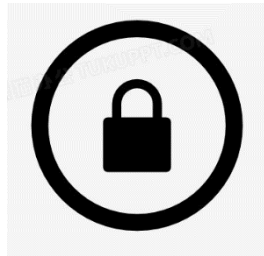


AES

期中報告



小組成員

組長 賴威成 407410058

組員 劉 杰 407410389

組員 周子善 406410950

組員 謝德興 207410381

目錄

一、摘要

二、背景

2-1 歷史

2-2 簡介

2-3 對稱加密與非對稱加密

2-4 優劣評比

三、規格

3-1 AES 參數簡介

3-2 AES 參數規格及關係

四、加密流程

4-1 AES Algorithm function

1. AES encrypt algorithm function
2. AES decrypt algorithm function
3. 共同部分
4. 線性轉換(linear transformation)/線性映射(linear mapping)/線性(linear)

4-2 AES 流程圖

4-3 AES 內部程序

1. Substitute bytes
2. Inverse substitute bytes

3. Shift rows
4. Inverse shift rows
5. Mix columns
6. Inverse Mix columns
7. Add round key

五、32-bit 查表加速版

5-1 簡介

5-2 加密過程

5-3 解密過程

六、程式碼

6-1 SubBytes and Inverse SubBytes

6-2 ShiftRows and Inverse ShiftRows

6-3 MixColumns and Inverse MixColumns

6-4 AddRoundKeys

七、資料參考來源

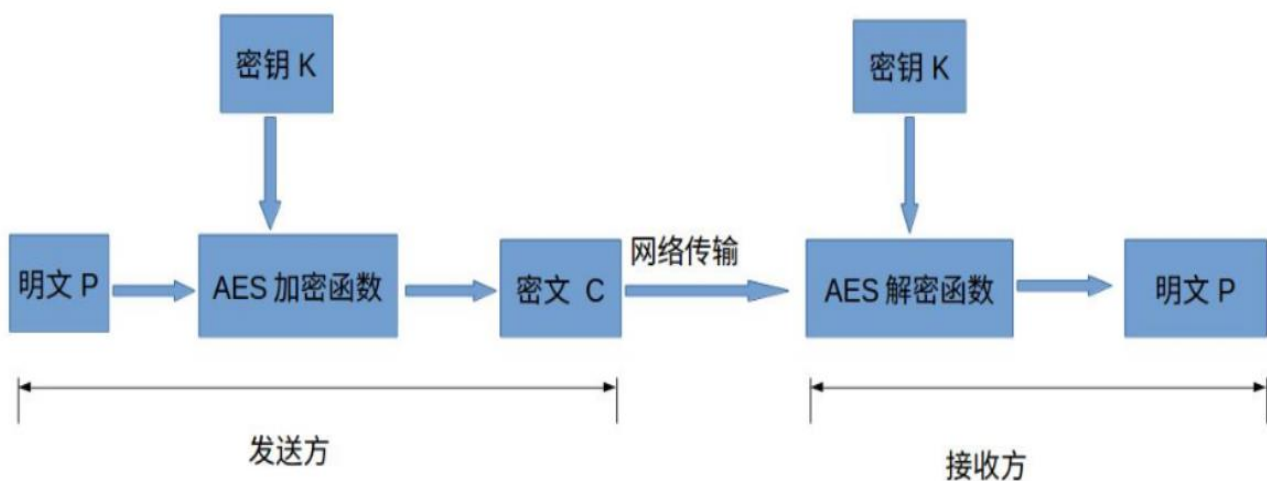
壹、摘要

甚麼是 AES 對稱加密算法？

假設有一個發送方在向接收方發送消息，如果沒有任何加密算法，接收方接收的是一個明文訊息:「我是 Paul」，如果訊息被中間人截獲到，即使中間人無法篡改訊息，也可以窺探到訊息內容，從而暴露了通信雙方的私密。因此我們不再直接傳送明文，而改用對稱加密的方式傳輸密文，發送方利用密鑰 123456，加密明文「我是 Paul」，加密結果為 TNRvx.....。

發送方把加密後的內容 TNRvx=.....傳輸給接收方。接收方收到密文 TNRvx...，利用密鑰 123456 還原為明文「我是 Paul」。

摘要算法是不可逆的，它的主要作用是對信息一致性和完整性的校驗。而對稱加密算法是可逆的，它的主要作用是保證私密信息不被泄露。



貳、背景

2-1 歷史

在 20 世紀末差分密碼分析及線性密碼分析的誕生，70 年代美國人開創的 DES (資料加密標準) 漸漸退出主流。

在已有的加密算法中，DES 的密鑰太短，3DES(三重資料加密標準)太慢,IDEA 受專利保護且速度不快，所以美國國家標準技術研究所在 1997 年宣布希望有一個安全性能更高的加密算法 (AES) 用以取代 DES，同時要求每一種候選算法應當支持 128、192 和 256bit 的密鑰長度。

	DES	3DES	AES
資料區塊	64 位元	64 位元	128 位元
金鑰長度	56 位元	168 位元	128/192/256 位元
重複運算次數	16 次	48 次	10/12/14 次

2-2 簡介

AES 得到了全世界很多密碼工作者的響應，先后有很多人提交了自己的設計方案。最終經過嚴格的性能測評，由 Rijndael 算法獲勝，因此 AES 算法也叫 Rijndael，是一個對稱分組密碼算法，而對稱加密算法應用的相對較早期，技術更為成熟。對稱密碼學簡單來說就是加密和解密使用同一把密鑰。

加密密鑰能夠從解密密鑰中推算出來，同時解密密鑰也可以從加密密鑰中推算出來。在通信過程中，密鑰的保密性對於通信的安全性起到至關重要的作用，從上世紀一直沿用至今的 DES 算法也是對稱加密算法。DES 算法是把 64 位的明文輸出塊變為數據長度為 64 位的密文輸出塊，其中 8 位為奇偶校驗位，另外的 56 位作為密碼的長度。首先把輸入的 64 位數據塊按位重新組合，並把輸出分為 L0，R0 兩部分，每部分各長 32 位，並進行前后置換，最終由 L0 輸出左 3 位，R0 輸出右 32 位，迭代 16 次后得到 L16 和 R16，將此作為輸入，進行與初始置換相反的逆置換，即得到密文的輸入。



2-3 對稱加密與非對稱加密

對稱加密法 v.s. 非對稱加密法

	對稱加密	非對稱加密
加解密的金鑰是否相同	相同	不相同
金鑰保管問題	如果與 n 人交換訊息 需保管 n 把加解密鑰匙	不管和多少人交換訊息 只需保管自己私密鑰匙
加解密速度	快	慢
金鑰能否公開	無法公開	公鑰可公開，私鑰無法公開
應用方面	加密長度較長的資料 例如:email	用於加密長度較短的資料例 如：數位簽章

對稱加密是加密數據的兩種主要方法之一，而另一種就是所謂的非對稱加密，也稱為公鑰加密。兩種方法的區別在於非對稱加密系統使用兩個不同的密鑰進行加解密，不同對稱加密中所使用的相同密鑰。在非對稱加密中，其中一個密鑰用於共享（公鑰），另一個密鑰必須保密（私鑰）。

