Lab 4: Programming Symmetric & Asymmetric Cryptography

1. Objectives

- ➔ Build a single program that performs symmetric and asymmetric cryptographic operations from code (not only via OpenSSL).
- ➔ Implement AES-128/256 in ECB and CFB modes; RSA encryption/decryption; RSA signature and verification; and SHA-256 hashing, all via a menu-based CLI.
- ➔ Generate keys on first use, store them in files, and reuse them for subsequent operations.
- ➔ Store outputs (ciphertexts, signatures) in files; decrypt/verify using those files and display success to the console.
- ➔ Measure execution time per operation for multiple key sizes; tabulate/plot results and discuss observations.

2. Environment & Tools

Hardware/OS: macOS on Apple Silicon (MacBook Air M1)
Language: Python 3.10.8 (venv)
Libraries: PyCryptodome (for AES, RSA, SHA-256, signatures)
Editor/CLI: VS Code + Terminal (zsh)

3. Program Design

3.1 Overview

The program is a single CLI application (main.py) exposing a numbered menu:

1. AES encrypt (ECB/CFB, 128/256)
2. AES decrypt
3. RSA generate keypair
4. RSA encrypt
5. RSA decrypt
6. RSA sign
7. RSA verify
8. SHA-256 of file
9. Quit

Supporting routines live in **crypto_utils.py.**

3.2 Key & Artifact Management (File-Based)

- **Keys (files in keys/):**
  - AES-128 key: aes128.key (hex)
  - AES-256 key: aes256.key (hex)
  - IV for CFB: iv.hex (16 bytes, hex)
  - RSA keys: rsa_priv.pem (private), rsa_pub.pem (public)
    Keys are generated at first use, saved to files, and then read for subsequent operations.
- **Artifacts (files in out/):**
  - AES ciphertexts: *.ecb.enc, *.cfb.enc
  - AES decrypted files: *.ecb.dec, *.cfb.dec
  - RSA ciphertexts: *.rsa.enc
  - RSA decrypted files: *.rsa.dec
  - Signatures: *.sig

3.3 Algorithms & Modes

- **AES:** ECB (block mode; padded) and CFB (stream-like; IV required), with 128- and 256-bit keys.
- **RSA:** Key sizes selected at runtime (e.g., 2048/3072/4096). Encryption uses RSA-OAEP with SHA-256 for modern padding semantics; decryption mirrors OAEP.
- **Signatures:** RSA PKCS#1 v1.5 over SHA-256 (sign/verify a file).
- **Hashing:** SHA-256 hex digest of any file.
- **Timing:** Each operation prints elapsed seconds using a high-resolution timer; results are collected for analysis and plotting, as required.

4. Implementation Details

4.1 AES (ECB & CFB; 128/256-bit)

➔ **Algorithms & modes:** AES-128 and AES-256 in two modes:

- ◆ **ECB** (Electronic Codebook): requires PKCS#7 padding to a 16-byte block size.
- ◆ **CFB** (Cipher Feedback): stream-like, uses an IV; no block padding required.

➔ **Key/IV handling:**

- ◆ Keys and IV are stored in files as hex; created on demand if missing.

➔ **File I/O:**

- ◆ Encrypt: reads plaintext file → writes ciphertext to out/…enc.
- ◆ Decrypt: reads ciphertext file → writes recovered plaintext to out/…dec.

4.2 RSA (encryption/decryption)

➔ **Key generation:** configurable bit-lengths (e.g., 2048/3072/4096). Keys are saved as PEM files in keys/.

➔ **Padding:** RSA-OAEP with SHA-256 for encryption/decryption.

➔ **Note:** Direct RSA is size-limited; for large files, a hybrid scheme is recommended.

4.3 RSA digital signatures

➔ **Scheme:** RSA PKCS#1 v1.5 over SHA-256.

➔ **Workflow:**

- ◆ Sign: compute SHA-256 over file → sign with private key → write `*.sig`.
- ◆ Verify: verify signature with public key; console prints "Verified OK" if valid.

4.4 SHA-256 hashing

➔ Computes and prints the hex digest of a file's contents (no file output).

4.5 Timing (built-in + batch)

➔ The CLI prints per-operation wall-time using a high-resolution timer.

