Costruzioni pratiche di primitive a chiave simmetrica

Paolo D'Arco

Costruzioni pratiche di primitive a chiave simmetrica

Abbiamo visto che, dati **PRG, PRF** e funzioni **Hash**, esistono:

- schemi simmetrici di cifratura sicuri in accordo a diverse nozioni
- codici per l'autenticazione dei messaggi di lunghezza arbitraria
- protocolli vari per funzionalitá di base utili nelle applicazioni

Due domande si pongono:

1. Esistono questi "oggetti"?
2. Come possono essere costruiti?

Vedremo: costruzioni **euristiche** efficienti di queste primitive

Stream cipher

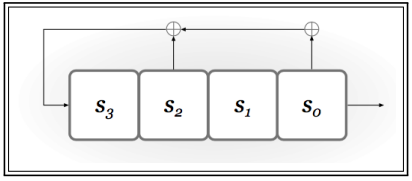
Due algoritmi deterministici: ($Init$, $GetBits$)

- $Init$: prende in input una chiave ed (opzionalmente) IV e dá in output uno stato iniziale st
- $GetBits$: puó essere invocata ripetutamente per avere in output una sequenza infinita di bit $y_1, y_2, \ldots$ a partire da st

$\Rightarrow$ la sequenza di output deve essere **indistinguibile** da una sequenza di bit scelti indipendentemente ed uniformemente

Stream cipher

Linear-feedback Shift Registers: *LFSR*



Storicamente usati per la generazione di numeri pseudocasuali. Di per sé non danno generatori pseudocasuali forti.

Tecnicamente: array di n registri con un meccanismo di retroazione (feedback loop) specificato da n coefficienti di feedback $c_0, c_1, \ldots, c_{n-1}$

Nell'esempio $c_0 = c_2 = 1$ e $c_1 = c_3 = 0$

Il numero n di registri é il **grado** dell'*LFSR*.

Stream cipher

Se lo stato al tempo t é $s_{n-1}^t, \ldots, s_0^t$, allora il prossimo stato, dopo il tick del clock, é:

$$\begin{aligned} s_i^{t+1} &= s_{i+1}^t, \quad \text{per} \quad i = 0, \ldots, n-2 \\ s_{n-1}^{t+1} &= \oplus_{i=0}^{n-1} c_i \cdot s_i^t \end{aligned}$$

Se denotiamo con y_i i bit di output, allora:

$$\begin{aligned} y_i &= s_i^0, \quad \text{per} \quad i = 1, \ldots, n \\ y_i &= \oplus_{j=0}^{n-1} c_j \cdot y_{i-n+j-1} \quad \text{per} \quad i > n. \end{aligned}$$

I primi n bit sono pertanto lo stato iniziale.

Un LFSR trasla i bit a destra, producendo un bit di output e liberando il registro piú a sinistra. Nell'esempio precedente, se lo stato iniziale é $(0, 0, 1, 1)$:

Stream cipher

$$(0,0,1,1)$$
$$(1,0,0,1)$$
$$(1,1,0,0)$$
$$(1,1,1,0)$$
$$(1,1,1,1)$$

La sequenza di output é $1,1,0,0,1$

Un *LFSR* puó ciclare attraverso al piú 2^n stati.

Un *LFSR* di **lunghezza massima** cicla attraverso $2^n - 1$ stati prima di ripetere la sequenza (lo stato $(0,\ldots,0)$ va escluso: entrando in esso non se ne esce piú).

La lunghezza dipende dai coefficienti di feedback.

Attacchi di ricostruzione

Gli *LFSR* hanno buone proprietá statistiche

Ogni stringa di n bit si presenta con uguale frequenza nella sequenza di output.

Tuttavia, **non** sono impredicibili i bit prodotti.

Un attaccante puó ricostruire l'intero stato di un LFSR di grado n dopo aver visto $2n$ bit di output

$$y_1, \ldots, y_n, y_{n+1}, \ldots, y_{2n}$$

I primi rappresentano lo stato iniziale. Dei secondi la forma é nota. Pertanto possiamo determinare i coefficienti di feedback $c_0, \ldots, c_{n-1}$

$$\begin{aligned} y_{n+1} &= c_{n-1}\, y_n \oplus \cdots \oplus c_0\, y_1 \\ &\vdots \\ y_{2n} &= c_{n-1}\, y_{2n-1} \oplus \cdots \oplus c_0\, y_n. \end{aligned}$$

Attacchi di ricostruzione

Si puó dimostrare che per un *LFSR* di grado n di lunghezza massima, le n equazioni sono *linearmente indipendenti modulo* 2

$\Rightarrow$ $\exists!$ sol. per $c_0, \ldots, c_{n-1}$. Facilmente calcolabile con i metodi dell'algebra lineare.

$\Rightarrow$ Tutti i bit seguenti sono noti.

Soluzioni. Aggiunta di **non linearitá**.

- Prima soluzione: rendere il feedback non lineare.

$$\begin{aligned} s_i^{t+1} &= s_{i+1}^t, \quad \text{per} \quad i = 0, \ldots, n-2 \\ s_{n-1}^{t+1} &= g(s_0^t, \ldots, s_{n-1}^t) \end{aligned}$$

dove g é una funzione non lineare. É possibile progettare *FSR* (non lineari) di lunghezza massima e con buona proprietá statistiche.

Stream cipher non lineari

- Seconda soluzione: combinazione.

  Introduce la non linearitá nella sequenza di output. Nella configurazione piú semplice abbiamo un *LFSR* modificato, in cui il bit di output é ottenuto calcolando una funzione non lineare g di **tutti** i registri

  $\Rightarrow$ g deve essere bilanciata, cioé

$$Pr[g(s_0^t, \ldots, s_{n-1}^t) = 1] \approx 1/2$$

Una variante della precedente consiste

- nell'usare diversi *LFSR*, combinando assieme i bit di output dei singoli *LFSR*, attraverso una funzione non lineare g (generatore della combinazione)

Stream cipher

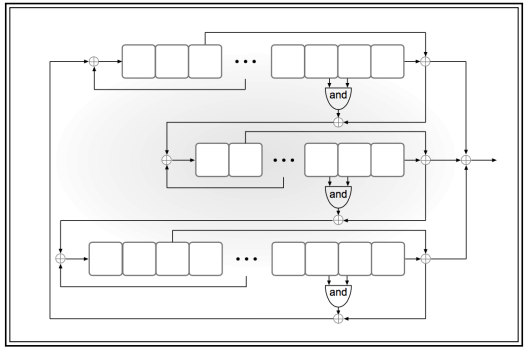
Gli *LFSR* non é richiesto che abbiano lo stesso grado.

In realtá la lunghezza del ciclo viene massimizzata se i gradi sono diversi.

Cura va posta nel far sí che il bit di l'output **non** dipenda maggiormente da uno degli *LFSR* tra i tanti.

Trivium (estream project, 2008)

Usa tre *FSR* non lineari A (grado 93), B (grado 84) e C (grado 111) accoppiati



Ha uno stato di 288 bit. Fu progettato per avere una descrizione semplice e un'implementazione **hardware** compatta.

Trivium (estream project, 2008)

L'output é l'xor dei tre bit piú a destra dei registri A, B e C.

$Init(\cdot)$ accetta

- una chiave di 80 bit, caricata nei registri di A piú a sinistra
- un vettore IV di 80 bit, caricato nei registri di B piú a sinistra

I registri restanti sono posti a 0, **eccetto** i tre registri piú a destra di C, posti a 1.