非對稱加密算法使用兩個不同的密鑰也是其與對稱密鑰產生功能差異的原因。非對稱算法比對稱算法更複雜，運算速度更慢。因為非對稱加密中使用的公鑰和私鑰在某種程度上是算數相關的，所以密鑰本身也必須足夠長，以此達到與對稱加密算法（使用較短加密密鑰）相同的安全級別。

2-4 優劣評比

對稱加密算法可以提供相對較高的安全級別，同時支持快速加密和解密消息。對稱加密系統的便捷性在邏輯上也是一種優勢，因為它比非對稱消耗更少資源。對稱加密提供的安全性可以通過增加密鑰長度來實現。隨著對稱密鑰長度的增加，暴力攻擊破解加密的難度也呈指數增長。

雖然對稱加密能夠提供諸多優勢，但是也存在一個嚴重的缺點。

用於加密和解密數據的密鑰是相同的。當這些密鑰在不安全的網絡連接中共享時，它們很容易被惡意的第三方攔截。如果未經授權的用戶獲得對特定密鑰的訪問權限，則使用該密鑰加密數據安全性都會受到破壞。為了解決這個問題，許多 Web 協議使用對稱和非對稱加密的組合來建立安全連接。這種混合協議最常見的例子是傳輸層安全加密協議（TLS），該協議被用於保護現代互聯網上大部分的網絡連接。還應注意，由於實施的不當，所有類型的計算機加密都會受到漏洞影響。雖然足夠長的密鑰在數學上可使暴力攻擊失效，但程序員的錯誤配置，常常也會產生漏洞，為網絡攻擊者開闢新的道路。

由於對稱加密的運算速度相對較快，易於使用 and 安全性較高，對稱加密被廣泛應用於互聯網流量防護和雲服務器上的數據保護等各種應用中。而為了解決傳輸密鑰的安全問題，它經常與非對稱加密配合使用，但對稱加密方案仍然是現代計算機安全的關鍵組成部分。

參、規格

3-1 AES 參數簡介

1.分組長度(Nb)：明文的資訊，英文全名為 Plaintext Block Size 以

32bits(1word)為一區塊單位。

2.金鑰長度(Nk)：金鑰長度，英文全名為 Key Size 以 32bits (1word)為一區

塊單位。

3.加密輪數(Nr)：加解密的運算回合次數，英文全名為 Number of Rounds。

3-2 AES 參數規格及關係

AES	金鑰長度 (Nk)			分組長度 (Nb)			加密輪數 (Nr)
單位	word	byte	bit	word	byte	bit	回合
AES-128	4	16	128	4	16	128	10
AES-192	6	24	192	4	16	128	12
AES-256	8	32	256	4	16	128	14

AES 的明文部分長度都是以 128bits(4 * 32bits)作為規格，而 AES 則是依照金鑰的長度作為分類，有 128bits、192bits 和 256bits 可供選擇。由上圖可知，金鑰長度越長加密的回合數也跟著增加，由此可知關係式為：

$$Nr = Nk + 6$$

肆、加密流程

4-1 AES Algorithm function

AES 演算法先將 128bits 分組區塊以每 8bits，也就是 1byte 切成 16 個小區塊並以 column major 來排列成矩陣形式，再經由下面函式進行加密解密。

1.AES encrypt algorithm function

SubBytes：透過 S-Box 將矩陣每個元素(1byte)經由查表置換掉。

ShiftRows：對每排 Row(4 bytes)進行左旋 circular shift。

MixColumns：對每列 Column(4bytes)進行 linear transform。

2.AES decrypt algorithm function

InvSubBytes：透過 Inverse S-Box 進行查表將矩陣中每個元素置換。

InvShiftRows：與上面 ShiftRows 很像，但是是進行右旋 circular shift。

InvMixColumns：一樣對每列 Column(4bytes)進行 linear transform，但

InvMixColumns 的 key 和 MixColumns 的 key 相乘會是單位矩陣。

3.共同部分

Add Round Key：明文區塊所形成的矩陣與每回合的金鑰做 Exclusive OR。

4.線性轉換(linear transformation)/線性映射(linear mapping)/線性(linear)

(1)定義

假設 $V = F$ 且 $V' = F$, $T: V \rightarrow V'$ 滿足:

$$a. \forall u, v \in V, T(u + v) = T(u) + T(v)$$

$$b. \forall \alpha \in F, v \in V, T(\alpha v) = \alpha T(v)$$

(2)矩陣轉換(matrix transformation)

$$T: \mathbb{R}^4 \rightarrow \mathbb{R}^4$$

$$T(x_1, x_2, x_3, x_4) = (2x_1 + 3x_2 + x_3 + x_4, x_1 + 2x_2 + 3x_3 + x_4, x_1 + x_2 + 2x_3 + 3x_4, 3x_1 + x_2 + x_3 + 2x_4)$$

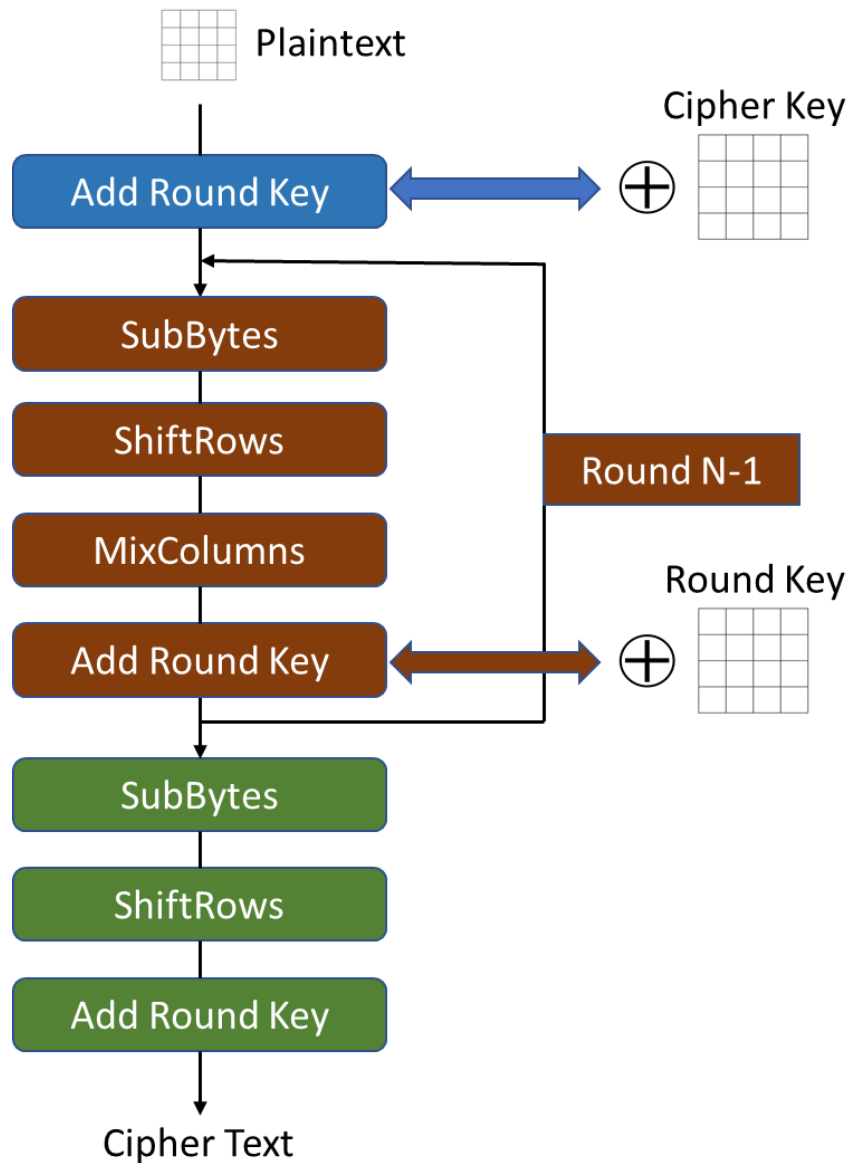
$$T \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 2x_1 + 3x_2 + x_3 + x_4 \\ x_1 + 2x_2 + 3x_3 + x_4 \\ x_1 + x_2 + 2x_3 + 3x_4 \\ 3x_1 + x_2 + x_3 + 2x_4 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

