cipher text c1 to the receiver 'B'.
    4. The Receiver 'B' tries to decrypt the cipher text c1 and gets a meaningless message.
    5. On decryption, the receiver 'B' obtains m1 = m^ed * r^ed .
    6. Since ed is 1, m1 = m*r. Receiver 'B' sends back the message m1 to the sender to confirm the message.
    7. The Adversary again intercepts the message m1 and divides the m1 with 'r' to obtain the plain text 'm'.

Learning Outcomes:

- This type of attack is possible if the RSA algorithm is deterministic in nature.
- Using RSA OAEP scheme, CCA attack can be prevented.
- OAEP-based encryption scheme is plaintext aware (in the random oracle model) i.e., roughly an adversary cannot produce a valid ciphertext without actually "knowing" the underlying plaintext

Hastad Broadcast Attack:

- This attack is possible when the parties involved in communication have correct implementation of the RSA algorithm but does the wrong usage.
- The attack is as follows:
    1. Consider a scenario where a message m has to be sent to multiple parties having same public key exponent 'e' but different values for p,q and n.
    2. This implies that each party has different private key exponent <d, n>.
    3. Suppose the leader wants to send the message to more than one party i.e., he wants to broadcast the message. He encrypts the message using the same public key exponent 'e' and the corresponding modulus n for each party and sends the cipher text.
    4. The receivers decrypt their respective cipher texts using their own private key exponent.
    5. The insecurity of the Hastad Broadcast Attack comes from the fact that Chinese Remainder Theorem can be used to recover the common message that was circulated to all the parties without the knowledge of the private key exponent.
    6. This attack is possible when public key exponent 'e' is small and the modulus n for all the parties are relatively prime.
       If n1, n2, n3……. are the moduli for different parties then gcd(n1, n2, n3,……) = 1

Learning Outcomes:

- A large value of public key exponent 'e' should be used to ensure that the attack is secure against Hastad Broadcast Attack.
- All the parties should not share the same public key exponent 'e'.
- The value of public key exponent should not be a generalized value like 3, 17 or 65537.

4. Summary:

Data Security has become a major preference in the present day scenario. One such technique widely used to secure data is by encrypting the data using RSA algorithm. As studied in the project, since plain RSA algorithm is subject to various attacks, a few security measures have been designed that make RSA algorithm highly secure. A few of the measures include the RSA OAEP encryption scheme which encodes the message prior to RSA encryption thus securing the system against attacks like Chosen Cipher Text Attacks and a few other attacks. Another measure that can be taken to prevent the RSA from attack is the generation of large prime numbers p and q. We have implemented certain measures that can be taken to ensure that the RSA algorithm is secure to certain attacks. We have also followed a few of the secure coding principles specified in the SEI guidelines. We have also studied the attacks which are possible on plain RSA and the possible measures which could be taken to prevent these attacks and have implemented a few measures thus securing the RSA against certain attacks.

5. References:

[1] https://tools.ietf.org/html/rfc8017#section-7.1 for RSA OAEP encryption scheme.

[2]https://csrc.nist.gov/csrc/media/publications/fips/186/3/archive/2009-06-25/documents/fips_186-3.pdf for Miller Rabin Primality Testing.

[3]Twenty Years of Attacks on the RSA Cryptosystem, Dan Boneh

[4]Attacks on RSA cryptosystem, Boise State University

[5]GMP Library manual

[6]https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards (Secure Coding Standard as per SEI CMU guidelines)

APPENDIX A:

Code:

RSA.cpp

```
#include<iostream>
#include<math.h>
#include<gmp.h>
#include<gmpxx.h>
#include <time.h>
#include "OAEP.h"

using namespace std;

const int BITS = 1024;
const int rounds = 64;

class RSA_Primes {

		mpz_t rsaP;
		mpz_t rsaQ;
		mpz_t n;
		mpz_t phi;
		mpz_t rsaE;
		mpz_t rsaD;
		mpz_t one;
public: mpz_t message;
public: mpz_t ciphertext;
public: mpz_t plaintext;

gmp_randstate_t rand;
unsigned long int seed;

public:
RSA_Primes() {
		seed = 1;
		mpz_init(rsaP);
		mpz_init(rsaQ);
		mpz_init(n);
		mpz_init(phi);
		mpz_init(rsaE);
		mpz_init(rsaD);
		mpz_init(message);
		mpz_init(ciphertext);
		mpz_init(plaintext);
		mpz_init(one);
		mpz_set_str(one, "1", 10);
		gmp_randinit_default (rand);
		gmp_randseed_ui(rand, seed);
}


// Miller-Rabin Primality Testing to find whether
//a randomly chosen odd number is a prime number or not
bool millerRabin_isPrimeCheck(mpz_t value) {
```

```
        if (mpz_even_p(value) == 1)
                return true;

        mpz_t tempValue;
        mpz_init(tempValue);
        mpz_sub_ui(tempValue, value, (unsigned int)1);

        mpz_t tempStore;
        mpz_init(tempStore);
        mpz_set(tempStore,tempValue);

        unsigned int a = 0;

        // Finding out the value for r which is (value / 2^d)
        while (true) {
                if (mpz_even_p(tempValue) == 0) {
                        break;
                }
                a++;
                mpz_cdiv_q_ui(tempValue, tempValue, (unsigned int)2);
        }

        // Doing the check for 64 rounds
        for (size_t i=0; i<rounds; i++) {
                if (millerTest(tempValue,tempStore,value,a) == false)
                        return false;
        }
        return true;
};

private: bool millerTest(mpz_t m, mpz_t tempStore, mpz_t value, unsigned int a) {

        mpz_t pickRandA;
        mpz_init(pickRandA);

        mpz_t storeX;
        mpz_init(storeX);

        // Picking up a random value between 0 and value-2
        mpz_urandomm(pickRandA, rand, tempStore);

        // Computing pickRandA ^ m mod given value
        mpz_powm(storeX, pickRandA, m, value);

        // Check whether the it is equal to 0 or (value-1)
        if (mpz_cmp_ui(value, (unsigned int )1) == 0 || mpz_cmp(value, tempStore) == 0)
        {
                return true;
        }

        for (size_t j=0; j<a-1; j++) {
                mpz_powm_ui(storeX, storeX, (unsigned long int)2, value);
                if (mpz_cmp_ui(storeX, (unsigned int )1) == 0) {
                        // Not a prime, hence return false
                        return false;
                }
                if (mpz_cmp(storeX, tempStore) == 0)
```

```
                        return true;
        }
        return false;
};


public: void generateLargePrimeNumbers() {

        bool isPrimeFlag = false;

        // get a random integer which has bits of length BITS using mpz_urandomb method
        mpz_urandomb(rsaP,rand,BITS);
        isPrimeFlag = millerRabin_isPrimeCheck(rsaP);
        while(!isPrimeFlag)
        {
                mpz_nextprime(rsaP,rsaP);
                isPrimeFlag = millerRabin_isPrimeCheck(rsaP);
        }

        isPrimeFlag = false;
        // get a random integer which has bits of length BITS using mpz_urandomb method
        mpz_urandomb(rsaQ,rand,BITS);
        isPrimeFlag = millerRabin_isPrimeCheck(rsaQ);
        while(!isPrimeFlag)
        {