Lo stato st_0 si ottiene eseguendo $GetBits$ 4288 volte (buttando via i bit di output).

Non sono noti attacchi contro Trivium (completo) piú efficienti della ricerca esaustiva.

I *FSR* hanno ottime performance in hardware ma sono lenti nelle implementazioni software. *RC*4, progettato da R. Rivest nel 1987, é semplice e veloce.

ALGORITHM 6.1
Init **algorithm for RC4**

Input: 16-byte key k
Output: Initial state (S, i, j)
(Note: All addition is done modulo 256)

for $i = 0$ to 255:
    $S[i] := i$
    $k[i] := k[i \bmod 16]$
$j := 0$
for $i = 0$ to 255:
    $j := j + S[i] + k[i]$
    Swap $S[i]$ and $S[j]$
$i := 0, \quad j := 0$
return (S, i, j)

Nota che $S[]$ contiene sempre una permutazione di $0, \ldots, 255$. Invece il vettore $k[]$ contiene i 16 byte della chiave ripetuti (puó gestire chiavi fino a 255 byte).

GetBits funziona come segue:

ALGORITHM 6.2
GetBits algorithm for RC4

Input: Current state (S, i, j)
Output: Output byte y; updated state (S, i, j)
(Note: All addition is done modulo 256)

$i := i + 1$
$j := j + S[i]$
Swap $S[i]$ and $S[j]$
$t := S[i] + S[j]$
$y := S[t]$
return $(S, i, j), y$

Osservazioni su $Init(\cdot)$ e $GetBits(\cdot)$ in $RC4$.

- $Init(\cdot)$: ciascun byte di S viene "swappato" almeno una volta in una locazione pseudocasuale
- $GetBits(\cdot)$: lo stato di S viene usato per generare la sequenza di output
  - i viene incrementato di 1 ad ogni invocazione
  - di nuovo, ciascun byte di S viene "swappato" almeno una volta ogni 256 iterazioni, assicurando un buon mix della permutazione S.

Nota: $RC4$ non fu progettato per usare un IV. Tuttavia, diverse implementazioni lo fanno, introducendo IV nell'array che contiene la chiave (prima o dopo la chiave)

Sfortunatamente questa modalitá introduce debolezze in $RC4$. Intuitivamente la ragione é che IV viene inviato in chiaro quando il cifrario é usato in modo asincrono.

$\Rightarrow$ parte dell'array $k[]$ é nota.

Cifrari a blocchi

Un cifrario a blocchi é una permutazione con chiave efficientemente calcolabile, cioé $F : \{0,1\}^n \times \{0,1\}^\ell \to \{0,1\}^\ell$ é tale che, $\forall k$:

$$F_k(x) \stackrel{def}{=} F(k,x) \quad \text{é una permutazione}$$

ed F_k ed F_k^{-1} sono efficientemente calcolabili.

n é la lunghezza della chiave ed ℓ la lunghezza di blocco (funzioni del par. di sicurezza).

In pratica n ed ℓ sono **costanti fissate**.

Un cifrario a blocchi viene generalmente considerato buono se l'attacco migliore che si conosce ha complessitá di tempo **equivalente** ad una ricerca esaustiva di k.

Differenza: analisi teorica, attacco con compl. $2^{n/2} \Rightarrow$ cifrario buono

scenario concreto: $n = 256$, attacco con compl. $2^{128} \Rightarrow$ cifrario non buono

Cifrari a blocchi

I cifrari a blocchi vengono progettati per esibire un comportamento pari a permutazioni pseudocasuali (forti).

Modellare poi un cifrario a blocchi con una *PRP* permette di fornire prove di sicurezza per costruzioni basate sui cifrari a blocchi.

Per esempio, nella *Call* per *AES* (Advanced encryption standard, ci torneremo a breve) la richiesta di pseudocasualitá era esplicita.

I cifrari a blocchi **non** sono schemi di cifratura. Tuttavia, la terminologia standard per attacchi contro un cifrario a blocchi F é la stessa.

Parliamo di (k non é nota in tutti i casi):

Cifrari a blocchi

- *Known-plaintext attack*: $\{(x_i, F_k(x_i))\}$, x_i fuori dal controllo di *Adv*
- *Chosen-plaintext attack*: $\{(x_i, F_k(x_i))\}$, x_i scelti da *Adv*
- *Chosen-ciphertext attack*: $\{(x_i, F_k(x_i))\}$, $\{(y_i, F_k^{-1}(y_i))\}$, x_i, y_i scelti da *Adv*

Obiettivi di *Adv*:

- *distinguere* F_k da una permutazione uniforme
- *recuperare* la chiave k

Nota che:

- Una permutazione pseudocasuale non puó essere distinta da una permutazione uniforme rispetto ad un attacco di tipo chosen-plaintext
- Una permutazione pseudocasuale **forte** non puó essere distinta da una permutazione uniforme rispetto ad un attacco di tipo chosen-ciphertext

Pseudocasualitá in pratica

Sfida nel costruire un cifrario a blocchi:

$\Rightarrow$ costruire un insieme di permutazioni con una rappresentazione concisa (i.e., chiave corta) che si comporti come una permutazione casuale

Cosa significa in pratica?

Intuizione: in una permutazione casuale, cambiare un singolo bit nell'input

$\Downarrow$

ottenere un output **quasi del tutto** (non possono essere uguali) indipendente dall'output associato all'input precedente

Similmente, cambiando un bit nell'input di $F_k(\cdot)$ (k uniforme), dovrebbe

$\Downarrow$

ottenere un output **quasi del tutto** indipendente dall'output precedente (ogni bit dell'output cambia con prob. $1/2$)

Paradigma della confusione e della diffusione

Idea: costruire una permutazione F che sembra casuale con una lunghezza di blocco grande da *molteplici* permutazioni $\{F_i\}_i$ piú piccole *casuali* o che sembrano casuali.

Definiamo F come segue: vogliamo lunghezza di blocco pari a 128 bits.

La chiave k specifica 16 permutazioni $f_1, \ldots, f_{16}$ con lungh. blocco di 8 bit.

Dato $x \in \{0,1\}^{128}$, lo vediamo come $x = x_1 x_2 \ldots x_{16}$ e poniamo

$$F_k(x) = f_1(x_1)||f_2(x_2)|| \ldots ||f_{16}(x_1 6).$$

In termini di memoria la strategia richiede per

$$f_i : \{0,1\}^8 \to \{0,1\}^8 \text{ circa } 8 \cdot 2^8 \approx 2000 \text{ bit}$$

$\Rightarrow F_k$ richiede circa $16 \cdot 2000$ (pochi kbyte), molto meno dei circa $128 \cdot 2^{128}$ richiesti da una permutazione casuale

Paradigma della confusione e della diffusione

Le funzioni f_i - dette funzioni di "round" - introducono **confusione** in F. Tuttavia:

$$F_k \textbf{ non } \text{é pseudocasuale}$$

Se x ed x' differiscono nel primo byte $\Rightarrow$ $F_k(x)$ ed $F_k(x')$ sono differenti nel primo byte

$$\Downarrow$$

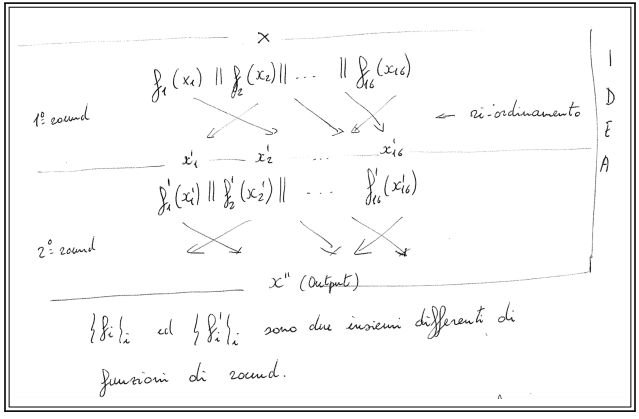
La confusione é *locale* al byte.

