Advanced Encryption Standard (AES)

Salvi Niccolò

20 marzo 2021

Indice

1 Introduzione

AES[1] o *Rijndael* è un algoritmo di cifratura a blocchi utilizzato come standard dal governo degli USA.

Nel 1997 il NIST[2] richiese pubblicamente proposte di algoritmi, tra i quali scegliere il sostituto del DES[3].

Il nuovo standard avrebbe dovuto consentire l'utilizzo di chiavi di 128, 192, 256 bit ed operare su blocchi di 128 bit in input.

Trascorso il periodo di analisi e confronto con altre proposte, nel 2002 diventa un effettivo standard.

L'algoritmo di cifratura *Rijndael* è stato progettato da Joan Daemen e Vincent Rijmen ed è un'evoluzione dell'algoritmo *Square* già esistente.

1.1 Criteri di valutazione

Il NIST utilizzò due diversi metri di giudizio nelle due fasi di valutazione degli algoritmi.

La prima valutazione si basò sulla verifica dei requisiti fondamentali:

- Sicurezza:
  - **Effettiva sicurezza**: da confrontare con gli altri algoritmi presenti
  - **Casualità**: il livello di indistinguibilità dell'output dell'algoritmo da un blocco, generato in modo casuale
  - **Solidità**: delle basi matematiche su cui si fonda l'algoritmo

  Siccome la dimensione minima della chiave era di 128 bit, gli attacchi a forza bruta con le tecnologie del tempo erano considerati impraticabili e per questo motivi era preferibile concentrarsi sull'analisi della resistenza dell'algoritmo rispetto agli altri attacchi noti.

- Costo:
  - **Requisiti di proprietà intellettuale**: l'algoritmo doveva essere accessibile e disponibile globalmente in modo gratuito
  - **Efficienza computazionale**: valutata prima l'implementazione software, poi la velocità dell'algoritmo
  - **Requisiti di memoria**: la memoria necessaria per l'esecuzione dell'algoritmo esaminato

- Altre caratteristiche

[1] **A**dvanced **E**ncryption **S**tandard
[2] **N**ational **I**nstitute of **S**tandards and **T**echnology
[3] **D**ata **E**ncryption **S**tandard

- **Flessibilità**: preferiti gli algoritmi che rispondevano ad un numero maggiore di utenti
- **Fattibilità hardware e software**: l'algoritmo non doveva essere restrittivo rispetto alla piattaforma hardware o software
- **Semplicità**: semplicità del progetto

Utilizzando questi criteri, gli algoritmi candidati vennero ridotti da 21 a 15 e poi 5.

I criteri utilizzati per la selezione finale:

- **Ambienti con spazio limitato**: valutata la memoria richiesta per il codice, la rappresentazione di *S-box* e sottochiavi
- **Implementazione hardware**
- **Crittografia e decrittografia**: se gli algoritmi sono differenti, richiedono ulteriore spazio; vi possono essere differenze nel tempo di esecuzione fra la cifratura e la decifratura
- **Agilità della chiave**: capacità di cambiare rapidamente la chiave, utilizzando la minima quantità possibile di risorse.
- **Flessibilità dei parametri**: il supporto di chiavi e blocchi di altre dimensioni e la possibilità di aumentare il numero di *round*
- **Potenzialità sfruttamento del parallelismo**: capacità di sfruttare le funzionalità di esecuzione parallela nei processi attuali e futuri

1.2 Valutazione *Rijndael*

I motivi per cui l'algoritmo di *Rijndael* è stato scelto sono i seguenti:

- **Sicurezza generale**: non esiste alcun attacco noto in grado di romperlo. *Rijndael* usa delle *S-box* come componenti non lineari che dovrebbero fornire un margine di sicurezza adeguato, ma ha ricevuto qualche critica per la semplicità della sua struttura matematica
- **Implementazione software**: *Rijndael* svolge adeguatamente le operazioni di cifratura e di decifratura in un'ampia varietà di piattaforme. Vi è una riduzione delle prestazioni quando aumentano le dimensioni delle chiavi. L'elevato parallelismo però facilita l'utilizzo efficiente delle risorse della CPU, ottenendo ottime prestazioni software
- **Ambienti con spazio limitato**: *Rijndael* è molto adatto ad ambienti con spazio limitato dove vengono implementate la crittografia e la decrittografia. I requisiti di memoria ROM aumentano quando la crittografia e la decrittografia sono entrambe implementate

