Lab 4 Report: Programming Symmetric & Asymmetric Crypto

Overview

The Cryptography.ipynb file provides implementations for various cryptographic operations as outlined below:

1. AES Encryption and Decryption
2. RSA Encryption and Decryption
3. RSA Digital Signature
4. SHA-256 Hashing
5. Execution Time Evaluation

Implementation Details

- ☐ Language: Python 3
- ☐ Libraries:
  - ☐ Cryptography for AES, RSA, and hashing
  - ☐ matplotlib for visualization
- ☐ Key Storage:
  - ☐ AES keys stored as Base64 files
  - ☐ RSA keys stored in PEM format

Task 1: AES Encryption & Decryption

In this task, AES encryption and decryption were implemented using both 128-bit and 256-bit keys in two different modes — **ECB** and **CFB**.

Key Features:

- ➔ Encryption keys were securely generated using `os.urandom()` to ensure randomness.
- ➔ Each key was saved as a Base64-encoded file (e.g., `aes_128_ECB.key`).
- ➔ For CFB mode, a random 16-byte Initialization Vector (IV**)** was generated and included at the start of the encrypted output.
- ➔ For ECB mode, proper padding was applied so that the plaintext size aligned with the 16-byte block requirement of AES.

Testing:

- ➔ The encryption and decryption processes were tested using a `sample.txt` file containing 27 bytes of data.
- ➔ The program successfully produced encrypted binary outputs, and decryption accurately recovered the original plaintext.
- ➔ Both AES-128 and AES-256 implementations in ECB and CFB modes functioned correctly, demonstrating successful encryption and decryption.

Task 2: RSA Encryption & Decryption

In this task, RSA encryption and decryption were implemented using **OAEP padding** with **SHA-256** for secure message handling.

Key Features:

- ➔ A 2048-bit RSA key pair was generated by default.
- ➔ The private key was saved as `rsa_private.pem`, and the public key as `rsa_public.pem`.
- ➔ Due to RSA's inherent key size limitations, the maximum amount of data that can be directly encrypted is restricted.

Testing:

- ➔ The process was tested using a `rsa_plain.txt` file containing 27 bytes of text.
- ➔ The encrypted output, `rsa_encrypted.bin`, was 256 bytes in size — matching the RSA key length.
- ➔ Decryption successfully recovered the original plaintext, confirming the accuracy and reliability of the RSA implementation.

Task 3: RSA Digital Signature

This task focused on implementing **digital signatures** to ensure data authenticity and integrity, using **PSS padding** with **SHA-256**.

Key Features:

- ➔ The private key was used to create (sign) the message.
- ➔ The public key was used to verify the signature.
- ➔ The generated signature was stored as a binary file (e.g., `sign.sig`).

Testing:

- ➔ The signature file `sign.sig` was 256 bytes in size.
- ➔ Verification confirmed that the signature was valid, proving that the message had not been altered and was signed by the correct private key.

Task 4: SHA-256 Hashing

In this task, **SHA-256** was used to generate message digests for checking file integrity.

Key Features:

- ➔ The program read an input file, calculated its SHA-256 hash, and displayed the result in hexadecimal format.

➔ The process is deterministic, meaning identical files always produce the same hash value.

Testing:

➔ The input file `sample.txt` (27 bytes) produced a 64-character SHA-256 hash.

➔ The results confirmed that the hashing function worked correctly and consistently.

Task 5: Execution Time Measurement

This task measured how long various cryptographic operations took to execute, depending on key size and algorithm type.

Key Features:

➔ AES was tested with 128-bit and 256-bit keys in both ECB and CFB modes.

➔ RSA was tested with 1024, 2048, 3072, and 4096-bit keys.

➔ The results were visualized using matplotlib, producing a timing graph (`timing_results.png`).

Graph & Performance Analysis:

➔ **AES:** Showed almost no performance difference between key sizes or modes — both encryption and decryption were very fast.

➔ **RSA:** Encryption time increased slightly with key size, but decryption time grew much faster, especially for larger keys.

Implications:

➔ AES is ideal for encrypting large amounts of data due to its high speed.

➔ RSA is better suited for small pieces of data, such as encrypting AES keys.

➔ A hybrid encryption model works best: use RSA to encrypt the AES key, and AES for the main data.

➔ Overall, larger keys improve security but come at the cost of slower performance.

References

1. Python Cryptography Library: https://cryptography.io/en/latest/

2. Cryptography Library Documentation:https://cryptography.io/en/latest/hazmat/primitives/

3. Python Official Documentation: https://docs.python.org/3/

4. Gemini Ai

5. Github Copilot Al