Assignment 2

Assigned: 31/10/2025
Due: 09/11/2025

In this assignment, you are going to implement from scratch two cryptographic methods: (i) **Elliptic Curve Diffie–Hellman (ECDH) Key Exchange with Key Derivation** and (ii) **RSA (Rivest–Shamir–Adleman) with Digital Signatures**. 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, as well as practical applications such as digital signatures and key derivation functions.

You will use the **GMP C library** to implement the RSA algorithm and **libsodium** for the ECDH algorithm and key derivation.

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)

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.

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. RSA can also be used for digital signatures, where the private key signs a message and the public key verifies the signature.

Task 1: Elliptic Curve Diffie-Hellman (ECDH) Key Exchange with Key Derivation

In this task, you will implement an Elliptic Curve Diffie-Hellman (ECDH) key exchange using the **libsodium** library. **Additionally**, you will derive multiple keys from the shared secret using a Key Derivation Function (KDF).

Scenario:

1. **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.
2. **Key Generation by Alice**:
    - Alice generates a private key (a large random integer).
    - Alice computes her public key A = a * G and sends it to Bob.
3. **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.
4. **Shared Secret Calculation**:

    - After receiving Bob's public key B, Alice computes the shared secret S_A = a * B.
    - 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.
5. **Key Derivation** :
    - Using the shared secret, both Alice and Bob derive **two separate keys**:
        - **Encryption Key** (32 bytes) - to be used for symmetric encryption
        - **MAC Key** (32 bytes) - to be used for message authentication
    - Use libsodium's `crypto_kdf_derive_from_key()` function for key derivation.
    - Both parties should derive the same keys from the shared secret.

Tool Specifications

Command Line Options for ECDH Tool:

-o path Path to output file

-a number Alice's private key (optional, hexadecimal format)

-b number Bob's private key (optional, hexadecimal format)

-c context Context string for key derivation (default: "ECDH_KDF")

-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
    - Shared secret (both parties)
    - Derived Encryption Key (both parties)
    - Derived MAC Key (both parties)
- All keys and secrets will be expressed as **hexadecimal values** in the output file.

Example Usage:

1. Random keys: ./ecdh_assign_2 -o output.txt
2. With fixed keys: ./ecdh_assign_2 -o output.txt -a 0x1a2b3c -b 0x1a2cb7
3. With custom context for KDF: ./ecdh_assign_2 -o output.txt -a 0x1a2b3c -b 0x1a2cb7 -c "koukou25"

4. This would generate public keys for Alice and Bob using the Curve25519 elliptic curve, compute the shared secret, derive two separate keys (encryption and MAC), and write all results to the `ecdh.txt` file.

Example Output (`ecdh.txt`):

Alice's Public Key:

9ffb17b364f12b40f335e802fe02983f295b679ce291785181f122764ea80370

Bob's Public Key:

cb4a27e877b8d5572fa4b9e92a8d8b5f892ac87f58ed053f116b0dc20fe80278

Shared Secret (Alice):

afc0b79e89270b54ac24e161434b7b99eedeeda2ee7907548b2502adf63ed40c

Shared Secret (Bob):

afc0b79e89270b54ac24e161434b7b99eedeeda2ee7907548b2502adf63ed40c

Shared secrets match!

Derived Encryption Key (Alice):

3b8f9c2e5a7d1f4e6b2c8a9e4d1f7c3a5b8e2f9c6d3a7e1f4b8c2e9a5d1f7c3a

Derived Encryption Key (Bob):

3b8f9c2e5a7d1f4e6b2c8a9e4d1f7c3a5b8e2f9c6d3a7e1f4b8c2e9a5d1f7c3a

Encryption keys match!

Derived MAC Key (Alice):

7c3a5b8e2f9c6d3a7e1f4b8c2e9a5d1f3b8f9c2e5a7d1f4e6b2c8a9e4d1f7c3a

Derived MAC Key (Bob):

7c3a5b8e2f9c6d3a7e1f4b8c2e9a5d1f3b8f9c2e5a7d1f4e6b2c8a9e4d1f7c3a

MAC keys match!

