

國立嘉義大學 資訊安全與管理

期末報告

4-2 RSA-OAEP 跟 RSA-PSS 先加密再簽名

與先簽名後加密

年度：一百一十二學年度

組別：第五組

組員：1102924 李名智

1102932 林微訢

1102943 顏莉諭

1102966 邱翊鉸

1103478 林虹佑

第一章、Optimal Asymmetric Encryption Padding (OAEP)

1.1 OAEP 概念

在 1994 年，由 Bellare 和 Rogaway 提出一種能跟任何陷門排列方案一起使用的加密方法，稱為 OAEP。Bellare 和 Rogaway 證明此做法利用加入一些 redundancy 提供了語意上的安全，但隨後發現他是 weakly plaintext-aware，他只能抵禦非自適應的選擇密文攻擊，僅符合 IND-CCA 1，無法證明他符合 IND-CCA2。

OAEP 是一種費斯妥演算法，他透過添加隨機性元素將確定性方案（傳統 RSA）轉變成機率性方案，且確保對手無法反轉陷門單向排列，所以無法恢復明文的任一部份，也不會造成資訊洩漏。

Plaintext-awareness 是指確切的明文無法透過不同的密文推論出來。

1.2 OAEP 加密作法

$$x = (M \parallel 0^k) \oplus G(r) \text{ and } y = r \oplus H(x)$$

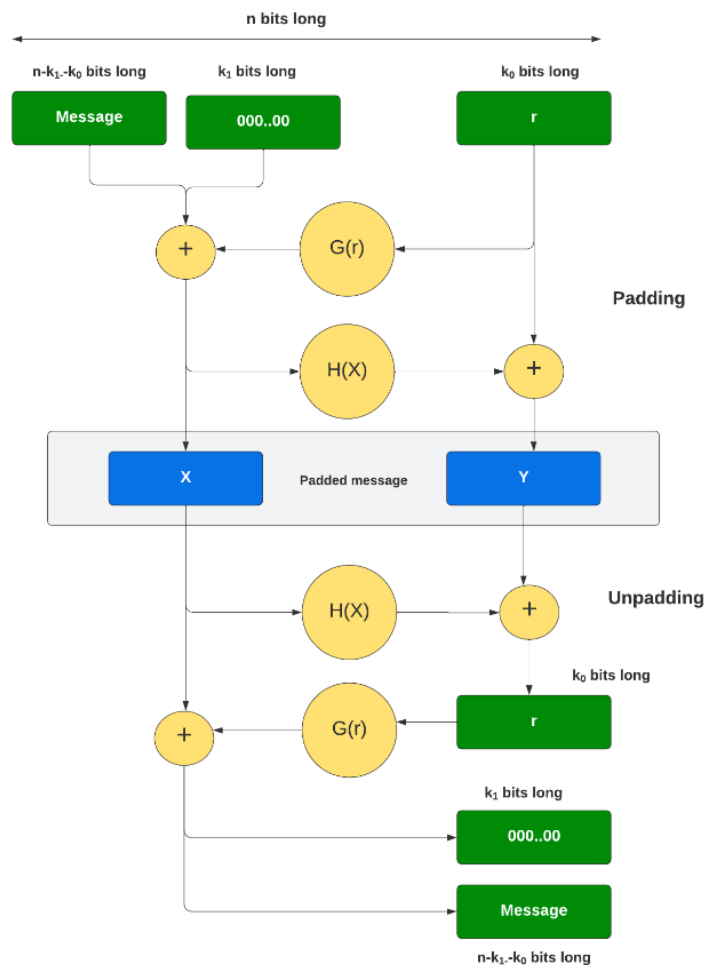


圖 1.1：OAEP 流程圖

1.2.1 將明文 M 用 OAEP 加密（圖 1.1）

- M 代表明文
- 0^{k_1} 代表 redundancy
- r 代表一組隨機字串
- G, H 代表兩個雜湊函數
- x 跟 y 代表結果
- 可表示成： $C=f(x, y)$