Abbiamo bisogno di un passo che introduca **diffusione**.

I bit dell'output vengono pertanto permutati (mixing permutation). L'effetto é diffondere i cambiamenti locali.

I passi che realizzano confusione e diffusione formano un **round**. Vanno ripetuti piú volte.

Paradigma della confusione e della diffusione



Solitamente le round function sono progettate specificamente e con cura, e sono fissate.

Reti a sostituzione e permutazione

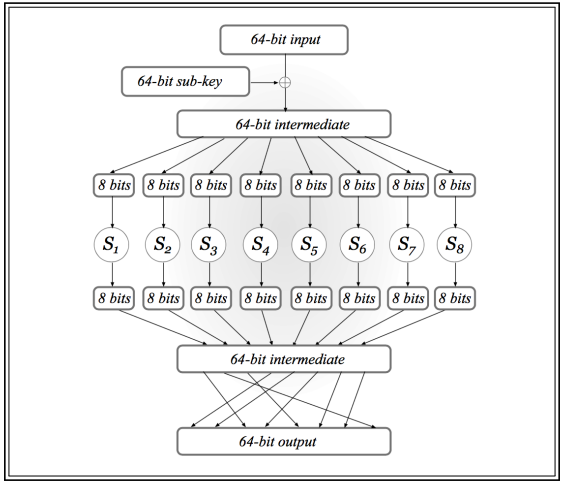
Sono un'implementazione diretta del paradigma della confuzione e della diffusione.

Idea di fondo:

- invece di usare una porzione della chiave k per specificare una f_i, fissiamo una **funzione di sostituzione** pubblica S
- chiameremo S S-box e useremo la chiave k o una porzione di essa per specificare la funzione f come
$$f(x) = S(k \oplus x).$$

Consideriamo una rete a sostituzione e permutazione (SPN in breve) con un blocco lungo 64 bit, basata su una collezione di S-box $S_1, S_2, \ldots, S_8$ di 8 bit.

Reti a sostituzione e permutazione



Reti a sostituzione e permutazione

La computazione procede attraverso una serie di round, dove in ciascun round ci sono i passi

- **Key mixing**: Poni $x := x \oplus k$, con k sottochiave del round corrente
- **Substitution**: Poni $x := S_1(x_1)||\ldots||S_8(x_8)$, con x_i i-esimo byte di x
- **Permutation**: Permuta i bit, producendo l'output del round

Le S-box e la mixing permutation sono **pubbliche** (Kerckoff's principle)

Sottochiavi **differenti** vengono usate in ciascun round

La chiave del cifrario a blocchi é una sorta di **master key**

Le sottochiavi sono derivate dalla master key in accordo ad un algoritmo di schedulazione delle chiavi (**key schedule**) spesso semplice.

Reti a sostituzione e permutazione

Una SPN ad r round ha r round pieni con key mixing, S-box substitution e mixing permutation piú un **passo finale** di key mixing.

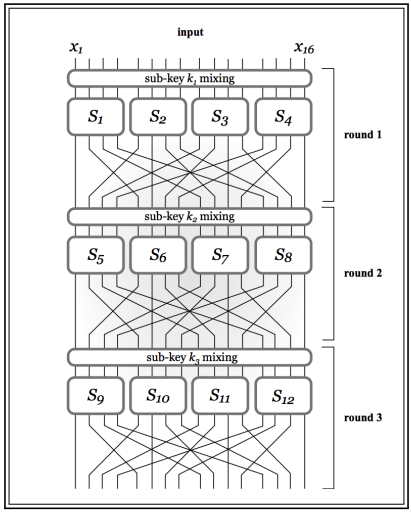
Una SPN é invertibile (data la chiave).

Proposizione 6.3. Sia F una funzione con chiave definita da una *SPN* in cui le S-box sono tutte permutazioni. Allora, indipendentemente dal numero di round e dall'algoritmo di schedulazione delle chiavi, F_k é una **permutazione** per qualsiasi valore di k.

Dim. Basta mostrare che il singolo round é invertibile $\Rightarrow$ tutta F_k é invertibile.

- **Mixing permutation** $\Rightarrow$ invertibile
- **S-Box**, permutazioni $\Rightarrow$ invertibili
- **Key mixing** (xor) $\Rightarrow$ invertibile, utilizzando la sottochiave opportuna

Struttura di una SPN



Il numero di round, le S-box, le mixing permutation e la schedulazione delle chiavi determinano la robustezza o la debolezza di uno schema.

Effetto valanga

Un piccolo cambiamento nell'input deve avere "effetto" su tutti i bit dell'output. Un modo per assicurare ció in una SPN é garantendo due proprietá:

1. le S-box sono tali che modificando **un singolo** bit di input si modificano almeno **due** bit di output della S-box
2. le mixing permutation sono progettate in modo tale che i **bit di output** di una data S-box sono usati **come input** in molteplici S-box nel round successivo

Perché funziona? Consideriamo due input che differiscono in **un solo** bit

1. Durante il primo round, i bit intermedi differiscono di un bit. Relativamente alla S-box in cui differiscono, l'output della S-Box differirá in due bit. La mixing permutation cambia le posizioni, ma i due output differiscono in **due bit** ancora

Effetto valanga

2. Nel secondo round ci sono due S-Box che ricevono input che differiscono in un bit. Pertanto, ragionando come prima, alla fine del round i valori intermedi differiscono in **quattro bit**.

3. Iterando il ragionamento ci aspettiamo che 8 bit vengano influenzati al termine del terzo round, 16 al termine del quarto ... e cosí via. Alla fine del 7-mo round tutti i 128 bit sono stati influenzati dalla computazione della SPN.

Nota: é sempre possibile che alla fine di un round ci siano meno bit differenti di quanti ce ne si aspetta $\Rightarrow$ Solitamente si usano piú di 7 round.

Sette round sono un **lower bound** per l'effetto valanga: meno round non possono produrlo.

Effetto valanga

S-Box scelte a caso **non** sono una buona scelta. Per esempio, sia S una S-Box con input di 4 bit, scelta a caso. Dati x ed x', siano $y = S(x)$ ed $y' = S(x')$

S casuale $\Rightarrow$ y' scelta uniformemente a caso.

Ci sono 4 stringhe che differiscono da y soltanto in un bit $\Rightarrow$ con prob. $4/15$ y' **non** differisce da y in almeno due bit.

Ovviamente il problema si amplifica quando consideriamo tutte le coppie di input che differiscono in un solo bit.

Pertanto, le S-Box **devono** essere progettate con cura.

Effetto valanga in PRP forti

Se un cifrario a blocchi deve realizzare una permutazione pseudocasuale **forte**, allora l'effetto valanga deve essere prodotto anche dalla permutazione inversa

$$\Downarrow$$

Cambiare un singolo bit dell'output deve aver effetti su **tutti** i bit di input.

É sufficiente che le S-box siano progettate in modo tale che, cambiando un singolo bit dell'output, si ottengano cambi in almeno due bit dell'input.

