資訊安全與密碼學—第一次作業

AES256-ECB
AES256-CBC
AES256-CTR
RSA2048
SHA512

4103056005
資工四
林佑儒

1 作業目標

1. 產生出一個大小為 512MB+7byte 的隨機檔案，目的在於使其在作加密時需要做 padding.
2. 使用 Python 的 PyCrypto 套件來做
   AES256-ECB、AES256-CBC、AES256-CTR、RSA2048、SHA512
3. 使用 Python 的 Cryptography 套件來做
   AES256-ECB、AES256-CBC、AES256-CTR、RSA2048、SHA512
4. 加密時，若遇到需要做 padding 的情況，需依照 PKCS 的 padding 規範
5. 測量兩套件各項功能所需的時間並比較之

2 系統環境

本次作業皆是在 macOS High Sierra + python2.7 的環境下測試執行

3 安裝套件

1. sudo pip install pycrypto
2. sudo pip install cryptography

4 PyCrypto

4.1 AES256-ECB

```
# PyCrypto_AES256_ECB
#=============================================================================
from Crypto.Cipher import AES
import os
import time
#=============================================================================
def start():
    global key
    key = os.urandom(32)                    #AES 256 bits = 32 bytes
    file = open("random.txt","r")
    global text
    text = file.read()                      #text will be the plaintext
    file.close()
#=============================================================================
def en_AES_ECB(key,text):
    padString = ''
    temp = 16-len(text)%16            #counting the padding number
    for i in range(0,temp):           #if surplus 7, then add 9 bytes 9
        padString = padString+chr(temp)
    text = text+padString

    cipher = AES.new(key)
    global encrypted
    encrypted = cipher.encrypt(text)                          #encrypt
#=============================================================================
def de_AES_ECB(key,encrypted):
    cipher = AES.new(key)
    decrypted = cipher.decrypt(encrypted)                      #decrypt
    temp = len(decrypted)                                     #unpadding
    temp = temp-int(decrypted[-1].encode('hex'),16)
    decrypted = decrypted[:temp]
#=============================================================================
start()
#AES256EBC
print "AES_256_ECB encode:",
startTime = time.time()
en_AES_ECB(key,text)
print time.time()-startTime

print "AES_256_ECB decode:",
startTime = time.time()
de_AES_ECB(key,encrypted)
print time.time()-startTime
#=============================================================================
```

4.2 AES256-CBC

```
# PyCrypto_AES256_CBC
#=============================================================================
from Crypto.Cipher import AES
import os
import time
#=============================================================================
def start():
    global key
    key = os.urandom(32)                        #AES 256 bits = 32 bytes
    file = open("random.txt","r")
    global text
    text = file.read()                          #text will be the plaintext
    file.close()
    global iv
    iv = os.urandom(16)                         #block size is 16 bytes
#=============================================================================
def en_AES_CBC(key,text,iv):
    padString = ''
    temp = 16-len(text)%16              #counting the padding number
    for i in range(0,temp):             #if surplus 7, then add 9 bytes 9
        padString = padString+chr(temp)
    text = text+padString

    cipher = AES.new(key,AES.MODE_CBC,iv)
    global encrypted
    encrypted = cipher.encrypt(text)                                #encrypt
#=============================================================================
def de_AES_CBC(key,encrypted,iv):
    cipher = AES.new(key,AES.MODE_CBC,iv)
    decrypted = cipher.decrypt(encrypted)                          #decrypt
    temp = len(decrypted)                                         #unpadding
    temp = temp-int(decrypted[-1].encode('hex'),16)
    decrypted = decrypted[:temp]
#=============================================================================
start()
#AES256CBC
print "AES_256_CBC encode:",
startTime = time.time()
en_AES_CBC(key,text,iv)
print time.time()-startTime

print "AES_256_CBC decode:",
startTime = time.time()
de_AES_CBC(key,encrypted,iv)
print time.time()-startTime
#=============================================================================
```