將 T 寫成矩陣型式為 $T(x) = Ax$, 其中 A 扮演線性轉換的角色。在 MixColumns 或

Inverse MixColumns 加密演算法中 , A 扮演 key 的角色 , 把輸入的矩陣透過矩陣相

乘加密成另一個矩陣 , 也是利用線性代數中矩陣轉換的特性來加密明文。

4-2 流程圖(自行製作)



由上圖可知，AES 加密先經由 Add Round Key 運算，再進行(N-1)回合 SubBytes、ShiftRows、MixColumns、Add Round Key，而最後一回合沒有 MixColumns，解密大同小異，只是內部稍有變更。

4-3 內部程序

1. Substitute bytes

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1x	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2x	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3x	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4x	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5x	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6x	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7x	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8x	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9x	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
ax	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
bx	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
cx	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
dx	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
ex	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
fx	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

圖 1 S-box

1E	20	39	BB
68	CB	14	CF
23	A1	30	CE
CA	10	AA	FA

表一

→

72	B7	12	EA
45	1F	FA	8A
26	32	04	8B
74	CA	AC	2D

表二


以表一第一列第一行作為範例，1E 可以經由查表 1x 和 Ex 找到 72 進行置換，以此類推，其他元素也是經由查表置換。

2. Inverse substitute bytes

		right (low-order) nibble															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
left (high-order) nibble	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

圖 2 Inverse S-box

72	B7	12	EA
45	1F	FA	8A
26	32	04	8B
74	CA	AC	2D



1E	20	39	BB
68	CB	14	CF
23	A1	30	CE
CA	10	AA	FA

同樣地，Inverse 也是經由查表將每個元素置換掉，但是唯一不太一樣的是表是經過 Inverse S-box 進行轉換。

3. Shift rows

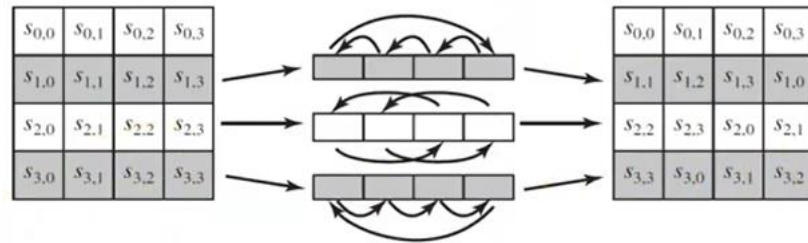


圖 3

1E	20	39	BB
68	CB	14	CF
23	A1	30	CE
CA	10	AA	FA

表一

→

1E	20	39	BB
CB	14	CF	68
30	CE	23	A1
FA	10	AA	CA

表二

Shift rows 是對每列進行左旋轉換，由表一來看，第一列(Row)不進行轉換，第二列往左移一行，第三列往左移兩行，第四列往左移三行就可以得到表二的矩陣。

4. Inverse shift rows

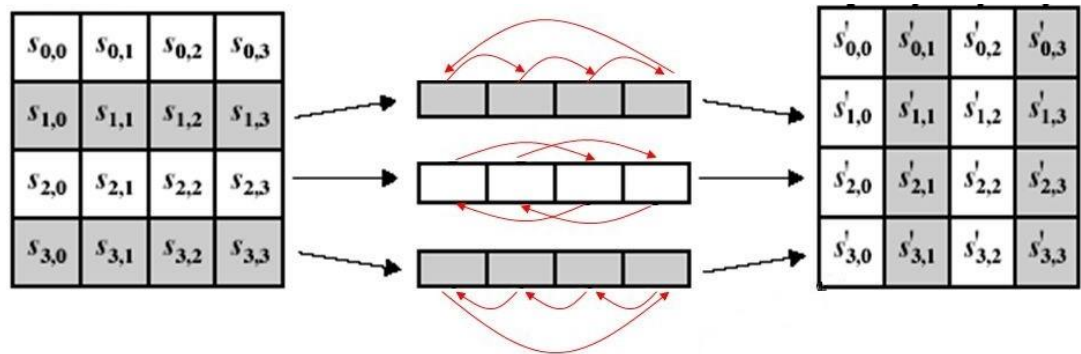
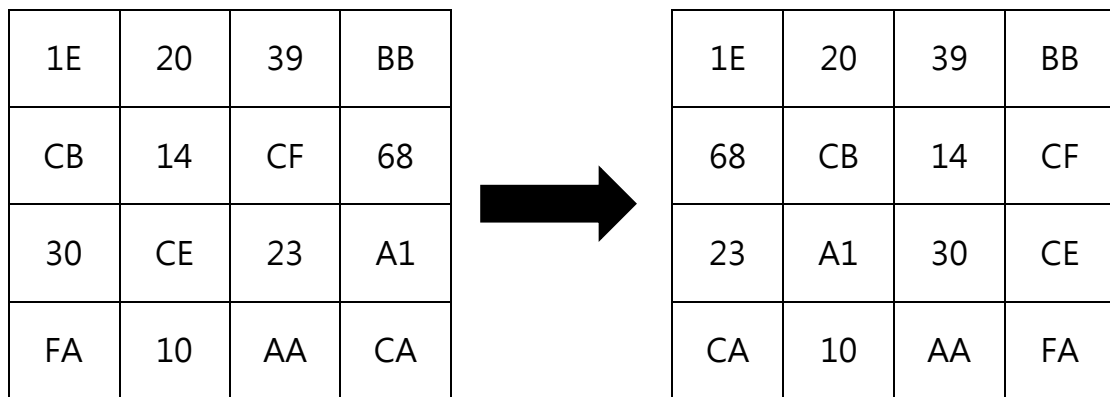