Ottenere l'effetto valanga in **entrambe** le direzione é un'altra ragione per incrementare il numero di round.

Attacchi contro SPN con un numero di round ridotto

Il numero di round é cruciale.

Un caso semplice. Un solo round senza passo finale di key-mixing.

Un Adv, data una **sola** coppia (x, y) recupera la chiave. A partire da y, valore di output:

- Inverte la mixing permutation $\rightarrow$ pubblica
- Inverte le S-Box $\rightarrow$ pubbliche
- Calcola l'xor tra l'input alle S-box ed x

In questo modo ottiene la sottochiave $\equiv$ chiave del singolo round.

Attacchi contro SPN con un numero di round ridotto

Consideriamo ora una SPN con un round **ed** il passo di key mixing finale.

Assumiamo che:

- la lunghezza del blocco sia 64 bit
- le S-box abbiano 8 bit di lunghezza di input/output
- le sottochiavi K_1 e K_2 usate nei due passi di key mixing sono **indipendenti**
- la master key é pertanto $K = K_1||K_2$ (128 bit)

Idea: estendere l'attacco semplice precedente per ottenere un attacco per il **recupero** della chiave che usa molto meno di 2^{128} passi

Attacchi contro SPN con un numero di round ridotto

Adv dispone di una coppia (x, y)

- enumera tutte le possibili chiavi K_2 (2^{64} in totale)
- per ognuna di esse, puó invertire il passo finale di key mixing
- usando l'attacco precedente ottiene, per ogni K_2, un singolo valore di K_1
- in 2^{64} passi produce una lista di 2^{64} chiavi $K = K_1||K_2$
- usando coppie (x_i, y_i) aggiuntive riduce la lista fino ad individuare la chiave giusta

Osservazione: un attacco migliore puó essere ottenuto notando che bit individuali dell'output dipendono soltanto da una parte della master key.

Attacchi contro SPN con un numero di round ridotto

Adv dispone di una coppia (x, y)

- enumera tutti i possibili valori del **primo byte** di K_2 (2^8 in totale)
- per ognuno di essi, puó invertire il passo finale di key mixing
- usando l'attacco precedente ottiene, per il byte di K_2, i valori di 8 bit di K_1

  $\Rightarrow$ le posizioni dipendono dalla mixing permutation usata nel round (pubblica)

Per ogni ipotesi sul primo byte di K_2 ottiene un'unica scelta possibile per 8 bit di K_1

Ripetendo l'attacco per tutti gli 8 byte di K_2, Adv ottiene 8 liste, ciascuna contenente 2^8 coppie che complessivamente rappresentano tutte le possibili master key

$$2^8 \cdot \ldots \cdot 2^8 \quad (8 \text{ volte}) = 2^{64} \quad \text{possibili master key.}$$

Attacchi contro SPN con un numero di round ridotto

Il tempo totale richiesto per far ció é $8 \cdot 2^8 = 2^3 \cdot 2^8 = 2^{11}$

$\Rightarrow$ precedentemente era 2^{64}

Adv, usando coppie (x_i, y_i) aggiuntive riduce la lista dei valori possibili

La chiave giusta deve essere consistente con una nuova coppia (x', y')

Euristicamente, un'ipotesi scorretta in ogni lista di 2^8 elementi é consistente con una nuova coppia (x', y') con probabilitá essenzialmente uniforme.

Poiché, dato x', ogni 16 bit della lista permettono di calcolare **un** byte di output

$\Rightarrow$ la prob. che il byte calcolato sia consistente con y' é $\frac{1}{2^8}$

$\Rightarrow$ coincide con la prob. con cui i 16 bit sono consistenti con (x', y')

Un piccolo numero di coppie (x_i, y_i) aggiuntive permette di far sí che le 8 liste contengano tutte un solo valore di 16 bit.

Attacchi contro SPN con un numero di round ridotto

L'attacco é possibile perché parti differenti della chiave possono essere **isolate** da altre

$\Rightarrow$ ulteriore diffusione é necessaria per essere sicuri che **tutti i bit** della chiave influenzino **tutti i bit** dell'output

$\Rightarrow$ piú round sono necessari

Le idee precedenti possono essere estese per ottenere un attacco migliore della ricerca esaustiva contro una SPN a due round che usa sottochiavi indipendenti nei due round

D'altra parte é facile vedere che una SPN a 2 round **non** é pseudocasuale. Infatti:

se Adv ottiene il risultato della valutazione della SPN su due input, x ed x', che differiscono in **un solo** bit, allora gli output corrispondenti differiranno in pochi bit

$\Rightarrow$ in una funzione casuale é molto diverso

Reti di Feistel

Un altro approccio alla costruzione di cifrari a blocchi

Vantaggio: le funzioni sottostanti, usate nelle reti di Feistel, contrariamente alle S-Box usate nelle SPN, **non** devono essere invertibili

$\Rightarrow$ rappresentano un modo per costruire una funzione invertibile tramite componenti non invertibili

Rispetto alle SPN c'é meno **struttura** nella rete.

Una rete di Feistel (FN in breve) opera attraverso una serie di round

In ogni round, viene applicata una funzione con chiave (del round)

$\Downarrow$

tipicamente costruita tramite S-Box e mixing permutation

Reti di Feistel

Nelle FN bilanciate (le uniche che considereremo) la i-esima funzione di round $\hat{f}_i$

- prende in input una sottochiave K_i ed una stringa di $\ell/2$ bit
- dá in output una stringa di $\ell/2$ bit.

Una volta scelta una master key K, che determina le sottochiavi K_i, definiamo

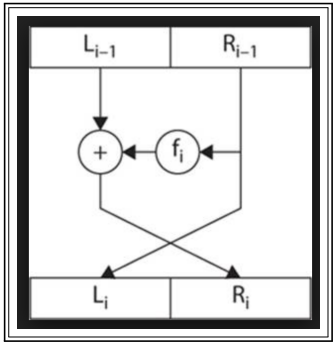
$$f_i : \{0,1\}^{\ell/2} \to \{0,1\}^{\ell/2} \qquad \text{come} \qquad f_i(R) \stackrel{def}{=} \hat{f}_i(K_i, R)$$

Nota:

- le $\hat{f}_i$ sono **fissate** e **pubblicamente** note
- le f_i dipendono dalla master key (**non** nota ad Adv)

Reti di Feistel

La lunghezza di blocco é ℓ bit. L'input é rappresentato da due sottostringhe di $\ell/2$ bit.



Il round i-esimo opera come segue:

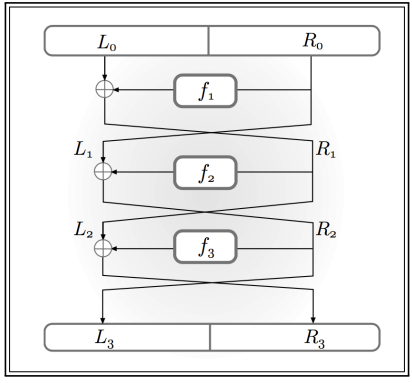
Input: L_{i-1} ed R_{i-1}.

Output: $L_i = R_{i-1}$ ed $R_i = L_{i-1} \oplus f_i(R_{i-1})$

In una rete di Feistel ad r round: $(L_0, R_0) \rightarrow (L_r, R_r)$

Reti di Feistel

Esempio di rete a 3 round.



Reti di Feistel

Le reti di Feistel sono invertibili.

Proposizione 6.4. Sia F una funzione con chiave definita da una FN. Indipendentemente dalle funzioni di round $\{\hat{f}_i\}_i$ e dal numero di round, F_k é una permutazione efficiente invertibile per qualsiasi valore di K.

