Information Security HW 1 Report

0856622 余家宏

- Github Link (https://github.com/yamiefun/Information-Security-HW1)
- It's recommended to read this report on hackmd for a better reading experience. Links are provided in Github above.

1. Generate Random File with Specific File Size

```
def create_plaintext(mbytes):
    ret = os.urandom(mbytes*1024*1024+1)
    return ret
```

I used this function to create random files as plaintext. The input of this function is the size of the random file in MBytes. Since the input of `os.urandom` is the bytes of target data, I multiplied `mbytes` by 1024^2 to convert the unit. Additionally, I added 1 extra byte to the plaintext to make it need to be padded while doing encryption.

- urandom DOC (https://docs.python.org/3/library/os.html#os.urandom)

2. Cryptography Library

- PyCryptodome (https://pycryptodome.readthedocs.io/en/latest/src/introduction.html)

In this project, all encryption and decryption functions are supported by `PyCryptodome(v3.9.9)` . You should checkout the link above to install `PyCryptodome` to your environment if you want to reproduce my works.

3. Padding

- pad (https://pycryptodome.readthedocs.io/en/latest/src/util/util.html?highlight=pad#Crypto.Util.Padding.pad)

The padding function I used in this project is supported by `PyCryptodome` . The default padding style is `PKCS7` . Please checkout the link above for more details.

4. Implementation of Different Encryption Methods

In the code `task4.py` , there's a function called `encrypt` , which contains all the encryption methods used in this project. The input of `encrypt` function is a byte stream plaintext and a encryption methods. In the main function, I used a `for` loop to iterate all kinds of encryption methods, running them with different length of plaintext, and compared the running time with a chart.

4.1 AES with Different Mode of Operations

Reference

- [AES-CBC](https://pycryptodome.readthedocs.io/en/latest/src/cipher/classic.html?highlight=cbc#cbc-mode) (https://pycryptodome.readthedocs.io/en/latest/src/cipher/classic.html?highlight=cbc#cbc-mode)
- [AES-OCB](https://pycryptodome.readthedocs.io/en/latest/src/cipher/modern.html?highlight=ocb#ocb-mode) (https://pycryptodome.readthedocs.io/en/latest/src/cipher/modern.html?highlight=ocb#ocb-mode)
- [AES-GCM](https://pycryptodome.readthedocs.io/en/latest/src/cipher/modern.html?highlight=gcm#gcm-mode) (https://pycryptodome.readthedocs.io/en/latest/src/cipher/modern.html?highlight=gcm#gcm-mode)
- [AES-CCM](https://pycryptodome.readthedocs.io/en/latest/src/cipher/modern.html?highlight=ccm#ccm-mode) (https://pycryptodome.readthedocs.io/en/latest/src/cipher/modern.html?highlight=ccm#ccm-mode)

Implementation

```
1  from Crypto.Cipher import AES
2  from Crypto.Util.Padding import pad
3
4  key = get_random_bytes(32)
5  if mode == "AES_CBC":
6      iv = os.urandom(16)
7      cipher = AES.new(key, AES.MODE_CBC, iv)
8      start_time = time.time()
9      ret = cipher.encrypt(pad(plaintext, AES.block_size))
10 elif mode == "AES_OCB":
11     cipher = AES.new(key, AES.MODE_OCB)
12     start_time = time.time()
13     ret, _ = cipher.encrypt_and_digest(pad(plaintext, AES.block_size))
14 elif mode == "AES_GCM":
15     cipher = AES.new(key, AES.MODE_GCM)
16     start_time = time.time()
17     ret, _ = cipher.encrypt_and_digest(pad(plaintext, AES.block_size))
18 elif mode == "AES_CCM":
19     cipher = AES.new(key, AES.MODE_CCM)
20     start_time = time.time()
21     ret, _ = cipher.encrypt_and_digest(pad(plaintext, AES.block_size))
```

- Line4: The key length should be 32 bytes for AES-256.
- Line6: IV is generated by `urandom` for CBC mode.
- The running time only includes the `encrypt` function of each mode. Creating the `cipher` object is not included.

RSA

Reference

- RSA (https://pycryptodome.readthedocs.io/en/latest/src/public_key/rsa.html?highlight=RSA#rsa)

Implementation

```
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

elif mode == "RSA1024":
    # generate private key
    key = RSA.generate(1024)
    encrypted_key = key.export_key()
    file_out = open("rsa_key.bin", "wb")
    file_out.write(encrypted_key)
    file_out.close()

    encoded_key = open("rsa_key.bin", "rb").read()
    key = RSA.import_key(encoded_key)

    cipher = PKCS1_OAEP.new(key)
    # max block size of RSA 1024 is 86 Bytes
    blk_size = 86
    start_time = time.time()
    for i in range(0, len(plaintext), blk_size):
        blk_plain = plaintext[i:i+blk_size]
        # ret = cipher.encrypt(blk_plain)
```

- I'm not showing both RSA1024 and RSA2048 codes because the only difference is the length of the key and the maxium block size.
- Since the length of plaintext is limited by the mechanism of RSA, the recommanded method using RSA is to encrypt an AES key with RSA, and encrypt message with AES.
- In this project, I splitted the long plaintext into small blocks, and encrypt them separately by RSA.
- After testing, the maximun length of plaintext is 86 for RSA1024, 214 for RSA2048.