圖 4



同樣地，Inverse shift rows 與上面 shift rows 操作很像，只是旋轉的方向是右旋。

5. Mix columns

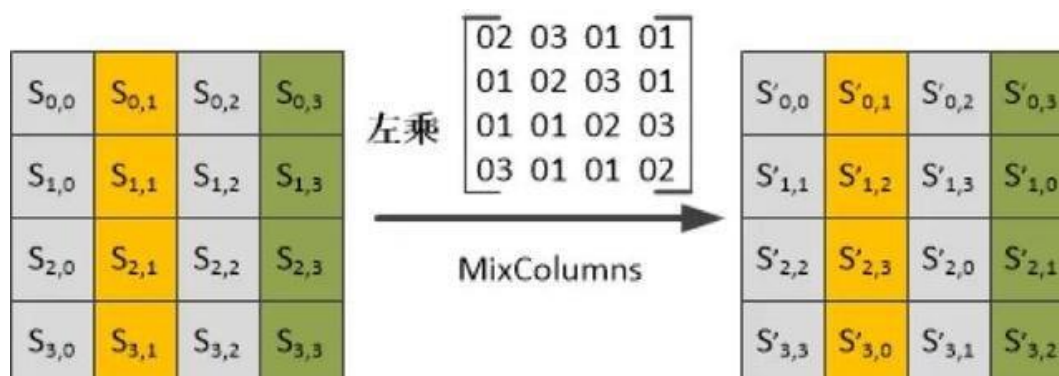


圖 5

$$s'_{0,j} = (2 \cdot s_{0,j}) \oplus (3 \cdot s_{1,j}) \oplus s_{2,j} \oplus s_{3,j}$$

$$s'_{1,j} = s_{0,j} \oplus (2 \cdot s_{1,j}) \oplus (3 \cdot s_{2,j}) \oplus s_{3,j}$$

$$s'_{2,j} = s_{0,j} \oplus s_{1,j} \oplus (2 \cdot s_{2,j}) \oplus (3 \cdot s_{3,j})$$

$$s'_{3,j} = (3 \cdot s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 \cdot s_{3,j})$$

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

$$02 \bullet C9 = 02 \bullet 11001001_B = 10010010_B \oplus 00011011_B = 10001001_B$$

$$03 \bullet 6E = (01 \oplus 02) \bullet 6E = 01101110_B \oplus 11011100_B = 10110010_B$$

$$01 \bullet 46 = 01000110_B$$

$$01 \bullet A6 = 10100110_B$$

圖 6

講解 Mix Columns 加密法之前，我先說明圖 6，一個二進位數乘以 1 時，得出來的值跟原本的一樣，當一個二進位數乘以 2 時，如果得出來的值沒有溢位的話，所有位元往左移一個 bit 就是答案。相反地，如果得出來的值有溢位的話，除了所有位元往左移一個 bit 之外，得出來的值還要 $00011011_{(2)} = 1B_{(16)}$ 做 XOR 運算，才是真正得出來的值。當一個二進位數乘以 3 時，首先把 3 拆成 1 xor 2，再利用分配律性質把答案求出來， $(01 \bullet \underline{0110} \underline{1110}) \text{ xor } (02 \bullet \underline{0110} \underline{1110}) = \underline{0110} \underline{1110} \text{ xor } \underline{1101} \underline{1100} = \underline{1011} \underline{0010}$ 。

Mix Columns 加密演算法，簡單來講，就是給你一塊 4×4 矩陣的明文，透過 4×4 矩陣的 key 加密成 4×4 的密文。關於詳細的內容，首先會拿到 4×4 的明文和 4×4 的 key，利用矩陣相乘的特性和二進位數乘法概念去對明文加密成密文，而 key 不管是 Mix Columns 還是 Inverse mix columns，都一律乘在明文的左邊。

最後，我想補充的一點是，無論是 Mix Columns 還是 Inverse mix columns 的 key 都有一個特性就是，第一行(column)所有元素不動，到了第二行(Column)的時候，第一行的所有元素每個都往下移動一格，到了第三行(Column)的時候，所有元素又往下移動一格。

6. Inverse mix columns

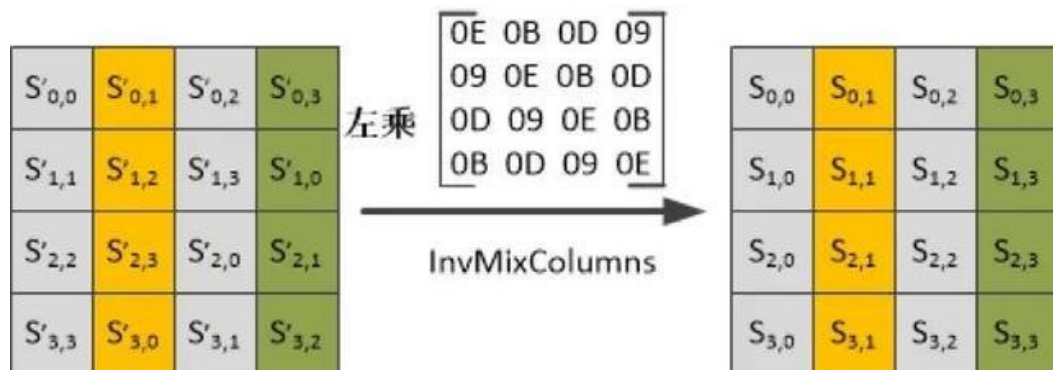


圖 7

$$\begin{bmatrix} \overline{0E} & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} \overline{02} & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} \overline{01} & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \\ 00 & 00 & 00 & 01 \end{bmatrix}$$

圖 8

關於 Inverse mix columns 加密演算法，其加密性質跟 Mix columns 類似，我覺得兩者的差別差在，一個是由明文加密(plaintext)成密文(cipher)，另一個 是由密文(cipher)解密成明文(plaintext)，另外，Mix columns 的 key 和 Inverse mix columns 的 key 相乘會形成單位矩陣。

7. Add round key

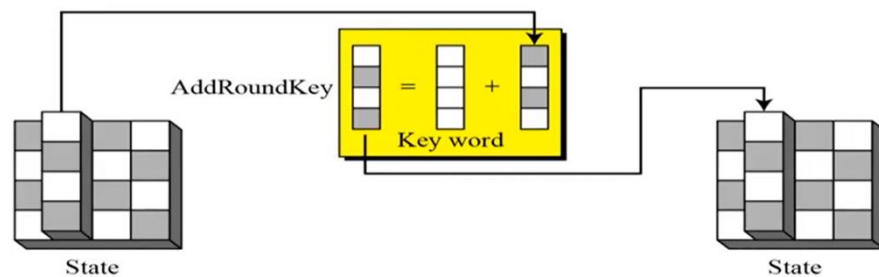


圖 9

Add round key 主要的功能是将原状态矩阵每行和所对应 keys 的每行做 Exclusive OR，而 Exclusive 的功能如下图 10，就是要被处理的矩阵(A)与 KEY 值(B)的值一样是输出就是 0，相反时则输出 1，而且在做一次 Exclusive OR 可以将其值转换回来，所以他是一个 one-one 且 onto 的函数，同个 Key 可以用在加密，也可以用在解密。

輸入		輸出
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

圖 10

伍、32-bit 查表加速版

5-1.簡介

使用 32 或更多位元定址的系統，可以事先對所有可能的輸入建立對應表，利用查表來實作 SubBytes，ShiftRows 和 MixColumns 步驟以達到加速的效果。這麼作需要產生 **4 個表**，每個表都有 **256 個格子**，一個格子記載 **32 位元** 的輸出；約佔去 4KB (4096 位元組) 記憶體空間，即每個表佔去 1KB 的記憶體空間。