                mpz_nextprime(rsaQ,rsaQ);
                isPrimeFlag = millerRabin_isPrimeCheck(rsaQ);
        }

        // Calculate RSA modulus n and phi values
        mpz_mul(n,rsaP,rsaQ);
        mpz_t tempP;
        mpz_t tempQ;

        mpz_init(tempP);
        mpz_init(tempQ);

        mpz_sub(tempP,rsaP,one);
        mpz_sub(tempQ,rsaQ,one);

        mpz_mul(phi,tempP,tempQ);

        gmp_printf("chosen rsaP value: %Zd\n", rsaP);
        cout << "\n";
        gmp_printf("chosen rsaQ value: %Zd\n", rsaQ);
        cout << "\n";
        gmp_printf("RSA modulus n value: %Zd\n", n);

        cout << "\n";
        gmp_printf("RSA Phi value: %Zd\n", phi);
        cout << "\n";

}

public: void calculatePublicPrivateKeys() {
        //                      unsigned long int coPrime = 65537;
        mpz_t gcdValue;
```

```
	mpz_init(gcdValue);

	mpz_t coPrime;
	mpz_init(coPrime);

	mpz_urandomb(coPrime, rand, BITS);

	while(true)
	{
		mpz_gcd(gcdValue,phi,coPrime);
		if(mpz_cmp_ui(gcdValue,(unsigned long int)1)==0)
			break;
		mpz_nextprime(coPrime, coPrime);
	}

	mpz_set(rsaE, coPrime);
	mpz_invert(rsaD, rsaE, phi);

	gmp_printf("chosen E value: %Zd\n", rsaE);
	cout << "\n";
	gmp_printf("chosen D value: %Zd\n", rsaD);
	cout << "\n";
}

void encryptMessage() {
	mpz_powm(ciphertext, message, rsaE, n);
	cout << "\n\nCipher Text: " <<ciphertext<<endl;
}

void decryptMessage() {
	mpz_powm(plaintext, ciphertext, rsaD, n);
	cout<< "\nDecrypted Text:"<<plaintext<<endl;
}

~RSA_Primes() {
	mpz_clear(rsaP);
	mpz_clear(rsaQ);
	mpz_clear(n);
	mpz_clear(phi);
	mpz_clear(rsaE);
	mpz_clear(rsaD);
	mpz_clear(message);
	mpz_clear(ciphertext);
}

};


int main() {

	RSA_Primes * rsa = new RSA_Primes;
	OAEP * oaep = new OAEP;

	// Generate 2 large prime numbers
	rsa->generateLargePrimeNumbers();

	// Calculating public and private keys using modulus n and phi
```

```
        rsa->calculatePublicPrivateKeys();

        char messageLabel[1000];
        cout<<"enter the message label :"<<"\n";
        cin>>messageLabel;

        cout<<"enter the message :"<<"\n";
        char message[1000];
        cin>>message;

        mpz_t encodedIntMessage;
        mpz_init(encodedIntMessage);

        // encoding the given message using OAEP scheme
        unsigned char * encodedOctetRep = oaep->getEncodedMessage(message, messageLabel);

        // Getting the integer representation of the encode message
        oaep->os2ip(encodedOctetRep, encodedIntMessage);
        cout << "Encoded Message in Integer Representation :"<<endl;
        cout<< encodedIntMessage;
        mpz_set(rsa -> message,encodedIntMessage);

        // encrypting the encoded message using the public key rsaE and modulus N
        rsa -> encryptMessage();

        // decrypting the cipher text
        rsa -> decryptMessage();

        // Getting the octet representation of the decrypted ciphter text to decode using OAEP scheme
        unsigned char * octetRep = oaep->i2osp(rsa->plaintext, modulus_n);

        // decoding the decrpted cipher text using OAEP scheme
        unsigned char * decodeMes =  oaep->getDecodedMessage(octetRep, messageLabel);

        // as per SEI coding standards MEM50-CPP
        delete rsa;
        delete oaep;

        // as per the SEI coding standards MEM31-C
        free (octetRep);
        free (decodeMes);
        free (encodedOctetRep);
        return 0;
}
```