- **Agilità della chiave**: *Rijndael* supporta il calcolo in tempo reale delle sottochiavi di crittografia; esso richiede una sola esecuzione per generare tutte le sottochiavi prima della decrittografia con una data chiave.
- **Potenzialità sfruttamento del parallelismo**: ottime potenzialità per sfruttare il parallelismo nella crittografia di un singolo blocco

2 Descrizione Algoritmo

2.1 La cifratura AES

La proposta *Rijndael* definiva una cifratura in cui la dimensione della chiave era 128, 192, 256 bit e vincolava la lunghezza del blocco a 128 bit.

La codifica consiste in:

- 10 *rounds* se la chiave è 128 bit
- 12 *rounds* se la chiave è 192 bit
- 14 *rounds* se la chiave è 256 bit

e ciascuno *round* è formato da 4 *layers*:

1. *Substitute Bytes* (*SubBytes*)
2. *ShiftRows Transformation*
3. *MixColumns Transformation*
4. *AddRoundKey*

L'ordine delle funzioni all'interno di ogni singolo *round* è il seguente:

$$SubBytes \rightarrow ShiftRows \rightarrow MixColumns \rightarrow AddRoundKey$$

Il *round* finale non utilizza *MixColumns* e quello preliminare, *round 0*, utilizza solo *AddRoundKey*.

Listing 1: Pseudo code for Chiper

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
    byte state[4,Nb]
    state = in

    AddRoundKey(state, w[0, Nb-1])

    for round = 1 step 1 to Nr-1
```

```
        SubBytes(state)
        ShiftRows(state)
        MixColumns(state)
        AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
    end for

    SubBytes(state)
    ShiftRows(state)
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

