Advanced Encryption Standard (AES)

Introduzione

Per realizzare degli strumenti complicati, prima sono stati realizzati degli strumenti semplici.

Quello che vorremmo è:

- che la soluzione sia efficiente da un punto di vista pratico;
- che ogni chiave possa essere riutilizzata.

Definizione di Cifrario

Un algoritmo di cifratura a blocchi è un algoritmo a chiave simmetrica che opera su gruppi di bit di lunghezza fissa, detto quindi blocco. Le operazioni di cifratura interessano un blocco come unità atomica e non i singoli bit.

Un cifrario a blocchi può essere rappresentato dalla seguente funzione:

$$E : \{0,1\}^k \times \{0,1\}^l \to \{0,1\}^l$$

Quindi scriviamo:

$$E(k, m) = c$$

Dove k indica la chiave, m il messaggio e c il cripto-testo.

Spesso è possibile trovare una scrittura così fatta:

$$E_k(m)$$

Che indica che la chiave non viene cambiate durante la cifratura di diversi messaggi.

I cifrari a blocchi rappresentano i pilastri della crittografia, ma dire che un cifrario a blocchi sia sicuro non è banale.

DES - Data Encryption Standard

In crittografia il Data Encryption Standard (DES, letteralmente "Norma per la crittografia dei dati") è un algoritmo di cifratura scelto come standard dal Federal Information Processing Standard (FIPS) per il governo degli Stati Uniti d'America nel 1976 e in seguito diventato di utilizzo internazionale. Si basa su un algoritmo a chiave simmetrica con chiave a 64 bit (ma solo 56 utili poiché 8 sono di controllo).
Questo algoritmo all'inizio ha suscitato molte discussioni per via della sua chiave di cifratura corta e per via di alcune scelte progettuali che erano segretate.
Attualmente DES è considerato insicuro per moltissime applicazioni. La sua insicurezza deriva dalla chiave utilizzata per cifrare i messaggi, che è di soli 56 bit.

Quando si è deciso di sostituire DES, questo era ancora sicuro. Ma il problema della sicurezza di un cifrario a blocchi è legato anche alla dimensione del blocco, infatti, può essere rotto in tempo $2^{n/2}$ ed essendo che il DES ha una dimensione del blocco pari a $n = 64$ con le tecnologie moderne può essere facilmente rotto. Inoltre, il DES è nato per essere veloce in hardware e aveva problemi di velocità in software. Quindi si cercava un nuovo cifrario che migliorava DES, ovvero:

- abbia dimensioni del blocco e della chiave più grandi rispetto al DES;
- sia più veloce del DES.

Un po' di storia

Nel 1998 il National Institute of Standards and Technology (NIST) annuncia una competizione per un nuovo cifrario a blocchi. A differenza del DES, il nuovo cifrario a blocchi deve essere costruito dal "pubblico".
Dopo 15 sottomissioni nel 2001 la scelta cade sull'algoritmo di **Rijandael**, che deriva dal nome degli sviluppatori: *Vincent Rihmen* e *Joan Daemen*.

Questo algoritmo venne successivamente chiamato **AES - Advanced Encryption Standard**.

Qualche informazione su AES

- La lunghezza del blocco è $n = 128$.
- La dimensione della chiave è variabile: 128, 192 o 256 bit. Per questo motivo lo standard specifica tre cifrari a blocchi differenti: AES128, AES192 e AES256.
- Con AES si indica l'algoritmo AES128.
- AES è il principale cifrario a blocchi per molte applicazioni commerciali.
- Nel 2003 l'NSA ha annunciato che avrebbe utilizzato AES per cifrare documenti classificati:
    - Fino al livello SECRET per ogni dimensione della chiave;
    - TOP SECRET per chiavi di 192 o 256 bit.
    - Nessun algoritmo pubblico era mai stato utilizzato per questi scopi.

Breve descrizione di AES

AES si compone di tre livelli fondamentali (layers), ogni layer manipola tutti i bit dello stato.