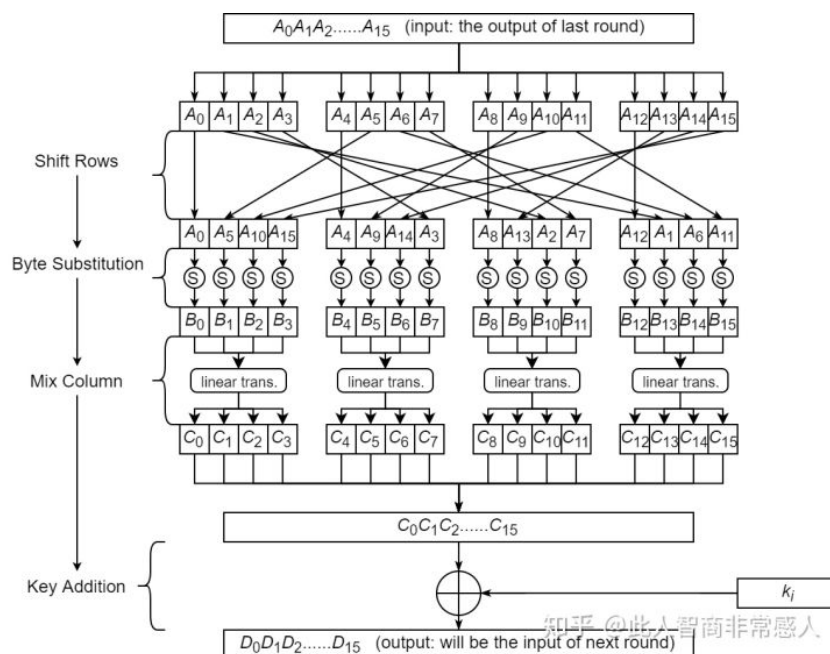
查表法的核心思想是將 SubBytes 層、ShiftRows 層和 MixColumn 層融合為查找表：每個表的大小是 32 bits(4bytes)乘以 256 項，一般稱為 T 盒(T- Box) 或 T 表。加密過程 4 個表(Te)，解密過程 4 個表(Td)，共 8 個。如此一來，**在每個加密迴圈中，只需要查 16 次表，作 12 次 32 位元的 XOR 運算，以及 AddRoundKey 步驟中 4 次 32 位元 XOR 運算。**

對於查表法實現，就是要將每一輪中的前三層操作(字節代換層、ShiftRows 層和 MixColumn 層)合併為查找表。

然而，實際實作中應避免使用這樣的對應表，否則可能因為產生快取命中與否的差別而使旁道攻擊成為可能。

5-2 加密過程

SubBytes 層和 ShiftRows 層是可以交換順序的——即先做 SubBytes 層再做 ShiftRows 層和先做 ShiftRows 層再做 SubBytes 層的結果是完全一樣的。如果將它們交換，那麼 ShiftRows 層實際上就是按照特定的順序去讀取這一輪操作的輸入數據。SubBytes 層和 MixColumn 層融合為查找表。密鑰加法層的本質是按位 XOR，這個操作很簡單，直接附在後面就行。



前面對 MixColumn 層給出了如下矩陣公式：

$$\begin{pmatrix} C_0 & C_4 & C_8 & C_{12} \\ C_1 & C_5 & C_9 & C_{13} \\ C_2 & C_6 & C_{10} & C_{14} \\ C_3 & C_7 & C_{11} & C_{15} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_1 & B_5 & B_9 & B_{13} \\ B_2 & B_6 & B_{10} & B_{14} \\ B_3 & B_7 & B_{11} & B_{15} \end{pmatrix}$$

聯繫矩陣乘法的定義，將其按列向量拆開，對於每一列，都有：(這裡只寫出了

第一列)

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix} \\ = \begin{pmatrix} 02 \\ 01 \\ 01 \\ 03 \end{pmatrix} B_0 + \begin{pmatrix} 03 \\ 02 \\ 01 \\ 01 \end{pmatrix} B_1 + \begin{pmatrix} 01 \\ 03 \\ 02 \\ 01 \end{pmatrix} B_2 + \begin{pmatrix} 01 \\ 01 \\ 03 \\ 02 \end{pmatrix} B_3$$

故有下式：(其中 W_{k0} 是本輪子密鑰 k_i 中對應的那 4 字節；這裡只寫出了第一

列)

$$\begin{pmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \end{pmatrix} = \begin{pmatrix} 02 \\ 01 \\ 01 \\ 03 \end{pmatrix} S(A_0) + \begin{pmatrix} 03 \\ 02 \\ 01 \\ 01 \end{pmatrix} S(A_5) + \begin{pmatrix} 01 \\ 03 \\ 02 \\ 01 \end{pmatrix} S(A_{10}) + \begin{pmatrix} 01 \\ 01 \\ 03 \\ 02 \end{pmatrix} S(A_{15}) + W_{k0}$$

我們將 T 盒(T-Box)作如下定義，即可基於前面的知識，通過編寫程序算法計算

出四個 Te 表：

$$Te_0(A_x) = \begin{pmatrix} 02 \\ 01 \\ 01 \\ 03 \end{pmatrix} S(A_x) \quad Te_1(A_x) = \begin{pmatrix} 03 \\ 02 \\ 01 \\ 01 \end{pmatrix} S(A_x)$$

$$Te_2(A_x) = \begin{pmatrix} 01 \\ 03 \\ 02 \\ 01 \end{pmatrix} S(A_x) \quad Te_3(A_x) = \begin{pmatrix} 01 \\ 01 \\ 03 \\ 02 \end{pmatrix} S(A_x)$$

即一輪操作為：

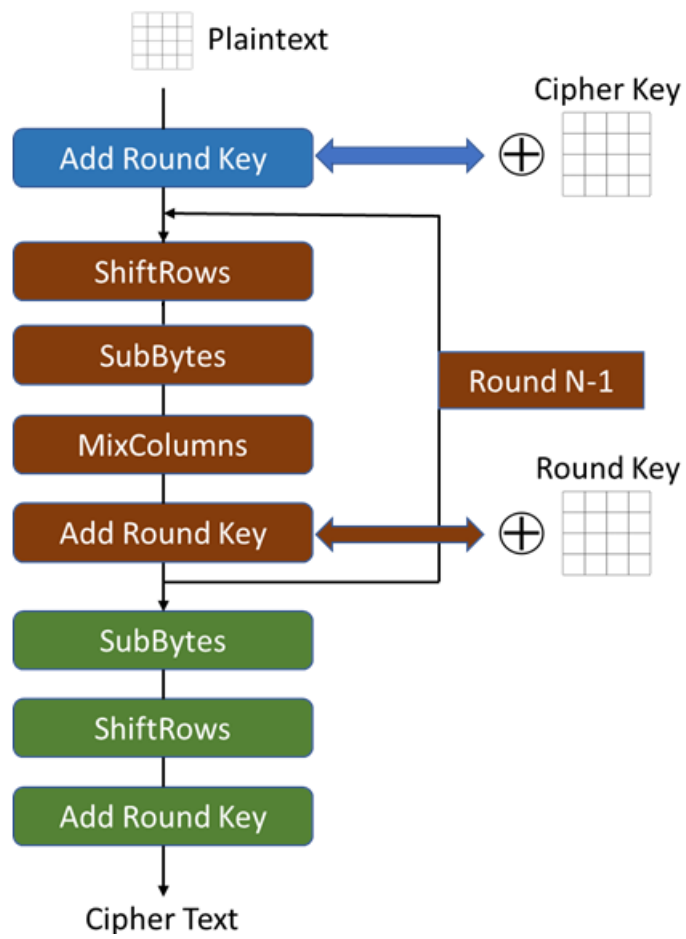
$$\begin{pmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \end{pmatrix} = Te_0(A_0) + Te_1(A_5) + Te_2(A_{10}) + Te_3(A_{15}) + W_{k0}$$