Prova. Per rendersene conto, basta considerare un singolo round e notare che:

$$(L_i, R_i) \qquad \Rightarrow \qquad \begin{cases} R_{i-1} = L_i \\ L_{i-1} = R_i \oplus f_i(R_{i-1}) \end{cases}$$

In particolare, si noti che le f_i vengono valutate in **una sola** direzione.

Data Encryption Standard (DES)

Sviluppato negli anni '70 (IBM-NSA). Adottato nel 1977.

É stato scrutinato a fondo. In pratica non sono stati trovati attacchi migliori della ricerca esaustiva.

Il DES é una rete di Feistel a 16 round

- lunghezza di blocco 64 bit
- lunghezza chiave 56 bit

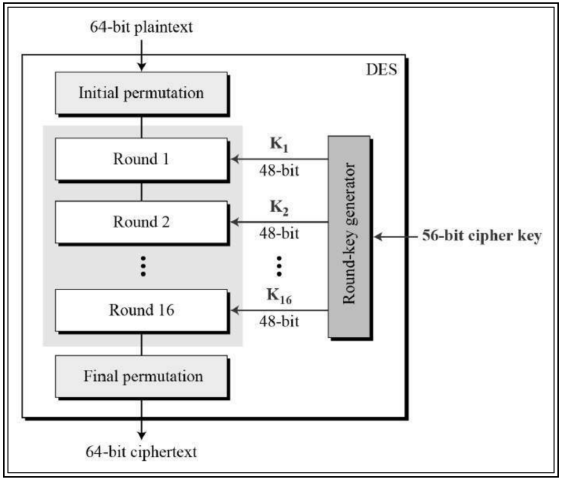
La funzione di round prende in input una sottochiave di 48 bit ed una stringa di 32 bit.

L'algoritmo di scheduling delle chiavi DES deriva le sottochiavi dalla master key

$$\text{master key} \quad K \quad (56 \text{ bit}) \quad \Rightarrow \quad K_1, K_2, \ldots, K_{16} \quad (48 \text{ bit ognuna})$$

I 48 bit di ogni sottochiave: 24 dalla parte sinistra di K e 24 dalla parte destra

DES

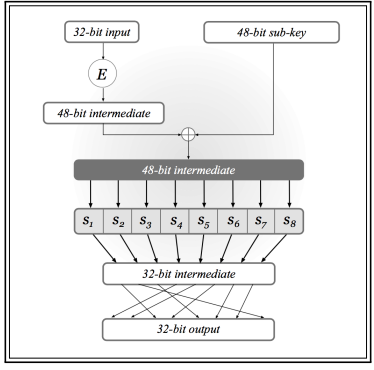

64-bit plaintext
DES
Initial permutation
Round 1
Round 2
Round 16
Final permutation
64-bit ciphertext
K1
48-bit
K2
48-bit
K16
48-bit
Round-key generator
56-bit cipher key

DES

La funzione di round del DES (DES Mangler function) é costruita usando il paradigma delle SPN

$$\hat{f}_i(K_i, R), \qquad K_i \in \{0,1\}^{48}, \qquad R \in \{0,1\}^{32}$$

La stessa funzione $\hat{f}_i$ per tutti i round (f_i differente per via delle sottochiavi K_i)



DES

I passi della funzione sono i seguenti

- R viene esteso attraverso la funzione di espansione $E : R \to E(R)$ a 48 bit
- $E(R) \oplus K_i$ é una stringa di 48 bit, input per le S-box

La funzione di espansione é

E BIT-SELECTION TABLE

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

DES

Le S-box **non** sono invertibili (gli input sono piú lunghi degli output)

La specifica dell'algoritmo é completamente pubblica. Soltanto la master key é segreta

Le S-box sono il cuore di $\hat{f}$ e sono cruciali per la sicurezza del DES

Anche un minimo cambiamento puó introdurre debolezze

Esibiscono le seguenti proprietá:

1. ciascuna S-box é una funzione 4-a-1
2. ciascuna riga della tabella contiene tutte le possibili stringhe di 4 bit esattamente **una** volta
3. cambiando un bit ad ogni stringa di input di una S-box, si cambiano sempre **almeno** due bit di output

DES

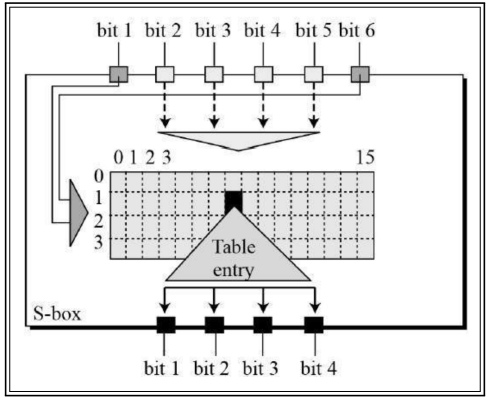
La struttura di una *S*-box

	Column Number															
Row No.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

DES

Sia $b_1 \dots b_6$ una generica stringa di input di 6 bit per la S-box.

I bit $b_1 b_6$ indicizzano una riga, i bit $b_2 b_3 b_4 b_5$ una colonna.



DES

Anche le mixing permutation furono progettate con cura

- i 4 bit di output di qualsiasi S-box influenzano l'input di 6 S-box nel round successivo

$\Uparrow$

reso possibile dalla presenza della funzione di espansione E

La permutazione é

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

DES

La funzione di round del DES fa sí che DES esibisca un forte **effetto valanga**.

Tracciamo le differenze tra i valori intermedi di DES di due input che differiscono in un singolo bit.

$$(L_0, R_0) \qquad \text{Assumiamo } R_0 = R_0' \qquad (L_0', R_0')$$

$$\Uparrow$$

$$\text{Il bit differente é in } L_0, L_0'$$

Dopo il primo round:

$$(L_1, R_1) \qquad\qquad (L_1', R_1')$$

Ancora differenti in un bit in R_1, R_1' essendo, $R_1 = L_0 \oplus f(R_0), R_1' = L_0' \oplus f(R_0')$

Dopo il secondo round:

$$(L_2, R_2) \qquad\qquad (L_2', R_2')$$

L_2, L_2' differiscono in un bit poiché $L_2 = R_1, L_2' = R_1'$

Assumiamo inoltre che il bit in cui R_1 ed R_1' differiscono **non** venga duplicato da E.

Per la proprietá 3. delle S-box, l'output dell'S-box in cui il bit é differente é **diverso** in almeno due bit $\Rightarrow$ R_2 ed R_2' differiscono in **almeno** due bit

$$\Downarrow$$

(L_2, R_2) ed (L_2', R_2') differiscono in **almeno** tre bit

La mixing permutation diffonde la differenza di due bit in R_2 ed R_2'.

Ciascuno dei due bit viene usato come input in diverse S-box.

Dopo il terzo round:

$$(L_3, R_3) \qquad\qquad (L_3', R_3')$$

L_3, L_3' differiscono in almeno due bit poiché $L_3 = R_2, L_3' = R_2'$

R_3, R_3' differiscono in almeno quattro bit per la prop. 3 della S-box

(se E duplica uno o entrambi i bit in cui R_2 ed R_2' differiscono, allora R_3, R_3' possono differire in piú di 4 bit)

$$\Downarrow$$

Come per le SPN, abbiamo un effetto esponenziale: dopo 7 round ci aspettiamo che tutti i 32 bit della parte destra siano influenzati dalla differenza di un bit nei due input.