1.2.2 編碼步驟

1. 用 k_1 位長的 0 將 M 填充至 $n-k_0$ 位的長度
2. 隨機生成 k_0 位長的串 r
3. 用 G 將 k_0 位長的 r 擴展至 $n-k_0$ 位長
4. $x = M00\dots 0 \oplus G(r)$
5. H 將 $n-k_0$ 位長的 x 縮短至 k_0 位長
6. $y = r \oplus H(x)$
7. 輸出為 $x || y$ ，在圖中 x 為最左邊的塊， y 位最右邊的塊

1.3 OAEP 解密作法

$$(x, y) = g(c), \text{ and next } r = y \oplus H(x) \text{ and } M = x \oplus G(r)$$

- g : secret key

1.3.1 解碼步驟

1. 恢復隨機串 r 為 $y \oplus H(x)$
2. 恢復消息 $M00\dots 0$ 為 $x \oplus G(r)$

1.4 RSA-OAEP

雖然 OAEP 不符合 IND-CCA2，但如果它跟 RSA 合用，即能達到 IND-CCA2 等級。

1.4.1 加密步驟 (圖 1.2)

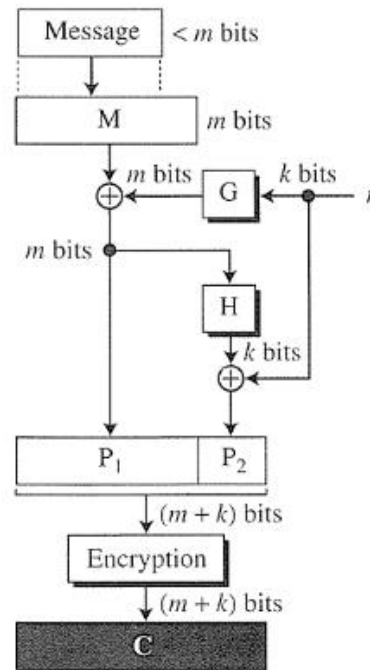


圖 1.2 RSA-OAEP 流程圖

$$\text{Enc} = \text{MessagePadded}^e \pmod{n} = (P_1 || P_2)^e \pmod{n}$$

先用 OAEP 將明文 pad，再加密此明文。

1.4.2 解密步驟

$$\text{MessagePadded} = \text{Enc}^d \pmod{n}$$

先用 RSA 解密出被 pad 過的密文，再將 pad 去除。

第二章、Probabilistic signature scheme (PSS)

PSS 是由 RSASA (RSA 數字簽名法) 演化而來，因為 RSASA 容易被選擇密文攻擊，所以加入 padding 讓明文的轉換多一些隨機性，讓它比較沒辦法被預測。

由於此簽章方案使用隨機數據，因此輸入的兩個簽名是不同的，並且都可以用來驗證原始資料，且 PSS 無法從簽章中恢復原來的簽章。

它的特性有：

1. 是隨機的，因此每次都會產生不同的簽名值。
2. 無法從 PSS 簽名中提取訊息摘要值，只能根據已知的訊息進行驗證。
3. PSS 具有安全性證明且比 PKCSV1_5 更健壯。

2.1 PSS 架構

PSS 在 RSASA 的基礎做法中進行一些改動，具體做法是在明文被 Hash 後，加入 PSS 填充的運算，之後再執行 RSA-PSS 做法 (圖 2.1)