$$\begin{pmatrix} D_4 \\ D_5 \\ D_6 \\ D_7 \end{pmatrix} = Te_0(A_4) + Te_1(A_9) + Te_2(A_{14}) + Te_3(A_3) + W_{k1}$$

$$\begin{pmatrix} D_8 \\ D_9 \\ D_{10} \\ D_{11} \end{pmatrix} = Te_0(A_8) + Te_1(A_{13}) + Te_2(A_2) + Te_3(A_7) + W_{k2}$$

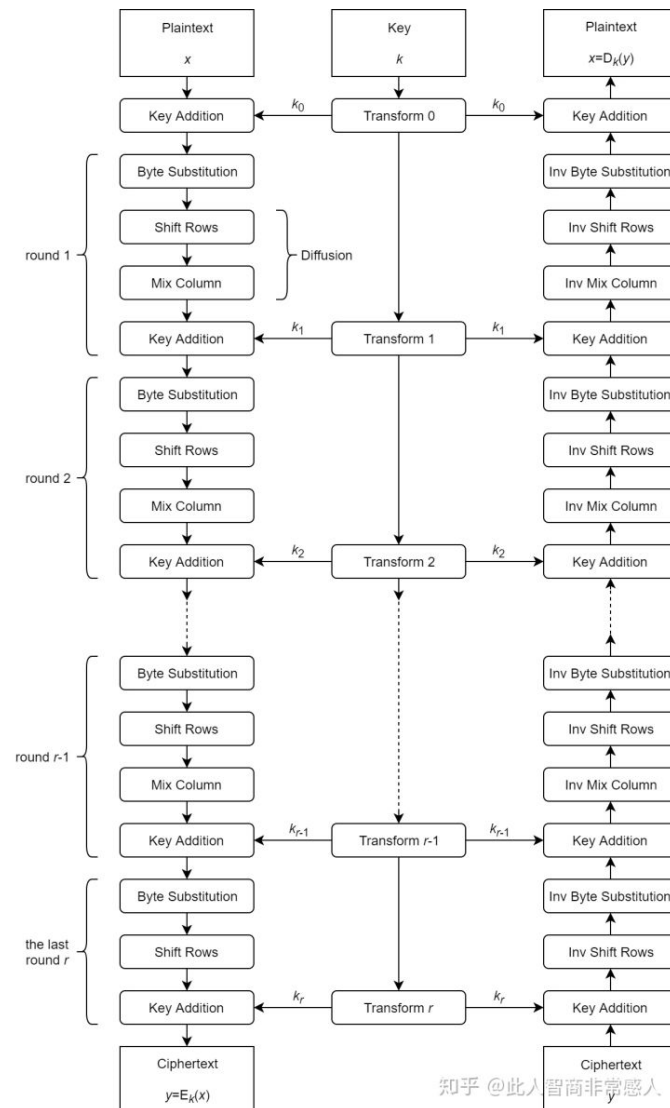
$$\begin{pmatrix} D_{12} \\ D_{13} \\ D_{14} \\ D_{15} \end{pmatrix} = Te_0(A_{12}) + Te_1(A_1) + Te_2(A_6) + Te_3(A_{11}) + W_{k3}$$

到了這裡，AES 的加密流程就變成了下圖所示：



5-3 解密過程

一個比較理想的情況是，解密也採用和加密過程相對稱的流程。但是觀察 AES



的原始流程圖，

對於解密過程，無論是否交換 InvSubBytes 層和 InvShiftRows 層，都無法將 InvSubBytes 層和 InvMixColumn 層按照先 InvSubBytes 再 InvMixColumn 的順序放在一起，即無法形成 ShiftRows->T-Box->KeyAddition 的一輪操作順序。這樣帶來的問題是，按照類似於構造 Te 表的方式構造的 Td 表將無法正確應用這個流程上！

而解決方案，是將原始解密方案中的 InvMixColumn 層和 AddRoundKey 層交換順序！這樣可以構成 ShiftRows->T-Box->KeyAddition 的一輪操作順序，但這是一個非常“冒險”的改變，因為 KeyAddition 和 InvMixColumn 這兩個操作是不可交換的。我們必需採取措施來防止錯誤的產生。

我們對 InvMixColumn 這個操作使用矩陣的乘法分配律：

$$\begin{aligned}
 \begin{pmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_1 & B_5 & B_9 & B_{13} \\ B_2 & B_6 & B_{10} & B_{14} \\ B_3 & B_7 & B_{11} & B_{15} \end{pmatrix} &= \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} C_0 & C_4 & C_8 & C_{12} \\ C_1 & C_5 & C_9 & C_{13} \\ C_2 & C_6 & C_{10} & C_{14} \\ C_3 & C_7 & C_{11} & C_{15} \end{pmatrix} \\
 &= \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \left(\begin{pmatrix} D_0 & D_4 & D_8 & D_{12} \\ D_1 & D_5 & D_9 & D_{13} \\ D_2 & D_6 & D_{10} & D_{14} \\ D_3 & D_7 & D_{11} & D_{15} \end{pmatrix} + \begin{pmatrix} K_0 & K_4 & K_8 & K_{12} \\ K_1 & K_5 & K_9 & K_{13} \\ K_2 & K_6 & K_{10} & K_{14} \\ K_3 & K_7 & K_{11} & K_{15} \end{pmatrix} \right) \\
 &= \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} D_0 & D_4 & D_8 & D_{12} \\ D_1 & D_5 & D_9 & D_{13} \\ D_2 & D_6 & D_{10} & D_{14} \\ D_3 & D_7 & D_{11} & D_{15} \end{pmatrix} + \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} K_0 & K_4 & K_8 & K_{12} \\ K_1 & K_5 & K_9 & K_{13} \\ K_2 & K_6 & K_{10} & K_{14} \\ K_3 & K_7 & K_{11} & K_{15} \end{pmatrix}
 \end{aligned}$$

這告訴我們：如果我們對子密鑰 k_i 也做一次 InvMixColumn 操作，然後再用於 AddRoundKey 層，那麼就可以保證上述交換 InvMixColumn 層和 AddRoundKey 層的改變不影響解密的正確性。