OAEP.h

```
#include <iostream>
#include <stdio.h>
#include "gmp.h"
#include <openssl/sha.h>
#include <string.h>
#include <cstdlib>
#include <math.h>

using namespace std;

const int modulus_n = 128;
const int hlen = 32;

class OAEP
{

	int messageLen;
	int messageLabelLen;
	char lHash[hlen+1];
	int dbLen = modulus_n - hlen - 1;

public :

	void os2ip(unsigned char octetString[], mpz_t result);
	unsigned char * i2osp(mpz_t integerValue, int xLen);
	unsigned char * getEncodedMessage(char message[], char messageLabel[]);
	unsigned char * getDecodedMessage(unsigned char encodedMessage[], char
messageLabel[]);
private:
	void maskGenerationFunction(int ceilValue, int len, unsigned char input[], unsigned
char output[]);
};

// Converting Octet Representation to Integer Representation using OS2IP technique

void OAEP::os2ip(unsigned char octetString[], mpz_t result) {
	int len = strlen((char *)octetString);
	mpz_t base;
	mpz_init(base);
	mpz_set_ui(base,(unsigned int)256);

	for (int index=0; index<len; index++) {
		mpz_t mulValue;
		mpz_init(mulValue);
```

```
                mpz_t value;
                mpz_init(value);
                mpz_pow_ui(value,base,(unsigned long int)index);

                mpz_mul_ui(mulValue, value, (unsigned long int)octetString[index]);
                mpz_add(result, result, mulValue);
        }

}

// Converting Integer Representation to Octet Representation using OS2IP technique
unsigned char * OAEP::i2osp(mpz_t integerValue, int xLen) {

        cout << "\n";
        unsigned char * result = (unsigned char *) malloc(2*xLen + 1);
        mpz_t modValue;
        mpz_init(modValue);
        mpz_set_ui(modValue, 256);

        for (int inc=0; inc<2*xLen; inc++) {

                mpz_t remainder;
                mpz_init(remainder);
                mpz_powm_ui(remainder, integerValue, (unsigned int )1, modValue);
                char * block = mpz_get_str(NULL , 10, remainder);
                int blocklen = strlen((char *)block)-1;
                std::string str;
                str.append(block);
                result[inc] = std::atoi(str.c_str());
                mpz_div_ui(integerValue, integerValue, (unsigned int)256);
        }
        result[2*xLen] = '\0';
        return result;
}

//Mask Generation Function to get the required number of output bits

void OAEP::maskGenerationFunction(int ceilValue, int len, unsigned char input[], unsigned
char output[])
{
        for (int counter=0; counter<=ceilValue; counter++)                  //generating hash
a particular number of times and appending the hash values
        {
                unsigned char * obuf;
                obuf = (unsigned char *)malloc(20+1);
                unsigned char bytes1[4];
                bytes1[0] = (counter >> 24) & 0xFF;
```

```
                bytes1[1] = (counter >> 16) & 0xFF;
                bytes1[2] = (counter >> 8) & 0xFF;
                bytes1[3] = counter | 0x00;
                unsigned char hashInput1[len+5];
                for (int i=0; i<len; i++)
                {
                        hashInput1[i] = input[i];
                }
                for (int i=0; i<4; i++)
                {
                        hashInput1[len+i] = bytes1[i];
                }
                // as per SEI coding standards STR32-C
                hashInput1[len+4] = '\0';

                SHA1((unsigned char *)hashInput1,strlen((char*)hashInput1),(unsigned char
*)obuf);
                for (int i=0; i<20; i++)
                {
                        int index = counter*20 + i;
                        output[index] = obuf[i];
                }
                free (obuf);
        }
}

// Encoding the given input message using the specifoed algorithm

unsigned char * OAEP::getEncodedMessage(char message[], char messageLabel[])

{

        int checkLen = modulus_n-(2*hlen)-2;
        messageLen = strlen(message);
        messageLabelLen = strlen(messageLabel);
        int psLen = checkLen - messageLen;
        unsigned char ps[psLen];

        // validating message length
        if (messageLen > checkLen) {
                cout << "message too long";
                return NULL;
        }

        // label length has to be validated
        // hashing the message label to get lhash
```

```
        SHA256((unsigned char *)messageLabel,strlen((char*)messageLabel),(unsigned char
*)lHash);
        cout<<"\n";

        // framing PS(padding string) containing psLen 0's
        for(int index=0;index<psLen;index++)
        {
                ps[index]='0';
        }


        // framing DB using lhash, PS and message
        unsigned char *db;
        db = (unsigned char *) malloc(dbLen+1);
        for(int i=0;i<hlen;i++)
