AES
Advanced Encryption Standard

di
Simone Forcella, Francesco Forconi, Simone Lai, Daniele Ninfo

Cenni Storici

Prima di cominciare a trattare l'algoritmo nelle sue parti, è sicuramente d'aiuto conoscere il contesto storico, in generale, nel quale questo si inserisce, sia per capire come sia nato, sia per avere un giusto approccio all'argomento.

Le Origini

Nel maggio del '73 l'NBS (National Bureau of Standard) decise di concentrare l'attenzione sullo sviluppo di algoritmi crittografici per la protezione dei dati; con una serie di note all'interno del "Federal Register" vennero definiti i requisiti necessari alla definizione di uno standard.
Un anno dopo l'IBM propose all'NBS l'evoluzione di un algoritmo chiamato "Lucifer", sul quale aveva iniziato le ricerche proprio dalla fine degli anni '60. Consegnato all' NSA (National Security Agency) questa s'incaricò sia dell'analisi che della revisione, per renderlo standard definitivo (15 gennaio 1977) con il nome di DES (Data Encryption Standard), con la clausola che ogni cinque anni ne fosse analizzata l'effettiva sicurezza, a fronte dell' evoluzione delle tecnologie e dei metodi di crittoanalisi. Queste analisi vennero presentate all'interno di workshop, durante i quali veniva discussa la possibilità di riconfermare l'algoritmo, o le eventuali modifiche o sostituzioni.
Nonostante fossero stati individuati molti punti deboli in DES (brevità della chiave o TrapDoor), il Data Encryption Standard resistette fino agli anni '90, quando ormai la potenza di calcolo dei calcolatori rese sempre più evidenti i punti deboli, al punto che una società americana chiamata RSA Data Security diede inizio a dei veri e propri Contest in cui sarebbe stato premiato chi fosse riuscito a violare il DES tramite attacchi a "Brute Force", cioè tentando tutte le chiavi possibili, nel minor tempo possibile. La prima violazione del DES avvenne nel 1997, e ci vollero ben 39 giorni, ma già un anno dopo bastarono 22 ore, questo grazie alla "DES Cracking Machine" una mainboard su cui erano montati solo processori.
Il governo U.S.A. si rese conto che ormai il DES stava giungendo alla fine della sua vita utile (anche nella versione 3DES che ancora oggi risulta inviolata, ma non di facile utilizzo a causa dei tempi calcolo necessari nella cifratura e decifratura).

Il concorso

Nel 1997 il NIST, (National Institute of Standard and Technology, ex NBS**)** l'agenzia del dipartimento di commercio, che ha il compito di approvare gli standard federali per il governo degli Stati Uniti decise che era il momento di avere un nuovo standard crittografico.
Si trovò però di fronte ad un problema di opinione pubblica, poiché in precedenza furono sollevati dubbi sull'effettiva segretezza degli algoritmi, in effetti quando uscì il DES si pensava che, essendo stato approvato dall' NSA, questa ne avesse anche una "BackDoor" con la quale tradurre i crittogrammi. Questo voleva dire che il nuovo algoritmo non poteva essere ideato dai laboratori dell'NSA, altrimenti nessuno si sarebbe fidato e l'algoritmo sarebbe subito caduto in disuso. Nacque così l'idea di un concorso. I ricercatori di tutto il mondo furono invitati ad inviare proposte di un nuovo standard che avrebbe preso il nome di **AES (advanced encryption standard)**, e avrebbe sostituito il DES.
I criteri adottati furono di due tipi, di scelta e di analisi.

I criteri di scelta furono i seguenti:

- Il cifrario doveva essere a chiave simmetrica;
- Il cifrario doveva essere a blocchi;
- La lunghezza della chiave doveva essere di 128 o 256bit
- I blocchi del testo in chiaro potevano essere di 64, 128 o 256 bit
- Doveva poter essere implementabile su Smart Card
- Doveva essere scritto in C o Java (scelti perché più diffusi)
- Doveva poter garantire una distribuzione dell'algoritmo a livello mondiale

I criteri d'analisi furono invece solamente tre:

1. Sicurezza, l'algoritmo doveva essere robusto, cioè resistente ad attacchi di varia natura

2. Costo, efficienza e velocità computazionale, insieme alla dimensione della memoria, dovevano essere considerate ai fini dell' applicazione in diversi campi di utilizzo.

3. Semplicità, l'oggetto che avrebbe crittato o decrittato il messaggio doveva poter essere facilmente costruibile, come anche il software doveva essere facilmente implementabile.

I passi fatti durante gli anni della selezione furono ufficializzati durante periodiche conferenze dette ROUND 1,2 e 3.

Durante il ROUND 1 (20 agosto del 1998) furono scelti solo 15 fra i candidati all' AES, e ne fu analizzata la versione con chiave a 128 bit.

Durante il ROUND 2 (nel 1999) furono analizzati i risultati che portarono alla determinazione di cinque finalisti:

1. MARS dell' IBM
2. RC6 della RCS
3. Rijndael di Joan Daemen e Vincent Rijmen
4. Serpent di Anderson, Biham, Knudsen
5. TwoFish di Schneider, Kelsey, Whiting, Wagner, Hall, Ferguson

Ai fini di una analisi approfondita dei candidati, il NIST costituì un forum pubblico e pubblicò una home page allo scopo di invitare la comunità internazionale a esprimere il proprio giudizio. Ai fini della selezione vennero presi in considerazione anche i documenti pervenuti in formato cartaceo ed elettronico.

Il ROUND 3 svoltosi alla fine del 2000, portò alla determinazione dell'algoritmo RIJNDAEL come Advanced Encryption Standard e successore del DES.

Concetti matematici di base

Molte operazioni in AES (o Rijndael) sono definite a livello byte, interpretando ogni bit come coefficiente dei polinomi del campo finito GF(2^8). Può essere utile capire in che cosa consistono i campi di Galois, ma è di fondamentale importanza capire come vengono effettuate le operazioni al loro interno.

Campi di Galois

Diamo dei cenni sui campi di Galois, in modo da rendere più comprensibile quella che poi sarà l'implementazione delle operazioni del Rijndael.

Definizione di campo (partendo dalla definizione di anello): *Si dice campo un anello commutativo unitario (A, +, •) (in cui l'operazione • è commutativa e esiste un elemento neutro per •, ad esempio 1) tale che:*

$$\text{per ogni } a \in A,\ a \neq 0$$

$$\text{esiste } a' \in A$$
$$\text{t.c. } a \bullet a' = a' \bullet a = 1$$

Definizione di polinomio irriducibile: *Un polinomio f(x) ∈ K[x] (anello commutativo unitario di polinomi con coefficienti nel campo K), ∂(f) = n, si dice irriducibile in K[x] se non si può scrivere come prodotto di due polinomi h(x), k(x) a coefficienti in K, con 0 < ∂(h) < n e 0 < ∂(k) < n.*

Facciamo un esempio:

verifichiamo che $f(x) = x^2+1$ sia riducibile in $\mathbb{R}[x]$ (campo dei polinomi a coefficienti reali). Dobbiamo trovare due polinomi $h(x) \in \mathbb{R}[x]$ e $k(x) \in \mathbb{R}[x]$, di grado maggiore di zero e inferiore a 2, tali che il loro prodotto sia uguale a f(x). Dunque, noi sappiamo che, in generale, un polinomio di secondo grado c(x) è rappresentabile come prodotto di due polinomi di primo grado nella forma:

$$(x - s_1)(x - s_2)$$

dove s_1 e s_2 sono le due soluzioni dell'equazione c(x)=0. Proviamo a scomporre f(x) trovando le due soluzioni:

$$x^2+1 = 0$$
$$x^2 = -1$$
$$x_{1,2} = \mp \text{ sqrt}(-1) \notin \mathbb{R}$$

Non avendo soluzioni reali, h(x) e k(x) non esistono in $\mathbb{R}[x]$ (mentre esistono in $\mathbb{C}[x]$), di conseguenza possiamo dire che il polinomio f(x) è *irriducibile* in $\mathbb{R}[x]$.