4.3 AES256-CTR

```
# PyCrypto_AES256_CTR
#=============================================================================
from Crypto.Cipher import AES
from Crypto.Util import Counter
import os
import time
#=============================================================================
def start():
    global key
    key = os.urandom(32)                        #AES 256 bits = 32 bytes
    file = open("random.txt","r")
    global text
    text = file.read()                          #text will be the plaintext
    file.close()
#=============================================================================
def en_AES_CTR(key,text):
    padString = ''
    temp = 16-len(text)%16              #counting the padding number
    for i in range(0,temp):             #if surplus 7, then add 9 bytes 9
        padString = padString+chr(temp)
    text = text+padString

    ctr = Counter.new(128)
    cipher = AES.new(key,AES.MODE_CTR,counter=ctr)
    global encrypted
    encrypted = cipher.encrypt(text)                                #encrypt
#=============================================================================
def de_AES_CTR(key,encrypted):
    ctr = Counter.new(128)
    cipher = AES.new(key,AES.MODE_CTR,counter=ctr)
    decrypted = cipher.decrypt(encrypted)                         #decrypt
    temp = len(decrypted)                                        #unpadding
    temp = temp-int(decrypted[-1].encode('hex'),16)
    decrypted = decrypted[:temp]
#=============================================================================
start()
#AES256CTR
print "AES_256_CTR encode:",
startTime = time.time()
en_AES_CTR(key,text)
print time.time()-startTime

print "AES_256_CTR decode:",
startTime = time.time()
de_AES_CTR(key,encrypted)
print time.time()-startTime
#=============================================================================
```

4.4 RSA2048

4.4.1 generate key

```
from Crypto.PublicKey import RSA
key = RSA.generate(2048)
f = open('privateKey.txt','w')
f.write(key.exportKey())
f.close()
f = open('publicKey.txt','w')
f.write(key.publickey().exportKey())
f.close()
print key.exportKey()
print key.publickey().exportKey()
```

4.4.2 Encode-Decode

```
# PyCrypto_RSA2048.py
#=============================================================================
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_v1_5
from Crypto.Hash import SHA512
from Crypto import Random
import os
import time
#=============================================================================
def en_RSA():
    file = open("publicKey.txt","r")                    #import public key
    key = file.read()
    file.close()
    rsakey = RSA.importKey(key)
    cipher = PKCS1_v1_5.new(rsakey)
    addnum = rsakey.size()/8-10  #256-11
    file1 = open("temp.txt","w")
    file = open("random.txt","r")
    while 1:
        text = file.read(addnum)
        if len(text)==0:
            break
        file1.write(cipher.encrypt(text))                          #encrypt
    file1.close()
    file.close()
#=============================================================================
def de_RSA():
    file = open("privateKey.txt","r")              #import privateKey key
    key = file.read()
    file.close()
    rsakey =  RSA.importKey(key)
    cipher = PKCS1_v1_5.new(rsakey)
    dsize = SHA512.digest_size
    file = open("temp.txt","r")
    file1 = open("ans.txt","w")
    while 1:
        text = file.read(256)
        if len(text)==0:
```

```
            break
        sentinel = Random.new().read(15 + dsize)
        file1.write(cipher.decrypt(text, sentinel))            #decrypt
    file.close()
    file1.close()
#========================================================================
#RSA2048
print "RSA_2048    encode:",
startTime = time.time()
en_RSA()
print time.time()-startTime

print "RSA_2048    decode:",
startTime = time.time()
de_RSA()
print time.time()-startTime
#========================================================================
```

4.5 SHA512

```
# PyCrypto_SHA512.py
#========================================================================
from Crypto.Hash import SHA512
import os
import time
#========================================================================
def start():
    file = open("random.txt","r")
    global text
    text = file.read()
    file.close()
#========================================================================
def en_SHA512(text):
    h = SHA512.new()
    h.update(text)
    encrypted = h.hexdigest()
#========================================================================
start()
#SHA512
print "SHA_512  hsah_func:",
startTime = time.time()
en_SHA512(text)
print time.time()-startTime
#========================================================================
```