Task 2: RSA Algorithm with Digital Signatures

In this task, you have to implement an RSA key-pair generation algorithm, encryption/decryption functions, and **digital signature creation and verification** .

Key Generation:

Follow the same steps as the original assignment:

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)**.
5. Choose a prime **e** where **(e % lambda(n) != 0) AND (gcd(e, lambda) == 1)**.
6. Choose **d** where **d is the modular inverse of (e, lambda)**.
7. The **public key** consists of **n** and **e**, in this order.
8. The **private key** consists of **n** and **d**, 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. When the decryption is over, the function stores the plaintext in an appropriate output file.

Digital Signatures:

1. **Sign Operation**:
    - Read plaintext from an input file
    - Compute the SHA-256 hash of the plaintext
    - Sign the hash using the **private key** (this is RSA signing: signature = hash^d mod n)
    - Store the signature in an output file
2. **Verify Operation**:
    - Read plaintext and signature from input files
    - Compute the SHA-256 hash of the plaintext
    - Verify the signature using the **public key** (compute: hash' = signature^e mod n)
    - Compare hash' with the original hash

- Output whether the signature is VALID or INVALID

Performance Analysis:

Compare the performance of RSA encryption, decryption, **signing**, and **verification** with different key lengths (1024, 2048, 4096) in terms of **computational time** and **memory usage**.

Tool Specifications

Command Line Options for RSA Tool:

-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

-s Sign input file and store signature to output

-v path Verify signature (path to signature file) against input file

-a Performance analysis with three key lengths (1024, 2048, 4096)

-h This help message

Notes:

- The arguments "i", "o", and "k" are always required when using "e", "d", or "s".
- When using "-v", you need "-i" (plaintext), "-k" (public key), and the path after "-v" (signature file).

Example Usage:

Key Generation:

./rsa_assign_2 -g 2048

Generates `public_2048.key` and `private_2048.key`.

Encryption:

./rsa_assign_2 -i plaintext.txt -o ciphertext.txt -k public_2048.key -e

Decryption:

./rsa_assign_2 -i ciphertext.txt -o decrypted.txt -k private_2048.key -d

Signing :

./rsa_assign_2 -i input.txt -o signature_output -k private_2048.key -s

Verification :

./rsa_assign_2 -i input.txt -k public_2048.key -v signature_output

Output: `Signature is VALID` or `Signature is INVALID`

Performance Analysis:

./rsa_assign_2 -a performance.txt

The tool will generate three sets of key pairs (1024, 2048, 4096 bits) and measure:

- Encryption time
- Decryption time
- Signing time
- Verification time
- Peak memory usage for each operation

Example Output (`performance.txt`):

Key Length: 1024 bits

Encryption Time: 0.02s

Decryption Time: 0.03s

Signing Time: 0.03s

Verification Time: 0.02s

Peak Memory Usage (Encryption): 12 KB

Peak Memory Usage (Decryption): 10 KB

Peak Memory Usage (Signing): 11 KB

Peak Memory Usage (Verification): 9 KB

Key Length: 2048 bits

Encryption Time: 0.08s

Decryption Time: 0.09s

Signing Time: 0.09s

Verification Time: 0.08s

Peak Memory Usage (Encryption): 25 KB

Peak Memory Usage (Decryption): 23 KB

Peak Memory Usage (Signing): 24 KB

Peak Memory Usage (Verification): 22 KB

Key Length: 4096 bits

Encryption Time: 0.30s

Decryption Time: 0.32s

Signing Time: 0.31s

Verification Time: 0.29s

Peak Memory Usage (Encryption): 50 KB

Peak Memory Usage (Decryption): 47 KB

Peak Memory Usage (Signing): 48 KB

Peak Memory Usage (Verification): 45 KB

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>_assign2 and then compress it as a .zip file. For example, if your login is 2022123456 the folder should be named 2022123456_assign2 you should commit 2022123456_assign2.zip.
2. **Google** your questions first.
3. Do not copy-paste code from online examples or AI tools, we will know ;)