Definizione di campi di Galois: *sia $\mathbb{Z}_p$ il campo finito degli interi mod p (p è primo, altrimenti $\mathbb{Z}_p$ non sarebbe un campo), $\mathbb{Z}_p[x]$ l'anello dei polinomi a coefficienti in $\mathbb{Z}_p$, e sia m(x) ∈ $\mathbb{Z}_p$, m(x) irriducibile in $\mathbb{Z}_p[x]$, ∂(m) = n. Chiamiamo Campo di Galois rispetto al polinomio m(x), il campo costituito dalle $q = p^n$ classi di congruenza mod m(x):*

$$GF(q) = GF(p^n) = \mathbb{Z}_p[x] / m(x)$$

Facciamo un esempio:

costruiamo $GF(2^2)$, rispetto al polinomio irriducibile in $\mathbb{Z}_2[x]$:

$$m(x) = x^2 + x + 1$$

Dobbiamo quindi determinare le 2^2 classi di congruenza mod $x^2 + x + 1$ (ovvero tutti i possibili resti di una divisione di un polinomio $f(x) \in \mathbb{Z}_2[x]$ per $m(x)$).

Essendo $m(x)$ di secondo grado, i resti della divisione di ogni polinomio $f(x) \in \mathbb{Z}_2[x]$ per $m(x)$ sono t.c.

$$r(x) = 0 \text{ oppure } 0 \leq \partial(r) < 2$$

quindi il grado dei resti può essere al massimo 1. I resti assumeranno la forma:

$$ax + b$$

Costruiamo tutte le possibili combinazioni ($2^2 = 4$) con a e b in $\mathbb{Z}_2$:

a	b	resto
0	0	0
0	1	1
1	0	x
1	1	x+1

Quindi $GF(2^2) = \{ 0, 1, x, x+1\}$.

I bytes in AES

Nel Rijndael ogni byte (sequenza di 8 bit) viene interpretato come un polinomio in $GF(2^8)$ in questo modo:

$$\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\} \Rightarrow b_7x^7+b_6x^6+b_5x^5+b_4x^4+b_3x^3+b_2x^2+b_1x+b_0$$

ogni bit equivale ad un coefficiente del polinomio (b_7 è il bit più significativo del byte).

Esempio (conversione da esadecimale a $GF(2^8)$):

$$\{57\}(hex) \Rightarrow \{01010111\}(bin) \Rightarrow x^6+x^4+x^2+x+1 \ (GF(2^8))$$

(d'ora in poi rappresenteremo con $\{x_1x_0\}$ un numero esadecimale e con $\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$ un numero binario).

Somma tra bytes

La somma di due elementi nel campo finito si ottiene sommando mod 2 i coefficienti delle corrispondenti potenze della x nei due polinomi. Di conseguenza, in Rijndael, la somma di due bytes si ottiene effettuando la somma mod 2 dei bit corrispondenti.

Implementazione: la somma tra due bytes può essere implementata come XOR ($\oplus$) bit a bit.

Esempio:

Notazione polinomiale:

$$(x^6+x^4+x^2+x+1) + (x^7+x+1) = x^7+x^6+x^4+x^2$$

Notazione binaria:

$$\{01010111\} \oplus \{10000011\} = \{11010100\}$$

Notazione esadecimale:

$$\{57\} \oplus \{83\} = \{d4\}$$

Moltiplicazione tra bytes

La moltiplicazione nel campo finito è ottenuta moltiplicando i due polinomi modulo un polinomio irriducibile di ottavo grado, che nell'AES è:

$$m(x) = x^8+x^4+x^2+x+1.$$

Questo polinomio in esadecimale coincide con {01}{1b} (sono necessari due bytes perchè nella conversione che stiamo analizzando con un byte si possono rappresentare al massimo polinomi di settimo grado). Naturalmente $m(x) \notin GF(2^8)$. Questo tipo di moltiplicazione è associativo è c'è un elemento neutro ({01}). Inoltre per qualunque polinomio b(x), diverso da zero e di grado inferiore a 8, si può ricavare l'inverso $b^{-1}(x)$ seguendo questo procedimento:

Effettuo l'algoritmo esteso di Euclide per trovare a(x) e c(x) tali che:

$$b(x)\cdot a(x) + m(x)\cdot c(x) = 1$$

Quindi, essendo:

$$a(x)\cdot b(x) \bmod m(x) = 1$$

Posso ricavare:

$$b^{-1}(x) = a(x) \bmod m(x)$$

Implementazione: purtroppo non esiste un'operazione semplice a livello byte per effettuare questo tipo di calcolo, ma si può realizzare un algoritmo funzionante partendo da un caso particolare, la moltiplicazione per x.

Moltiplicazione per x: quando moltiplichiamo un polinomio $f(x) \in GF(2^8)$ per x ({00000010} o {02}) otteniamo:

$$f(x)\cdot x = p(x) = b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x$$

nel caso $\partial(f) < 7$, p(x) al massimo avrà grado 7, quindi apparterrà a $GF(2^8)$

$$\partial(p) < 8 \Rightarrow p(x) \in GF(2^8)$$

ma se $\partial(f) = 7$ (quindi $b_7 = 1$), $p(x) \notin GF(2^8)$, e bisognerà ridurre p(x) mod m(x). La riduzione può essere effettuata sottraendo p(x) a m(x) (prendiamo per buono questo procedimento di riduzione). Siamo in $\mathbb{Z}_2$, una sottrazione equivale ad una somma, quindi possiamo sommare p(x) a m(x) invece di sottrarli. Sappiamo effettuare una somma tra bytes in $GF(2^8)$ (abbiamo visto precedentemente che equivale ad uno XOR), quindi non ci sono problemi. Il prodotto per x può essere implementato con un left shift (equivale a moltiplicare per {02} il byte) e successivamente, se $b_7 = 1$, con uno XOR ($\oplus$) con {1b} (la rappresentazione esadecimale di m(x)). Chiamiamo la funzione che effettua il prodotto di un polinomio per x (o di un byte per {02}) *xtime()*.

Moltiplicazione tra polinomi: Una volta definita xtime(), possiamo, sfruttando la proprietà distributiva della moltiplicazione, definire un algoritmo che risolva un prodotto tra una qualunque coppia di polinomi (a(x) e b(x)). Ma spieghiamo meglio. Sappiamo effettuare il prodotto di a(x) per x; quindi sappiamo anche calcolare il prodotto a(x) per x^n (basta infatti applicare n volte xtime()). b(x) è una somma di potenze di x, noi sappiamo calcolare il prodotto di a(x) per ogni termine di b(x) preso singolarmente. Ma il prodotto in $GF(2^8)$ è distributivo rispetto alla somma; vediamo come possiamo sfruttare questa importante proprietà a nostro vantaggio. Definiamo a(x) e b(x):

$$a(x) = x^6+x^4+x^2+x+1 = \{57\}$$
$$b(x) = x^4+x+1 = \{13\}$$

Effettuiamo $a(x) \cdot b(x)$:

$$a(x) \bullet b(x) = a(x) \bullet (x^4+x+1) = (a(x) \bullet x^4) + (a(x) \bullet x) + (a(x) \bullet 1)$$

Ci ritroviamo con una somma di termini che sappiamo calcolare (sono prodotti di polinomi per potenze di x), è importante notare come questo tipo di procedimento sia valido per qualunque polinomio a(x) e b(x) in $GF(2^8)$. Bene, vediamo ora come questo algoritmo può essere tradotto a livello byte. Innazitutto calcoliamo i prodotti per le potenze della x che ci servono:

$$a(x) \bullet x^1 = \{57\} \bullet \{02\} = xtime(\{57\}) = \{ae\}$$
$$a(x) \bullet x^2 = \{57\} \bullet \{04\} = xtime(xtime(\{57\})) = xtime(\{ae\}) = \{47\}$$
$$a(x) \bullet x^3 = \{57\} \bullet \{08\} = xtime(\{47\}) = \{8e\}$$
$$a(x) \bullet x^4 = \{57\} \bullet \{10\} = xtime(\{8e\}) = \{07\}$$

Poi effettuiamo i calcoli:

$$\{57\} \bullet \{13\} = \{57\} \bullet (\{10\} \oplus \{02\} \oplus \{01\}) =$$

$$= (\{57\} \bullet \{10\}) \oplus (\{57\} \bullet \{02\}) \oplus (\{57\} \bullet \{01\}) =$$

$$= \{07\} \oplus \{ae\} \oplus \{57\} = \{fe\}.$$

Le word in AES

Nell'AES per i calcoli tra word vengono utilizzati dei polinomi di 4 termini con coefficienti in $GF(2^8)$:

$$[a_3, a_2, a_1, a_0] = a(x) = a_3x^3+a_2x^2+a_1x+a_0$$

Ogni coefficiente a_x viene trattato come un byte e i calcoli tra coefficienti vengono effettuati come spiegato nelle sezioni precedenti.