5 Cryptography

5.1 AES256-ECB

```
# Cryptography_AES256_ECB
#=========================================================================
import os
import time
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms
from cryptography.hazmat.primitives.ciphers import modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
#=========================================================================
def start():
    global key
    key = os.urandom(32)                    #AES 256 bits = 32 bytes
    file = open("random.txt","r")
    global text
    text = file.read()                      #text will be the plaintext
    file.close()
#=========================================================================
def en_AES_ECB(key,text):
    global encrypted
    backend = default_backend()
    padder = padding.PKCS7(128).padder()              #set padding mode
    padded_data = padder.update(text)
    padded_data += padder.finalize()                            #padding
    cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=backend)
    encryptor = cipher.encryptor()
    #encrypt
    encrypted = encryptor.update(padded_data) + encryptor.finalize()
#=========================================================================
def de_AES_ECB(key,encrypted):
    backend = default_backend()
    cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=backend)
    decryptor = cipher.decryptor()
    #decrypt
    tdata = decryptor.update(encrypted) + decryptor.finalize()
    unpadder = padding.PKCS7(128).unpadder()
    data = unpadder.update(tdata)+ unpadder.finalize()       #unpadding
#=========================================================================
start()
#AES256ECB
print "AES_256_ECB encode:",
startTime = time.time()
en_AES_ECB(key,text)
print time.time()-startTime

print "AES_256_ECB decode:",
startTime = time.time()
de_AES_ECB(key,encrypted)
print time.time()-startTime
#=========================================================================
```

5.2 AES256-CBC

```
# Cryptography_AES256_CBC
#=============================================================================
import os
import time
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms
from cryptography.hazmat.primitives.ciphers import modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
#=============================================================================
def start():
    global key,iv
    key = os.urandom(32)                    #AES 256 bits = 32 bytes
    iv = os.urandom(16)                     #block size is 16 bytes
    file = open("random.txt","r")
    global text
    text = file.read()                      #text will be the plaintext
    file.close()
#=============================================================================
def en_AES_CBC(key,text,iv):
    global encrypted
    backend = default_backend()
    padder = padding.PKCS7(128).padder()              #set padding mode
    padded_data = padder.update(text)
    padded_data += padder.finalize()                            #padding
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv),backend=backend)
    encryptor = cipher.encryptor()
    #encrypt
    encrypted = encryptor.update(padded_data) + encryptor.finalize()
#=============================================================================
def de_AES_CBC(key,encrypted,iv):
    backend = default_backend()
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv),backend=backend)
    decryptor = cipher.decryptor()
    #decrypt
    tdata = decryptor.update(encrypted) + decryptor.finalize()
    unpadder = padding.PKCS7(128).unpadder()
    data = unpadder.update(tdata)+ unpadder.finalize()       #unpadding
#=============================================================================
start()
#AES256CBC
print "AES_256_CBC encode:",
startTime = time.time()
en_AES_CBC(key,text,iv)
print time.time()-startTime

print "AES_256_CBC decode:",
startTime = time.time()
de_AES_CBC(key,encrypted,iv)
print time.time()-startTime
#=============================================================================
```

5.3 AES256-CTR

```
# Cryptography_AES256_CTR
#=============================================================================
import os
import time
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms
from cryptography.hazmat.primitives.ciphers import modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
#=============================================================================
def start():
    global key,ctr
    key = os.urandom(32)                    #AES 256 bits = 32 bytes
    ctr = os.urandom(16)                    #block size is 16 bytes
    file = open("random.txt","r")
    global text
    text = file.read()                      #text will be the plaintext
    file.close()
#=============================================================================
def en_AES_CTR(key,text,ctr):
    global encrypted
    backend = default_backend()
    padder = padding.PKCS7(128).padder()              #set padding mode
    padded_data = padder.update(text)
    padded_data += padder.finalize()                            #padding
    cipher = Cipher(algorithms.AES(key),modes.CTR(ctr),backend=backend)
    encryptor = cipher.encryptor()
    #encryptor
    encrypted = encryptor.update(padded_data) + encryptor.finalize()
#=============================================================================
def de_AES_CTR(key,encrypted,ctr):
    backend = default_backend()
    cipher = Cipher(algorithms.AES(key),modes.CTR(ctr),backend=backend)
    decryptor = cipher.decryptor()
    #decrypt
    tdata = decryptor.update(encrypted) + decryptor.finalize()
    unpadder = padding.PKCS7(128).unpadder()
    data = unpadder.update(tdata)+ unpadder.finalize()       #unpadding
#=============================================================================
start()
#AES256CTR
print "AES_256_CTR encode:",
startTime = time.time()
en_AES_CTR(key,text,ctr)
print time.time()-startTime

print "AES_256_CTR decode:",
startTime = time.time()
de_AES_CTR(key,encrypted,ctr)
print time.time()-startTime
#=============================================================================
```