    out = state
end
```

2.1.1 Numero di *round*

Il numero di *round* è determinato in modo tale da rendere impossibile un attacco di forza bruta, ossia un attacco basato sulla ricerca esaustiva della chiave.

Il numero 10 è già un margine di sicurezza, poichè per il cifrario a blocchi di 128 bit non sono possibili questi tipi di attacco già con più di 6 *rounds*, per le seguenti motivazioni:

1. All'algoritmo *Rijndael* sono sufficienti due soli *rounds* per ottenere una diffusione completa dei bytes, per il fatto che i bit di *State* dipendono dai byte di *State* di due *rounds* precedenti a quello corrente.

2. Attacchi di crittografia lineare e differenziale sfruttano le correlazioni tra output ed input di n *round* per attaccare un cifrario di $n+1$ e $n+2$ *round*.

   Sono stati aggiuntu ulteriori 4 *rounds* ai 6 che ne avrebbero garantito la sicurezza, per raddoppiare la difficoltà dell'attacco che passa da 4 a 8 *rounds*.

2.1.2 State

L'input dell'algoritmo è un singolo blocco di 128 bit, rappresentato da un matrice quadrata di bytes nel *FIPS PUB 197*[4].

Analogamente la chiave di 128 bit è rappresentata anch'essa come una matrice quadrata di bytes e fornita come input in un array di 44 *word* a 32 bit.

Questo blocco viene copiato nell'array *State* che viene modificato in ogni fase della crittografia.

[4]Documento che contiene le caratteristiche ufficiali dell'AES.

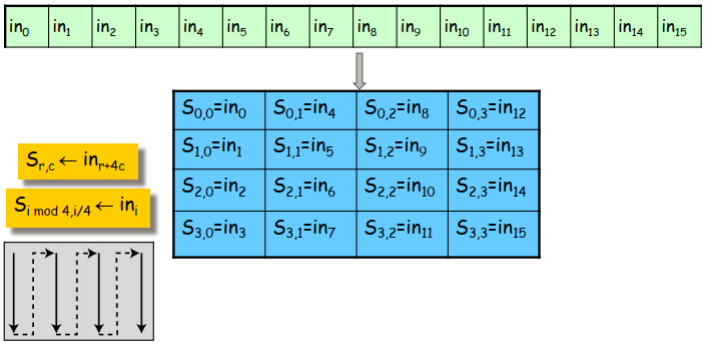

Figura 1: Input bytes → *State*

Dopo la fase finale, *State* viene copiato in una matrice di output.

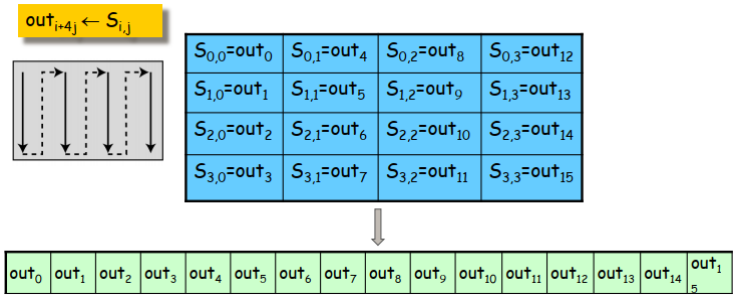

Figura 2: *States* → Output bytes

2.1.3 *S-box*

La *S-box* viene costruita nel seguente modo:

1. Inizializzare la *S-box* con il valore dei byte in una sequenza ascendente riga per riga. Il valore del byte nella riga x e colonna y è {xy}.
   La prima riga contiene {00}, {01}, {02}, ..., {0F}
   La seconda riga contiene {10}, {11}, {12}, ..., {1F}

2. Associare a ciascun byte della *S-box* il suo inverso moltiplicativo nel campo finito $GF(2^8)$.
   {00} rimane {00}

3. Considerare che ciascun byte della *S-box* è costituito da 8 bit con $(b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0)$. Applicare la seguente trasformazione a ciascun bit di ciascun byte della *S-box*:

$$b'_i = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus c_i$$

   dove c_i è l'i-esimo bit del byte c con il valore {63}: 01100011.

$$b'_0 = b_0 \oplus b_4 \oplus b_5 \oplus b_6 \oplus b_7 \oplus 0$$

$$b'_1 = b_1 \oplus b_5 \oplus b_6 \oplus b_7 \oplus b_0 \oplus 1$$

		y														
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
x	**0**	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4B	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Tabella 1: *S-box*

Le proprietà della *S-box* sono le seguenti:

- L'output non è una funziona lineare dell'input: si tratta dell'unica operazione non lineare dell'intero algoritmo
- $InvSbox(Sbox(a)) = a \rightarrow$ invertibile
- $Sbox(a) \neq InvSbox(a) \rightarrow$ no self-invertibile
- Progettata per resistere ad attacchi crittoanalitici

2.1.4 SubBytes

La *Trasformazione SubBytes* è una ricerca su tabella. AES definisce una matrice di 16x16 byte chiamata *S-box* che contiene una permutazione di tutti i 256 valori a 8 bit.

Ogni singolo byte di *State* viene mappato in nuovo byte: i primi 4 bit indicano la riga ed i secondi 4 bit la colonna. Tali valori di riga e colonna rappresentato gli indici della *S-box* per selezionare un valore univoco di output a 8 bit.

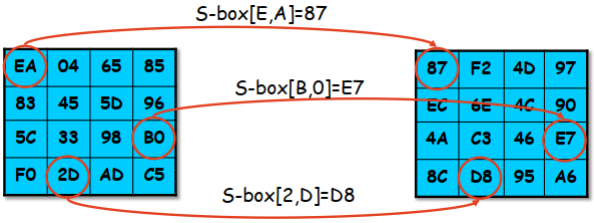