雖然這樣我們在生成解密用的子密鑰的時候要多做一步操作，但是這是值得的，因為在分組密碼中，處理一長段內容都採用同一個密鑰，即使用同一組子密鑰。我們一般只進行一次密鑰編排，而將生成的子密鑰保存在內存中，所以這個額外的過程只需要做一次。而在處理長內容的過程中，AES 核心要對多個塊依次進行處理，這裡會節省下來非常多的時間。

所以，我們仍然仿照加密過程來定義解密過程中需要的 Td 表：

$$Td_0(D_x) = \begin{pmatrix} 0E \\ 09 \\ 0D \\ 0B \end{pmatrix} S^{-1}(D_x) \quad Td_1(D_x) = \begin{pmatrix} 0B \\ 0E \\ 09 \\ 0D \end{pmatrix} S^{-1}(D_x)$$

$$Td_2(D_x) = \begin{pmatrix} 0D \\ 0B \\ 0E \\ 09 \end{pmatrix} S^{-1}(D_x) \quad Td_3(D_x) = \begin{pmatrix} 09 \\ 0D \\ 0B \\ 0E \end{pmatrix} S^{-1}(D_x)$$

即一輪操作為：

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = Td_0(D_0) + Td_1(D_{13}) + Td_2(D_{10}) + Td_3(D_7) + W_{k0}$$

$$\begin{pmatrix} A_4 \\ A_5 \\ A_6 \\ A_7 \end{pmatrix} = Td_0(D_4) + Td_1(D_1) + Td_2(D_{14}) + Td_3(D_{11}) + W_{k1}$$

$$\begin{pmatrix} A_8 \\ A_9 \\ A_{10} \\ A_{11} \end{pmatrix} = Td_0(D_8) + Td_1(D_5) + Td_2(D_2) + Td_3(D_{15}) + W_{k2}$$

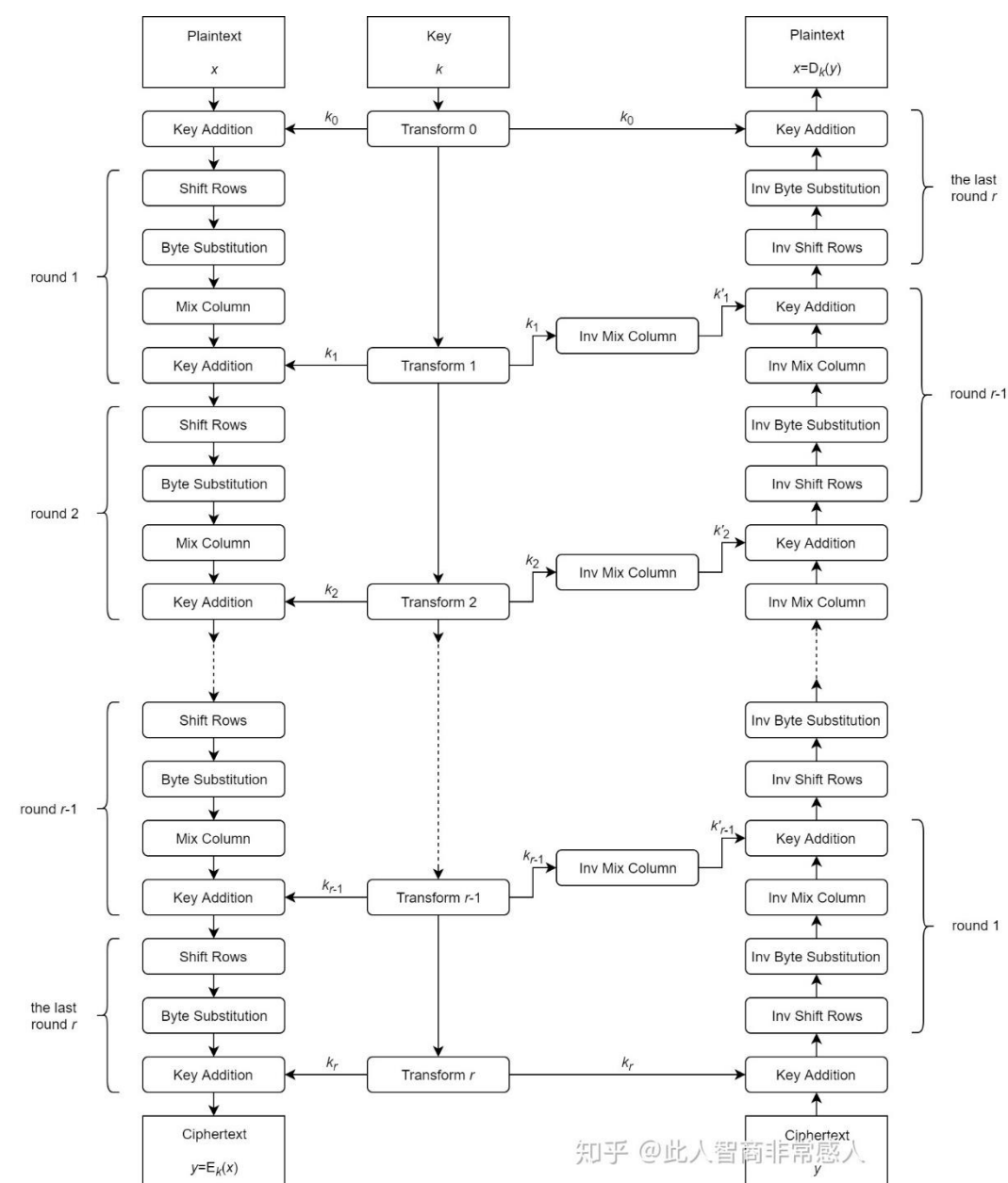
$$\begin{pmatrix} A_{12} \\ A_{13} \\ A_{14} \\ A_{15} \end{pmatrix} = Td_0(D_{12}) + Td_1(D_9) + Td_2(D_6) + Td_3(D_3) + W_{k3}$$

注意輪的劃分也發生了改變：一個完整的一輪操作在這裡定義為(依次)

InvShiftRows 層、InvSubBytes 層、InvMixColumn 層和 AddRoundKey 層。

解密過程中，第 1 輪之前還有一次 AddRoundKey 操作，最後一輪沒有

InvMixColumn 操作——這幾乎和加密過程的層操作順序一模一樣了！



陸、資料參考資源

6-1 SubBytes and Inverse SubBytes

S_Box

```
unsigned char S_Box[256] =
{
    //0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 //F
};
```

S_Box_Inv

```
unsigned char S_Box_Inv[256] =
{
    //0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,
    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,
    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,
    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,
    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,
    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,
    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,
    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,
    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d
};
```

SubBytes

```
void SubBytes(unsigned char state[4][4]) {
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            //The content of state is the S_Box index
            state[i][j] = S_Box[state[i][j]];
        }
    }
}
```

Inverse SubBytes

```
void Inv_SubBytes(unsigned char state[4][4]) {
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            //The content of state is the S_Box_Inv index
            state[i][j] = S_Box_Inv[state[i][j]];
        }
    }
}
```

