Lab Report: Programming Symmetric & Asymmetric Crypto (Lab 4)

Files Submitted

- `crypto_lab.py` — single Python script that implements all required functionalities and the benchmarker.
- `README.md` — short instructions for running the program (this report substitutes the README for submission).
- `plots/` — output directory generated by the script when the `benchmark` command is run (contains PNG files).
- `keys/` and `data/` — directories created by the script at runtime.

Tools and Libraries

- **Language:** Python 3.8+.
- **Crypto library:** PyCryptodome (`pip install pycryptodome`).
- **Plotting:** matplotlib (`pip install matplotlib`).
- **Other:** standard Python libraries: `argparse`, `time`, `hashlib`, `json`, `os`.

All code is contained in `crypto_lab.py`.

How to run

1. Install dependencies:

```
pip install pycryptodome matplotlib
```

1. View help / usage:

```
python crypto_lab.py -h
```

1. Example flows:

2. Generate AES-128 key:

```
python crypto_lab.py gen-aes --bits 128
```

- Encrypt with AES-128 in CFB:

```
python crypto_lab.py aes-encrypt secret.txt secret.txt.enc --bits 128 --mode
CFB
```

- Decrypt:

```
python crypto_lab.py aes-decrypt secret.txt.enc --bits 128 --mode CFB
```

- Generate RSA keypair (2048 bits):

```
python crypto_lab.py gen-rsa --bits 2048
```

- RSA encrypt and decrypt:

```
python crypto_lab.py rsa-encrypt keys/rsa_public_2048.pem secret_small.bin
secret_small.bin.enc
python crypto_lab.py rsa-decrypt keys/rsa_private_2048.pem
secret_small.bin.enc
```

- Sign and verify:

```
python crypto_lab.py sign keys/rsa_private_2048.pem file.txt file.sig
python crypto_lab.py verify keys/rsa_public_2048.pem file.txt file.sig
```

- Hash a file with SHA-256:

```
python crypto_lab.py hash file.txt
```

- Run full benchmark (creates plots in `plots/`):

```
python crypto_lab.py benchmark
```

The program creates `keys/`, `data/`, and `plots/` directories automatically if they do not exist.

Implementation Details

High-level design

The program is a single command-line tool with subcommands for each operation: `gen-aes`, `aes-encrypt`, `aes-decrypt`, `gen-rsa`, `rsa-encrypt`, `rsa-decrypt`, `sign`, `verify`, `hash`, and `benchmark`. Keys are stored as raw bytes (AES) or PEM files (RSA) in the `keys/` directory. Encrypted outputs and signatures are written to files in `data/` (or any path provided by the user).

The implementation uses `time.perf_counter()` to measure elapsed wall-clock time for each operation.

AES (Symmetric)

- **Key sizes supported:** 128, 192, 256 bits (the AES standard). The CLI allows 128 or 256 as requested by the lab.
- **Modes:** ECB and CFB.
- **Padding:** PKCS#7 padding is used for ECB mode. For CFB mode no padding is required (CFB is a streaming mode).
- **File I/O conventions:**
- **ECB:** ciphertext file contains raw ciphertext bytes.
- **CFB:** ciphertext file contains `IV || ciphertext` (IV prepended) so that the decryptor can extract the IV.
- **Notes and security remarks:**
- ECB leaks patterns and is not recommended for real data; used here only for educational / comparison purposes.
- For production systems prefer authenticated encryption modes (AES-GCM).

RSA (Asymmetric)

- **Key generation:** RSA keypairs are generated using PyCryptodome's `RSA.generate(bits)` and stored in PEM format.
- **Encryption/decryption:**
- Standard RSA (PKCS#1 OAEP) is used for encryption; however RSA can only encrypt messages smaller than the key modulus minus padding overhead. For demonstration, a simple chunking strategy is implemented: the plaintext is split into chunks small enough to be encrypted individually and the ciphertext file stores a small header plus length-prefixed ciphertext chunks for robust parsing.
- **Recommended practice:** Hybrid encryption (generate a random symmetric key, encrypt the bulk data with AES, encrypt the AES key with RSA) is the usual and efficient approach. The implemented chunking is for lab demonstration.
- **Signatures:** RSA signing uses PKCS#1 v1.5 with SHA-256 as the hash function. Signatures are stored as raw bytes in the signature file.

SHA-256

- The script computes SHA-256 of a file using Python's `hashlib.sha256` and prints the hex digest.

Key storage and usage

- AES keys are exact-length raw binary files `keys/aes_key_<bits>.bin`.
- RSA keys are PEM files `keys/rsa_private_<bits>.pem` and `keys/rsa_public_<bits>.pem`.
- The program looks up the appropriate key files by convention when `--bits` is specified or a key file path is provided directly.

Timing & Benchmarking Methodology

The lab requests timing as a function of key size (N), trying at least five different N values starting from 16 bits for AES and RSA. The lab manual explicitly requests starting from 16 bits. In practice:

- **AES:** The AES standard requires key sizes of 128, 192, or 256 bits. Libraries typically do not accept artificially small AES keys (e.g., 16 bits). To comply with the lab's intent (collect timings across different key sizes) while producing meaningful and valid results, the benchmark measures AES across the standard key sizes (128, 192, 256) and across multiple plaintext sizes (e.g., 16 B, 128 B, 1 KB, 8 KB, 64 KB). This lets you observe encryption time scaling with plaintext size while comparing standard AES key sizes.

- **RSA:** For RSA we benchmark across a variety of key sizes including small (but practical) values. The provided script uses the following default RSA key sizes: 512, 1024, 2048, 3072, and 4096 bits. These give five data points and show how key generation and encryption/decryption costs scale with modulus size. *Note:* extremely small RSA keys (e.g., 16 bits) are insecure and will often be rejected by libraries.

Timing approach: For each tested configuration (key size, plaintext size, mode), the script:

1. Starts `time.perf_counter()` immediately before the cryptographic operation.
2. Executes the operation (keygen/encrypt/decrypt/sign/verify).
3. Stops the timer and records the elapsed time.

For more robust measurements, run each configuration multiple times (e.g., 10 trials) and compute mean ± standard deviation. The script currently runs single trials; you can modify it to repeat trials for statistical stability.

Plotting: The benchmark routines produce PNG plots in `plots/` showing:

- AES: plaintext size (log scale) vs encryption time for each AES key size.
- RSA: key size vs key-generation/encrypt/decrypt time.