DES ha 16 round. Assicurano che su input simili gli output sembrino indipendenti.

Attacchi contro DES ridotto

Consideriamo attacchi contro DES ad un round, a due round ed a tre round.

Nessuna di queste varianti puó essere pseudocasuale. Non c'é effetto valanga.

Consideriamo attacchi di recupero della chiave di tipo known-plaintext.

$$Adv \quad \text{conosce} \quad \{(x_i, y_i)\}_i, \quad \text{dove } y_i = DES_K(x_i)$$

ONE round. Data la coppia (x, y) risulta

$$y = (L_1, R_1) \qquad x = (L_0, R_0)$$

Poiché $R_0 = L_1$ ed $R_1 = L_0 \oplus f_1(R_0)$, possiamo ricavare $f_1(R_0) = L_0 \oplus R_1$

$\Rightarrow$ conosciamo una **coppia** input/output per f_1, ovvero $(R_0, L_0 \oplus R_1)$.

Attacchi contro DES ridotto

Applicando l'inversa della mixing permutation ad $L_0 \oplus R_1$ otteniamo le stringhe di output delle S-box

Ci sono 4 possibili valori di input (di 6 bit) **per ognuna** delle S-box

L'input alle S-box é l'xor tra $E(R_0)$ e K_1.

Essendo R_0 noto, risulta $E(R_0)$ noto $\Rightarrow$ possiamo calcolare 4 **possibili valori** per ciascuna porzione di 6 bit di K_1

- semplicemente, **per ognuna** delle S-box, calcoliamo l'xor tra ognuno dei 4 possibili input alla S-box e la porzione di $E(R_0)$ corrispondente

Abbiamo ridotto il numero di valori possibili per la sottochiave K_1 da 2^{48} a $4^8 = 2^{16}$

Un'altra coppia (x', y') é sufficiente per individuare quella giusta tra le 2^{16}.

Attacchi contro DES ridotto

TWO round. Data una coppia (x, y) sono noti: (L_0, R_0) ed (L_2, R_2). D'altra parte

$$\mathbf{L_1} = R_0, \quad R_1 = L_0 \oplus f_1(R_0), \quad L_2 = \mathbf{R_1}, \text{ ed } R_2 = L_1 \oplus f_2(R_1)$$

per cui conosciamo anche L_1 ed R_1

$\Rightarrow$ disponiamo di una coppia input/output per f_1, ovvero $(R_0, L_0 \oplus R_1)$

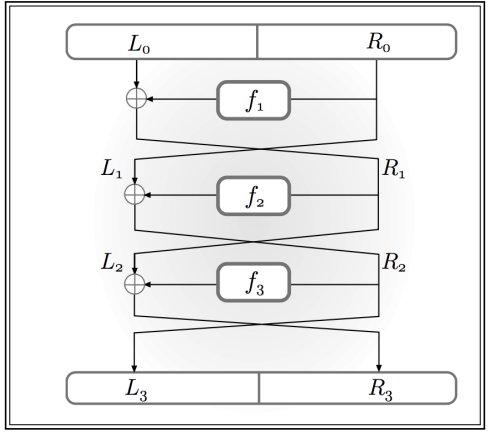
$\Rightarrow$ disponiamo di una coppia input/output per f_2, ovvero $(R_1, L_1 \oplus R_2)$

Pertanto, lo stesso metodo usato nel caso precedente puó essere utilizzato per determinare K_1 e K_2 in tempo circa $2 \cdot 2^{16}$

Nota: l'attacco funziona anche se K_1 e K_2 sono chiavi totalmente indipendenti.

Attacchi contro DES ridotto

THREE round. Piú complicato. Diversa strategia di attacco.



Attacchi contro DES ridotto

Data una coppia (x, y), conosciamo (L_0, R_0) ed (L_3, R_3).

Poiché $L_1 = R_0$ ed $R_2 = L_3$, le uniche incognite sono R_1 ed L_2 (che sono **uguali**).

Pertanto, **non** disponiamo di una coppia input/output per f_1, f_2 ed f_3, e.g.,

$$R_2 = L_1 \oplus f_2(R_1) \quad \Rightarrow \quad f_2(R_1) = L_1 \oplus R_2 \quad \text{ma } R_1 \textbf{ non é noto.}$$

Ma conosciamo una **relazione** tra input ed output di f_1 ed f_3. Precisamente, risulta

$$f_1(R_0) \oplus f_3(R_2) = (L_0 \oplus \mathbf{R_1}) \oplus (L_2 \oplus R_3) = (L_0 \oplus \mathbf{L_2}) \oplus (L_2 \oplus R_3) = (L_0 \oplus R_3)$$

Ovvero, l'xor degli output di f_1 ed f_3 é uguale ad $L_0 \oplus R_3$ (che é noto).

Inoltre, R_0 ed R_2, input di f_1 ed f_3, rispettivamente, sono noti.

Attacchi contro DES ridotto

Possiamo allora determinare gli **input** di f_1 ed f_3 e l'**xor dei loro output**.

Useremo queste informazioni per progettare un attacco di recupero della chiave!

Ricordiamo che la master key in DES K (56 bit) é $K = K_L||K_R$, due metá di 28 bit.

Ciascuna sottochiave K_i (48 bit) prende 24 bit da K_L e 24 da K_R.

$$\Downarrow$$

K_L influenza i primi 24 bit, K_R influenza i secondi 24 bit

Proviamo ad indovinare (e a **fissare**) un valore per K_L

Conosciamo R_0 per f_1. Usiamo K_L per calcolare l'input per le prime 4 S-box e, quindi, metá dei bit di output di f_1.

Attacchi contro DES ridotto

Usando R_2 possiamo calcolare gli stessi bit di output di f_3 per lo stesso valore di K_L

A questo punto possiamo verificare l'ipotesi fatta su K_L

- calcoliamo l'xor tra i bit di output ottenuti valutando f_1 ed f_3
- li compariamo con i bit corrispondenti di $L_0 \oplus R_3$

Indicando con $\overline{f_1(R_0)}|_{16}$ e $\overline{f_3(R_2)}|_{16}$ i bit di output calcolati, ed $\overline{L_0 \oplus R_3}|_{16}$ i bit di $L_0 \oplus R_3$ nelle **stesse** posizioni, se

$$\overline{f_1(R_0)}|_{16} \oplus \overline{f_3(R_2)}|_{16} \neq \overline{L_0 \oplus R_3}|_{16}.$$

allora l'ipotesi su K_L é sbagliata e occorre ripetere il processo ipotizzando un nuovo valore per K_L.

Attacchi contro DES ridotto

Un'ipotesi corretta **supera sempre** il test. Una scorretta soltanto con prob. $\approx 2^{-16}$.

Poiché sono possibili 2^{28} ipotesi per K_L, alla fine del processo si aspettiamo circa $2^{28} \cdot 2^{-16} = 2^{12}$ valori possibili per K_L.

Lo stesso attacco puó essere sferrato contro K_R

$\Rightarrow$ in tempo prossimo a $2 \cdot 2^{28}$ abbiamo circa $2^{12} \cdot 2^{12} = 2^{24}$ possibili candidati per la master key.

