Cryptography Course
Programming Assignment
(Report)

Rickard Johansson 19930415-4512 (rickjoh@student.chalmers.se)
Vaios Taxiarchis 19910309-8514 (vaios@student.chalmers.se)

December 2016

1 A Math Library for Cryptography

1.1 Introduction

In this assignment we implemented a library that provides a number of mathematical functions commonly used in cryptography, such as Euler's Phi Function (Totient), the Extended Euclidean Algorithm and some Primality Test. Our library successfully passed the test suite provided with the skeleton code. Personal number for this task was 199103098514.

1.2 Implementation

In this assignment we are required to follow the skeleton code provided by the course site, to make sure that the test suite works. All functions work for arbitrary integers and we did not use already implemented libraries (such as BigInteger or GMP). We chose the C++ programming language to implement this programming assignment. The cryptolib_cpp folder contains the skeleton code and the test suite.

1.2.1 Extended Euclidean Algorithm

The function **void EEA(int a, int b, int result[])** assigns to the array **result[]** the values **result[0] = gcd**, **result[1] = s** and **result[2] = t** such that **gcd** is the greatest common divisor of **a** and **b**. The formula for EEA algorithm:

$$gcd = a * s + b * t$$

We calculate all the temporary values inside the function using local variables and finally we assign the three semantic values to the array.

1.2.2 Euler's Phi Function (Totient)

The function **int EulerPhi(int n)** is very simple and returns Euler's Totient for value "n". Euler's Phi function counts the number of positive integers less than or equal to n which are relatively prime to n, i.e. do not have any common divisor with n except 1. The formula for Euler's function:

$$\phi(n) = n \prod_{p \text{ prime}} \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

1.2.3 Modular Inverse

The function **int ModInv(int n, int m)** returns the modular multiplicative inverse, the value 'v', such that:

$$n * v = 1 \ mod(m)$$

If the modular inverse does not exist, the function returns 0. In order to calculate **n mod(m)** we implemented an extra function **int modulo(int n, int p)** and we call it whenever is neccessary.

1.2.4 Fermat Primality Test

The function **int FermatPT(int n)** is the Fermat primality test and it is based on Fermat's little theorem. According to Fermat's little theorem, if n is a prime number and if a is an integer that is relatively prime to n, then the following congruence relationship holds:

$$a^{n-1} = 1 \ mod(n)$$

The function returns 0, if the number is a Fermat Prime, otherwise it returns the lowest Fermat Witness. We did not check random values but all the values starting from 2 (inclusive) to "n/3" (exclusive). We also implemented an extra function **ll modulo_extended(ll base, ll exponent, ll mod)** to calculate the modular exponentiation for very large integers using the 'namespace std' and defining a new type for the large integers with name 'll'.

1.2.5 Hash Collision Probability

The function **double HashCP(double n_samples, double size)** returns the probability that calling a perfect hash function with 'n_samples' (uniformly distributed) will give one collision (i.e. that two samples result in the same hash), where 'size' is the number of different output values that the hash function can produce. In order to implement this functionality our solution is based on the birthday paradox (or birthday problem) such that (where n = n_samples):

$$P(n, size) = \frac{size}{size} \cdot \frac{size-1}{size} \cdot ... \cdot \frac{size-n}{size} = 1 \cdot \frac{size-1}{size} \cdot ... \cdot \frac{size-n}{size}$$

Finally, P = 1 - P(n,size) where P is the probability that calling a perfect hash function gives one collision and P(n,size) is the probability described before for birthday paradox.

2 Special Soundness of Fiat-Shamir sigma-protocol

2.1 Introduction

We eavesdropped on a number of Fiat-Shamir protocol runs and we found that the same nonce was used twice. Due to the special soundness property we should now be able to retrieve the secret key used in the protocol. The fiat_shamir_java folder contains the skeleton code and the test suite.

2.2 Background Theory

n = modulus used, X = public key, x = private key, r = random integer chosen by prover, R = r^2 random value sent by prover, c = challenge sent by verifier, s = rx^c = proof sent by prover.

1 : Prover chooses random r and sends $R = r^2 \ mod(n)$ to the verifier.

2 : Verifier chooses challenge c (= 0 *or* 1) and sends it to the prover.

3 : Prover computes $s = rx^c \ mod(n)$ and sends it to the verifier.

4 : Verifier checks that $s^2 = RX \ mod(n)$.

If the prover uses the same r twice with different c values it is possible for an adversary that eavesdrops to calculate s. Lets call the random number used twice r_0 and the two proofs s_1 and s_2 where s_1 had the challenge c = 0 and s_2 had the challenge c = 1. Then

$$s_1 = r_0 x^0 = r_0 \ mod(n)$$
$$s_2 = r_0 x^1 = r_0 x \ mod(n)$$

the goal is to find the secret key x from this. That is done by solving

$$x = r_0^{-1} s_2 = s_1^{-1} s_2 \ mod(n)$$

All that is needed is to find the inverse of s_1 and multiply with s_2 and x is found.

2.3 Implementation

The implementation follows the above procedure by first checking if any R (nonce) have been used twice by comparing each run with every other run. If the same nonce have been used twice it finds s_1 and s_2, calculates $x = s_1^{-1} s_2 \ mod(n)$ and returns x. If no R have been used twice the function returns zero.

2.4 Execution

Personal number for this task was 199304154512.

Recovered message:
1202203607812597209493110499710324507964215487010040564744487575299733559330713082857559695886952605977533361252079918918363946260879327779957959458954184924128240383294510458915882768432198464920156307842545778336053685692551399170115398710096664176413397103918707721083909960487083967073408154642187722265215890024780932354827418661647219

Decoded text:
Think left and think right and think low and think high. Oh, the thinks you can think up if only you try! ——————————- Dr. Seuss