Somma tra word

Dati due polinomi a(x) e b(x):

$$a(x) = a_3x^3+a_2x^2+a_1x+a_0$$
$$b(x) = b_3x^3+b_2x^2+b_1x+b_0$$

la somma a(x)+b(x) si effettua sommando in $GF(2^8)$ i coefficienti delle x^n con la stessa n:

$$a(x) + b(x) = (a_3 \oplus b_3)x^3 + (a_2 \oplus b_2)x^2 + (a_1 \oplus b_1)x + (a_0 \oplus b_0)$$

Implementazione: data una coppia di word, la somma viene effettuata calcolando lo XOR tra ogni byte della prima word e il corrispettivo della seconda:

$$[a_3, a_2, a_1, a_0] + [b_3, b_2, b_1, b_0] =$$
$$= [(a_3 \oplus b_3), (a_2 \oplus b_2), (a_1 \oplus b_1), (a_0 \oplus b_0)]$$

Prodotto tra word (prodotto modulare)

Dati due polinomi a(x) e b(x):

$$a(x) = a_3x^3+a_2x^2+a_1x+a_0$$
$$b(x) = b_3x^3+b_2x^2+b_1x+b_0$$

il prodotto $a(x)\cdot b(x)$ si effettua calcolando inizialmente il prodotto algebrico tra a(x) e b(x):

$$c(x) = c_6x^6+c_5x^5+c_4x^4+c_3x^3+c_2x^2+c_1x+c_0$$

In cui:

$c_0 = a_0 \bullet b_0$

$c_1 = a_1 \bullet b_0 \oplus a_0 \bullet b_1$

$c_2 = a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2$

$c_3 = a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3$

$c_4 = a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3$

$c_5 = a_3 \bullet b_2 \oplus a_2 \bullet b_3$

$c_6 = a_3 \bullet b_3$

Ora, il nostro risultato deve essere anch'esso rappresentabile con una parola di 4 bytes (dev'essere al massimo un polinomio di terzo grado, con 4 coefficienti), quindi dobbiamo ridurre c(x) modulo un polinomio di quarto grado. Nell'AES, questo polinomio è x^4+1. Quindi, se:

$$x^i \bmod (x^4+1) = x^{i \bmod 4}$$

c(x) diventerà:

$$c(x) = c_6x^2 + c_5x + c_4 + c_3x^3 + c_2x^2 + c_1x + c_0$$

raccogliendo avremo:

$$c(x) = c_3x^3 + (c_2 \oplus c_6)x^2 + (c_1 \oplus c_5)x + (c_0 \oplus c_4)$$

Implementazione: possiamo definire il *"prodotto modulare"* $d(x) = a(x) \otimes b(x)$ come:

$$d(x) = d_3x^3+d_2x^2+d_1x^1+d_0$$

in cui:

$$d_0 = (a_0 \bullet b_0) \oplus (a_3 \bullet b_1) \oplus (a_2 \bullet b_2) \oplus (a_1 \bullet b_3)$$

$$d_1 = (a_1 \bullet b_0) \oplus (a_0 \bullet b_1) \oplus (a_3 \bullet b_2) \oplus (a_2 \bullet b_3)$$

$$d_2 = (a_2 \bullet b_0) \oplus (a_1 \bullet b_1) \oplus (a_0 \bullet b_2) \oplus (a_3 \bullet b_3)$$

$$d_3 = (a_3 \bullet b_0) \oplus (a_2 \bullet b_1) \oplus (a_1 \bullet b_2) \oplus (a_0 \bullet b_3)$$

I coefficienti d_x possono essere scritti in forma matriciale:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Reincontreremo questo prodotto matriciale nella funzione MixColums dell'AES.

Note: x^4+1 non è irriducibile in $\mathbb{Z}_2[x]$, quindi la moltiplicazione per un polinomio di quattro termini fissati non è necessariamente invertibile (questa conclusione va presa per buona); l'AES ha bisogno di un polinomio di 4 termini *invertibile* perchè, mentre il primo andrà utilizzato nella cifratura, l'inverso sarà necessario quando bisognerà decifrare i dati. Si è risolto questo problema specificando un polinomio di quattro termini che ha un'inversa:

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

Infatti:

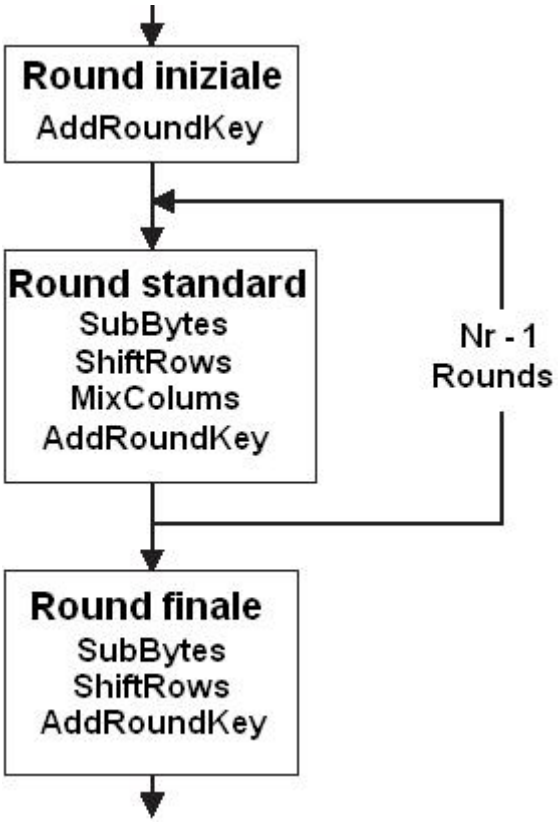
$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}.$$

Algoritmo di cifratura

L'AES è un cifrario a blocchi, ovvero il testo in chiaro, espresso in bit, sarà diviso in blocchi di lunghezza fissa che verranno processati autonomamente e poi riuniti per dare il messaggio cifrato: sulla base delle specifiche del concorso, gli autori di Rijndael decisero che la dimensione dei blocchi del cifrario dovesse essere di 128 bit. La chiave, sempre in base alle specifiche del concorso, può assumere tre lunghezze diverse, ovvero 128, 192 o 256 bit, dando quindi origine a tre versioni diverse dell'algoritmo. L'input e l'output sono sequenze di bit, che nell'implementazione saranno trattati come array, ma durante la cifratura sono considerati logicamente come delle matrici (d'ora in avanti si parlerà di input e di chiave in termini di matrice): gli elementi di queste matrici sono byte. L'unità di elaborazione dell'algoritmo quindi non è tanto il bit quanto il byte, perciò la matrice in input, dato che 128 bit sono 16 byte, sarà una matrice con 16 elementi, ovvero una matrice quadrata 4x4. Allo stesso modo la chiave sarà rappresentata, nelle tre versioni, da matrici 4x4, 4x6 o 4x8: vengono mantenute 4 righe per consentire operazioni con il blocco in input, come si vedrà in seguito. Seguono alcune notazioni che verranno utilizzate:

- Word – Una singola colonna di un blocco, sia di quello in input che di quello della chiave.
- Nb – Numero di word del blocco in input. Nella versione ufficiale di AES, sulla base del fatto che il blocco in input è di 128 bit, si ha Nb=4.
- Nk – Numero di word della chiave. In base a quale delle tre versioni si sta usando, si ha Nk=4,6,8.
- Nr – Numero di round in cui avverrà la codifica.

