Lab 4 user manual

Folder Structure :

```
.
├── Crypto.py
├── data
│   ├── ani1_cfb.json
│   ├── ani1_ecb.json
│   ├── ani2_cfb.json
│   ├── ani2_ecb.json
│   ├── ani3_cfb.json
│   ├── ani3_ecb.json
│   ├── obi1.bin
│   ├── obi2.bin
│   └── obi3.bin
├── ExecutionLog.csv
├── keys
│   ├── AESKey-16-1.bin
│   ├── AESKey-16-3.bin
│   ├── AESKey-24-1.bin
│   ├── AESKey-24-3.bin
│   ├── AESKey-32-1.bin
│   ├── AESKey-32-3.bin
│   ├── private-1024.pem
│   ├── private-2048.pem
│   ├── private-4096.pem
│   ├── public-1024.pem
│   ├── public-2048.pem
│   └── public-4096.pem
├── Lab4-documentation.md
├── plots
│   ├── aes_stuff-AES.png
│   └── RSA_logs.png
├── __pycache__
│   ├── Crypto.cpython-36.pyc
│   └── Crypto.cpython-38.pyc
├── sample.txt
├── sample_sig_1024.bin
├── sample_sig_2048.bin
└── sample_sig_4096.bin
```

Crypto.py :

The main code that handles all 4 crypto operations.

data

Contains encrypted data generated during crypto operations

`keys`

Contains keys generated during crypto operations

`plots`

Stores plot generated from the log file.

`ExecutionLog.csv`

Contains various statistics regarding the crypto operations.

Instructions for running code :

```
$ cd /lab4

$ pip install -r requirements.txt

$ python `Crypto.py`
```

After running the script, follow the prompted instructions on the terminal to perform different crypto operations.

Sources :

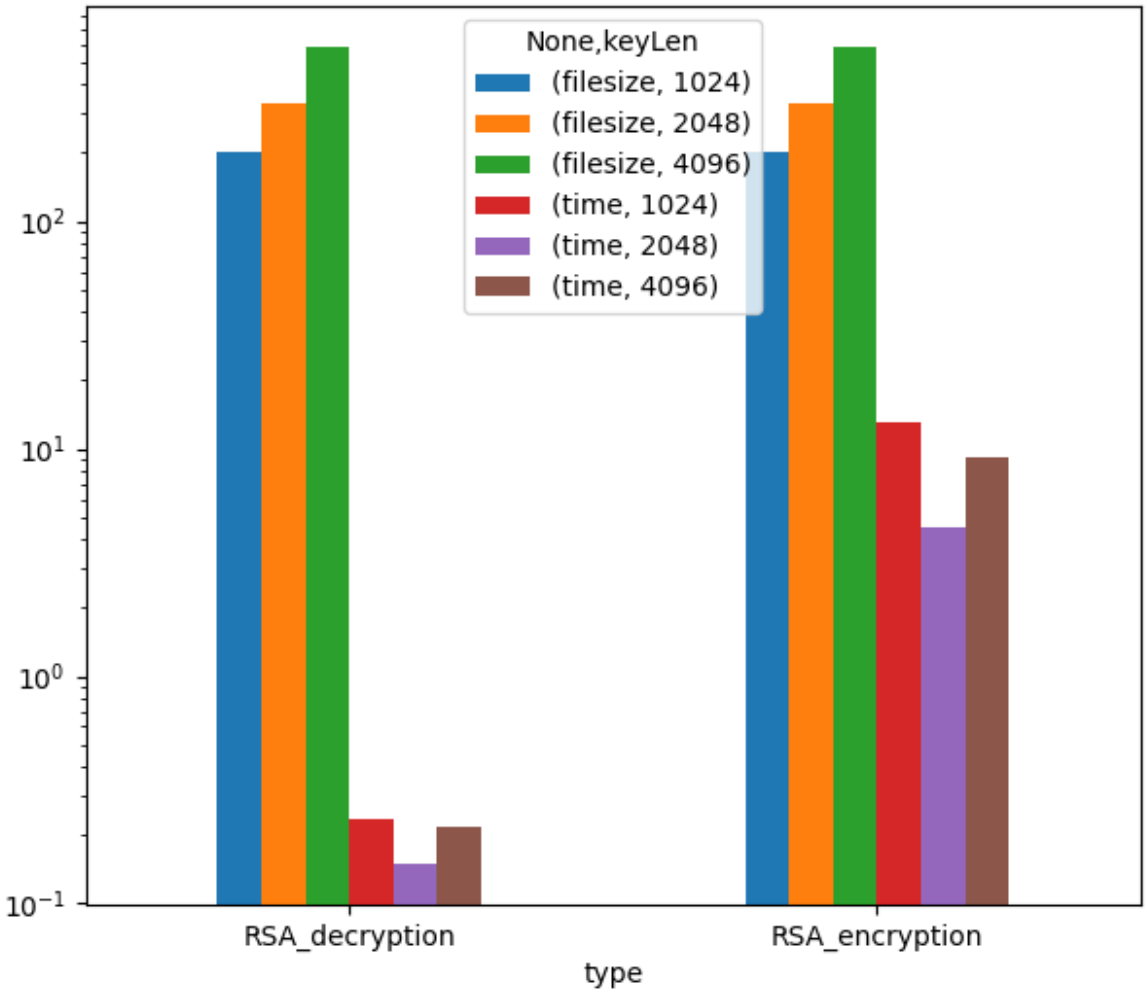
PyCryptodome Documentation.