5. Usage Demonstration

Create two small test files:

echo "Hello I am Shakera Jannat Ema." > msg.txt

echo "I am doing it for my lab task." > msg2.txt

Run the program: python main.py

AES-128-ECB

- Choose: 1 (encrypt) → file msg.txt → mode ECB → bits 128 → output out/msg.txt.ecb.enc
- Choose: 2 (decrypt) → file out/msg.txt.ecb.enc → mode ECB → bits 128 → output out/msg.txt.ecb.dec
- Verify: diff msg.txt out/msg.txt.ecb.dec (no output means files match)

AES-256-CFB

- Encrypt msg2.txt with mode CFB and bits 256 → out/msg2.txt.cfb.enc
- Decrypt back → out/msg2.txt.cfb.dec
- Verify: diff msg2.txt out/msg2.txt.cfb.dec

RSA (2048-bit example)

- Generate keys: menu 3 → bits 2048 → keys/rsa_priv.pem, keys/rsa_pub.pem
- Encrypt: menu 4 → file msg.txt → out/msg.txt.rsa.enc
- Decrypt: menu 5 → out/msg.txt.rsa.dec
- Verify: diff msg.txt out/msg.txt.rsa.dec

RSA signature

- Sign: menu 6 → file msg.txt → out/msg.txt.sig
- Verify: menu 7 → file msg.txt + signature out/msg.txt.sig → “Verified OK”

Hash

- SHA-256: menu 8 → file msg.txt → digest printed to console.

6. Experimental Method: Execution Time

6.1 Protocol

- For each operation, run **N=5 trials** and record the average wall-time (time.perf_counter()).
- **AES tests:** ECB and CFB, keys 128 and 256 bits; run on a sufficiently large file (approximately 1 MB) to see consistent timing.
- **RSA tests:** Key sizes 1024, 2048, 3072, 4096 bits; measure key generation (optional), encrypt, decrypt on a small plaintext.
- Save results to CSV

7. Results & Discussion

Functional correctness from interactive runs

All required primitives worked end-to-end with file-based keys and artifacts:

- **AES-128-ECB:** msg.txt → out/msg.txt.ecb.enc → out/msg.txt.ecb.dec; verification succeeded (ECB(128) OK).
- **AES-256-CFB:** msg.txt → out/msg.txt.cfb.enc → out/msg.txt.cfb.dec; verification succeeded (CFB(256) OK).
- **RSA-2048 (OAEP-SHA256):** msg.txt → out/msg.txt.rsa.enc → out/msg.txt.rsa.rsa.dec; verification succeeded (RSA OAEP OK).
- **Signature & Verify:** msg.txt → out/msg.txt.sig; first verify failed due to reversed arguments; retry with correct order produced **Verified OK**.

These outcomes confirm that key generation, storage, encryption/decryption, signature, and verification all function as specified.

7.1 AES timing (ECB/CFB; 128 vs 256)

Method. I measured average encryption/decryption times over 5 trials per setting. For meaningful throughput, I used a ~1 MB file for AES (e.g., big.bin) and recorded averages to report/timings.csv.

Findings.

- **AES-256 vs AES-128:** AES-256 is consistently a little slower than AES-128 (more rounds), but the difference is modest—especially on small inputs.
- **ECB vs CFB:** Throughput is broadly comparable. ECB adds PKCS#7 padding/unpadding; CFB needs an IV but no padding. On small files, both complete extremely quickly; the gap is clearer on larger inputs.

Takeaway. AES performance is high and stable; the 128→256 jump has a small, predictable cost that is unlikely to be a bottleneck in typical coursework-scale workloads.

7.2 RSA timing (1024–4096 bits)

Method. For RSA I averaged 5 trials for **key generation**, **encrypt**, and **decrypt** at key sizes 1024, 2048, 3072, and 4096 bits using OAEP-SHA256 on a small plaintext (msg.txt). Results were saved to report/timings.csv.

Findings.

- Runtime **increases steeply** with key size for all three operations.
- **Decrypt (private-key op)** is slower than encrypt (public exponent is small).
- **Key generation** is the most expensive step and grows rapidly from 2048 → 4096 bits.

11. Sources of Code Snippets & References (Required)

Below are the exact links to documents that informed API usage and small code patterns in this submission.