Figura 3: *State* $\xrightarrow{S\text{-}box}$ *State'*

2.1.5 ShiftRows

La *Trasformazione ShiftRows* consiste nello scorrimento di byte nell'array *State*:

- 1^a riga non viene modificata
- 2^a riga scorrimento circolare a sinistra di 1 byte
- 3^a riga scorrimento circolare a sinistra di 2 byte
- 4^a riga scorrimento circolare a sinistra di 3 byte

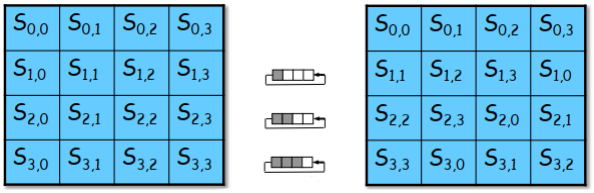

Figura 4: ShiftRows di *State*

2.1.6 MixColumns

La *Trasformazione MixColumns* opera su ogni singola colonna. Ciascun byte di una colonna viene mappato in un nuovo valore che è una funzione dei 4 byte presenti nella colonna. L'operazione è definita dalla seguente moltiplicazione di matrici su *State*:

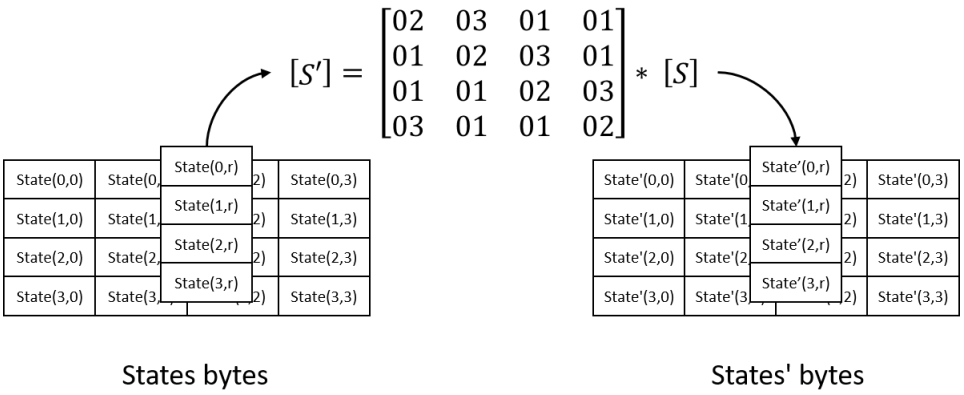

Figura 5: *State* * Matrix = *State'*

Ciascun elemento è somma dei prodotti degli elementi di una riga e una colonna.

$$0 \leq j \leq 3$$

$$s'_{0,j} = (\{02\} \cdot s_{0,j}) \oplus (\{03\} \cdot s_{1,j}) \oplus s_{2,j} \oplus s_{3,j}$$

$$s'_{1,j} = s_{0,j} \oplus (\{02\} \cdot s_{1,j}) \oplus (\{03\} \cdot s_{2,j}) \oplus s_{3,j}$$

$$s'_{2,j} = s_{0,j} \oplus s_{1,j} \oplus (\{02\} \cdot s_{2,j}) \oplus (\{03\} \cdot s_{3,j})$$

$$s'_{3,j} = (\{03\} \cdot s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (\{02\} \cdot s_{3,j})$$

Dove $\cdot$ si utilizza per indicare la moltiplicazione sul campo finito $GF(2^8)$ e $\oplus$ per indicare l'operazione XOR bit a bit, che corrisponde alla somma in $GF(2^8)$.

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

$\rightarrow$

47	40	A3	4C
37	D4	70	9F
94	E4	3A	42
ED	A5	A6	BC

$$s'_{0,0} = (\{02\} \cdot \{87\}) \oplus (\{03\} \cdot \{6E\}) \oplus \{46\} \oplus \{A6\} = \{47\}$$

$$s'_{1,0} = \{87\} \oplus (\{02\} \cdot \{6E\}) \oplus (\{03\} \cdot \{46\}) \oplus \{A6\} = \{37\}$$

$$s'_{2,0} = \{87\} \oplus \{6E\} \oplus (\{02\} \cdot \{46\}) \oplus (\{03\} \cdot \{A6\}) = \{94\}$$

$$s'_{3,0} = (\{03\} \cdot \{87\}) \oplus \{6E\} \oplus \{46\} \oplus (\{02\} \cdot \{A6\}) = \{ED\}$$

Il documento AES descrive anche un altro modo per eseguire la *Trasformazione MixColumns*: ciascuna colonna di *State* viene definita come polinomio di quattro termini con coefficienti in $GF(2^8)$.

Ciascuna colonna viene moltiplicata modulo $(x^4 + 1)$ per il polinomio fisso $a(x)$ dato da:

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$

2.1.7 AddRoundKey

Nella *Trasformazione AddRoundKey* viene inserita la chiave segreta che rende il cifrario sicuro (le altre trasformazioni sono note e quindi facilmente invertibili).

Nella *Trasformazione AddRoundKey* i 128 bit di *State* vengono sottoposti a uno XOR bit a bit con i 128 bit della *Round Key*[5]: si tratta di una somma vettoriale in $\mathbb{Z}_2^{128}$ tra i 4 byte della colonna di *State* ed *Nb* della chiave schedulata.

La sicurezza è garantita dalla complessità dell'espansione della chiave di fase e dalla complessità degli altri stadi di AES.

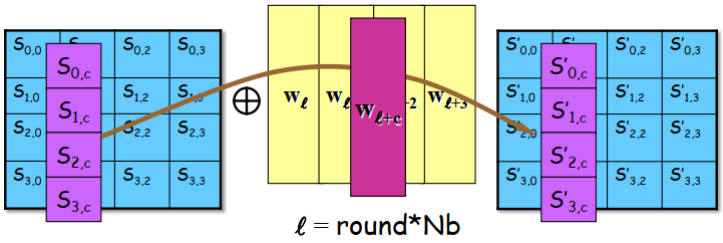

Figura 6: *AddRoundKey*

2.1.8 Espansione chiave

Rijndael implementa una routine di espansione della chiave per generare una chiave schedulata, partendo da quella presa in input.

L'algoritmo di espansione della chiave genera un totale di $Nb * (Nr + 1)$ *words*: il processo di cifratura richiede un insieme iniziale di *Nb words* ed ad ogni esecuzione di un *round* vengono richiesti *Nb words* della chiave.

La chiave schedulata risultante consiste di un array lineare di *words* a 32 bit.

Listing 2: Pseudo Code for the Key Expansion

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
```