Si parla di round in quanto il messaggio viene crittato non una, ma più volte, in base alla versione. Ogni singola crittazione del messaggio è considerata un round. Con chiave a 128 bit si hanno 10 round, con chiave a 192 bit 12 round e con chiave a 256 bit 14 round: il numero elevato di round è un grave problema per il crittoanalista in quanto crescono moltissimo le possibili combinazioni di crittazione.
L'algoritmo è formato da due funzioni principali, KeyExpansion e Cipher. La seconda è la funzione che effettivamente codifica il messaggio, mentre la prima è un generatore di chiavi: la crittazione infatti avviene utilizzando una chiave diversa per ogni round, e KeyExpansion fornisce a Cipher tutte le chiavi necessarie generandole a partire da quella principale. Il numero delle chiavi è una più del numero di round Nr, ed in seguito si capirà il motivo. Le operazioni non verranno effettuate sull'input, che all'inizio è un array, ma su una matrice di appoggio detta State, su cui l'input verrà copiato e da cui verrà estratto l'output: la copiatura avverrà per colonne, ovvero i primi 4 byte dell'input formeranno la prima word di State e così via. Ora concentreremo l'attenzione su Cipher, quindi su come il messaggio viene crittato. Cipher implementa gli Nr round: di questi, Nr-1 sono identici, mentre uno è diverso dagli altri. Gli Nr-1 round identici codificano il messaggio attraverso l'applicazione di quattro funzioni in cascata: SubBytes, che opera su ogni byte, ShiftRows, che opera sulle righe di byte, MixColums, che opera sulle word, e AddRoundKey, che critta utilizzando la chiave. Il round rimanente è diverso dagli altri perché sostituisce a MixColums un'altra occorrenza di AddRoundKey, e quindi ha bisogno di due chiavi, inoltre le funzioni vengono applicate in ordine diverso. Il round diverso inizia prima degli altri e termina dopo gli altri, questo è comprensibile osservando lo schema a blocchi dell'algoritmo:



In questo schema il round diverso dagli altri è formato da Round Iniziale, con l'applicazione di AddRoundKey, e Round Finale, con le altre tre funzioni. Da come si evince nello schema a blocchi, per realizzare gli Nr-1 round sarà necessario realizzare un ciclo. Ora che sono definite le operazioni da eseguire è possibile scrivere lo pseudocodice di Cipher:

```
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
```

In input la funzione prende tre array, in, out e w: in è il blocco in input, di dimensioni 4*Nb ovvero 16 byte. Out è il blocco in output, che ha le stesse dimensioni di quello in input e che è ovviamente vuoto visto che dovrà essere scritto al termine della funzione. W è un array di word, ovvero un array di colonne, di dimensione Nb*(Nr+1), cioè quattro colonne per ogni round più uno: la chiave infatti è composta da quattro word, e dato che un round ha bisogno di due chiavi queste saranno Nr+1. W viene passato a Cipher da KeyExpansion, perciò a Cipher non interessa sapere come queste chiavi vengano generate, l'importante è che abbia tutte le chiavi di cui ha bisogno. Le prime operazioni di Cipher sono creare lo State e copiarvi sopra l'input. In seguito vengono implementati i round, allo stesso modo in cui sono descritti nello schema a blocchi: per gli Nr-1 round viene realizzato un ciclo con un'istruzione for. Infine nell'array di output viene copiato lo State e la funzione restituisce il testo cifrato.
Ora vediamo in dettaglio le quattro funzioni che realizzano un round:

SubBytes

Sub sta per substitution, infatti i byte vengono semplicemente sostituiti tramite una funzione che presenta due caratteristiche principali: è invertibile, in quanto è necessario poter realizzare la funzione inversa, ma più importanza ha il fatto che sia non lineare, costituendo un punto di forza di Rijndael, dato che la non-linearità è un grave problema per il crittoanalista. La sostituzione avviene in maniera semplice: esiste una tabella, chiamata Substitution-Box, S-Box, in cui ogni possibile valore di un byte ha un valore corrispondente. La funzione non farà altro che sostituire ad ogni byte il suo corrispettivo presente nella S-Box. La S-Box, dato che un byte può assumere 256 valori, dovrà avere 256 elementi, e quindi sarà una matrice 16x16. Il problema di implementazione sta nel costruire una S-Box in modo che garantisca la non linearità e l'invertibilità: per costruirla si parte da una matrice 16x16 in cui sono scritti ordinatamente tutti i possibili valori di un byte, e ad ogni elemento di questa matrice si applicano queste due trasformazioni:

1. Ad ogni elemento viene sostituito il suo inverso in $GF(2^8)$ (il byte 00000000 rimane tale)
2. Ad ogni bit del byte viene applicata questa trasformazione affine:

   $$b_i = b_i \oplus b_{(i+4)\bmod 8} \oplus b_{(i+5)\bmod 8} \oplus b_{(i+6)\bmod 8} \oplus b_{(i+7)\bmod 8} \oplus c_i$$

   c_i è l'i-esimo bit del byte 01100011, che è un valore fisso utilizzato per ogni byte. In pratica, questa operazione corrisponde a mettere in xor ogni bit con i quattro bit che lo precedono e con un bit esterno: l'operazione è modulo 8 quindi è applicabile anche ai primi bit

Applicando queste due trasformazioni viene realizzata la S-Box, che risulta così costruita:

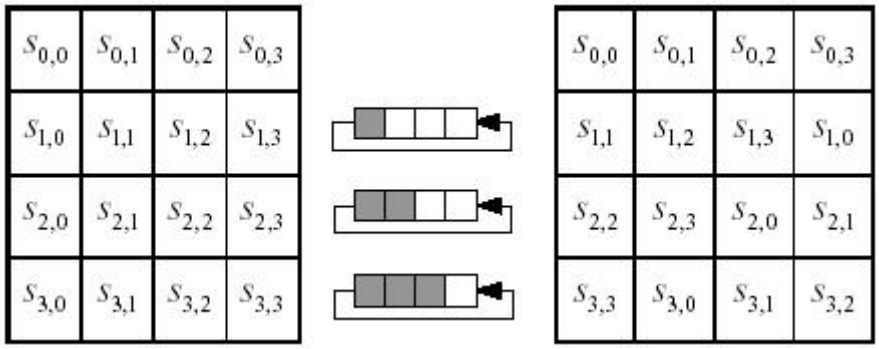
		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	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	4d	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

Qui i valori sono espressi in esadecimale: in questa numerazione un byte è esprimibile con due cifre, in particolare l'insieme x delle righe rappresenta i primi 4 bit, mentre l'insieme y delle colonne rappresenta gli ultimi 4, quelli meno significativi. SubBytes trova i valori in x e y e sostituisce l'elemento corrispondente.

ShiftRows

Questa operazione viene applicata a State dopo MixColums. La funzione shifta le righe di byte secondo una legge data da:

$S'_{r,c} = S_{r,(c-shift(r,Nb))mod\ Nb}$ $0<r<4$, $0\leq c<Nb$ con shift(1,4)=1, shift(2,4)=2, shift(3,4)=3 nella versione a 128 bit. Nelle altre due versioni vi sono altri valori analoghi di shift.



In pratica, la prima riga resta uguale (shift di zero posizioni), la seconda riga si muove verso sinistra di una posizione, la terza riga di due posizioni e la quarta riga di tre posizioni. L'operazione è modulo 4, quindi i byte che muovendosi a sinistra "escono" dalla matrice, rientrano da destra.

MixColums

Questa funzione opera sulle colonne. Ogni byte della matrice viene trattato come un polinomio in $GF(2^8)$, ma anche la singola colonna viene considerata come un polinomio, che ha i byte come coefficienti: una word quindi è un polinomio che ha coefficienti che sono polinomi in $GF(2^8)$. Ogni colonna viene moltiplicata modulo x^4+1 per un polinomio fissato a(x) che è dato da:

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

a(x) ha le stesse caratteristiche delle singole colonne, è un polinomio con coefficienti che sono polinomi in $GF(2^8)$. La moltiplicazione avviene secondo le regole viste nelle basi matematiche, e può essere espressa come segue:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \qquad \text{for } 0 \leq c < Nb$$