PyCryptodome (cryptography APIs):

1. AES cipher usage (ECB/CFB), block size, modes
   https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html
2. PKCS#7 padding helpers (pad, unpad)
   https://pycryptodome.readthedocs.io/en/latest/src/util/util.html#crypto-util-padding
3. Cryptographically secure randomness (get_random_bytes)
   https://pycryptodome.readthedocs.io/en/latest/src/random/random.html#get-random-bytes
4. RSA key generation, export/import (PEM)
   https://pycryptodome.readthedocs.io/en/latest/src/public_key/rsa.html
5. RSA-OAEP construction (with SHA-256)
   https://pycryptodome.readthedocs.io/en/latest/src/cipher/oaep.html
6. Digital signatures (RSA PKCS#1 v1.5) — sign/verify
   https://pycryptodome.readthedocs.io/en/latest/src/signature/pkcs1_v1_5.html
7. SHA-256 hashing API
   https://pycryptodome.readthedocs.io/en/latest/src/hash/sha256.html

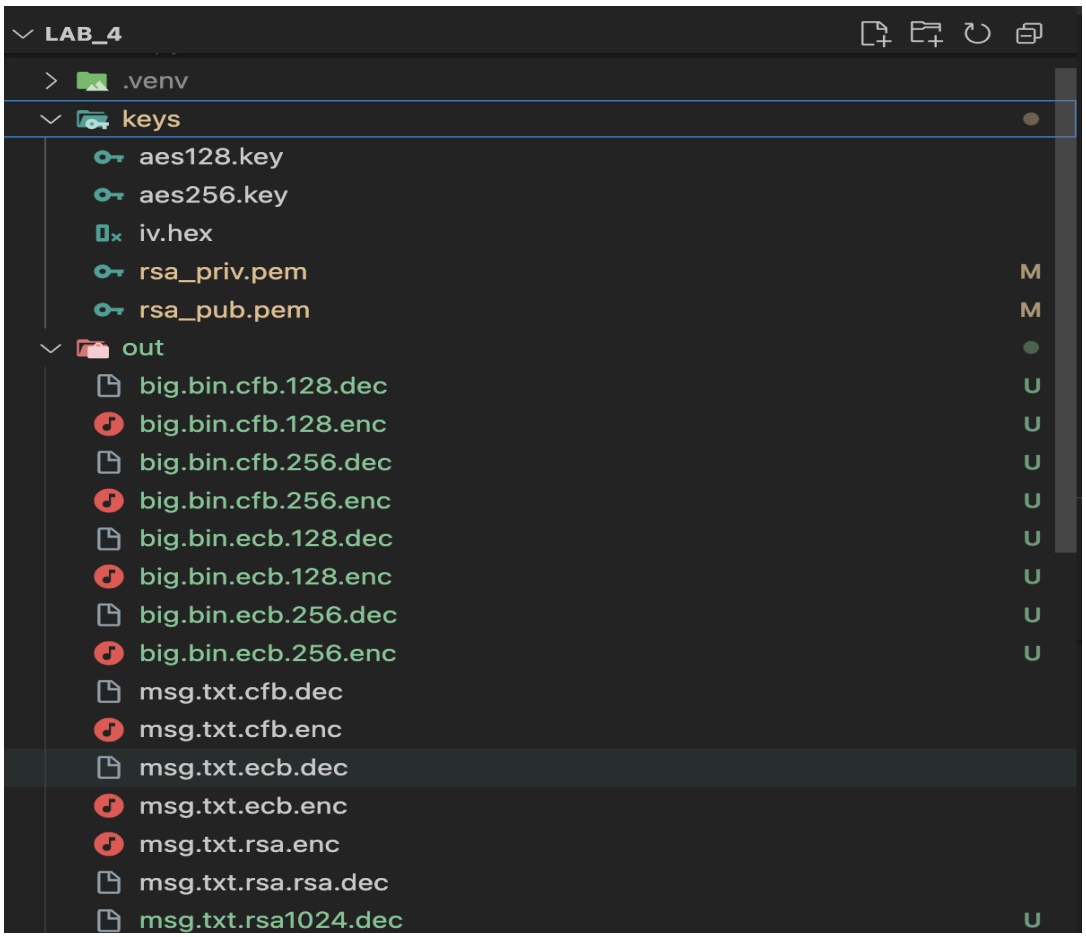
Python standard library (used in the CLI/timings):