- **Key Addition Layer**: in cui lo stato attuale viene aggiornato usando una chiave di round di 128 bit.
- **Byte Substitution Layer (S-box)**: lo stato viene modificato utilizzando una trasformazione non lineare.
- **Diffusion Layer**: "Diffonde" le trasformazioni effettuate su tutti i bit dello stato. Consiste di due procedure, entrambe effettuano operazioni lineari.

Algoritmo AES

```
AES_k(M) { // |M| = 128 e |K| = 128
    (K_0, ..., k_10) <- Espandi_Chiave(K) // |K_i| = 128
    s <- M XOR K_0 // s è chiamato lo stato
    for r = 1 to 10 do:
        s <- S(s)
        s <- Shift_Rows(s)
        if r <= 9 then mix_cols(s)
        s <- s XOR K_r
    return s
}
```

Il valore **s** è chiamato stato. Tale valore è inizializzato a quello del messaggio M. Lo stato finale è il cripto-testo C. Lo stato viene cambiato ad ogni iterazione del ciclo *for* che è chiamato round. AES consiste di 10 round.

Descriviamo AES in termini delle 4 funzioni:

- Espandi_Chiave;
- S;
- Shift_Rows;
- Mix_Cols.

La funzione Espandi_Chiave

E' la funzione di espansione della chiave. Prende in input una stringa di 128 bit e produce 11 sotto chiavi di 128 bit ciascuna.

Le restanti tre funzioni mappano biettivamente 128 bit a 128 bit. In realtà, saremo più generali per S, lasciando che sia una mappa su $(((\{0,1\})^8)^+$.

La funzione S-Box

Questa funzione rappresenta il cuore pulsante di AES. Per capire cosa succede all'interno di questa funzione è necessario un ripasso di algebra.

Descriviamo un modo per fare calcoli sui byte. Identifichiamo ogni byte:

$$a = a_7a_6a_5a_4a_3a_2a_1a_0$$

con il polinomio formale:

$$a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0$$

Supponiamo che l'insieme di tutti i possibili byte possano essere rappresentati con la configurazione appena vista sopra e che questi appartengano ad un insieme A. Quello che si vuole fare è definire delle operazioni su questo insieme affinché si abbia una struttura chiamata **campo**.

Creiamo un'operazione che, innanzitutto, permetta la somma tra due byte. Questa nuova operazione verrà rappresentata dal simbolo $\oplus$, che è uguale alla somma mod-2 dei polinomi corrispondenti.

Esempio:

Supponiamo di avere due byte, rappresentati utilizzando i polinomi corrispondenti dell'insieme A:

$$a = x^7 + x^6 + x^3 + x^2 + 1$$
$$b = x^7 + x^5 + x^3 + x + 1$$

Sommiamo i due polinomi usando la nuova funzione di somma appena definita:

$$2x^7 + x^6 + x^5 + 2x^3 + x^2 + x + 2 =$$
$$= x^6 + x^5 + x^2 + x$$

Se si parlasse di byte l'operazione appena definita corrisponderebbe all'XOR.

A questo punto bisogna inventarsi una nuova operazione tipo la moltiplicazione, ricordando che il grado del polinomo deve essere minore o uguale a 7. Per fare ciò due polinomi possono essere moltiplicati tra di loro, in maniera tradizionale, ottenendo così un polinomio di grado 14 (o meno), e quindi successivamente prendere il resto della divisione di questo polinomio per un polinomio **irriducibile fisso**:

$$m = x^8 + x^4 + x^3 + x + 1$$

Questo è un polinomio di grado al massimo sette che, come prima, può essere considerato come un byte. In questo modo, possiamo aggiungere e moltiplicare due byte qualsiasi.

Esempio:

Consideriamo i seguenti polinomi:

$$a = x^7 + x^5 + x^4 + x + 1$$
$$b = x^5 + x^3 + x^2 + x$$
$$a \cdot b = x^{12} + x^6 + x^4 + x$$

Il risultato della moltiplicazione deve essere diviso per m e il resto rappresenta il risultato dell'operazione di moltiplicazione nell'insieme A.

$$x^{12} + x^6 + x^4 + x / x^8 + x^4 + x^3 + x + 1 = x^4 + 1$$
$$Resto = x^7 + x^6 + x^5 + x^4 + x^3 + x$$

Osserviamo che il $Resto$ è ancora un elemento dell'insieme A.

