Assignment 1

Assigned: 14/10/2024
Due: 24/10/2024

In this assignment, you are going to implement from scratch two cryptographic methods: (i) **Elliptic Curve Diffie–Hellman (ECDH)** 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 with the opportunity to get familiar with the internals and implementations of two popular encryption schemes, namely RSA and ECDH Key Exchange. You will use the **GMP C library** to implement the RSA algorithm and **libsodium** for the ECDH 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).

Libsodium

Libsodium is a modern, easy-to-use cryptographic library that implements a variety of encryption, decryption, and key exchange algorithms. One of its key features is built-in support for **Elliptic Curve Diffie-Hellman (ECDH)** using Curve25519, which provides strong security with smaller key sizes and better performance compared to traditional Diffie-Hellman.

Elliptic-Curve Diffie-Hellman: Elliptic Curve Diffie-Hellman (ECDH) is a cryptographic protocol that allows two parties to securely establish a shared secret over an insecure communication channel. It is an adaptation of the classical Diffie-Hellman key exchange, but it leverages the mathematics of elliptic curves to

provide the same level of security with significantly smaller key sizes. This efficiency makes ECDH particularly suited for modern applications such as mobile devices, IoT systems, and blockchain technologies, where both security and performance are critical.

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. on the recipient's side.

Elliptic Curve Diffie-Hellman (ECDH) Key Exchange

In this task, you will implement an Elliptic Curve Diffie-Hellman (ECDH) key exchange using the **libsodium** library. Follow the steps below to create a tool that enables secure key exchange between two parties (Alice and Bob):

Scenario:

- **Agreement on Elliptic Curve Parameters**: Alice and Bob agree to use the Curve25519 elliptic curve for the ECDH key exchange. This curve is built into libsodium, and no additional curve parameters need to be specified.
- **Key Generation by Alice**:
    - Alice generates a private key (a large random integer).
    - Alice computes her public key `A = a * G` (elliptic curve point multiplication using the base point `G`) and sends it to Bob.
- **Key Generation by Bob**:
    - Bob generates a private key (a large random integer).
    - Bob computes his public key `B = b * G` and sends it to Alice.
- **Shared Secret Calculation**:
    - After receiving Bob's public key `B`, Alice computes the shared secret `S_A = a * B` (where `B` is Bob's public key).
    - After receiving Alice's public key `A`, Bob computes the shared secret `S_B = b * A`.
    - Both shared secrets are the same: `S_A = S_B = (a * b) * G`.

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 2 numbers. Let's name them **p** and **q** and a **key_length** (e.g. 1024, 2048, 4096).
2. Calculate if these 2 numbers are primes or not (**Note**: p and q should each be key_length/2).
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.

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 a function called *generateRSAKeyPair,* following the steps in the **Key generation** algorithm above. 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.

Performance Analysis: Compare the performance of RSA encryption and decryption with different key lengths in terms of **computational time** and **also memory usage**.

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.

ECDH Key Exchange Tool

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

Command Line Options for ECDH Tool:

```
-o  path        Path to output file
-a  number  Alice's private key (optional)
-b  number  Bob's private key (optional)
-h       This help message
```

If the private keys are not provided, the tool should randomly generate them using **libsodium**.

The output file must contain Alice's public key, Bob's public key, and the shared secret. The keys and shared secret will be expressed as hexadecimal values in the output file.

Example:

./ecdh_assign_1 -o ecdh.txt -a 5 -b 3

This would generate public keys for Alice and Bob using the **Curve25519** elliptic curve, compute the shared secret, and write the public keys and shared secret to the `ecdh.txt` file.

Example of the ecdh.txt:

Alice's Public Key:
9ffb17b364f12b40f335e802fe02983f295b679ce291785181f122764ea80370
Bob's Public Key:
cb4a27e877b8d5572fa4b9e92a8d8b5f892ac87f58ed053f116b0dc20fe80278
Shared Secret (Alice):
afc0b79e89270b54ac24e161434b7b99eedeeda2ee7907548b2502adf63ed40c
Shared Secret (Bob):
afc0b79e89270b54ac24e161434b7b99eedeeda2ee7907548b2502adf63ed40c
Shared secrets match!

Clarifications:

Private Keys: If private keys are provided via the `-a` or `-b` options, they will be used directly for Alice and Bob. If not provided, the tool will generate random private keys for each.

RSA Different key length 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	length	Perform RSA key-pair generation given a key length “length”
-d		Decrypt input and store results to output.
-e		Encrypt input and store results to output.
-a		Compare the performance of RSA encryption and decryption with three different key lengths (1024, 2048, 4096 key lengths) in terms of computational time.
-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 given a key length “length” and stores them to the public_length.key and private_length.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.

Using -a the user generates three distinct sets of public and private key pairs, allowing for a comparison of the encryption and decryption times for each.

Example:
./rsa_assign_1 -g “length”

The tool will generate a public and a private key given a length “length” and store them in the files public_length.key and private_length.key respectively.

Example:
./rsa_assign_1 -i plaintext.txt -o ciphertext.txt -k public_length.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”.

Example:
Step 1: ./rsa_assign_1 -a performance.txt

The tool will generate three distinct sets of public and private key pairs, each with different key lengths (1024, 2048, 4096). These key pairs will be saved in files named "public_1024.key," "private_1024.key," "public_2048.key," "private_2048.key," "public_4096.key," and "private_4096.key." Afterward, it will encrypt and decrypt the contents of the "plaintext.txt" file using each key pair and record the **time taken** (see the **<sys/time.h>** library) and the **memory usage** (see the **<sys/resource.h>** library) during each process in the "performance.txt" file. This setup allows for a direct comparison of encryption and decryption times, as well as memory consumption, for each key length.

An example of the output of the performance.txt file is:

Key Length: 1024 bits
Encryption Time: 0.02s
Decryption Time: 0.03s
Peak Memory Usage (Encryption): 12 Bytes
Peak Memory Usage (Decryption): 10 Bytes

Key Length: 2048 bits
Encryption Time: 0.08s
Decryption Time: 0.09s
Peak Memory Usage (Encryption): 25 Bytes
Peak Memory Usage (Decryption): 23 Bytes

Key Length: 4096 bits
Encryption Time: 0.30s

Decryption Time: 0.32s
Peak Memory Usage (Encryption): 50 Bytes
Peak Memory Usage (Decryption): 47 Bytes

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

Useful Links

- **Libsodium Library**: https://libsodium.gitbook.io/doc/
  This documentation covers all functionalities of **libsodium**, including elliptic curve Diffie-Hellman (ECDH) using **Curve25519**.
- **Libsodium API Documentation**: https://doc.libsodium.org/
  Provides a detailed description of the APIs available in libsodium, such as key generation, key exchange (ECDH), encryption, and other cryptographic primitives.
- **Libsodium Installation Guide**: https://libsodium.gitbook.io/doc/installation
  Instructions for installing **libsodium** on different operating systems, including Windows, Linux, and macOS.
- **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. Do not copy-paste code from online examples, we will know ;)