Hash

Reference

- Crypto-hash (https://pycryptodome.readthedocs.io/en/latest/src/hash/sha256.html?highlight=sha256#sha-256)
- Hashlib-hash (https://docs.python.org/3/library/hashlib.html#module-hashlib)

Implementation

```
from Crypto.Hash import SHA256

elif mode == "SHA":
    hash_obj = SHA256.new()
    start_time = time.time()
    hash_obj.update(plaintext)
    ret = hash_obj.hexdigest()
```

After testing, I found that `SHA256` provided in `Crypto` is surprisingly slow. So I tried to use another `SHA256` function in `hashlib`, and the speed was as fast as expected.

```
import hashlib

elif mode == "SHA":
    m = hashlib.sha256()
    start_time = time.time()
    m.update(plaintext)
    ret = m.digest()
```

ChaCha20

Reference

- [ChaCha20 (https://pycryptodome.readthedocs.io/en/latest/src/cipher/chacha20.html?highlight=chacha20#chacha20-and-xchacha20)](https://pycryptodome.readthedocs.io/en/latest/src/cipher/chacha20.html?highlight=chacha20#chacha20-and-xchacha20)

Implementation

```
from Crypto.Cipher import ChaCha20

elif mode == "CC20":
    key = get_random_bytes(32)
    cipher = ChaCha20.new(key=key)
    start_time = time.time()
    ret = cipher.encrypt(plaintext)
```

- ChaCha20 is a stream cipher designed by Daniel J. Bernstein.
- The length of the key of ChaCha20 is 32 bytes.

ChaCha20-Poly1305

Reference

- [ChaCha20-Poly1305 (https://pycryptodome.readthedocs.io/en/latest/src/cipher/chacha20_poly1305.html#chacha20-poly1305-and-xchacha20-poly1305)](https://pycryptodome.readthedocs.io/en/latest/src/cipher/chacha20_poly1305.html#chacha20-poly1305-and-xchacha20-poly1305)

Implementation

```
from Crypto.Cipher import ChaCha20_Poly1305

elif mode == "CCP":
    key = get_random_bytes(32)
    cipher = ChaCha20_Poly1305.new(key=key)
    start_time = time.time()
    ret, _ = cipher.encrypt_and_digest(plaintext)
```

- ChaCha20-Poly1305 is an authenticated cipher with associated data (AEAD).
- The length of the key is 32 bytes, and must never be reused.

MAC

Reference

- [Poly1305 (https://pycryptodome.readthedocs.io/en/latest/src/hash/poly1305.html?highlight=poly1305#poly1305)](https://pycryptodome.readthedocs.io/en/latest/src/hash/poly1305.html?highlight=poly1305#poly1305)
- [HMAC (https://pycryptodome.readthedocs.io/en/latest/src/hash/hmac.html?highlight=HMAC#hmac)](https://pycryptodome.readthedocs.io/en/latest/src/hash/hmac.html?highlight=HMAC#hmac)
- [CBC-MAC (https://pycryptodome.readthedocs.io/en/latest/src/hash/cmac.html?highlight=CMAC#cmac)](https://pycryptodome.readthedocs.io/en/latest/src/hash/cmac.html?highlight=CMAC#cmac)