Lab 4 Observation Report

For both `AES` and `RSA` we use the same sample text (`It's over Anakin! I have the high ground!`) for encryption and we record the execution time, ciphertext size for different keys during both encryption and decryption. Visulization was done in `base-10 logarithmic scale` on the Y-axis. Both the algorithms were emulated through the `PyCryptodome` library.

RSA

The time algorithmic complexity of RSA is O(n^2). It is observed that as the size of private key length increases, the increase in time becomes nonlinear and exponential.



Here, we successively tried the keylengths [1024, 2048, 4096] as per the instructions.

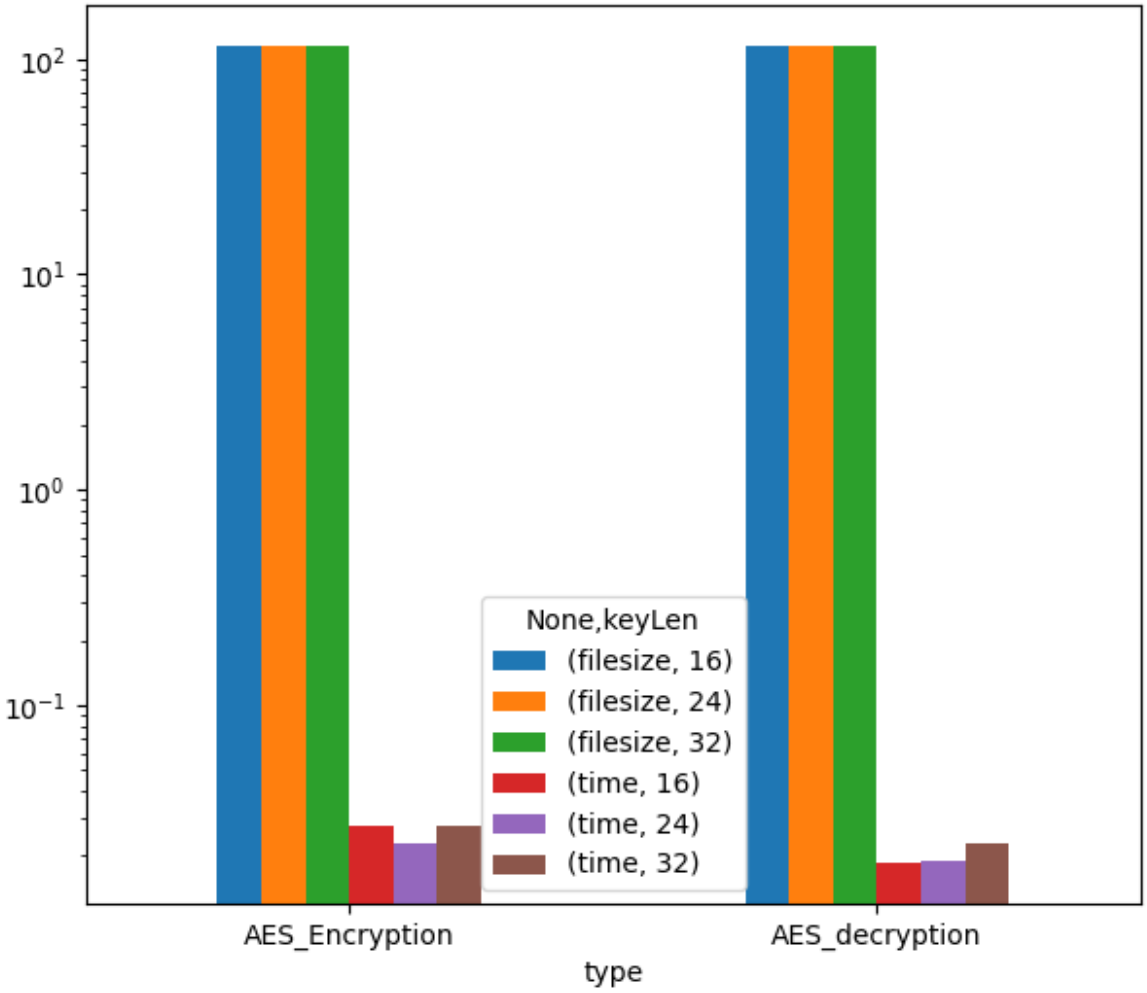
From our primary observations we see that :