Usando una coppia aggiuntiva (x', y') é possibile individuare quella giusta

- con una ricerca esaustiva su tutte le 2^{24} possibili master key
- applicando nuovamente l'attacco presentato con le 2^{12} possibili K_L e le 2^{12} possibili K_R

Attacchi contro DES ridotto

Homework

La complessitá dell'attacco é data da

- relativamente all'uso di (x, y), circa $2 \cdot 2^{28}$ passi
- relativamente all'uso di (x', y'), circa 2^{24} passi per la ricerca esaustiva oppure circa $2 \cdot 2^{12}$ passi per ripetere l'attacco

In entrambi i casi, non piú di 2^{30} passi, che é molto meno di 2^{56}.

Sicurezza del DES

15 January	1977	DES is published as a FIPS standard FIPS PUB 46
	1983	DES is reaffirmed for the first time
	1986	Videocipher II, a TV satellite scrambling system based upon DES, begins use by HBO
22 January	1988	DES is reaffirmed for the second time as FIPS 46-1, superseding FIPS PUB 46
July	1991	Biham and Shamir rediscover differential cryptanalysis, and apply it to a 15-round DES-like cryptosystem.
	1992	Biham and Shamir report the first theoretical attack with less complexity than brute force: differential cryptanalysis. However, it requires an unrealistic 2^{47} chosen plaintexts.
30 December	1993	DES is reaffirmed for the third time as FIPS 46-2
	1994	The first experimental cryptanalysis of DES is performed using linear cryptanalysis (Matsui, 1994).
June	1997	The DESCHALL Project breaks a message encrypted with DES for the first time in public.
July	1998	The EFF's DES cracker (Deep Crack) breaks a DES key in 56 hours.
January	1999	Together, Deep Crack and distributed.net break a DES key in 22 hours and 15 minutes.
25 October	1999	DES is reaffirmed for the fourth time as FIPS 46-3, which specifies the preferred use of Triple DES, with single DES permitted only in legacy systems.
26 November	2001	The Advanced Encryption Standard is published in FIPS 197
26 May	2002	The AES becomes effective
26 July	2004	The withdrawal of FIPS 46-3 (and a couple of related standards) is proposed in the *Federal Register*.[19]
19 May	2005	NIST withdraws FIPS 46-3 (see Federal Register vol 70, number 96)
April	2006	The FPGA-based parallel machine COPACOBANA of the Universities of Bochum and Kiel, Germany, breaks DES in 9 days at a $10,000 hardware cost.[20] Within a year software improvements reduced the average time to 6.4 days.
Nov.	2008	The successor of COPACOBANA, the RIVYERA machine, reduced the average time to less than a single day.
July	2017	A chosen-plaintext attack utilizing a rainbow table can recover the DES key for a single specific chosen plaintext *1122334455667788* in 25 seconds. A new rainbow table has to be calculated per plaintext. A limited set of rainbow tables have been made available for download.[21]

Parametri insufficienti oggi

Non solo la lunghezza della chiave é troppo corta per la potenza computazionale attuale.

Anche la lunghezza del blocco in alcune applicazioni é fonte di preoccupazione.

Esempio: la prova di sicurezza che CRT-mode é CPA sicura dipende dalla lunghezza del blocco.

A vince con probabilitá $\frac{2 \cdot q^2}{2^\ell}$ se ottiene q coppie (m, c)

Pertento, se $\ell = 64 \Rightarrow$ per $q = 2^{30}$ risulta

$$\frac{2 \cdot q^2}{2^\ell} = \frac{2 \cdot (2^{30})^2}{2^\ell} = \frac{2 \cdot 2^{60}}{2^{64}} = \frac{1}{2^3} = 1/8$$

che é una probabilitá altissima!

Conclusione

La progettazione del DES é "quasi perfetta". Non si conoscono debolezze strutturali.

I suoi parametri sono solo troppi corti oggi.

In pratica il miglior attacco disponibile é la ricerca esaustiva.

Esistono due tecniche di attacco sofisticate

- crittoanalisi differenziale (Biham e Shamir 1991)
  - attacco chosen plaintext: richiede 2^{47} coppie e rompe in tempo 2^{37}
- crittoanalisi lineare (Matsui, 1993)
  - attacco known plaintext: richiede 2^{43} coppie e rompe in tempo 2^{39}

Gli attacchi sono piú di interesse teorico che di rilevanza pratica.

Come usare DES oggi?

1. Apportare modifiche interne
   - mantenere la funzione di round, incrementare la taglia della master key
   - cambiare S-box ed usare sottochiavi piú lunghe
   - svantaggio: **perdiamo** la confidenza che abbiamo guadagnato circa la struttura DES negli anni

2. Usare il DES come una scatola nera
   - lo consideriamo un cifrario a blocchi perfetto con chiave da 56 bit
   - un nuovo cifrario a blocchi lo invoca come subroutine

Cifratura doppia

Sia F un cifrario a blocchi con chiave di n bit e lunghezza di blocco di ℓ bit.

Un nuovo cifrario con chiave di $2n$ bit puó essere definito come segue:

$$F'_{K_1,K_2}(x) \stackrel{def}{=} F_{K_2}(F_{K_1}(x))$$

Nel caso che F sia DES, otterremmo $2DES$, con chiave da 112 bit.

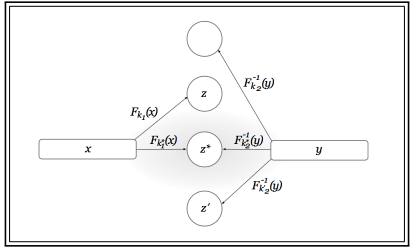
Se la ricerca esaustiva fosse l'attacco migliore, saremmo protetti rispetto ad attacchi di tempo $\leq 2^{112}$

Purtroppo non é cosí.

Attacco contro la cifratura doppia

A dispone di $(x, y) = (x, F'_{K_1^*, K_2^*}(x))$. Opera come segue

- Per ogni $K_1 \in \{0,1\}^n$, calcola $z = F_{K_1}(x)$ e memorizza (z, K_1) nella lista L_1
- Per ogni $K_2 \in \{0,1\}^n$, calcola $z = F^{-1}_{K_2}(y)$ e memorizza (z, K_2) nella lista L_2
- Le coppie $(z_1, K_1) \in L_1$ e $(z_2, K_2) \in L_2$ costituiscono un **match** se $z_1 = z_2$. Per ciascun match memorizza (K_1, K_2) nella lista S



Attacco contro la cifratura doppia

L'attacco richiede

$$\text{Tempo } O(n \cdot 2^n) \qquad \text{Spazio } O((n+\ell) \cdot 2^n)$$

La lista S contiene (K_1, K_2) tali che $F_{K_1}(x) = F_{K_2}^{-1}(y)$

Una coppia $(K_1, K_2) \neq (K_1^*, K_2^*)$ soddisfa l'equazione con prob. $2^{-\ell}$, trattando F_K ed F_K^{-1} come funzioni totalmente casuali

$$\Downarrow$$

$$\text{La taglia attesa della lista } S \text{ é } 2^{2n} \cdot 2^{-\ell} = 2^{2n-\ell}$$

Usando altre poche coppie (x_i, y_i), S viene sfoltita e la chiave (K_1^*, K_2^*) correttamente individuata.

In conclusione, $2DES$ non viene usato.

Cifratura tripla

Vengono usate due varianti

1. Tre chiavi K_1, K_2 e K_3 indipendenti

$$F''_{K_1,K_2,K_3}(x) \stackrel{def}{=} F_{K_3}(F^{-1}_{K_2}(F_{K_1}(x)))$$

2. Due chiavi K_1 e K_2 indipendenti

$$F''_{K_1,K_2}(x) \stackrel{def}{=} F_{K_1}(F^{-1}_{K_2}(F_{K_1}(x)))$$

L'invocazione centrale di F^{-1}_K é per ragioni di compatibilitá, i.e.,

$$K_1 = K_2 = K_3 \qquad \Rightarrow \qquad 3DES_{K_1,K_2,K_3} = DES_{K_1}$$

Cifratura tripla

D'altra parte, se F é una SPRP, allora F^{-1} é una SPRP e quindi non ci sono problemi in generale in questo uso.