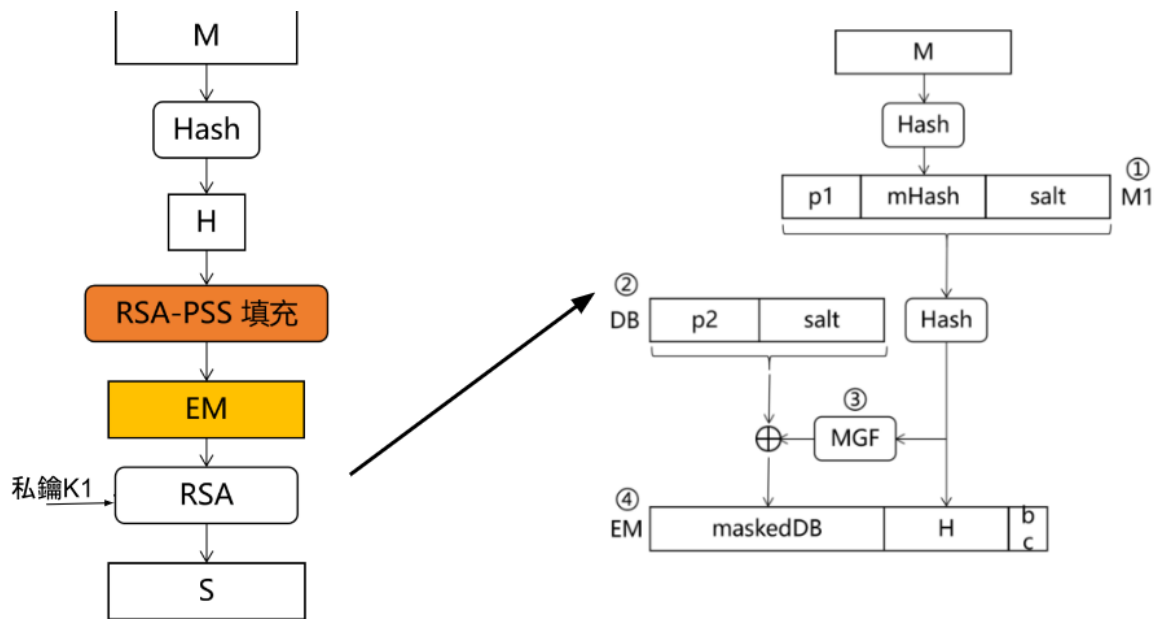


圖 2.1：先加密再簽署（左） RSA-PSS 流程圖（右）

預設的 function 值：

- `hashAlgorithm` : `sha1`
- `maskGenAlgorithm` : `mgf1SHA1` (the function MGF1 with SHA-1)
- `saltLength` : 20
- `trailerField` : `trailerFieldBC` (the byte 0xbc)

推薦 MGF 的 hash function 要跟 hash function 有一樣的 scheme，salt 的長度跟 hash function 長度一樣 (`hLen`)。

2.2 RSA-PSS 填充演算法，分為四步驟

1. M 轉換成 $M1$

- $M1 = p1 || mHash || salt$

- $p1$ 就是 8 位元組的 0
- $mHash = Hash(M)$ ， M 是待簽署的訊息。SHA-1 的輸出是 20 個位元組，所以 $mHash$ 的長度 $hLen = 20$;
- $salt$ (鹽值) 就是偽隨機數，它的長度 (記為 $sLen$) 一般等於 $hLen$ ，目前的選擇就是 $sLen = hLen$

2. 建構 DB (Data Block)

- DB (Data Block) 的建構方式為， $DB = p2 || salt$
 - $p2$ 的值等於若干個位元組的 0x00 後面跟著 1 位元組的 0x01
 - 這若干個位元組記為 $xLen$ ，則 $xLen$ 等於 $xLen = emLen - sLen - hLen - 2$ ， $emLen$ 是上圖中 EM 的長度
 - $salt$ 的值等於第 1 節所描述的 $salt$ 的值。特別強調，兩者必須相等，否則無法驗證數位簽章

3. MGF (Mask Generation Function, 掩碼生成函數)

- MGF 所對應的輸入與輸出分別是 $mask = MGF1(mgfSeed, maskLen, hash)$
 - $mgfSeed = Hash(M1)$ ， $Hash$ 函數選擇 SHA-1
 - $maskLen = emLen - hLen - 1$
 - $hash$ 函數選擇 SHA-1

4. 建置 EM (Encoded Message)

- $EM = maskedDB || H || bc$
 - $maskedDB = DB \oplus mask$
 - $H = Hash(M1)$

- $bc = 0xBC$ ， bc 的長度是 1 個位元組， H 的長度是 $hLen$ ， $maskedDB$ 的長度（記為 $mdbLen$ ）， $mdbLen = emLen - hLen - 1$
- $emLen$ 的長度就是 EM 的長度。由於接下來要對 EM 進行 RSA 加密計算，所以 EM 的長度符合 RSA 的要求即可