- Encryption takes several multitudes more time than decryption.
- The size of ciphertext increases with the increase of `KeyLength`
- The time consumption follows similar trend for both encryption and decryption

- The execution time seems arbitrary for both encryption and decryption

AES

Considering that this algorithm is operating on a fixed block size, and each block will take the same amount of time regardless of the input, the time complexity of this algorithm is only O(1).



Here, we successively tried the keylengths [128, 192, 256] as per the instructions.

From our primary observations we see that :

- The generated ciphertext has the same size regardless of the keylength, as is expected.
- The encryption decryption method does not follow the same trend.
- The execution time seems arbitrary for both encryption and decryption

Source :

Algohub

RSA Signature

Both signature generation and verification uses 3 sizes of keys. 1024, 2048, and 4096. The keys are generated the first time the program is run. For further runs, the stored keys are used which are saved in the folder `keys` .

Signature Generation

1. Run the program and select option 3
2. To generate a new signature enter option 1
3. Enter the path to the file to generate the signature. Example, `sample.txt`
4. Enter the size of the key to use. One of 1024, 2048, 4096
5. The signature file will be saved in the same location as the input file with a `_sig.bin` suffix. Example, for 1024, it is saved in `sample_sig_1024.bin`
6. To generate a new signature with a new key, delete the `.pem` files from the `keys` folder. This will generate a new set of keys during the next run.

```
Select Action:
1. AES encryption/decryption
2. RSA encryption/decryption
3. RSA Signature
4. SHA-256 Hashing
5. Plot Data(if exists)
Enter Option: 3

Select Action:
1. Generate signature for a file.
2. Verify signature.
Enter Option: 1
Enter path to input filename: sample.txt
Enter keylen (1024, 2048 or 4096): 1024
Using previously generated key found in: keys\rsa_1024_private.pem, keys\rsa_1024_public.pem
Saved signature in: sample_sig_1024.bin
```