La struttura algebrica risultante ha tutte le proprietà necessarie per essere chiamata **campo finito**. In particolare, questa è una rappresentazione del campo finito noto come $GF(2^8)$ - il campo di Galois su $2^8 = 256$ punti. Come campo finito, si può trovare l'inverso di qualsiasi punto di campo diverso da zero (l'elemento zero è il byte zero) e si può distribuire l'addizione sulla

moltiplicazione.

Ci sono alcuni trucchi utili quando si vogliono moltiplicare due byte. Poiché m è un altro nome per **zero**, $x^8 = x^4 + x^3 + x + 1 = \{1b\}$, qui le parentesi graffe indicano un numero esadecimale. Quindi è facile moltiplicare un byte a per il byte $x = \{02\}$: cioè, spostare il byte a di una posizione a sinistra, lasciando che il primo bit "cade" e spostando uno zero nella posizione dell'ultimo bit. Adesso:

- se il primo bit che abbiamo eliminato era 0, l'operazione si conclude;
- se il primo bit che abbiamo eliminato era 1, bisogna aggiungere (con l'operazione di xor) $x^8 = 1b$.

In sintesi, per un byte a, $a \cdot x = a \cdot \{02\}$ se il primo bit di a è zero, altrimenti $a \cdot x = (a \cdot \{02\}) \oplus \{1b\}$, perché in questo caso l'operazione di moltiplicazione mi porterebbe fuori dal range dell'insieme A.

Ha senso considerare solo $\{1b\}$ perché nel momento in cui si esce fuori dal range dei possibili valori di A con lo shift, il bit a 1 viene eliminato, cosa che succederebbe anche nel caso in cui si effettuasse la divisione per m, quindi bisogna semplicemente sommare, con l'XOR, al risultato dell'operazione di shift il valore $\{1b\} = x^4 + x^3 + x + 1$.

Le operazioni consistono quindi solo di *shift* e *xor*, che sono due operazioni efficienti. Nonostante questo, eseguire queste operazioni per un certo numero di volte, potrebbero appesantire il sistema, rendendo l'algoritmo "lento". Si vogliono quindi effettuare delle pre-computazioni su uno spazio extra, in maniera da velocizzare le operazioni descritte precedentemente. Però, bisogna fare attenzione alle parti dove è necessario usare chiavi segrete.

Come detto prima, ogni byte a diverso da zero ha un inverso moltiplicativo, $inv(a) = a^{-1}$. La mappatura indicata con:

$$S : \{0,1\}^8 \to \{0,1\}^8$$

si ottiene da:

$$inv : a \to a^{-1}$$

Per prima cosa, si modifica la mappatura per renderla totale su $\{0,1\}^8$ settando che $inv(\{00\}) = \{00\}$.

Tornando alla funzione S-Box, questa consiste in un primo step in cui viene calcolato il polinomio inverso corrispondente al byte di input, ed un secondo step dove vengono applicate delle trasformazioni lineari. Allora, per calcolare $S(x)$, prima si sostituisce x con $x = inv(x)$ e ritorna x' così fatto:

$$\boldsymbol{x'_0 x'_1 x'_2 x'_3 x'_4 x'_5 x'_6 x'_7} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Tutta l'aritmetica è in GF(2), questo significa che le addizioni di bit è il loro XOR e la moltiplicazione di bit è la loro congiunzione AND.

Quindi si definisce $S : \{0,1\}^8 \to \{0,1\}^8$ con la seguente procedura:

```
S(x) {
    x <- inv(x)
    x' <- AFF(x)
    return x'
}
```

Ora che sappiamo come calcolare la funzione di base, estendiamo la funzione S in modo che possa prendere input più grandi. Data una stringa di m byte $A = A[1] \dots A[m]$, poniamo:

$$S(A) = S(A[1]) \dots S(A[m])$$

Dunque S prende in input uno stato s (16 byte) e lo modifica secondo le procedure descritte.

