컴퓨터 보안

-과제3 (RSA 구현)-

12171651 컴퓨터공학과 오윤석

우선 RSA의 전체적인 과정을 설명하겠다. 처음 서로 다른 소수인 p와 q를 구한다. 그리고 n은 p*q로 두고 phi(n)을 구한다. Phi(n)은 (p-1)*(q-1)이다. 그 다음 e를 구해야하는데 e는 phi(n)과 서로수인 1보다 크고 phi(n)보다 작은 값이다. 그리고 나서 e*d = 1 mod phi(n)을 만족시키는 d를 구해야 한다. 이 과정들을 거치면 key setup을 완료하게 된다.

공개키 = {e, n}

비밀키 = {d, n}

그리고 암호화 복호화 과정은 간단하다.

암호화: c = M^e mod n (여기서 c는 cipher, M은 암호화하고자 하는 메시지고 0보다 같거나 크고 n보다 작아야한다.)

복호화: M = c^d mod n

여기까지가 RSA의 전체적인 암호화, 복호화 과정이다.

지금부터는 구현한 코드에 대해 설명하겠다.

p, q 생성

```
while (true) //p가 홀수인지 짝수인지 판별
{
    Sleep(1000);
    srand((unsigned int)time(NULL));

    p = 16384 + rand() % 16383;

    if(p%2 == 0)  //짝수일때 다시 random으로 범위안 무작위 수로 세팅
    {
        continue;
    }

    p = miller_rabin(p);

    if (p != 1)
    {
        break;
    }
}
```

P와 q의 값의 범위는 2^14~2^15-1이다. 따라서 코드에서는 rand와 srand를 사용하여 초마다 바뀌는 해당 범위에서 나오는 난수를 구현하였다. 난수가 생성되면 우선 홀수인지 짝수인지 판단한다. 짝수이면 다른 난수를 생성하고 홀수이면 그 다음 과정은 밀러 라빈 소수 판별기를 구현한 함수로 이동하게 했다.

```
int miller_rabin(int p)
{
    int s = 0;
    int d = 0;
    int p2 = 0; // p-1 값
    int a[20] = { -1 }; //p보다 작은 임의의 양수
    int flag = 1; //소수인지 아닌지 확인하는 flag 0일때 소수
    int random = 0;


    p2 = p - 1;

    while (true)
    {
        if ((p2 % 2) == 0) //2로 소인수 분해
        {
            s++;
            p2 = p2 / 2;
        }

        else //2로 안나눠지면 그 값이 d가 된다.
        {
            d = p2;
            break;
        }

    }
```

Miller_rabin 함수에서는 우선 여러가지 값들을 구해야 한다. 우선 들어오는 값 p에대해 소수인지 판별을 해주는 함수임으로 많은 값들이 필요하다. 우선 p2는 p-1이고, 이 값을 2^s*d형태로 나타내야 한다. 따라서 소인수 분해를 통해 s와 d값을 구하였다. (p2값은 무조건 짝수이기 때문에 2로 s값은 1이상이다.)

```
srand((unsigned int)time(NULL));

for (int i = 0; i < 20; i++)
{
    //Sleep(1000);
    a[i] = 1 + rand() % p - 1; //총 20개의 p보다 작은 난수 생성
}

for (int count = 0; count < 20; count++)
{

    if ((count == 0 && square_multi(a[count], d, p) == p+1) || (count == 0 && square_multi(a[count], d, p) == 1))
    {
        flag = 0;
        break;
    }


    for (int c = 0; c < s - 1; c++)
    {
        if (square_multi(a[count], d * pow(2, c), p) == p - 1 || square_multi(a[count], d * pow(2, c), p) == -1)
        {

            flag = 0;

            goto here;
        }
    }
}

here:

if (flag == 0)
{
    return p;
}
```

우리의 테스트는 20회로 한정되어있기 때문에 총 20개의 랜덤한 난수를 a 배열에 저장해둔다.

