Assignment 1

Assigned: 21/10/2022
Due: 31/10/2022

In this assignment you are going to implement from scratch two cryptographic methods: (i) **Diffie-Hellman Key Exchange** and (ii) **RSA (Rivest–Shamir–Adleman)**. Both implementations will be in the **C programming language**. The purpose of this assignment is to provide you the opportunity to get familiar with the internals and implementations of two popular encryption schemes, namely RSA and DH Key Exchange. You will use the **GMP C library** to implement the RSA algorithm.

Basic Theory

GNU Multiple Precision Arithmetic Library (GMP): GMP is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. It aims to provide the fastest possible arithmetic for all applications that need higher precision than is directly supported by the basic C types. Many applications use just a few hundred bits of precision, but some applications may need thousands or even millions of bits. GMP is designed to give good performance for both, by choosing algorithms based on the sizes of the operands, and by carefully keeping the overhead at a minimum. The speed of GMP is achieved by using full words as the basic arithmetic type, by using sophisticated algorithms, by including carefully optimized assembly code for the most common inner loops for many different CPUs, and by a general emphasis on speed (as opposed to simplicity or elegance).

Diffie–Hellman Key Exchange: Diffie-Hellman key exchange is a method of securely exchanging cryptographic keys over a public channel and was one of the first public-key protocols conceived by Ralph Merkle and named after Whitfield Diffie and Martin Hellman. DH is one of the earliest practical examples of public key exchange implemented within the field of cryptography.

RSA Algorithm: The RSA Algorithm involves two keys, i.e. a public key and a private key. One key can be used for encrypting a message which can only be decrypted by the other key. As an example let’s say we have two peers communicating with each other in a channel secured by the RSA algorithm. The sender will encrypt the plain text with the

recipient's public key. Then the receiver is the only one who can decrypt the message using their private key. The public key will be available in a public key repository. n the recipient's side.

Diffie-Hellman Key Exchange Details

The Diffie-Hellman algorithm is primarily used to exchange cryptography keys for use in symmetric encryption algorithms. You have to implement a simple tool in C, that will apply the scenario mentioned below:
Let's assume that Alice wants to establish a shared secret with Bob. The protocol for this requires the following actions:

- Alice and Bob agree to use a prime number *p* and base *g*. (These two values are chosen in this way to ensure that the resulting shared secret can take on any value from 1 to *p*–1)
- Alice chooses a secret integer *a*<*p*, then sends Bob: $A = g^a \bmod p$.
- Bob chooses a secret integer *b*<*p*, then sends Alice: $B = g^b \bmod p$.
- Alice computes $s = B^a \bmod p$.
- Bob computes $s = A^b \bmod p$.
- Alice and Bob now share a secret.

RSA Algorithm Details

In this task, you have to implement an RSA key-pair generation algorithm. In order to do so, you will first need to study RSA's internals:
https://en.wikipedia.org/wiki/RSA_(cryptosystem).

Key generation:

1. Give from the command line 2 numbers. Let's name them **p** and **q**.
2. Calculate if these 2 numbers are primes or not.
3. If they are, compute **n** where **n = p * q**.
4. Calculate **lambda(n)** where **lambda(n) = (p - 1) * (q - 1)**. This is Euler's totient function, described in the original RSA paper "*A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*". You may find out that other implementations use different totient functions. However, for this implementation, we are going to use Euler's.
5. Choose a prime **e** where **(e % lambda(n) != 0) AND (gcd(e, lambda) == 1)** where gcd() is the Greatest Common Denominator.

6. Choose **d** where **d** is the **modular inverse of (e, lambda)**.
7. The public key consists of **n** and **d**, in this order.
8. The private key consists of **n** and **e**, in this order.

Data Encryption: Develop a function that provides RSA encryption functionality, using the keys generated in the previous step. This function reads the data from a file and encrypts them using one of the generated keys. Then, it stores the ciphertext in an output file. For each character (1 byte) of the plaintext, the tool generates an 8-byte ciphertext (size_t on 64-bit machines). For example, if the plaintext is “hello” then the 5 bytes (5 chars) of the plaintext will produce 40 bytes (5 * sizeof(size_t)) of ciphertext.