        {
                db[i] = lHash[i];

        }
        for (int j=0; j<psLen; j++) {
                db[j+hlen] = ps[j];

        }

        db[hlen+psLen] = '1';

        for (int k=0; k<messageLen; k++) {
                db[hlen+psLen+1+k] = message[k];


        }
        // according the SEI coding standards STR31-C
        db[dbLen] = '\0';

        cout << "\n";

        // forming a random seed of length hlen
        unsigned char seed[hlen+1];
        srand(time(NULL));

        for(int i=0;i<hlen;i++)
        {
                seed[i] = (char)rand()%128;
        }
        // according the SEI coding standards STR31-C
        seed[hlen] = '\0';
```

```
//mask generation function
cout <<"\n";
int ceilValue = ceil((double)dbLen/20)-1;
unsigned char extendedSeed[20*(ceilValue+1)];
maskGenerationFunction(ceilValue, hlen, seed, extendedSeed);



// Performing XOR Operation on DB and output of MGF(seed)
unsigned char maskedDB[dbLen+1];
for(int i = 0; i < dbLen; i++)
{
        maskedDB[i] = extendedSeed[i] ^ db[i];
}
maskedDB[dbLen] = '\0';

//Mask Generation Function 2
cout<<"\n";

int ceilValue1 = int(ceil((double)hlen/20))-1;
unsigned char extendedSeed1[20*(ceilValue1+1)];
maskGenerationFunction(ceilValue1, dbLen, maskedDB, extendedSeed1);


// Performing XOR Operation on DB and output of MGF(seed)
unsigned char masked_Seed[hlen+1];
for(int i = 0; i < hlen; i++)
{
        masked_Seed[i] = extendedSeed1[i] ^ seed[i];

}
masked_Seed[hlen] = '\0';
cout << "\n";


unsigned char * encoded_message;
encoded_message = (unsigned char *)malloc(modulus_n+1);
encoded_message[0]='0';

int cnt=1;
```

```
        for(int i=0;i<hlen;i++)
        {
                encoded_message[cnt]=masked_Seed[i];
                cnt++;
        }

        for(int i=0;i<dbLen;i++)
        {
                encoded_message[cnt]=maskedDB[i];
                cnt++;
        }

        // According to SEI coding standards STR31-C
        encoded_message[modulus_n] = '\0';

        // as per SEI coding standards MEM31-C
        free (db);
        return encoded_message;
}

//Decoding the message obtained after decryption

unsigned char * OAEP::getDecodedMessage(unsigned char encodedMessage[], char
messageLabel[])
{

        unsigned char * seed;
        unsigned char * db;
        unsigned char * maskedSeed;
        unsigned char * maskedDB;
        seed = (unsigned char *)malloc(hlen);
        db = (unsigned char *)malloc(dbLen);
        maskedSeed = (unsigned char *)malloc(hlen);
        maskedDB = (unsigned char *)malloc(dbLen);

        for (int index=0; index<hlen; index++)
                maskedSeed[index] = encodedMessage[index+1];
        for (int index=0; index<dbLen; index++)
                maskedDB[index] = encodedMessage[index+hlen+1];

        // XORing masked seed with MGF(maskedDB, hlen) to get seed
        int ceilValue = int(ceil((double)hlen/20))-1;
        unsigned char extendedMask[20*(ceilValue+1)];
        maskGenerationFunction(ceilValue, dbLen, maskedDB, extendedMask);

        for(int i = 0; i < hlen; i++)
        {
```

```
                seed[i] = extendedMask[i] ^ maskedSeed[i];

        }



cout << "\n";

        int ceilV = ceil((double)dbLen/20)-1;
        unsigned char * extendedSeed;
        extendedSeed = (unsigned char *)malloc(20*(ceilV+1));

        //Calculating extended seed which is the output of Mask Generation Function

        maskGenerationFunction(ceilV, hlen, seed, extendedSeed);



/*
        for (int index=0; index<dbLen; index++)
                cout << maskedDB[index];*/
        cout << "\n";

        // Xoring extended seed with Masked DB
        for(int index = 0; index < dbLen; index++)
        {
                db[index] = extendedSeed[index] ^ maskedDB[index];

        }
        cout << "\n";

        unsigned char * decodedMessage;
        int lengthMess = 0;
        bool flag = false;

        cout << "Message obtained after decoding : \n";
        for (int index=hlen; index<dbLen; index++)                              //Taking
the message from the encoded message

        {
                if (db[index] == '0' || db[index] == '1')
                        continue;
                lengthMess = index;
                break;
        }
```

```
        decodedMessage = (unsigned char *)malloc(dbLen-lengthMess);
        for (int index=lengthMess; index<dbLen; index++) {
                decodedMessage[index-lengthMess] = db[index];
                cout << db[index];
        }




cout << "\n";

        // as per SEI coding standards MEM31-C
        free (seed);
        free (db);
        free (maskedSeed);
        free (maskedDB);
        return decodedMessage;
}
```