그 다음 a^d mod p가 1이거나 a^d*2^I mod p가 -1이면 소수이므로 p값을 return한다. 그렇지 않으면 1을 return하여 다시 무작위로 p값을 생성하도록 구현하였다. 그리고 여기서 mod연산을 square and multiply로 구현하였다.

```
int square_multi(int M, int e, int n)
{
    int x = 1;
    int power = M % n;
    std::bitset<32> bs(e);
    int flag = 0;
    int arr[32] = { 0 };

    for (int count = 31; count > 0; count--) //1이 시작되는 왼쪽 부분 찾기
    {
        if (bs[count])
        {
            flag = count;
            break;
        }
    }

    for (int count = 0; count < flag+1; count++)
    {
        arr[count] = bs[count];
    }

    for (int i = 0; i < flag + 1; i++)
    {
        if (arr[i] == 1)
        {
            x = (x * power) % n;
        }
        power = (power * power) % n;
    }

    return x;

}
```

이 함수는 우선 M^e mod n을 구현하려고 만든 함수이다. 우선 count를 통해 bs배열에 이진수 값을 담고 만약 그 값이 1이면 x = (x*power)%n을 해주고, power를 제곱뒤 n으로 mod연산을 해준다. 이 함수는 뒤에 나오는 exp_mod와 비슷하기 때문에 mod 연산에 관한 내용은 거기에서 다루도록 하겠다.

```
n = p * q; // n값 구하기
phi = (p - 1) * (q - 1); //phi 구하기
```

그 다음은 n값과 phi값을 구하였다. 이 부분은 단순 연산이라 생략한다.

```
while(true)
{
    srand((unsigned int)time(NULL));

    e = 1 + rand() % phi - 1; //e값 난수 생성

    tmp = Euclid(e, phi); //확장된 유클리드에서 b3값이 1이면 그 수들은 서로소이다.

    if (tmp == 1) //서로수일때 루프탈출, 아니면 다른 e값으로 하기
    {
        break;
    }
}
```

그리고 나서 e값을 구해야한다. e또한 1보다 크고 phi보다 작은 난수를 생성해야한다. 그 다음은 e와 phi가 서로 서로소인지 확인을 해야한다. 여기서는 확장된 유클리드 호제법을 사용하였다.

```
int Euclid(int e, int phi) //확장된 유클리드 호제법
{

    int q = 0; //몫 1번째인자와 두번째 인자를 나눈 몫
    int a1 = 1;
    int a2 = 0;
    int a3 = e; //1번째 인자
    int b1 = 0;
    int b2 = 1;
    int b3 = phi; //2번째 인자
    int t1 = 0; //a1 - (q * b1)
    int t2 = 0; //a2 - (q * b2)
    int t3 = 0; //a3 - (q * b3)


        while (b3 > 0 || b3 == 1)
        {
            q = a3 / b3;
            t1 = a1 - (q * b1);
            t2 = a2 - (q * b2);
            t3 = a3 - (q * b3);
            a1 = b1;
            a2 = b2;
            a3 = b3;
            b1 = t1;
            b2 = t2;
            b3 = t3;

            if (b3 == 1)
            {
                return b3;
            }
        }

        return b3;
}
```

확장된 유클리드 호제법은 이론과 그대로 코드에 적용하였다. 우선은 a1에는 1, a2 = 0 b1 = 0, b2=1을 두고 1번쨰 인자(a3)와 2번째 인자(b3) 값을 나누어 주고 몫을 q라고 한다. 그리고 나서 b3가 음수나 1이 될때까지 루프를 돌린다. 여기서 a1, a2, a3는 전 단계 b1, b2, b3 값이 되고, a1은 t1, t2, t2값이 된다. 여기서 t1, t2, t3는 주석에 있는 수식 그대로이다. 그리고 나서 루프를 빠져나왔을때 b3값이 1이 되면 이 두수는 서로소이다. 이 함수를 거치고 나서 다시 main으로 돌아오면 b3가 1이면 루프를 탈출하고, 아니면 다른 무작위 e값을 잡고 다시 반복한다.

```
d = return_d(e, phi);
```

그리고나서 우리는 d값을 구해야한다. D는 앞에서 말했던 것처럼 e*d = 1 mod phi(n)를 만족해야 한다.

```
q = a3 / b3;
t1 = a1 - (q * b1);
t2 = a2 - (q * b2);
t3 = a3 - (q * b3);
a1 = b1;
a2 = b2;
a3 = b3;
b1 = t1;
b2 = t2;
b3 = t3;
if (b3 < 2)
{

    if(b1 > 0)
    {
        return b1;
    }

    else
    {

        return b1+phi;
    }
```

이건 return_d라는 함수에서 가져왔다. 이 함수는 방금 소개한 확장된 유클리드 호제법과 차이점이 맨 마지막 return부분밖에 없다. 수식 보면 결국 d값은 확장된 유클리드 호제법에서 b1에 해당하는 n보다 작은 정수라는 사실을 알 수 있다.

```
cout << "d = " << d << endl;
cout << "Message Input : ";
cin >> input; //message 입력
cout << "Message = " << input << endl;


cout << "**Encryption" << endl;
cipher = exp_mod(input ,e, n);  //암호화 시작
cout << "cipher = " << cipher << endl;
cipher = encryption(input, e, n);
cout << "**Decryption" << endl;
```