2.3 RSA-PSS 簽章

- $EM = \text{RSA-PSS}(M)$
- $S = \text{RSAEP}(EM)$

2.4 RSASA-PSS 的簽章驗證

1. 解密

- $EM = \text{RSADP}(S)$
- 拿到簽章 S 以後，執行 RSA 解密演算法，得到解密後的資訊 EM

2. 分割 EM

- 得到 EM 以後，接下來就是分割和驗證
- $maskedDB, H, bc = \text{Split}(EM)$
- 最右一個位元組是 bc ，從 bc 往左數 $hLen$ 個位元組是 H ，剩下的是 $maskedDB$
- 如果最右一個位元組不是 $0xBC$ ，則簽章驗證停止（該數位簽章是非法的）

3. 計算 $salt$

- 得到 H 以後，就可以計算 $mask$ ， $mask = \text{MGF1}(H, maskLen, hash)$ ，因為 $maskedDB = DB \oplus mask$ ，所以 $DB = maskedDB \oplus mask$

- 得到 DB 以後，就可以對其分割，
- $p2, salt = Split(DB)$
- salt 是 DB 最右邊的 sLen 個字節，剩下的是 p2。
- 如果 p2 的值不等於若干個位元組的 0x00 後面跟著 1 位元組的 0x01，那麼驗證停止（該數位簽章是非法的）。

4. 校驗 Hash

- 透過所接收到的 M，計算 Hash， $mHash = Hash(M)$
- 建構 M1， $M1 = p1 || mHash || salt$ ，salt 就是第 3 步所計算出來的 salt
- 計算 M1 的 Hash， $H1 = Hash(M1)$
- 比較 H1 與 H，如果兩者相等，則簽章驗證通過
- 如果不相等，則簽名非法
- 其中，H 就是 2. 步所得到的雜湊值

經過以上四步驟以後，就完成了 RSASA-PSS 的數位簽章驗證。

第三章、RSA-OAEP 跟 RSA-PSS 先簽名再加密 實作

3.1 實作程式碼與說明

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Signature import pss
from Crypto.Hash import SHA256
```

- 簽署

```
# 簽名
def sign_message (private_key, message) :
    h = SHA256.new (message)
    signature = pss.new (private_key).sign (h)
    return signature
# 驗證簽名
def verify_signature (public_key, message, signature) :
    h = SHA256.new (message)
    try :
        pss.new (public_key).verify (h, signature)
        return True
    except (ValueError, TypeError) :
        return False
# 生成 RSA 密鑰對
key = RSA.generate (2048)
# 待加密/簽名的訊息
message0="HE1L00000oo0000 WWWWWWoRLd"
message= message0.encode ( )
# 創建 RSA-OAEP 密碼器
cipher_rsa = PKCS1_OAEP.new (key)
# 先簽名再加密
signature = sign_message (key, message)
print (" 簽名後的 message :", signature)
print ("\n")
```

- 加密

Encrypt a message with PKCS#1 OAEP.

Parameters:: **message** (*bytes/bytearray/memoryview*) – The message to encrypt, also known as plaintext. It can be of variable length, but not longer than the RSA modulus (in bytes) minus 2, minus twice the hash output size. For instance, if you use RSA 2048 and SHA-256, the longest message you can encrypt is 190 byte long.

Returns:: The ciphertext, as large as the RSA modulus.

Return type:: bytes

Raises:: **ValueError** – if the message is too long.

圖 3.1：OAEP 加密最大明文長度計算公式

```
# OAEP 加密的最大明文長度計算 (公式來源：圖 3.1)
max_data_length = key.size_in_bytes() - 2*32-2
# 分塊加密 (因明文長度太長)
final_sign=message+signature
encrypted_message = b''
for i in range(0, len(final_sign), max_data_length):
    block = final_sign[i:i + max_data_length]
    encrypted_block = cipher_rsa.encrypt(block)
    encrypted_message += encrypted_block
print("加密後的 signature:", encrypted_message)
print("\n")
```

● 結果