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

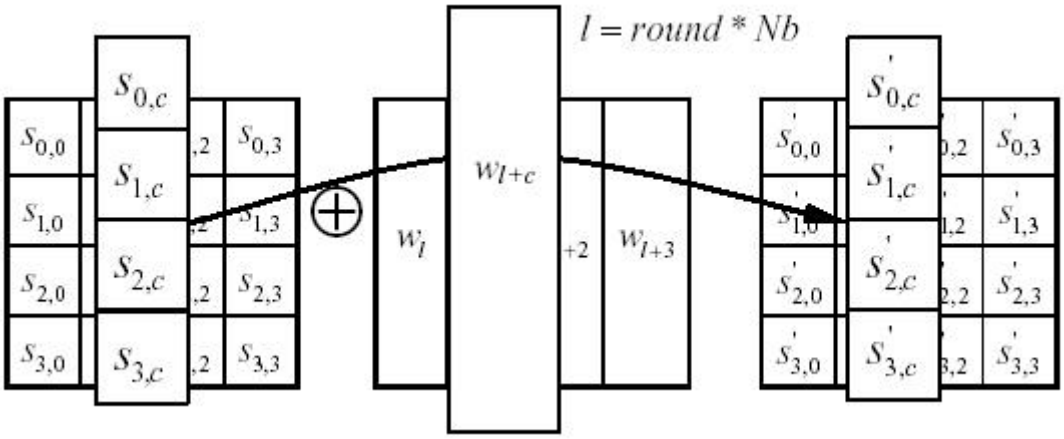
$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

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

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).$$

AddRoundKey

È l'ultima operazione effettuata in un round. Fino ad ora la chiave non era stata utilizzata da alcuna funzione, ed è qui che interviene. Come si può vedere nello pseudocodice, AddRoundKey viene invocata ricevendo in input quattro delle colonne prese dall'array w: sono quelle quattro colonne che formeranno la chiave utilizzata. La chiave viene aggiunta allo State semplicemente con un operazione di xor su ogni byte:



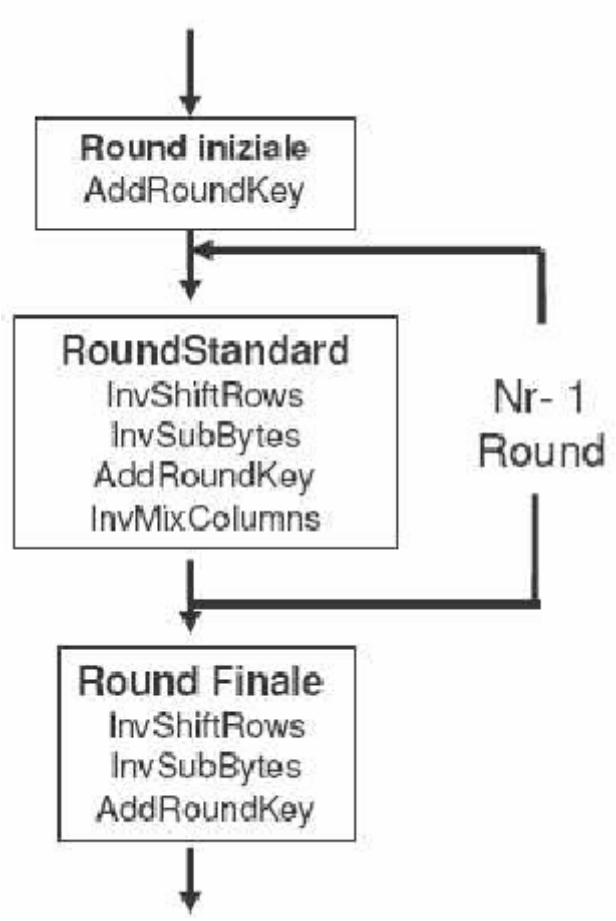
Con l'applicazione di AddRoundKey un singolo round (uno degli Nr-1 round uguali) termina. Alla fine di tutti i round il messaggio sarà effettivamente crittato.

Algoritmo di decifratura

Analogie con l'algoritmo di cifratura

Nell' analisi dell' algoritmo di decifratura di AES, anche detto Inverse Cipher, è facile notare come vengano ripercorse le stesse strutture dell'algoritmo di cifratura. InvCipher infatti, come Cipher opera su box da 128 bit (o 16 byte), quindi l'unità fondamentale di operazione è ancora una volta il byte, e la chiave è la proprietà che distingue le tre versioni dell' algoritmo inverso a seconda se viene utilizzata a 128,192 o 256 bit (standard a 128 bit). Inoltre anche InvCipher distingue i box, considerati come matrici, in box Input/Output e box di Stato, ovvero la matrice che viene modificata dalle iterazioni dell' algoritmo. Anche le unità di riferimento sono le stesse:

- N_b – Numero di word del blocco d' input. Nella versione standard di AES vale 4.
- N_k – Numero di word della chiave.
- N_r – Numero di round.



Un altro elemento invariante è lo schema dell'algoritmo, infatti nonostante le funzioni siano differenti, i round (passi) di InvCipher e Cipher conservano la stessa struttura: InvCipher implementa Nr round. Di questi Nr-1 sono eseguiti ciclicamente, mentre il Round Iniziale e il Round Finale compongono un unico passo che comincia prima e termina solo dopo le iterazioni del ciclo.

Le Funzioni di InvCipher

L' algoritmo è composto da 4 funzioni:

- InvShiftRows()
- InvSubBytes()
- InvMixColumns()
- AddRoundKey()

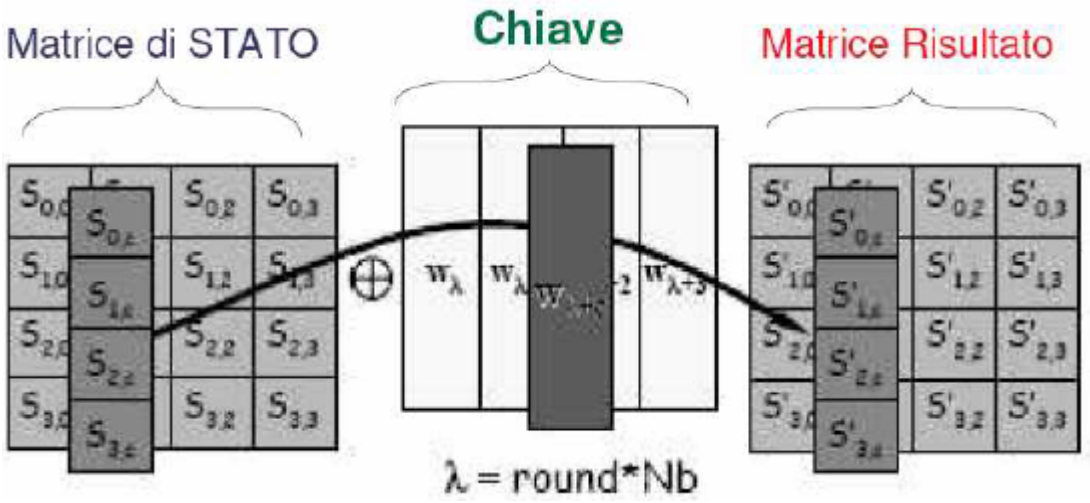
Le prime tre sono le funzioni caratterizzanti l'InvCipher, mentre l'ultima (AddRoundKey) è la stessa incontrata in Cipher. Indicate le funzioni e mostrato l'ordine di esecuzione nel diagramma a blocchi, è possibile analizzare lo pseudo-codice di InvCipher:

```
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
```

Nella prima riga del codice sono definiti gli input della funzione, e sono rispettivamente tre array: in, out e w. In e Out hanno le stesse dimensioni (128 bit) e sono gli array rappresentanti la matrice di Ingresso e la matrice di Uscita. L'array w è invece formato da word, o meglio è un array i cui elementi sono colonne, la dimensione di w, Nb*(Nr+1), e viene passato ad InvCipher dal codice di KeyExpansion che si occupa di fornire le chiavi. Le parti cerchiate indicano come l'algoritmo definisce la matrici di stato e successivamente definisce questa come Input, all' inizio, e come Output, in fondo al codice. Le parti evidenziate da rettangoli rossi servono per rendere evidente l'analogia tra il codice e il diagramma di flusso.