APPENDIX B:

CRYPTO CODING PRACTICES:

Use strong Randomness: As our algorithm RSA is based on large prime numbers and Random number generation is a major part of the implementation. We see that many pseudorandom number generators (PRNG) fail to meet them. For random number generations for primes we have used the "mpz_urandomb" defined in the GMP library.

Clean memory of secret data: Since we deal with Public key cryptography, we can see there is usage of private key for decryption. If the calculated key is passed on from one process to another, there is a chance of key getting compromised/exposed. So, in our project we are clearing all the variables before they go out of scope using delete method.

Use unsigned bytes for data representation: We can see that usage of signed and unsigned bytes might vary and if the bite signed, there is chance that it might behave differently according the sign of the data. In this project, this practice has been implemented and all the unsigned bytes have been used for the variables.

Use separate types for secret and non-secret information: As stated already we are dealing with Public Key cryptography which has public key and private key and this practice states that we should use different data types for these two keys in order to ensure security. But we have used the same data type for both public and private keys. We have used mpz_t to store both private and public keys.

Compare secret strings in constant time: As we know RSA includes the calculations of big prime number and their exponentiation, it is susceptible to Timing attacks by an attacker who can analyze the algorithm for the time required to carry out the operations and can compare the strings byte-by-byte may make the cryptosystem vulnerable to timing attacks. So, it is recommended to compare them in constant time.

APPENDIX C:

SECURE CODING PRACTICES:

EXP54-CPP. Do not access an object outside of its lifetime: This coding practice states that the lifetime of an object begins when sufficient, properly aligned storage has been obtained for it and ends when either destructor is called or the storage for the object has been reused or released. In order to follow this standard, we made sure that the local variables are defined for the storage and initialized within the function.

MEM31-C. Free dynamically allocated memory when no longer needed: For the efficient usage of memory and not to expect any undefined behavior of the variables we need to free the memory once the usage of object ends, we have followed this practice by explicitly freeing the objects using free or delete keywords thereby freeing the memory.

MEM50-CPP. Do not access freed memory: Don't try to evaluate a pointer which has been deallocated using a memory management function in any way (dereferencing the pointer, type casting it, using it an arithmetic expression etc). We followed this practice in our project. We made sure that we have not used a pointer which been freed or deallocated.

STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator: While copying data to a temporary variable one should make sure that it has sufficient place to hold the entire information. Otherwise, it my lead to Buffer Overflow which are frequent when doing operations with strings. In our project, we made sure that while copying the buffer space has sufficient place to hold the entire space and to hold the null termination character at the end.

STR32-C. Do not pass a non-null-terminated character sequence to a library function that expects a string: Passing a string or character sequence which is not properly null terminated to a function might result in accessing the memory that is outside the bounds of this object. We have terminated the character sequences in our project with null to ensure that this doesn't happen.

EXP50-CPP. Do not depend on the order of evaluation for side effects: This coding practice states that if we modify an object or defining a function which might modify object can be qualified as side effects. To overcome this, we have defined all the variables and called the functions in the same order so that they wouldn't exhibit any unspecified or undefined behavior and when there is a single variable evaluated more than one time, we have let the execution happen statement by statement.

EXP46-C. Do not use a bitwise operator with a Boolean-like operand: This coding standard states the mixing of bitwise and relational operators in the same expression can result in a logical error. So, wherever we wanted to do a comparison, we made sure that bit-wise operators are not used and instead used a parenthesized expression carefully even in the case when the bitwise operator is intended.

EXP45-C. Do not perform assignments in selection statements: This coding standard explicitly states some scenarios where assignments are not allowed to be done (Various contexts are stated like controlling expressions of "if", "while" etc). In our project, we have taken care of this coding standard by letting the assignment happen after or before the condition/selection statements to avoid unexpected behaviors and errors.

EXP37-C. Call functions with the correct number and type of arguments: If the functions are not called with correct number and type of arguments the function throws exception or exhibits undefined behavior. So, complying with the standard and since we have number of methods to be called to execute our program we have checked that all functions are called with correct number and type of argument.

STR37-CPP. Arguments to character handling functions must be representable as an unsigned char: The rule states that while using character handling functions, the input parameter should be of unsigned char type. In our project we have used unsigned chars only.