```
# 分塊解密訊息
decrypted_message=b''
for i in range(0, len(encrypted_message), 256):
    block = encrypted_message[i:i + 256]
    decrypted_block = cipher_rsa.decrypt(block)
    decrypted_message += decrypted_block
# 印出明文 (全部的 decrypted_message=明文部分+簽名部分) 所以只取前面
print("解密後的 message:", decrypted_message[0:len(decrypted_message) -
key.size_in_bytes()].decode())
print("\n")
# 驗證簽名 (要分離 decrypted_message 分離成明文部分+簽名部分 再丟入驗證簽名的
Function)
verification = verify_signature(key.publickey(), decrypted_message[0:len
(decrypted_message)-key.size_in_bytes()], decrypted_message[len
(decrypted_message)-key.size_in_bytes():len(decrypted_message)])
```

```
print ("簽名是否吻合:", verification)
```

- 程式碼運行結果 (圖 3.2)

```
簽名後的 message: b'xE>\x8a\xa8\x12\xdd\x7f\x1e\xfcRt\x14Bv.\xcb\xee\xbb#\x88\xbc\xce7\xd7D\xab\x86,\xa3\x1ar\x97\x12\xc2\xf8Yo}\x9f3\xbb\xea\x0b\x07,\xe5\xe3X\x82\xd8Y\xcf|\xa3\x14u\xc6\xdb}\x1c(@\xeb))\x0f\x0b\x82\x87\xe5\xce\xa7\xa9\xa5D\xfc]\xfa\\ \xbf\x06v\x15\x15:\xb0\xe8*\xbd\x83\x0c8\x81\x9eu\xbeA7;7\xbe\x1b3\xeb\xbe\xebu0"\V\x98\x1d\xfe\xa7\xee\xca\x16\x95!L\x90=\xf9\x18\xd8`@89*\xbfv\x11\x04\xfdA\xc6\xfb\x00)\x85\x1c\xcf\x1c\xa1s\xc50\x1f\xd8\x0e#\xdc\xa5\x19` \xd4\xa d)\x18\x0c\xf2\xebw\xebC\xb3\x9b\xf2\x1c\xe9199\xc6\xd1w\xddGZ]*\x80=;\xcF\xeaD\xc2I-\\ \x13h\xfb\x08\xca\x14\xba0\xd4Y\x9de\x9c\x9f\xfe;[\xbdV\xc8\xca<=\xca'\xa8X34` \x87\x15\xd5\xd6\x11/5\x92\xe6r\xf9E\xfb\xcctH\x12q \xd9|n\xf6\x1e\xa3X\xb3\xd9\xcc\xa7\xdd\xab'

加密後的 signature: b''a\x1d\xcd$\xa80\x07\xa0\xadn\xbd\xfaP\xf6w0\xaf\\ \xcc\x90\x8aZ\xfc09o\x9b\x1eX\xfd\x03\x04\x8dv\x92R\xfaK\xe68y6&\xb9\xd5\xba\x02F\xd6\x18\xb9\xbeY\x06x1\xd9\xce6\xfb\x07\x03'\xa3\xe7\x9b\x8cx!D\x87 \xaf\xd4b1\x1f[\xa6\x93<+\x8f\xec\x13Vr\xe5\xa9\xfb1jHL\x91*\xf6\xd9\x00sfBw\xdb\xfb31\xd9\xad}\xb1\x9c\x00\xaeH\x17\x9e\x89\xd1\x1a:0w\xd0K\xfc@)g\x88v+\xf6\xde\x03pE\xff\x93\xd2X\x8c{\xc1\x89\n\x7f\x04\xbb\x08\xab\xfe<\x83x\xab\x03\x9d\x0c\xe4Z\xdc\xe4\xa9\x0fu\xea\x03'jD\xd8\x10\xa0\xd9<):\x95\x8b\x9f|\x95n\x9f\xfb7\xd7\x91\xe9T\xcd\x0b1\rM\xea\xa1\xe1\xe48\x1c\xcb\xeeL\xdf\x0b1\xdd\xbb\x88/\xeb\xafBT\xd6\x03\x02\xd9\x80\xbb~\x802\xaf\xdf\x9e]d\xfb sbPA\xff\x94\tpz\x18\xae\x0b\x8b\x1b\xa7\xa7V-\xdf\x0b\xcd\xe1e*\xe1MZ\x1e\x83sedWmg!ZjU\xd4w\x8a\x02}\x0e1\xa7\x8e\xaf\xa0}\xd9\xb9\xc4B\x04\xa1\xc4\xaf\xff\xcc;(\xe7T\xc7\x07i\xb56\xd0\x83^\x83tx\eck%\x8b\x17JF\xc7\xdf\xdbj\xcb.\xd8\x9c\x15[\x19\x1f\xca\xd1\x0f\x91n\xfb4\x86(\x1d\xad\x96\x99\xb6Ed[\xb1\xc5{\x17r'E\x93\x89PGw\xad\xfad\x03\x9a\xe1\xba\xafG\xc6\xd2Li\x9f5\xc0\xdd\x01\x0b5\x11\xda1P\x84\xc2\x0b8}\x8b\x9f^\xe2\xd5\x8e3n\x85s8F\xb8\xe9\xff\xc4/\xf8\xbbd4\x07\x08\xfb2j\x0c\x0f\x0b1=\x07Q5wA\xbc\x07\xec@i\x93\xa8\xfb1\x9e\xea\xd9\xd8<\xc5\x86H\xfe~@\x9b\x81\x0bchz\x0b0\x0fu\xda1wX%\xbfb3\x18\xbc\x9d\x09\xe5\xaa\x0b1?m>\x03\xe7\xec\x1dRRz\xba\x02\x9d\x92\xeaJz!\x9c)7\x82\x1e\x0e\x00\xd1u\x855X6\xdb\xfa\x84d\x04\xe3\x96\xb2o\x09!@\xaa\x95s:5\xf9\xc4\xe5y\xec\xea\x0e?\xa7\x15"