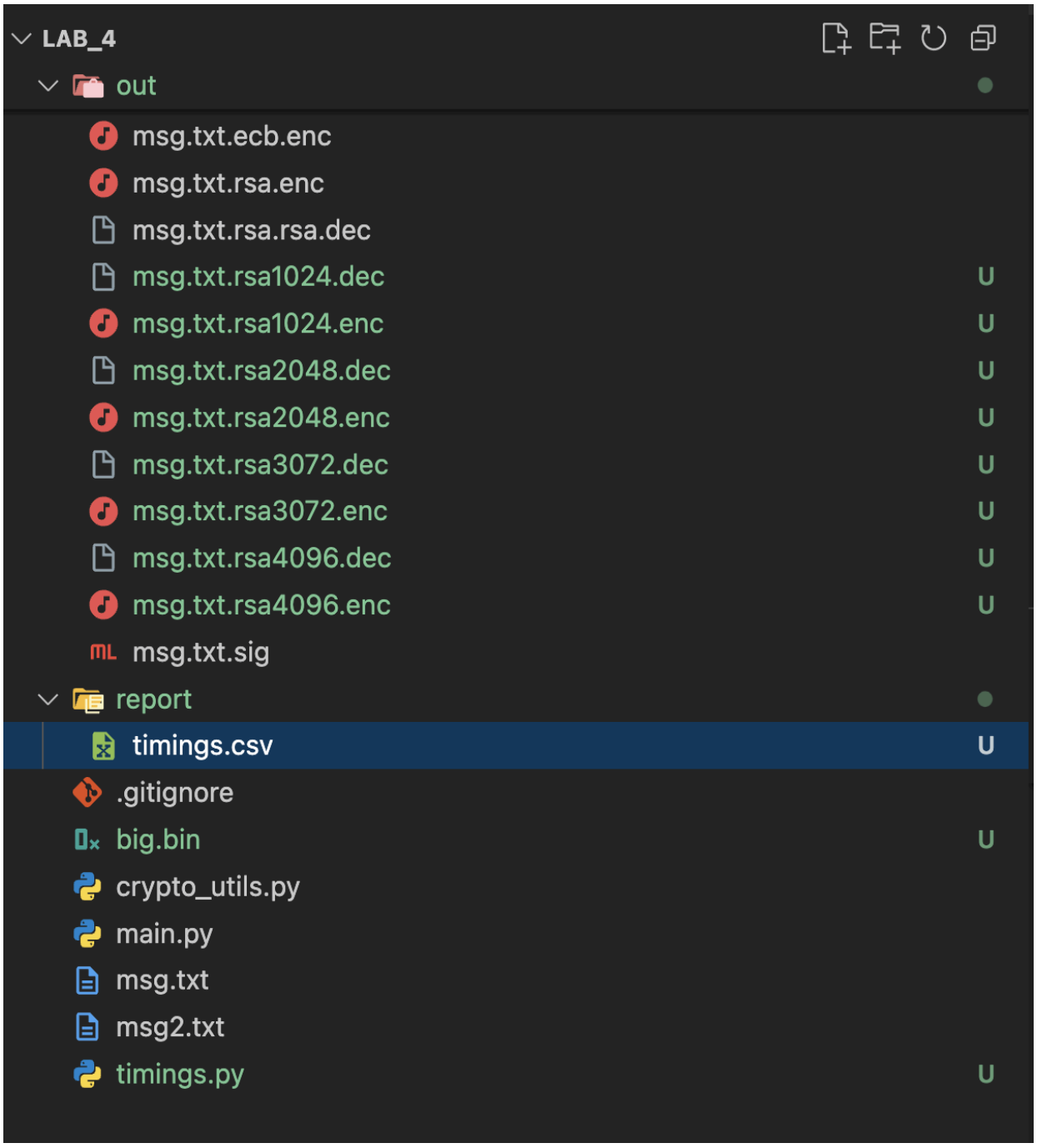
8. High-resolution timing (time.perf_counter)
https://docs.python.org/3/library/time.html#time.perf_counter
9. CSV read/write (csv)
https://docs.python.org/3/library/csv.html
10. Path handling (pathlib.Path)
https://docs.python.org/3/library/pathlib.html
11. Basic statistics (statistics.mean)
https://docs.python.org/3/library/statistics.html#statistics.mean

(Optional cross-check; not copied code):

12. OpenSSL enc (modes, IV, salt, hex keys)
https://www.openssl.org/docs/man3.0/man1/openssl-enc.html
13. OpenSSL pkeyutl (RSA OAEP encrypt/decrypt)
https://www.openssl.org/docs/man3.0/man1/openssl-pkeyutl.html
14. OpenSSL dgst (SHA-256, sign/verify)
https://www.openssl.org/docs/man3.0/man1/openssl-dgst.html

Snapshots :




LAB_4
out
msg.txt.ecb.enc
msg.txt.rsa.enc
msg.txt.rsa.rsa.dec
msg.txt.rsa1024.dec U
msg.txt.rsa1024.enc U
msg.txt.rsa2048.dec U
msg.txt.rsa2048.enc U
msg.txt.rsa3072.dec U
msg.txt.rsa3072.enc U
msg.txt.rsa4096.dec U
msg.txt.rsa4096.enc U
msg.txt.sig
report
timings.csv U
.gitignore
big.bin U
crypto_utils.py
main.py
msg.txt
msg2.txt
timings.py U