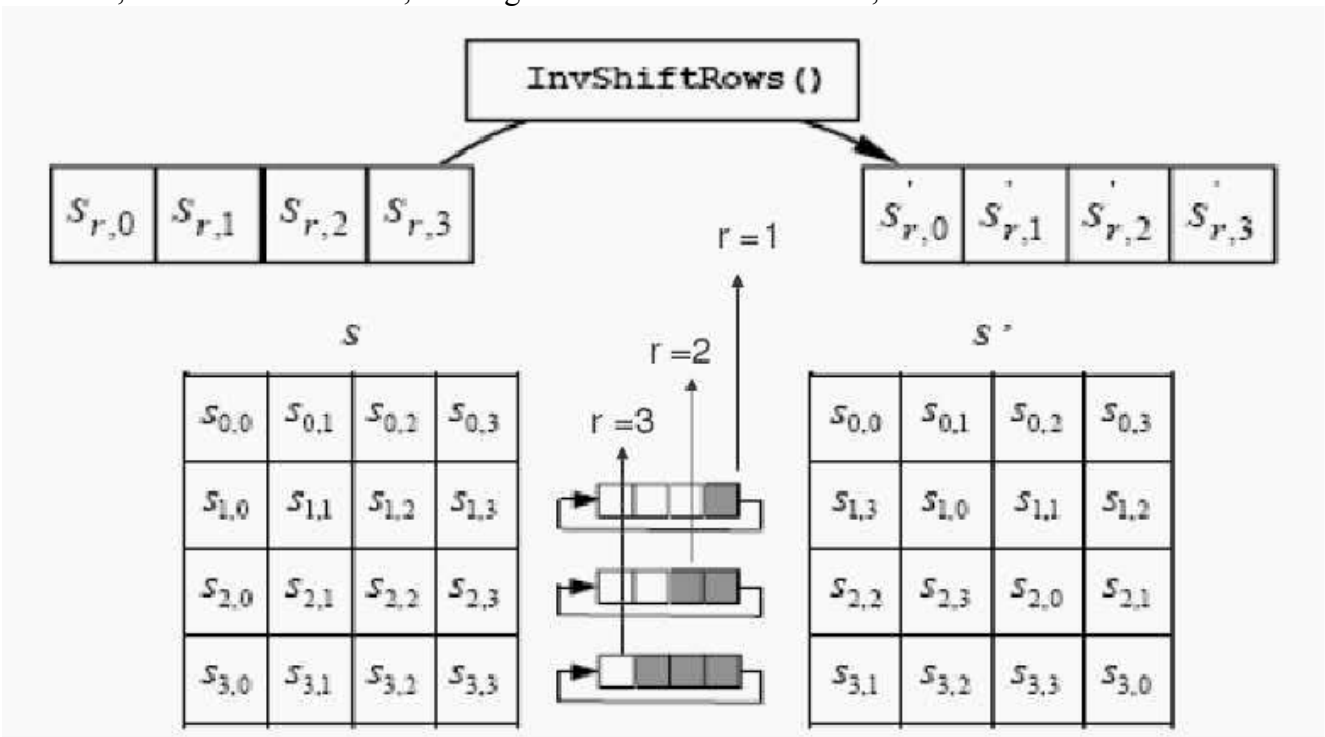
Funzione AddRoundKey()

La funzione AddRoundKey() è la prima ad essere eseguita. Come parametri riceve la matrice di STATO e la Chiave, ad ogni iterazione modificata dal proprio codice. Dal disegno è evidente come AddRoundKey() considera la "STATE" una colonna alla volta e la pone in XOR con una colonna della matrice Chiave, (ricordo che XOR è quell'operazione logica che è verificata, cioè restituisce 1, solo se gli input sono differenti), il risultato è una nuova matrice di STATO.



Funzione InvShiftRows()

InvShiftRows(), a differenza di AddRoundKeuy() opera sulle righe, "shiftandole" di un offset a seconda della riga considerata. Se la riga considerata è la prima, la numero zero il valore di offset r sarà zero, ovvero non cambierà, se la riga considerata è la numero 1, r=1 così fino ad r=3.



Osservando la funzione si nota che non è altro che la stessa di Cipher con il senso dello shift invertito:

$S'_{r,(c+shift(r,Nb))} \bmod Nb = S_{r,c}$ con $0 <= c < Nb$ e $0 < r < 4$

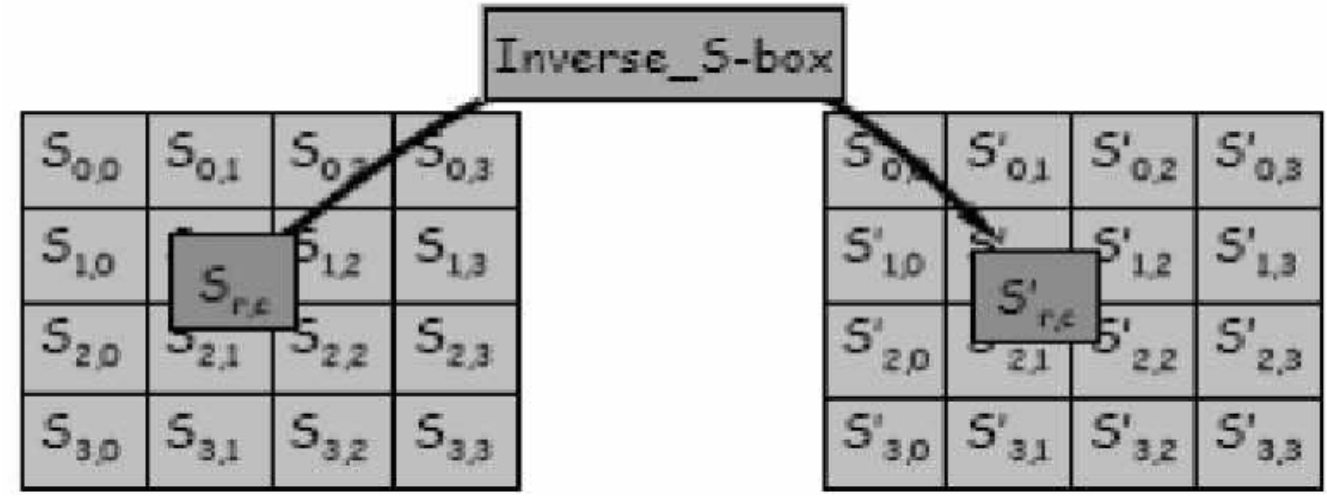
con shift(4,1)=1, shift(4,2)=2, shift(4,3)=3 nella versione a 128 bit.

InvSubBytes()

Per poter analizzare questa funzione è necessario ricordare alcuni aspetti relativi alla matrice di stato. In STATE i valori dei singoli byte sono rappresentati da un numero esadecimale di due cifre. Queste due cifre indicano rispettivamente un ascissa (Y) e un ordinata (X) di una tabella di sostituzione. Questa tabella è chiamata Inverse S_Box, i valori al proprio interno sono definiti con la stessa logica con cui sono definiti quelli all'interno della S_Box di Cipher :

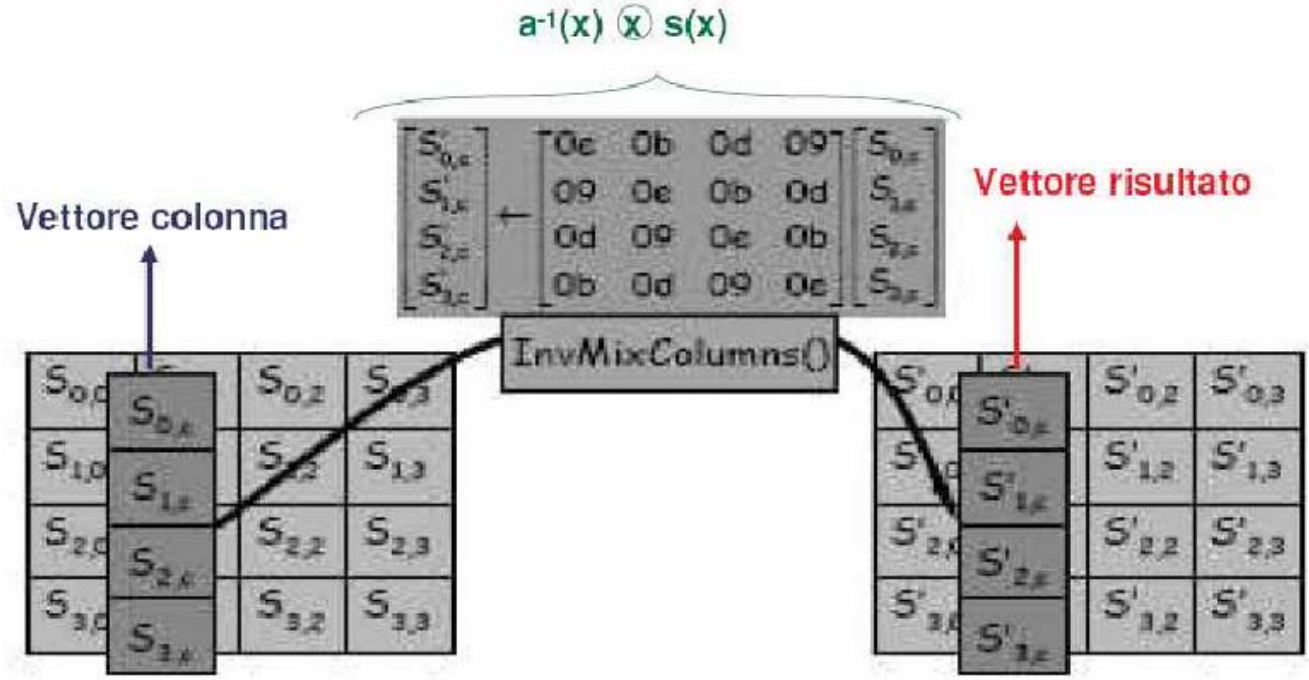
x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	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