解密後的 message: HELLO0000o00000 WWWWwWwORLD

簽名是否吻合: True
```

圖 3.2：先簽名再加密實作結果圖

第四章、RSA-OAEP 跟 RSA-PSS 先加密再簽名 實作

4.1 實作程式碼與說明

- 加密

```
# 生成 RSA 密鑰對
key = RSA.generate (2048)
# 待加密/簽名的訊息
message0="HE1L00000oo0000 WWWWWWWoRLd"
message= message0.encode ( )
# 創建 RSA-OAEP 密碼器
cipher_rsa = PKCS1_OAEP.new (key)
# 先加密再簽名
encrypted_message = cipher_rsa.encrypt (message)
print ("加密後的 message :", encrypted_message)
print ("\n")
```

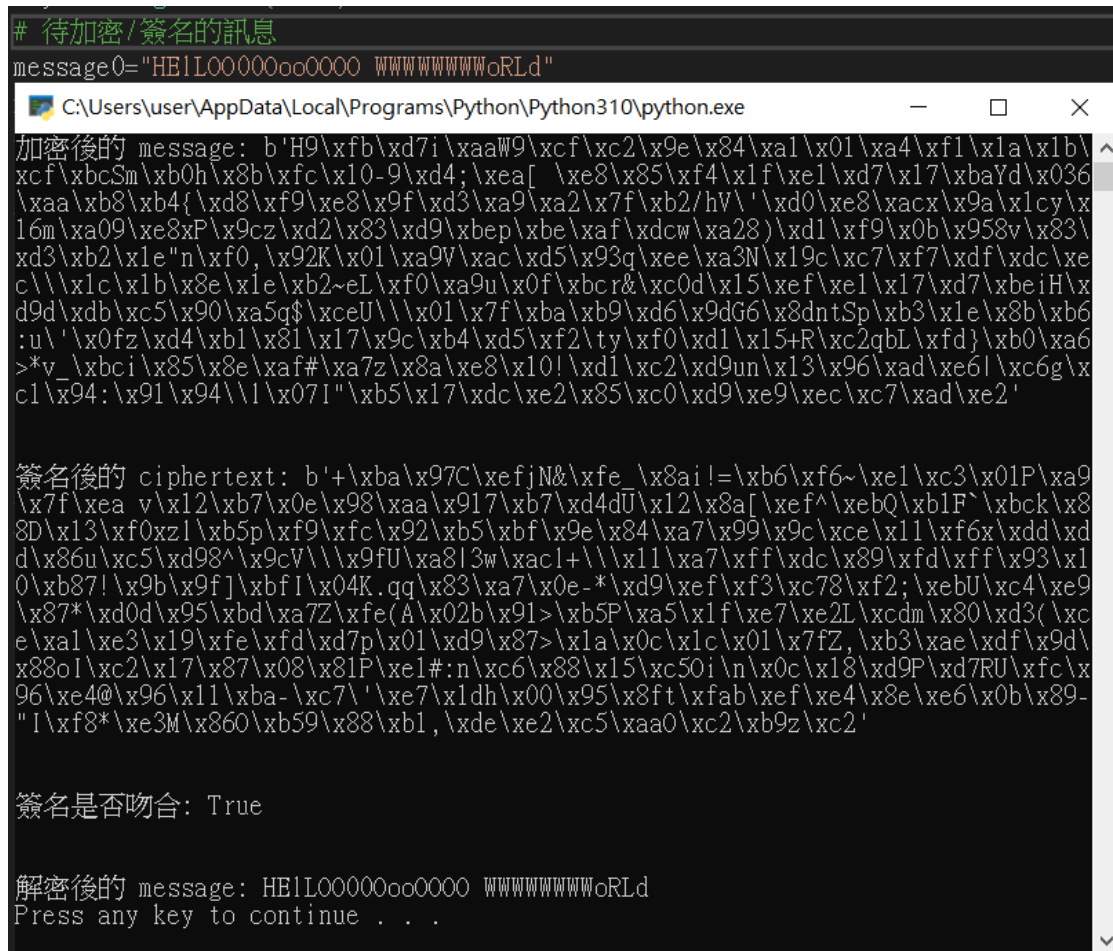
- 簽署