5.4 RSA2048

5.4.1 generate key

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization

private_key = rsa.generate_private_key(
    public_exponent=65537,key_size=2048,backend=default_backend())
pem = private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.TraditionalOpenSSL,
    encryption_algorithm=serialization.NoEncryption()
)
f = open('privateKey.txt','w')
f.write(pem)
f.close()
pem = private_key.public_key().public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo
)
f = open('publicKey.txt','w')
f.write(pem)
f.close()
```

5.4.2 Encode-Decode

```
# Cryptography_RSA2048
#=========================================================================
import os
import time
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
#=========================================================================
def en_RSA():
    def en_RSA():
    file = open("publicKey.txt","r")
    public_key = serialization.load_pem_public_key(
        file.read(),
        backend=default_backend()
    )
    file.close()
    addnum = 245
    file1 = open("temp.txt","w")
    file = open("random.txt","r")
    while 1:
        text = file.read(addnum)
        if len(text)==0:
            break
        file1.write(public_key.encrypt(text,padding.PKCS1v15()))
    file1.close()
    file.close()
```

```
#=============================================================================
def de_RSA():
    file = open("privateKey.txt","r")
    private_key = serialization.load_pem_private_key(
        file.read(),
        password=None,
        backend=default_backend()
    )
    file.close()
    file = open("temp.txt","r")
    file1 = open("ans.txt","w")
    while 1:
        text = file.read(256)
        if len(text)==0:
            break
        file1.write(private_key.decrypt(text,padding.PKCS1v15()))
    file.close()
    file1.close()
#=============================================================================
#RSA2048
print "RSA_2048    encode:",
startTime = time.time()
en_RSA()
print time.time()-startTime

print "RSA_2048    decode:",
startTime = time.time()
de_RSA()
print time.time()-startTime
#=============================================================================
```

5.5 SHA512

```
# Cryptography_SHA512
#=============================================================================
import os
import time
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend
#=============================================================================
def start():
    file = open("random.txt","r")
    global text
    text = file.read()
    file.close()
#=============================================================================
def en_SHA_512(text):
    digest = hashes.Hash(hashes.SHA512(), backend=default_backend())
    digest.update(text)
    ans = digest.finalize().encode('hex')
#=============================================================================
start()
#SHA512
print "SHA_2_512   encode:",
startTime = time.time()
en_SHA_512(text)
print time.time()-startTime
#=============================================================================
```

6 比較

Table 1: 執行時間 (單位: 秒)

	PyCrypto	**Cryptography**
Encode AES256EBC	4.91898	1.47442
Decode AES256EBC	5.09097	1.43221
Encode AES256CBC	5.48372	2.39367
Decode AES256CBC	5.33132	1.44257
Encode AES256CTR	6.44351	1.45824
Decode AES256CTR	6.42914	1.46113
Encode RSA2048	1608.21340	79.65453
Decode RSA2048	24,809.77850	1673.72978
Encode SHA512	1.71591	0.87150

由上表可知:

1. Cryptography 的執行效率高於 PyCrypto
2. Hash function 速度比加密快很多
3. AES 除了在 Cryptography 的 CBC 中加解密速度不一致，其餘加解密時間差不多
4. 由於 PyCrypto 的 AES CTR 時間在 AES 中最久，因此推測 PyCrypto AES CTR 並未作分散式處理
5. RSA 運算十分費時
6. RSA 解密時間會比加密來的大許多