[5]Ad ogni *round* la chiave aggiunta è diversa e ricavata ricorsivamente dalle precedenti.

```
begin
    word temp
    i = 0

    while (i < Nk)
        w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
        i = i+1
    end while

    i = Nk
    while (i < Nb * (Nr+1))
        temp = w[i-1]
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if (Nk > 6 and i mod Nk = 4)
            temp = SubWord(temp)
        end if

        w[i] = w[i-Nk] xor temp
        i = i + 1
    end while
end
```

Le prime Nk (rappresenta il numero di *words* a 32 bit che formano la chiave di cifratura) *words* della chiave espansa sono ottenute direttamente dalla chiave di cifratura presa in input.

Ogni word successiva, $W[i]$, è uguale allo XOR tra la *word* precedente e quella di Nk posizioni più indietro.

$$W[i] = W[i-1] \oplus W[i-Nk]$$

Per le *words* con posizioni multiple di Nk, viene applicata a $W[i-1]$ una trasformazione, seguita da uno XOR con una costante di round *Rcon[i]* ed infine viene eseguito la XOR con la *word* di Nk posizioni precedenti (come definito nel punto precedente).

Questa trasformazione consiste in:

RotWord() compie uno scorrimento circolare a sinistra di un byte sulla *Word*

SubWord() utilizzando la *S-box* di AES sostituisce ogni byte della *Word* in input

L'algoritmo di espansione della chiave per una chiave di 256 bit si compota diversamente rispetto ad una chiave di 128 o 192 bit, in quanto cambia il valore di Nk.

2.2 La decifratura di AES

AES non è un cifrario di *Feistel*, al contrario del precedente algoritmo DES.

Pertanto, la cifratura e la decifrazione utilizzano due algoritmi differenti.

Ogni funzione della cifratura ha una sua funzione inversa:

1. *InvSubBytes* utilizza la permutazione inversa *InvS-box*
2. *InvShiftRows* si ottiene facendo scorrere le righe a destra, invece che a sinistra.
3. *InvMixColumns* sfrutta la matrice invertita usata in *MixColumns*
4. *AddRoundKey* coincide con il suo inverso

L'ordine delle funzioni all'interno di ogni singolo *round* è il seguente:

InvShiftRows → *InvSubBytes* → *AddRoundKey* → *InvMixColumns*

Listing 3: Pseudo Code for the Inverse Cipher

```
InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
    byte state[4,Nb]
    state = in

    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

    for round = Nr-1 step -1 downto 1
        InvShiftRows(state)
        InvSubBytes(state)
        AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
        InvMixColumns(state)
    end for

    InvShiftRows(state)
    InvSubBytes(state)
    AddRoundKey(state, w[0, Nb-1])

    out = state
end
```

2.2.1 *InvS-box*

La *InvS-box* è un'altra tabella che implementa la permutazione inversa, calcolata nella seguente maniera:

$$b'_i = b_i \oplus b_{(i+2)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+7)mod8} \oplus d_i$$

$$d = \{05\}$$

		y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
x	**0**	52	09	6A	D5	30	36	A5	38	DF	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

Tabella 2: *InvS-box*

2.2.2 InvShiftRows

La *Trasformazione InvShiftRows* applica all'array *State* uno scorrimento circolare nelle righe, allo stesso modo della corrispettiva funzione di cifratura, ma nella direzione opposta: verso destra.