```
# 簽名
def sign_message (private_key, message) :
    h = SHA256.new (message)
    signature = pss.new (private_key).sign (h)
    return signature
# 驗證簽名
def verify_signature (public_key, message, signature) :
    h = SHA256.new (message)
    try :
        pss.new (public_key).verify (h, signature)
        return True
    except (ValueError, TypeError) :
        return False
signature = sign_message (key, encrypted_message)
print ("簽名後的 ciphertext :", signature)
print ("\n")
# 驗證簽名
verification = verify_signature (key.publickey ( ), encrypted_message,
signature)
```

- 結果

```
print ("簽名是否吻合:", verification)
print ("\n")
# 解密訊息
decrypted_message = cipher_rsa.decrypt (encrypted_message)
print ("解密後的 message:", decrypted_message.decode ())
```

- 程式碼運行結果 (圖 4.1)



```
# 待加密/簽名的訊息
message0="HELLO0000oo0000 WWWWWWoRLd"

加密後的 message: b'H9\xfb\xd7i\xaaW9\xcf\xc2\x9e\x84\xal\x01\xa4\xfl\xla\xlb\xcf\xbcSm\xb0h\x8b\xfc\x10-9\xd4;\xea[ \xe8\x85\xf4\x1f\xel\xd7\xl7\xbaYd\x036\xaa\x8b\x84\x8d\x9f\x8e\x9f\x8d3\xa9\xa2\x7f\xb2/hV\''\xd0\xe8\xac\x9a\xlcylx16m\xa09\xe8xP\x9cz\xd2\x83\xd9\xbeP\xbe\xaf\xdcw\xa28)\xd1\xf9\x0b\x958v\x83\xd3\xb2\xle"n\xfo,\x92K\x01\xa9V\xac\xd5\x93g\xee\xa3N\x19c\x7\x7f\xdf\xdc\xec\\\xlc\xlb\x8e\xle\x2~eL\xfo\xa9u\x0f\xbc&\xc0d\x15\xef\xel\xl7\xd7\xbeiH\x9d9\xdb\xc5\x90\xa5q$\xceU\\\x01\x7f\xba\x8b9\xd6\x9dG6\x8dntSp\xb3\xle\x8b\x86:u\''\x0fz\xd4\xbl\x81\xl7\x9c\x84\xd5\xf2\ty\xfo\xdl\xl5+R\xc2qbL\xfd}\xb0\xa6>*v_\xbci\x85\x8e\xaf#\xa7z\x8a\xe8\xl0!\xd1\xc2\xd9un\xl3\x96\xad\xe6l\xc6g\xcl\x94:\x9l\x94\\\l\x07I"\xb5\xl7\xdc\xe2\x85\xc0\xd9\xe9\xec\x7\xad\xe2'