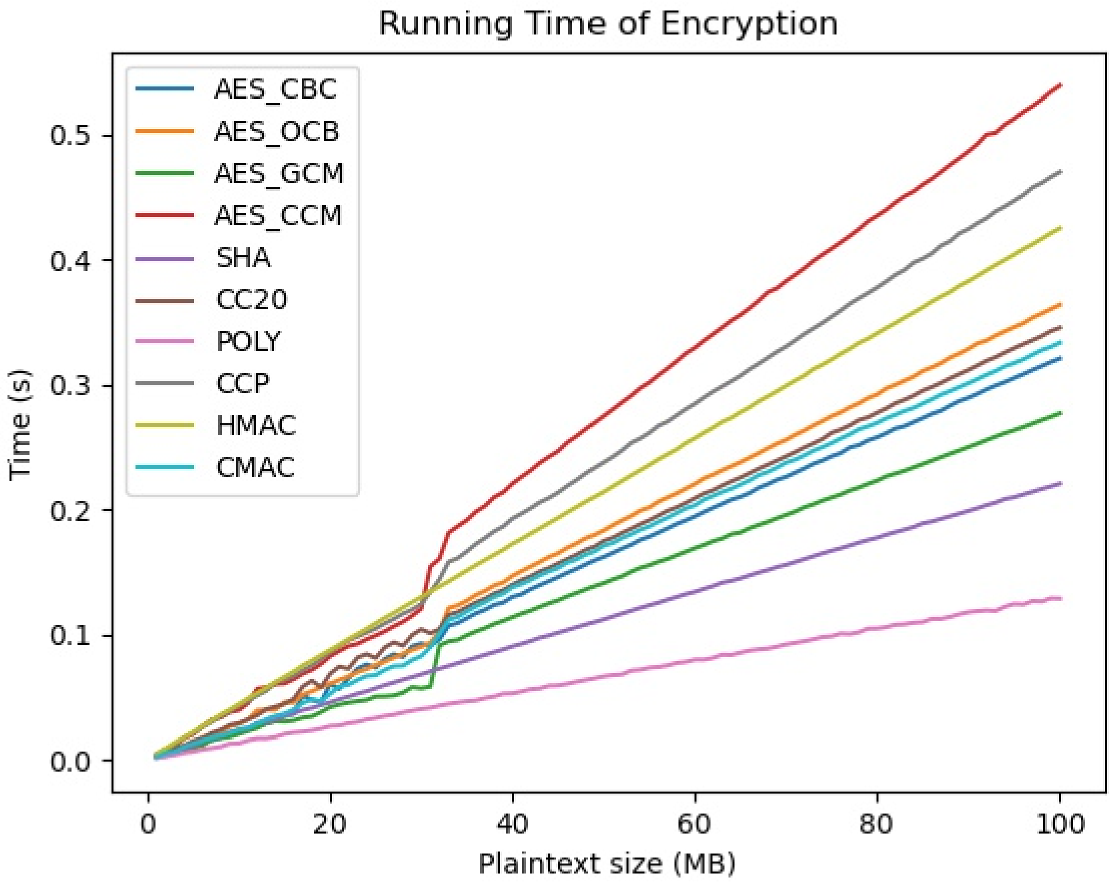
Implementation

```
from Crypto.Hash import Poly1305
from Crypto.Hash import HMAC
from Crypto.Hash import CMAC

elif mode == "POLY":
    key = get_random_bytes(32)
    mac = Poly1305.new(key=key, cipher=AES)
    start_time = time.time()
    mac.update(plaintext)
    ret = mac.hexdigest()
elif mode == "HMAC":
    h = HMAC.new(key, digestmod=SHA256)
    start_time = time.time()
    h.update(plaintext)
    ret = h.hexdigest()
elif mode == "CMAC":
    cobj = CMAC.new(key, ciphermod=AES)
    start_time = time.time()
    cobj.update(plaintext)
    ret = cobj.hexdigest()
```

- The length of the key of Poly1305 is 32 bytes.
- The cipher of Poly1305 could be ChaCha20.

Result



- Each line is average of 100 times testing.
- RSA is not shown in this image because it's too slow to compare with other methods.

5. Why Key as IV of AES-CBC is Bad

The code below follows the steps in cource slides. If user takes key as IV in AES-CBC, man in the middle can guess the key by using chosen ciphertext attack.

Step 0

Define key and IV as the same random value.

```
key = get_random_bytes(16)
iv = key
```

Step 1

Create plaintext P. The length of P is 3 AES blocks.

```
P = get_random_bytes(16*3)
```

Step 2

Encrypt plaintext P and get ciphertext C.

```
e_cipher = AES.new(key, AES.MODE_CBC, iv)
C = e_cipher.encrypt(P)
```

Step 3

Man in middle receive C, modify the middle of C to bytes with all zeros, call C'.

```
z = bytearray(16)
C_pr = C[0:16]+z+C[0:16]
```

Step 4

Decrypt C' by chosen ciphertext attack, get P', and separate it to 3 parts, P'_1, P'_2, and P'_3.

```
d_cipher = AES.new(key, AES.MODE_CBC, iv)
P_pr = d_cipher.decrypt(C_pr)
P1_pr = P_pr[0:16]
P2_pr = P_pr[16:32]
P3_pr = P_pr[32:48]
```

Step 5

Man in middle now can guess key by XORing P'_1 and P'_3.

```
guess_key = bytes(a ^ b for a, b in zip(P3_pr, P1_pr))
```

Step 6

Verify that guessed key is the same as the real key.

```
print("Original key: {}".format(base64.b64encode(key).decode('utf-8')))
print("Guessed key:  {}".format(base64.b64encode(guess_key).decode(
                                'utf-8')))
if guess_key == key:
    print("Guessed key is correct.")
else:
    print("Guessed key is not correct.")
```

Result

```
Original key: XxDJmwBW+7AGtUJYHVXiug==
Guessed key:  XxDJmwBW+7AGtUJYHVXiug==
Guessed key is correct.
```