Ipotizzando che un valore generico della matrice di stato $S_{r,c}$ sia '5°', la cifra '5' rappresenta il valore scelto tra le X della tabella, mentre 'a' è la cifra scelta tra le Y della InvS_Box. Incrociando i valori sullo schema, si determina un nuovo numero esadecimale che sostituirà il precedente in $S'_{r,c}$.



InvMixColumns()

InvMixColumns() come l'analoga funzione in Cipher opera sulle colonne. Ogni singolo byte della matrice di stato viene definito come un polinomio in $GF(2^8)$, il vettore colonna è quindi rappresentabile come un polinomio di 4° grado con coefficienti in $GF(2^8)$, cioè i cui coefficienti sono a loro volta polinomi in $GF(2^8)$.



La funzione utilizza una particolare operazione matematica chiamata prodotto modulare, che consiste nella moltiplicazione del vettore colonna per un polinomio di terzo grado fissato, tutto riportato in modulo x4+1. Il polinomio fissato è:

$$a^{-1}(x) = \text{'0b'}x3 + \text{'0d'}x2 + \text{'09'}x + \text{'0e'}$$

Sia s(x) il vettore colonna con coefficienti in GF(28) della matrice di stato, allora il nuovo vettore s'(x) può essere così definito:

$$s'(x) = a^{-1}(x) \otimes s(x)$$

Ricordando le proprietà matematiche del prodotto modulare si osserva come è possibile ottenere una rappresentazione matriciale:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

quindi, quella che poteva sembrare un'operazione computazionalmente complessa si è rivelata come un prodotto tra matrici.

Cifrario Inverso Equivalente

A differenza dell' algoritmo di cifratura InvCipher ha la caratteristica di poter invertire tra loro alcune funzioni:

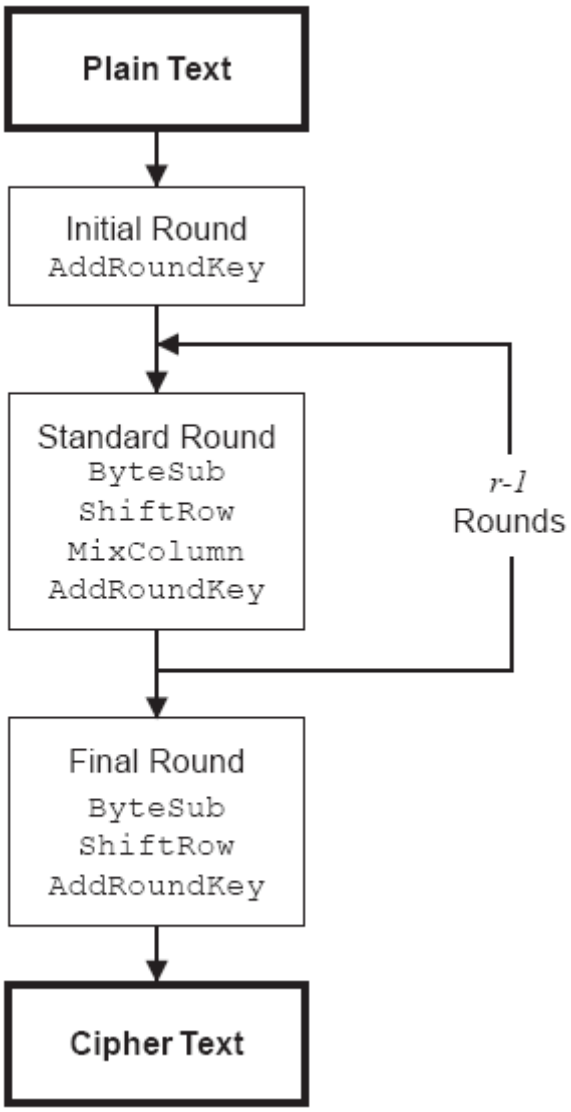
- La prima proprietà riguarda l' inversione della funzione di InvShiftRows() con InvSubBytes(). Nel codice visto in precedenza (nel riquadro blu), InvShiftRows() può essere eseguita dopo InvSubBytes() senza bisogno di modificare il codice, questo perché le due funzioni ricevono in ingresso solamente la matrice di stato;
- Anche InvMixColums() e AddRoundKey() possono essere invertite tra loro, ma Questa volta è necessario modificare il codice, poiché la funzione AddRoundKey() riceve in ingresso la chiave, che essendo schedulata ad ogni iterazione varia se considerata un round prima;

Nel codice viene così implementato un nuovo array di word dw[],contenente la nuova chiave di decifrazione schedulata, al posto del precedente w. Il codice quindi diviene:

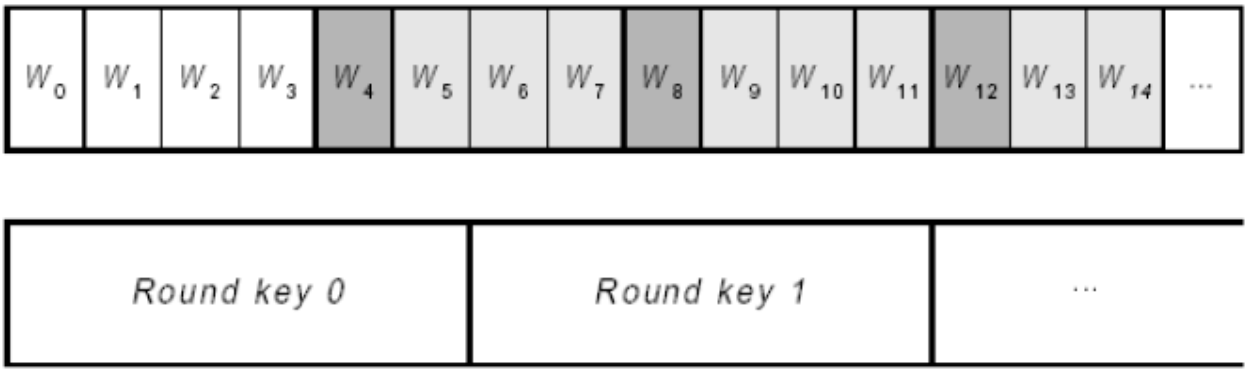
```
EqInvCipher(byte in[4*Nb], byte out[4*Nb], word dw[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   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
```

Key Schedule

La fase dell'algoritmo indicata solitamente con "Key Schedule", è la parte di AES che definisce l'utilizzo della chiave di cifratura.
Delle quattro operazioni di base che compongono un round di cifratura, soltanto l'operazione AddRoundKey implica l'utilizzo della chiave di cifratura. Essa infatti effettua uno XOR bit a bit tra le colonne della matrice State e quelle di un'altra matrice indicata come "Round Key", diversa ad ogni applicazione della AddRoundKey.
Le Round Keys non sono altro che una serie di matrici formate da un numero di parole di 4 byte pari al valore di Nb, ovvero al numero di colonne (e quindi di parole di 4 byte) che compongono un blocco di ciffratura (nello standard AES, che prevede solo blocchi di 128 bit, Nb è sempre uguale a 4). Le Round Keys sono ottenute dalla chiave di cifratura tramite l'operazione di "Key Expansion". Questa operazione produce la cosiddetta "Expanded Key", ovvero un array lineare, identificato con W, contenente tutte le Round Keys che verranno utilizzate nel corso dell'algoritmo.
Il numero di parole di cui è composta la Expanded Key è dovuto alla configurazione dell'algoritmo: come è stato visto in precedenza, infatti, ad ogni esecuzione dell'operazione AddRoundKey vengono "consumate" un numero di parole pari a Nb. Inoltre, come si può vedere dal diagramma di flusso dell'algoritmo di cifratura, il numero di applicazioni della AddRoundKey è pari a Nr + 1.