Calcolo dell'inverso

L'inverso per un elemento a su un campo finito può essere calcolato in differenti modi, uno di questi è quello di moltiplicare a per ogni numero del campo finché non si ottiene 1. Questa è una ricerca **brute-force**.

Quando si sviluppano algoritmi per il calcolo del campo di Galois su piccoli campi di Galois, un approccio comune all'ottimizzazione delle prestazioni consiste nel trovare un **generatore** g.

Chiamiamo **generatori** del nostro campo finito quei polinomi che, moltiplicati tante volte per se stessi, riescono a generare l'intero dominio, che ricordiamo essere limitato a 255 polinomi totali. Tra la vastità di generatori ne scegliamo uno e lo chiamiamo g. Possiamo dedurre che $g^{255} = 1$ da qualche osservazione: g^n dove $n = 0, 1, \dots, 255$ deve necessariamente riprodurre tutti i possibili polinomi poiché g è un generatore. Questo avviene in maniera ciclica, per cui ad un certo punto, moltiplicando g per se stesso, otterremo nuovamente il valore di partenza ed il ciclo ricomincerà. Se per assurdo $g^{255} \neq 1$, allora vi sarebbero conseguenze nel ragionamento:

- Se l'1 si trovasse prima nella sequenza delle potenze, allora g non potrebbe produrre tutti i 256 polinomi, poiché il ciclo ricomincerebbe nuovamente.
- Se l'1 si trovasse dopo, allora g avrebbe prodotto lo stesso polinomio più volte durante il ciclo.

Per cui dimostriamo in maniera informale che $g^{255} = 1$.

I generatori di Rijandael sono i seguenti:

3 5 6 9 11 14 17 18 19 20 23 24 25 26 28 30 31 33 34 35 39 40 42 44 48 49 60
62 63 65 69 70 71 72 73 75 76 78 79 82 84 86 87 88 89 90 91 95 100 101 104
105 109 110 112 113 118 119 121 122 123 126 129 132 134 135 136 138 142
143 144 147 149 150 152 153 155 157 160 164 165 166 167 169 170 172 173
178 180 183 184 185 186 190 191 192 193 196 200 201 206 207 208 214 215
218 220 221 222 226 227 229 230 231 233 234 235 238 240 241 244 245 246
248 251 253 254 255

- Oppure, in esadecimale

03 05 06 09 0b 0e 11 12 13 14 17 18 19 1a 1c 1e 1f 21 22 23 27 28 2a 2c 30 31
3c 3e 3f 41 45 46 47 48 49 4b 4c 4e 4f 52 54 56 57 58 59 5a 5b 5f 64 65 68 69
6d 6e 70 71 76 77 79 7a 7b 7e 81 84 86 87 88 8a 8e 8f 90 93 95 96 98 99 9b 9d
a0 a4 a5 a6 a7 a9 aa ac ad b2 b4 b7 b8 b9 ba be bf c0 c1 c4 c8 c9 ce cf d0 d6
d7 da dc dd de e2 e3 e5 e6 e7 e9 ea eb ee f0 f1 f4 f5 f6 f8 fb fd fe ff

Preso un generatore g, è possibile calcolare una **look-up table** andando a considerare come chiave il polinomio a e come valore la potenza b : $g^b = a$. Utilizzando la tavola calcoliamo $b = log_g a$, poniamo $c = 255 - b$ e l'inverso cercato è:

$$c = 255 - b = 255 - log_g a \rightarrow g^c = g^{255-b}$$

Se $c = b$, allora:

$$a \cdot a^{-1} = g^b \cdot g^{255-b} = g^{255} = 1$$

Fare l'inverso si riduce quindi ad effettuare una sottrazione dato che effettuare $log_g a$ si riduce a trovare il risultato su una tabella pre-computata.

Ottimizzazioni

Avendo solo 256 possibili polinomi, potremmo calcolare $S(0), \ldots, S(255)$ a priori ed inserirli in una tabella, così da ridurre la funzione S ad una look-up in tempo costante. L'unico drawback di questo metodo è il dover archiviare da qualche parte tale tabella ed assicurarsi che essa non venga manomessa.