Signature Verification

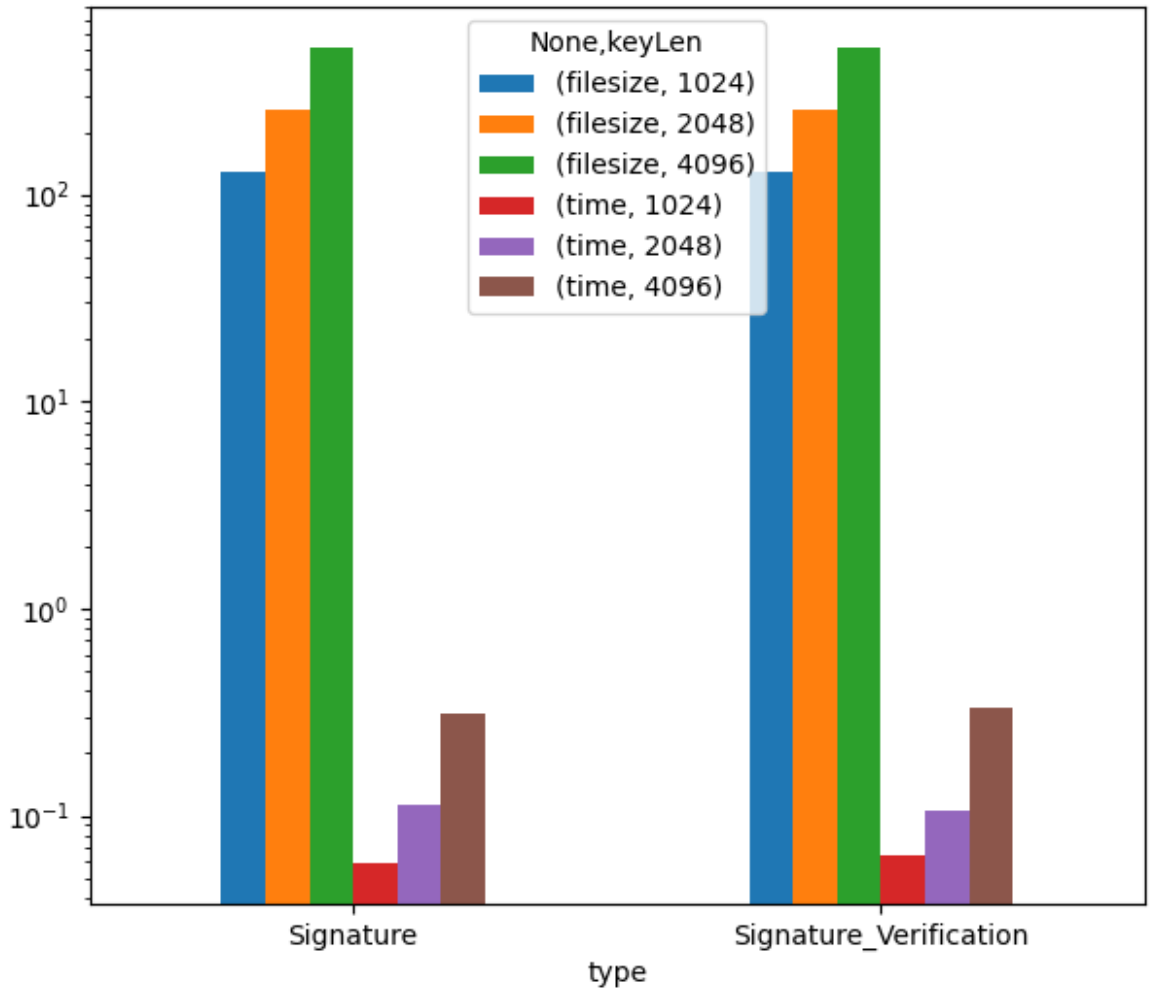
1. Run the program and select option 3
2. To verify a previously generated signature select option 2
3. Enter the path to the main content file. Example, `sample.txt`
4. Enter the path to the previously generated signature file to verify. Example `sample_sig_1024.bin`
5. Enter the keylen that was used to generate the signature. The key must be same, otherwise the signature won't match.

```
Select Action:
1. AES encryption/decryption
2. RSA encryption/decryption
3. RSA Signature
4. SHA-256 Hashing
5. Plot Data(if exists)
Enter Option: 3

Select Action:
1. Generate signature for a file.
2. Verify signature.
Enter Option: 2
Enter path to input filename: sample.txt
Enter path to signature file: sample_sig_1024.bin
Enter keylen used to generate signature (1024, 2048 or 4096): 1024
Using previously generated key found in: keys\rsa_1024_private.pem, keys\rsa_1024_public.pem
Signature is valid.
```

Time Comparison

RSA Signature generation and verification both use the RSA algorithm to encrypt the digest of the input message. Since we are using pre-generated keys the following chart compares the algorithm runtime only, without considering the key generation phase.



As we can see from the above plot, the runtime for signature generation and verification are identical. As expected the runtime changes with the size of the key used. 1024 needs the least time, while 4096 needs the most. A Similar pattern is found for file size as well. Higher size keys produce larger signature files.

SHA256

Follow the steps to perform SHA256 Hashing

1. Run the program and enter option 4 to use SHA256 Hashing.
2. Enter the path to the file that you want to hash. A sample file called `sample.txt` is provided in the same folder.
3. The output hash is displayed to the console.

```
Select Action:
1. AES encryption/decryption
2. RSA encryption/decryption
3. RSA Signature
4. SHA-256 Hashing
5. Plot Data(if exists)
Enter Option: 4

Enter the input filename to hash: sample.txt
SHA256 Hash for sample.txt :  0c655a91d827498a97980cc07d8b0b8b2fc0248b2a8d37b9015769088a5593de
Completed in : 0.0 seconds.
```