Cryptography Course 2020 (TDA352)

Programming Assignement Report

Félix Defrance - Alexis Laude

Prelude

The personnummer used to generate our assignment is : **199809077237**.

All the programs are written in Java, and we provide Makefile(s) in order to run the code. Read all the README(s) for each exercices to get additionnal information on how to use the Makefile, but it should be pretty straightforward.

0 A Math Library for Cryptography

0.1 Extended Euclidean Algorithm

The implementation follows `ALGORITHM B.10` from the **INTRODUCTION TO MODERN CRYPTOGRAPHY** textbook.

0.2 Euler's Phi Function (Totient)

Straightforward implementation from the Wikipedia definition : *Euler's totient function counts the positive integers up to a given integer n that are relatively prime to n.*

0.3 Modular Inverse

Our function has to *"return the value "v" such that "n*v = 1 (mod m)""*, given n and m.

First, if n is negative, we add the modulo m until it gets positive.

We then run the Extended Euclidean Algorithm defined above with n and m.

If the modular inverse exists ($gcd(n, m) = 1$), we check if v is positive, and we return v if it is and $v + m$ if it is negative so that it gets positive.

If there are no modular inverse, we return 0.

0.4 Fermat Primality test

We first defined a helper function `ModularExponantiation` that takes a, b and m and returns $a^b \pmod m$.

Then, to check if a given n is a Fermat Prime, we iterate over i from 2 to $\frac{n}{3}$, and check at each iteration if $i^{n-1} \not\equiv 1 (\text{mod } m)$.

If i verify this property, we return i, which is the lowest Fermat Witness.

If no i is a Fermat Witness, we return 0.

0.5 Hash Collision Probability

The probability P that calling a perfect hash function with "n_samples" (uniformly distributed) will give one collision, where "size" is the number of different output values the hash function can produce, is :

$$\begin{aligned} P &= 1 - \frac{size!}{size^{n\_samples}(size - n\_samples)!} \\ &= 1 - \frac{1}{size^{n\_samples}} \cdot \frac{size!}{(size - n\_samples)!} \end{aligned}$$

$\frac{size!}{(size-n\_samples)!}$ is the number of permutation of $n\_samples$ of $size$. We can write permutations this way :

$$Perm(n,k) = \frac{n!}{(n-k)!} = \underbrace{n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)}_{k \text{ factors}}$$

If we rewrite P :

$$\begin{aligned} P &= 1 - \frac{1}{size^{n\_samples}} \cdot \frac{size!}{(size - n\_samples)!} \\ &= 1 - \frac{1}{\underbrace{size \cdot size \cdots size}_{n\_samples \text{ factors}}} \cdot \underbrace{size \cdot (size-1) \cdot (size-2) \cdots (size - n\_samples + 1)}_{n\_samples \text{ factors}} \\ &= 1 - \prod_{i=size-n\_samples+1}^{size} \frac{i}{size} \end{aligned}$$

That's the formula we implemented with a for loop.

1 Special Soundness of Fiat-Shamir sigma-protocol

We eavesdropped on a number of Fiat-Shamir protocol runs and we found that the same nonce was used twice! The goal is to retrieve the secret key used in the protocol.

The same nonce r has been used twice for different challenge values : $c = 0$ and $c = 1$.

We know that during the Fiat-Shamir protocol, when the Verifier sends a challenge $c \in \{0, 1\}$, the Prover needs to send her back the value $s = rx^c \pmod n$, where x is the secret key.

For the first challenge $c = 0$, he Prover will sends back the value $s = rx^0 = r$.

Thus, we know the value of r, it is s.

For the second challenge $c = 1$, he Prover will sends back the value $s = rx^1 = rx$.

We know the value of s and r, we can then find the value $x = sr^{-1} \pmod n$.

In order to solve the problem, we can iterate two-by-two over all runs, and compute for each iteration the value $s_1 s_2^{-1} \pmod n$.

At each run, we can check wether $X = x^2 \pmod n$ stands, and if it does, we return $x \pmod n$.

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

2 Decrypting CBC with simple XOR

We have intercepted a message that was encrypted using cypher-block chaining, and we also know the plain-text value of the first block.

We know that Cipher Block Chaining ciphers this way :

$$C_i = K \oplus (M_i \oplus C_{i-1}) \ ; \ \text{where } C_0 = IV$$

We are given M_1, and the whole encrypted message C.

The first goal is to find the key K.

$$C_i = K \oplus (M_i \oplus C_{i-1}) \iff K = M_i \oplus C_{i-1} \oplus C_i$$

Applied to our case, we get :

$$K = M_1 \oplus IV \oplus C_1$$

Once we found K, we only have to iterate over all encrypted block, and decipher it with the key K, since we know :

$$C_i = K \oplus (M_i \oplus C_{i-1}) \iff M_i = C_i \oplus K \oplus C_{i-1}$$

We can then reconstruct the complete plain-text message.

Recovered message: 199809077237
Do or do not. There is not try. - Master Yoda000

3 Attacking RSA

The same message has been encrypted using RSA to three different recipients. All recipients have the same public key ($e = 3$) but different modulus (N_1, N_2, N_3). We have intercepted the three ciphertexts. The goal is to find the initial message.

Knowing how the RSA protocol works, we have the following encryptions :

$$\begin{cases} c_1 \equiv m^e (\text{mod } N_1) \\ c_2 \equiv m^e (\text{mod } N_2) \\ c_3 \equiv m^e (\text{mod } N_3) \end{cases}$$

With the Chinese Remainder Theoremn we can find c such that the solutions to the system above are all number congruent to c modulo $N_1 N_2 N_3$.

Having found c, we know that :

$$c \equiv m^e (\text{mod } N_1 N_2 N_3) \Longleftrightarrow m \equiv \sqrt[3]{c \ (\text{mod } N_1 N_2 N_3)}$$

The last step is then to compute m with the helper function provided in the `CubeRoot.java` file.

Decoded text: Taher ElGamal

4 Attacking ElGamal

In this exercice, we are given :

- **p** : the group size
- **g** : the generator of the group
- **y** : the public key of the receiver
- **time** : the time at which the message was encrypted (and the weak technique used to choose the "randomness")
- **(c1, c2)** : the ElGamal encryption of the message

The first goal is to find the random number r used for the encryption.

We know that $c_1 = g^r$, and we are given g, c_1 and the technique to find r.

The pseudo code for the 'pseudo random number generator' is the following :

```
integer createRandomNumber:
    return YEAR*(10^10)+month*(10^8)+days*(10^6)+hours*(10^4)+minute*(10^2)+sec+millisecs;
```

We only miss the *millisecs* value.

We can then loop over all the possible value of *milliseconds* and check at each iteration if $c_1 = g^r$.

When the equality above is true, it means that we have found the right r.

Now the last goal is to find the message m.

From ElGamal encryption protocol, we know that $c_2 = my^r$.

We are given c_2, y and we just found r.

$$c_2 \equiv my^r \pmod{p} \iff m \equiv c_2(y^{-1})^r \pmod{p}$$

We can get the value of m by solving the above equation.

After decryption, we get the following message :

Decoded text: Crytanalysis doesn't break cryptosystems. Bruce Schneier breaks cryptosystems.