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

Le funzioni shift-rows e mix-cols

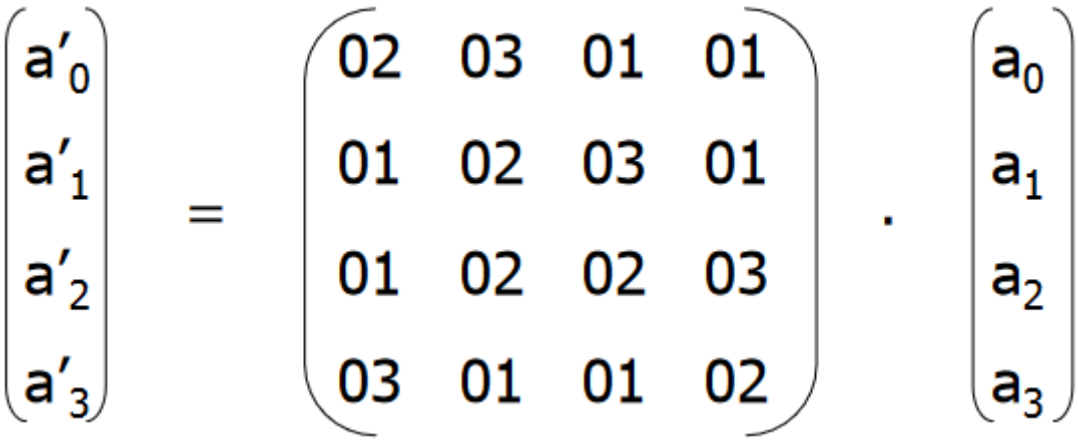
Si dispongano i 16 byte dello stato s come una matrice 4×4:

$$\begin{matrix} S_0 & S_4 & S_8 & S_{12} \\ S_1 & S_5 & S_9 & S_{13} \\ S_2 & S_6 & S_{10} & S_{14} \\ S_3 & S_7 & S_{11} & S_{15} \end{matrix}$$

La funzione shift-rows consiste nei seguenti step:

- Nessuna azione sulla prima riga;
- 1 shift circolare della seconda riga;
- 2 shift circolari della terza riga;
- 3 shift circolari della quarta riga.

La funzione mix-cols prende gruppi da 4 byte e applica la seguente funzione:



Osservazioni

Sorprendentemente AES non contiene **feistel rounds**. Inoltre si hanno più informazioni del DES. La sicurezza dei sistemi a blocchi considerati affidabili, tuttavia si basa sul fatto che non siamo riusciti a romperli.

Sicurezza basata su Key Recovery

Fino ad ora abbiamo analizzato la sicurezza dei sistemi a blocchi carcando di curare la seguente domanda: date q coppie (messaggio, crittotesto), quanto è difficile trovare la chiave? Abbiamo considerato sicuro un sistema a blocchi capace di resistere ad un tale attacco (**security against key recovery**).

Consideriamo $c = E_k(m)$ una cifratura del messaggio m sotto la chiave k. Per un avversario in possesso di c dovrebbe risultare infattibile trovare interamente (o in parte) m. La security against key recovery è sicuramente una condizione necessaria, poiché se l'avversario riuscisse a trovare k, allora troverebbe anche m. Non è tuttavia una condizione sufficiente.

Supponiamo che vi sia un cifrario $E : \{0,1\}^{128} \times \{0,1\}^{256} \to \{0,1\}^{256}$. Sia $M[1]$ la stringa contenente i primi 128 bit del messaggio, mentre $M[2]$ quella contenente gli ultimi 128. Definiamo E_k come segue:

$$E_k(m) = AES_k(M[1])||M[2]$$

Ovvero cifriamo con AES $M[1]$ e concateniamo in chiaro $M[2]$. AES soddisfa la security against key-recovery, di conseguenza anche la soddisfa. Tuttavia, l'avversario è in grado di leggere parte del messaggio in chiaro, quindi il metodo non è considerabile sicuro. Vedremo che una proprietà sufficiente a garantire la sicurezza è la **pseudorandomicità**.