Perciò l'array W conterrà un numero di parole pari a Nb*(Nr+1). Nel caso classico di blocchi e chiave da 128 bit si avranno quindi 4*(10 + 1) = 44 parole di 4 byte.
La i-esima Round Key, ovvero la round key utilizzata dall'i-esima applicazione della AddRoundKey è data dalle parole che vanno da W[Nb*i] a W[Nb*(i+1)]. In figura è illustrato un esempio della selezione della Round Key in rapporto all'array W in un caso che, pur essendo possibile dal punto di vista di Rijndael, non è in realtà previsto dallo standard AES, ovvero il caso in cui i blocchi sono di 192 bit, e quindi con Nb pari a 6.



La costruzione dell'array W viene normalmente eseguita prima dell'esecuzione dell'algoritmo di cifratura, fornendolo quindi come parametro della funzione Cipher. E' tuttavia possibile un'implementazione in cui le Round Key vengono costruite ad ogni passo quando necessario.
Di seguito viene mostrato lo pseudo codice del primo caso.

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

La funzione di Key Expansion prende come parametro la chiave di cifratura, il valore Nk, ovvero il numero di parole di 4 byte che compongono la chiave stessa e un riferimento all'array che dovrà contenere le Round Keys.
Come si può vedere nel primo ciclo **`while`** le prime Nk posizioni di W vengono riempite con le Nk parole che compongono la chiave di cifratura. Dopo questa fase si entra in un ulteriore ciclo **`while`** orientato alla parola destinato a riempire le rimanenti posizioni di W, ovvero quelle che vanno dalla posizione Nk alla posizione Nb*(Nr +1) – 1. La generazione dell'i-esima parola dipende dall'indice **`i`** stesso. In particolare si ha:

```
W[i] = W[i - Nk] ⊕ temp
```

con temp che sarà il risultato di una diversa serie di operazioni a seconda dell'indice **`i`**, applicate alla parola della posizione precedente ovvero **`W[i – 1].`**
In particolare si hanno tre possibili casi:

- se l'indice **`i`** è un multiplo di Nk, ovvero **`i mod Nk = 0`**
  - **`temp = SubWord(RotWord(W[i – 1]) ⊕ Rcon[i / Nk]`**
- se l'indice **`i mod Nk = 4`** e **`Nk = 8`**
  - **`temp = SubWord(W[i – 1])`**
- altrimenti
  - **`temp = W[i – 1]`**

Le funzioni applicate hanno la seguente semantica:

- SubWord: questa funzione prende in input un parole di 4 byte restituisce in output un'altra parole di 4 byte ottenuta applicando la S-box ad ogni singolo byte della parole di input
- RotWord: data un parole di 4 byte in input nella forma [a0,a1,a2,a3], la RotWord applica uno shift di una posizione a sinistra ai byte della parola, restituendo in output la parola [a1,a2,a3,a0]
- Rcon[k]: Rcon è un elemento di 4 byte che ha una struttura fissata del tipo [x^(i – 1),{00},{00},{00}], dove x è un elemento di GF(2^8), {02} in esadecimale. Il primo byte dell'elemento Rcon quindi, essendo una potenza di x che dipende dall'indice i, procederà nel seguente modo: {01}, {02}, {04}, {08}, {10}, …

La funzione di espansione della chiave è stata progettata per resistere ad attacchi tipo:

- attacchi in cui il crittoanalista conosce una parte della chiave di cifratura
- attacchi in cui la chiave è interamente conosciuta o può essere scelta dal crittoanalista
- attacchi related-key, ovvero in cui il crittoanalista è in possesso di testi cifrati con più chiavi di cifratura, di cui conosce solo le differenze (ad esempio sa che due chiavi di cifratura differiscono solo in un bit).

In particolare una condizione necessaria per resistere ad attacchi di tipo related key è quella di non avere chiavi di cifratura differenti che producano un certo numero di round key uguali.
L'espansione della chiave gioca inoltre un ruolo molto importante nell'eliminazione della simmetria introdotta dalla cifratura:

- simmetria all'interno dei round di trasformazione: tutti i byte dello state vengono trattati in maniera simile nel corso delle operazioni di un round
- simmetria tra i round: le trasformazioni applicate durante un round sono le stesse per tutti i round

Sicurezza ed attacchi

AES è attualmente lo standard di crittazione utilizzato dal governo degli USA per proteggere documenti classificati SECRET, crittati con chiave a 128 bit, e TOP SECRET, crittati con chiave a 192 o 256. Infatti ad oggi non sono noti attacchi ad AES andati a buon fine né è stata evidenziata alcuna chiave debole o semi debole come per il DES. Gli unici successi sono stati ottenuti da attacchi "parziali", ovvero portati ad implementazioni di AES con un numero di round inferiore a quello previsto dallo standard. I risultati migliori sono stati i seguenti:

- (2000) Ferguson et al.: AES a 7 round su chiave a 128 bit e AES a 9 round su chiave a 256 bit tramite l'attacco Square
- (2003) Jakimoski et al.: AES a 8 round su chiave a 192 bit tramite un attacco di crittoanalisi differenziale basato sulle chiavi

In particolare l'attacco Square è una tipologia di attacco che Rijndael ha ereditato dal suo predecessore, l'algoritmo di cifratura Square appunto, e che è stato documentato dagli stessi autori al momento della presentazione di Rijndael al concorso del NIST. L'attacco Square è un attacco di tipo chosen plaintext, ovvero in cui il crittoanalista può scegliere una serie di testi in chiaro e i rispettivi testi cifrati per poi risalire alla chiave studiando le differenze prodotte dalla cifratura dei diversi testi. Esso si basa sulla struttura orientata al byte di Rijndael, utilizzando degli insiemi di state tutti uguali tranne che per un byte, detto "attivo". Seguendo l'evoluzione della posizione e delle trasformazioni di tale byte, l'algoritmo è in grado di eliminare alcuni valori possibili della chiave riducendo lo spazio di ricerca. L'attacco Square pur essendo valido, rimane comunque un attacco parziale e necessita di un numero di plain text pari a 2^32.
Alcuni crittografi hanno invece avanzato dubbi sulla sicurezza dell'AES a causa della sua struttura matematica. Infatti a differenza di altri algoritmi, AES possiede una struttura molto semplice e molto ben definita e documentata.
Proprio su questo spunto si basa una nuova tipologia di attacco che ha suscitato grande clamore in relazione ad AES, ovvero l'attacco chiamato XSL. L'attacco XSL è un attacco di tipo algebrico per il key recovery virtualmente applicabile a quasi tutti i principali algoritmi di cifratura a blocchi, quindi non soltanto ad AES ma anche ad esempio a SERPENT, ovvero un altro dei cinque algoritmi finalisti del concorso del NIST e in particolare quello riconosciuto come il più sicuro. In particolare XSL sfrutta la struttura matematica di AES, riducendo il problema di recuperare la chiave di cifratura al problema di risolvere un sistema di equazioni quadratiche multivariate, un sistema che risulta inoltre essere particolarmente sparso. Secondo uno studio del 2002 combinando la tecnica XSL ad un sistema per la rappresentazione concisa di AES (BES), si può arrivare ad una complessità di 2^100 operazioni.
Comunque al di là del clamore suscitato dagli studi che hanno proposto l'attacco XSL, sono stati avanzati numerosi dubbi sull'efficacia di tale tecnica. Inoltre, pur essendo teoricamente corretta, non è al momento implementabile né si sa se lo sarà mai.
Ad oggi quindi, considerando che la ricerca esaustiva dovrebbe operare su un numero di possibilità pari a 3,4*10^38 su chiave a 128 bit, l'AES risulta essere inviolato e rappresenta ancora una delle scelte più sicure nel panorama degli algoritmi di crittografia a chiave privata.