Data Decryption: Implement a function that reads a ciphertext from an input file and performs RSA decryption using the appropriate one of the two keys, depending on which one was used for the ciphertext encryption. The keys will be generated using the KDF described in Task A. When the decryption is over, the function stores the plaintext in an appropriate output file.

In order to successfully decrypt the data, you have to use the appropriate key. If the ciphertext is encrypted using the public key, you have to use the private key in order to decrypt the data and vice versa. Also, every 8-bytes of the ciphertext produces a 1-byte plaintext. For example, a 40-byte ciphertext will produce a 5-byte plaintext.

NOTE: For the Key generation Algorithm, Encryption, and Decryption you **must** use the GMPLibrary (https://gmplib.org/). See “Useful Links” for more details.

Tools Specifications

In order to assist you in the development of the two tools, we provide a basic skeleton of both of them.

Diffie-Hellman Key Exchange Tool

The tool will receive the required arguments from the command line upon execution as such:

Options:

-o	*path*	Path to output file
-p	*number*	Prime number
-g	*number*	Primitive Root for previous prime number
-a	*number*	Private key A

-b *number* Private key B
-h This help message

The argument *-p* will include the will be the public prime number.
The argument *-g* will be the public primitive root of the previous prime number.
The argument *-a* will be the private key of user A.
The argument *-b* will be the private key of user B.

The command line tool will return the public key of user A, the public key of user B, and the shared secret. The output file **must be** in the following format:
<public key A>, <public key B>, <shared secret>

The compiled name of the command line tool **must be** dh_assign_1.

Example:
./dh_assign_1 -o output.txt -p 23 -g 9 -a 15 -b 2

RSA Tool

The tool will receive the required arguments from the command line upon execution as such:

Options:
-i path Path to the input file
-o path Path to the output file
-k path Path to the key file
-g Perform RSA key-pair generation
-d Decrypt input and store results to output
-e Encrypt input and store results to output
-h This help message

The arguments “i”, “o” and “k” are always required when using “e” or “d”
Using -i and a path the user specifies the path to the input file.
Using -o and a path the user specifies the path to the output file.
Using -k and a path the user specifies the path to the key file.
Using -g the tool generates a public and a private key and stores them to the public.key and private.key files respectively.
Using -d the user specifies that the tool should read the ciphertext from the input file, decrypt it and then store the plaintext in the output file.

Using -e the user specifies that the tool should read the plaintext from the input file, encrypt it and store the ciphertext in the output file.

Example:
./rsa_assign_1 -g

The tool will generate a public and a private key and store them in the files public.key and private.key respectively.

Example:
./rsa_assign_1 -i plaintext.txt -o ciphertext.txt -k public.key -e

The tool will retrieve the public key from the file public.key and use it to encrypt the data found in “plaintext.txt” and then store the ciphertext in “ciphertext.txt”

The name of the command line tool **must be** rsa_assign_1.

Useful Links

- GMP Library: https://gmplib.org/
- GMP Library functions that you can find useful for RSA Algorithm implementation: (1) Integer Comparison, (2) Integer Exponation, (3) Number Theoretic Functions, (4) Integer Import & Export
- RSA Algorithm Original Paper: https://people.csail.mit.edu/rivest/Rsapaper.pdf

Important notes

1. You need to submit all the source codes of your tools, including a README file and a Makefile. The README file should briefly describe your tool. You should place all these files in a folder named <AM>_assign1 and then compress it as a .zip file. For example, if your login is 2022123456 the folder should be named 2022123456_assign1 you should commit 2022123456_assign1.zip.
2. **Google** your questions first.
3. Use the tab “Συζήτηση” in courses for questions.
4. Do not copy-paste code from online examples, we will know ;)