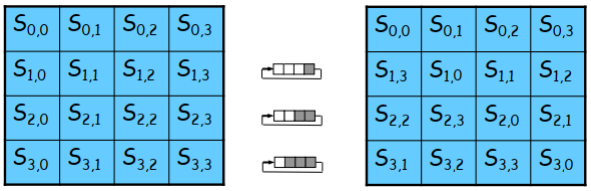

Figura 7: *InvShiftRows*

2.2.3 InvSubBytes

La *Trasformazione InvSubBytes* funziona esattamente allo stesso modo della sua corrispettiva diretta, cambia solo la *S-box* è una tabella che implementa la permutazione inversa.

2.2.4 InvMixColumns

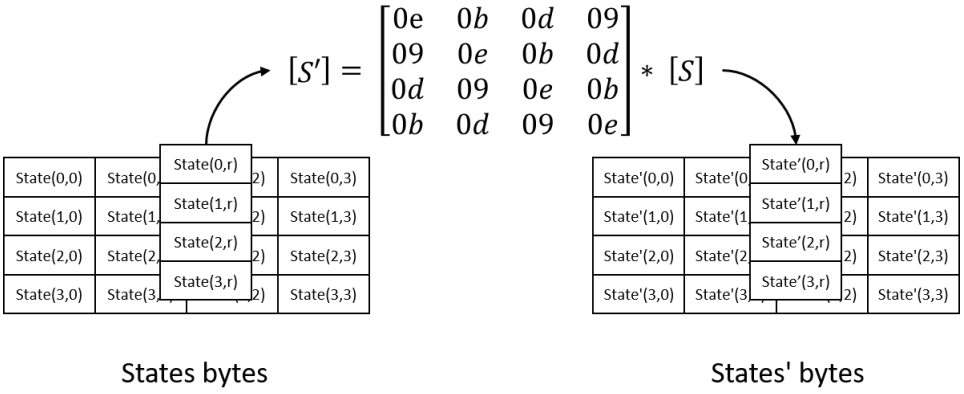

Figura 8: *InvMixColumns*

2.2.5 Cifratura inversa equivalente

Predisporre di due algoritmi differenti per le operazione di cifratura e di decifratura rappresenta uno svantaggio dal punto di vista implementativo.

E' stato implementato un algoritmo equivalente a quello di decrittografia, che utilizza sempre le funzioni inverse nel medesimo ordine del processo di cifrazione.

InvSubBytes → *InvShiftRows* → *InvMixColumns* → *AddRoundKey*

Listing 4: Pseudo Code for the Equivalent Inverse Cipher

```
EqInvCipher(byte in[4*Nb], byte out[4*Nb], word dw[Nb*(Nr+1)])
begin
    byte state[4,Nb]
    state = in

    AddRoundKey(state, dw[Nr*Nb, (Nr+1)*Nb-1])
    for round = Nr-1 step -1 downto 1
        InvSubBytes(state)
        InvShiftRows(state)
```

```
        InvMixColumns(state)
        AddRoundKey(state, dw[round*Nb, (round+1)*Nb-1])
    end for

    InvSubBytes(state)
    InvShiftRows(state)
    AddRoundKey(state, dw[0, Nb-1])

    out = state
end
```

Rispetto all'algoritmo di decifratura precedente è necessario scambiare le prime due trasformazioni tra loro, così come le seconde due.

Scambio tra *InvShiftRows* e *InvSubBytes* *InvShiftRows* altera la sequenza dei bytes di *State* ma non il suo contenuto, mentre *InvSubBytes* ne altera il contenuto ma non la sequenza dei bytes. Per un determinato *State* S_i:

$$InvShiftRows[InvSubBytes(S_i)] = InvSubBytes[InvShiftRows(S_i)]$$

Scambio tra *AddRoundKey* e *InvMixColumns* Per un determinato *State* S_i ed una determinata chiave di fase W_j:

$$InvMixColumns(S_i \oplus W_j) = [InvMixColumns(S_i)] \oplus [InvMixColumns(W_i)]$$

Per poter applicare lo scambio, basta applicare *InvMixColumns* alla chiave di fase prima di aggiungerla allo *State* corrente.