이제 우리는 keysetup을 끝냈다. 따라서 암호화 작업을 시작할 수 있다. 암호화는 M^e mod n (여기서 c는 cipher, M은 암호화하고자 하는 메시지고 0보다 같거나 크고 n보다 작아야한다.)를 통해 진행된다. 여기서 문제점은 지수곱이 매우커서 일반적인 방법으로는 하기가 힘들다.

```
int exp_mod(int a, int n, int z)
{
    int result = 1;
    long long x = 0;

    x = a % z;

    while (n > 0)
    {
        if ((n % 2) == 1)
        {

            result = (result * x) % z;

        }
        x = (x * x) % z;
        n = n / 2;

    }

    return result;
}
```

따라서 이 함수를 통해 mod연산을 할 수 있었다. 우선 해당 지수승에 해당하는 수를 bit로 표현한다. 그리고 해당 비트마다 mod 연산을 달리해주면된다. Square and Multiply Algorithm을 사용한다는 이야기다. 여기서 지수 승에 해당하는 n을 이진법으로 본다는 이야기다. 예를 들어 3^644 mod 645를 구한다고 하면 645는 이진법 표현으로 1010000100이 된다. 이는 배열에 0010000101이 되는데 여기서 루프를 돌면서 만약 2로 나누어 지지 않으면 (result * x) mod x를 해주고 x = (x*x)%z, n = n/2를 해주면 된다. 만약 2로 나누어지면 result 값에 대한 연산은 필요가 없다. 결론적으로 Mod 연산은 합동 연산이 가능해 해당 비트가 1일때는 정리한 식으로 표현하면 x = x*(M%n) %n이 된다. 0 일 때는 연산을 하지않고 (M%n) = (M%n)* (M%n)%n이되고 비트가 1일때만 계산을 하면 된다. 그리고 나서 최종적으로 나오는 x값을 return해주면 지수승 mod 연산이 된다.

마지막은 복호화이다. 여기서는 중국인의 나머지 정리를 활용하여 문제를 풀어야한다. 앞에서 설명했던 것처럼 복호화 과정은 M = c^d mod n 이 식을 통해 구현이 된다.

```
int real_china(int c, int d, int p, int q)
{
    int dp = exp_mod(d, 1, p - 1);
    int dq = exp_mod(d, 1, q- 1);
    int qinv = inverse(53, 61);
    int m1 = exp_mod(c, dp, p);
    int m2 = exp_mod(c, dq, q);
    int h = ((qinv * (m1 - m2))) % p;
    int m = m2 + (h * q);

    return m;

}
```

코드는 이렇게 작성하였다. 우선 복호화시 dp, dq, qinv는 항상 사용되는 값이므로 미리 계산하여 저장한다고 한다. 각 변수를 설명하면 아래 그림과 같다.

$$d_p = d \pmod{(p-1)}$$
$$d_q = d \pmod{(q-1)}$$
$$q_{Inv} = q^{-1} \pmod{p} =$$

여기서 q-1mod p 연산은 구현한 inverse함수를 통해 진행하였다.

```
int inverse(int a, int m) // a mod m 역원 구하기
{
    int x = 0;

    for (int i = 1; i < m; i++) {
        if ((a * i) % m == 1) {
            x = i;
        }


    }

    return x;
}
```

그 다음 암호문을 복호화 하려면 m1, m2, h변수가 필요하다. m1, m2는 아까 계산했던 dp, dq를 필요로 한다. 계산은 다음과 같다.

$$m_1 = c^{a_p} \pmod{p} =$$
$$m_2 = c^{d_q} \pmod{q} =$$

또한 h값을 구해야하는데 h는

$$h = (q_{Inv} \times (m_1 - m_2)) \pmod{p}$$

이렇게 정의된다. 이를 통해

$m = m_2 + h \times q$ 로 평문을 구할 수 있었다.

```
p = 30557
q = 30577
N = 934341389
phi = 934280256
e = 14213
d = 724916813
Message Input : 123
Message = 123
**Encryption
cipher = 844248350
**Decryption
decrypted cipher : 123
```

최종 결과입니다.

구현 환경 및 언어: 비주얼 스튜디오 2019, c++

주의: srand값을 무작위로 구현하기 위해 sleep 함수를 사용해서 초반 밀러 라빈 판별이 느립니다.