6-2 ShiftRows and Inverse ShiftRows

ShiftRows

```
void ShiftRows(unsigned char state[4][4]) {  
    unsigned char temp[4][4];  
    //1's row not rotate  
    temp[0][0] = state[0][0];  
    temp[0][1] = state[0][1];  
    temp[0][2] = state[0][2];  
    temp[0][3] = state[0][3];  
  
    //2's row left rotate 1 unit  
    temp[1][0] = state[1][3];  
    temp[1][1] = state[1][0];  
    temp[1][2] = state[1][1];  
    temp[1][3] = state[1][2];  
  
    //3's row left rotate 2 unit  
    temp[2][0] = state[2][2];  
    temp[2][1] = state[2][3];  
    temp[2][2] = state[2][0];  
    temp[2][3] = state[2][1];  
  
    //4's row left rotate 3 unit  
    temp[3][0] = state[3][1];  
    temp[3][1] = state[3][2];  
    temp[3][2] = state[3][3];  
    temp[3][3] = state[3][0];  
  
    //state copy from temp  
    for(int i = 0; i < 4; i++) {  
        for(int j = 0; j < 4; j++) {  
            state[i][j] = temp[i][j];  
        }  
    }  
}
```

Inverse ShiftRows

```
void Inv_ShiftRows(unsigned char state[4][4]){  
    unsigned char temp[4][4];  
    //1's row not rotate  
    temp[0][0] = state[0][0];  
    temp[0][1] = state[0][1];  
    temp[0][2] = state[0][2];  
    temp[0][3] = state[0][3];  
  
    //2's row right rotate 1 unit  
    temp[1][0] = state[1][1];  
    temp[1][1] = state[1][2];  
    temp[1][2] = state[1][3];  
    temp[1][3] = state[1][0];  
  
    //3's row right rotate 2 unit  
    temp[2][0] = state[2][2];  
    temp[2][1] = state[2][3];  
    temp[2][2] = state[2][0];  
    temp[2][3] = state[2][1];  
  
    //4's row right rotate 3 unit  
    temp[3][0] = state[3][3];  
    temp[3][1] = state[3][0];  
    temp[3][2] = state[3][1];  
    temp[3][3] = state[3][2];  
  
    //state copy from temp  
    for(int i = 0; i < 4; i++) {  
        for(int j = 0; j < 4; j++) {  
            state[i][j] = temp[i][j];  
        }  
    }  
}
```

6-3 MixColumns and Inverse MixColumns

xTime

```
unsigned char xTime(unsigned char x){
    //if the largest bit is 1 then * 0x1b, else just left rotate one bit
    return ((x << 1) ^ ((x >> 7) & 1) * 0x1b);
}
```

Multiply

```
unsigned char multiply(unsigned char a, unsigned char b) {
    unsigned char c = 0;
    unsigned char d = b;
    //to check 8 bits, if it is 1, we do xor
    for (int i = 0; i < 8; i++) {
        if (a & 2 == 1)    c ^= d;
        a /= 2;    //go to check next bit
        d = xTime(d);    //to check this largest
    }
    return c;
}
```

MixColumns

```
void MixColumns(unsigned char state[4][4]) {
    unsigned char temp[4]; //temporary for one column
    for(int i = 0; i < 4; i++){
        //to calculate column by column
        temp[0] = state[0][i];
        temp[1] = state[1][i];
        temp[2] = state[2][i];
        temp[3] = state[3][i];
        /*left multiply by
        02 03 01 01
        01 02 03 01
        01 01 02 03
        03 01 01 02
        over GF(2^8)*/
        state[0][i] = multiply(0x02, temp[0]) ^ multiply(0x03, temp[1]) ^ temp[2] ^ temp[3];
        state[1][i] = temp[0] ^ multiply(0x02, temp[1]) ^ multiply(0x03, temp[2]) ^ temp[3];
        state[2][i] = temp[0] ^ temp[1] ^ multiply(0x02, temp[2]) ^ multiply(0x03, temp[3]);
        state[3][i] = multiply(0x03, temp[0]) ^ temp[1] ^ temp[2] ^ multiply(0x02, temp[3]);
    }
}
```

Inverse MixColumns

```
void Inv_MixColumns(unsigned char state[4][4]){
    unsigned char temp[4]; //temporary for one column
    for(int i = 0; i < 4; i++){
        //to calculate column by column
        temp[0] = state[0][i];
        temp[1] = state[1][i];
        temp[2] = state[2][i];
        temp[3] = state[3][i];
        /*multiply left by
        0E 0B 0D 09
        09 0E 0B 0D
        0D 09 0E 0B
        0B 0D 09 0E
        over GF(2^8)*/
        state[0][i] = multiply(0x0e, temp[0]) ^ multiply(0x0b, temp[1]) ^ multiply(0x0d, temp[2]) ^ multiply(0x09, temp[3]);
        state[1][i] = multiply(0x09, temp[0]) ^ multiply(0x0e, temp[1]) ^ multiply(0x0b, temp[2]) ^ multiply(0x0d, temp[3]);
        state[2][i] = multiply(0x0d, temp[0]) ^ multiply(0x09, temp[1]) ^ multiply(0x0e, temp[2]) ^ multiply(0x0b, temp[3]);
        state[3][i] = multiply(0x0b, temp[0]) ^ multiply(0x0d, temp[1]) ^ multiply(0x09, temp[2]) ^ multiply(0x0e, temp[3]);
    }
}
```


6-4 AddRoundKeys

AddRoundKeys

```
void AddRoundKey(unsigned char state[4][4], unsigned char RoundKey[4][4]) {  
    //Each element do XOR with RoundKey  
    for(int i = 0; i < 4; i++) {  
        for(int j = 0; j < 4; j++) {  
            state[i][j] ^= RoundKey[i][j]; //XOR RoundKey  
        }  
    }  
}
```

柒、資料參考資源

1. https://www.google.com.tw/search?q=s-box&tbm=isch&ved=2ahUKEwirr8Kf3vDsAhWxzIsBHa4zBcUQ2-cCegQIABAA&oq=s-box&gs_lcp=CgNpbWcQAzICCAAYAggAMgYIABAHEB4yBggAEAcQHjIGCAAQBxAeMgYIABAHEB4yBggAEAcQHjIGCAAQBxAeMgYIABAHEB4yBggAEAcQHIDDpQIYw6UJYKOnCWgAcAB4AIABZlgBZJIBAzAuMZgBAKABAaoBC2d3cy13aXotaW1nwAEB&sclient=img&ei=l7umX-uHL7GZr7wPrueUqAw&bih=610&biw=1280&hl=zh-TW#imgsrc=896NvhMLHc5xpM
2. <https://crypto.stackexchange.com/questions/55975/aes-s-box-input-and-output-question>
3. https://www.youtube.com/watch?v=7vN2J_kHnzs
4. <https://slideplayer.com/slide/3380067/>
5. <https://kknews.cc/zh-tw/code/rn24jmo.html>
6. https://www.youtube.com/watch?v=RTThGL7n_0s
7. <https://zhuanlan.zhihu.com/p/42264499>
8. <https://zhuanlan.zhihu.com/p/41716899>
9. <https://zh.wikipedia.org/wiki/%E9%AB%98%E7%BA%A7%E5%8A%A0%E5%AF%86%E6%A0%87%E5%87%86>