簽名後的 ciphertext: b'+\xba\x97C\xefjN&\xfe \x8ai!=\xb6\xf6~\xel\xc3\x01P\xa9\x7f\xea v\xl2\xb7\x0e\x98\xaa\x9l7\xb7\xd4dU\xl2\x8a[\xef^\xebQ\xblF'\xbck\x88D\xl3\xfoxzI\x85p\x9f\xfc\x92\xb5\xbf\x9e\x84\xa7\x99\x9c\xce\xl1\x6f\xxdd\xd\x86u\xc5\xd98^\x9cV\\\x9fU\xa8l3w\xacI+\\\x11\xa7\xff\xdc\x89\xfd\xff\x93\xl0\xb87!\x9b\x9fj\xbfI\x04K.qq\x83\xa7\x0e-.*\xd9\xef\x83\x78\x82;\xebU\xc4\xe9\x87*\xd0d\x95\xbd\xa7Z\xfe(A\x02b\x91>\xb5P\xa5\xl1f\xe7\xe2L\xcdm\x80\x83(\xc\xel\xal\x83\xl9\xfe\xfd\x7p\x01\xd9\x87>\xla\x0c\xlc\x01\x7fZ,\xb3\xae\xdf\x9d\x88oI\xc2\xl7\x87\x08\x81P\xel#:n\xc6\x88\xl5\x50i\n\x0c\xl8\xd9P\xd7RU\xfc\x96\xe4@\x96\xl1\xba-\xc7\''\xe7\xldh\x00\x95\x8ft\xfab\xef\xe4\x8e\xe6\x0b\x89-"I\x8*\xe3M\x860\xb59\x88\xbl,\xde\xe2\xc5\xaa0\xc2\xb9z\xc2'

簽名是否吻合: True

解密後的 message: HELLO0000oo0000 WWWWWWoRLd
Press any key to continue . . .
```

圖 4.1：先加密再簽名實作結果圖

第五章、比較差異

先簽署再加密有進行分塊加密（因明文簽署後長度太長），另一個則沒有。

選擇兩種流程的其中一種通常取決於應用場景和需求。若需要確保訊息在傳輸過程中不被竄改且來源可信，則選擇先簽名再加密的方式。若主要關注訊息的機密性，並且信任傳輸管道安全，則可以先加密再簽名。

傳統上，數位簽名是基於明文訊息生成的，因為簽名是用來驗證訊息完整性和來源的真實性。對密文進行簽名可能會導致一些安全風險，例如可能洩露加密密鑰或者導致安全性降低。

為了確保安全性和避免潛在的風險，傳統上仍然建議在加密之前對明文進行簽名，而不是對密文進行簽名。

References

1. [OAEP](#) : Optimal Asymmetric Encryption Padding
2. [RSA - OAEP](#) is Secure under the RSA Assumption
3. [RSA-OAEP](#).pic
4. [CryptoSys](#) PKI Pro Manual
5. [Formal](#) Proof for the Correctness of RSA-PSS ?
6. [Evaluation](#) of Security Level of Cryptography : RSA-OAEP, RSA-PSS, RSA Signature
7. [OAEP](#) Reconsidered*
8. [Optimal](#) asymmetric encryption padding
9. [Everything](#) you need to know about RSASSA-PSS
10. [RSA](#) 簽名的 PSS 模式
11. [RSA-PSS](#) 演算法的原理和應用
12. [RSA-PSS](#) 數位簽章演算法
13. [PKCS](#) #1 : RSA Cryptography Specifications/Version 2.0
14. [Optimal](#) asymmetric encryption padding
15. [Updates](#) for RSAES-OAEP and RSASSA-PSS Algorithm Parameters
16. [RSA - OAEP is](#) Secure under the RSA Assumption?
17. [Optimal](#) Asymmetric Encryption—How to Encrypt with RSA
18. [Python PKCS#1 OAEP \(RSA\)](#)

工作分配

- 整理資料及書面報告：1102924 李名智、1102932 林微訢
- 做簡報：1102943 顏莉諭
- 程式實作：1102966 邱翊鉸
- 上臺報告：1103478 林虹佑