Sicurezza della prima variante: la chiave é lunga $3n$ bit ma, per l'attacco precedente contro la cifratura doppia - che si applica anche qui - otteniamo sicurezza rispetto ad attacchi di tempo $\leq 2^{2n}$.

Sicurezza della seconda variante: la chiave é lunga $2n$ bit. Al momento non si conoscono attacchi di complessitá migliore di 2^{2n}, dato soltanto un piccolo numero di coppie (x, y)

É una buona scelta in pratica

Svantaggi: lunghezza di blocco relativemente piccola e cifrario complessivamente lento (3 invocazioni di DES).

Advanced Encryption Standard (AES)

Cifrario a blocchi, adottato ufficialmente dal NIST nel 2001.

Prescelto tra 15 algoritmi finalisti nella call per sicurezza, efficienza in implementazioni HW e SW e per flessibilitá.

Ad oggi, non sono noti attacchi crittoanalitici migliori della ricerca esaustiva

Ha lunghezza di blocco di 128 bit. Puó usare chiavi di $128, 192$ 2 256 bit

La lunghezza della chiave

- influisce sulla schedulazione delle chiavi e sul numero di round
- **non** influisce sulla struttura di alto livello di ciascun round

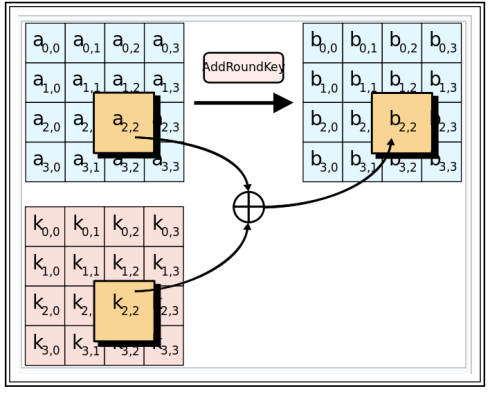
AES é essenzialmente una rete SPN

Un array di 16 byte di taglia 4×4 (lo stato di AES) viene modificato durante la computazione.

AES: Struttura

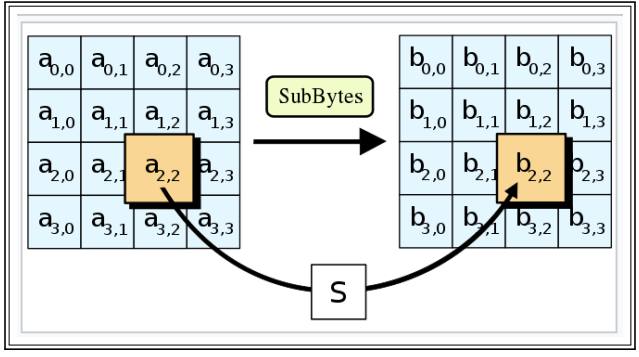
Lo stato iniziale corrisponde all'input. La struttura di ogni round é costituita da 4 passi.

Passo 1: AddRoundKey. Una sottochiave di round di 128 bit viene "aggiunta" tramite xor allo stato



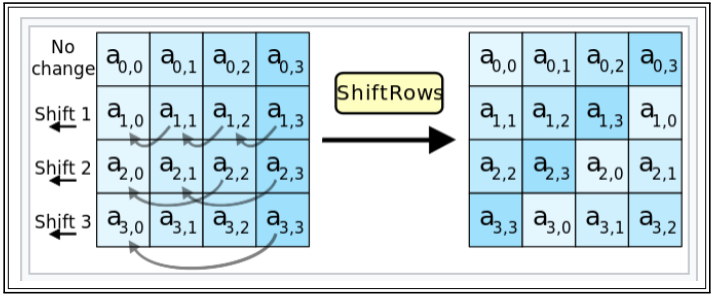
AES: Struttura

Passo 2: **SubBytes**. Ciascun byte dello stato viene sostituito con un altro byte, in accordo ad una tabella di lookup S.



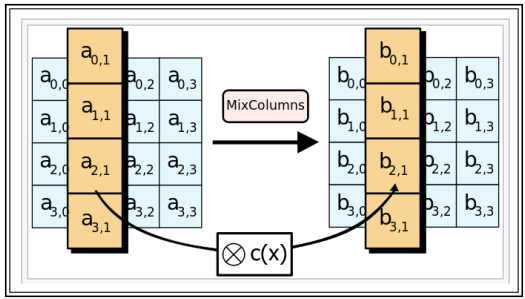
AES: Struttura

Passo 3: **ShiftRows**. I byte di ciascuna riga vengono traslati (shift) a sinistra. Riga $i \rightarrow$ shift di i posizioni.



AES: Struttura

Passo 4: MixColumns. Una trasformazione invertibile viene applicata ai 4 byte di ogni colonna (moltiplicazione matriciale).



Proprietá: se due input differiscono in $b > 0$ byte, allora applicando la trasformazione, i due output differiscono in almeno $5 - b$ byte.

AES: Struttura

Nel round iniziale viene effettuato soltanto il passo **AddRoundKey**.

Nei round intermedi i passi sono **SubBytes**, **ShiftRows**, **MixColumns** e **AddRoundKey**.

Nel round finale, **MixColumns** non viene eseguito.

Si noti che i passi 3 e 4 corrispondono al mixing step in una rete SPN.

Circa il numero di round, una chiave da

- 128 bit $\Rightarrow$ 10 round 192 bit $\Rightarrow$ 12 round 256 bit $\Rightarrow$ 14 round

AES é una scelta eccellente d'uso in tutti gli schemi che richiedono una permutazione pseudocasuale forte (SPRP).

Funzioni hash usate in pratica

$$H : \{0,1\}^* \to \{0,1\}^\ell \qquad \text{collision resistant}$$

Deve essere impraticabile trovare $x, x' : H(x) = H(x')$

Output di ℓ bit $\Rightarrow$ il meglio che possiamo sperare di ottenere é che risulta impraticabile trovare collisioni usando meno di $2^{\ell/2}$ invocazioni di H.

Generalmente costruite in due passi:

1. una funzione di compressione per input di lunghezza fissata viene progettata
2. la funzione viene estesa a domini arbitrari, per esempio usando la trasformazione di Merkle-Damgard

Come costruire una buona funzione di compressione?

Davies-Meyer construction

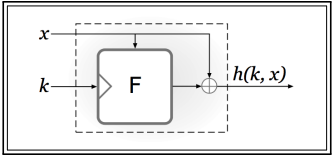
Un modo di costruire una funzione di compressione collision resistant da **un cifrario a blocchi** che soddisfa proprietá aggiuntive.

Sia

$$F : \{0,1\}^n \times \{0,1\}^\ell \to \{0,1\}^\ell$$

un cifrario a blocchi con lungh. chiave n e lungh. blocco ℓ. Definiamo

$$h : \{0,1\}^{n+\ell} \times \{0,1\}^\ell \to \{0,1\}^\ell \qquad \text{come} \qquad h(k,x) \stackrel{def}{=} F_K(x) \oplus x.$$



MD5

Funzione hash con output di 128 bit

- Progettata nel 1991 da R. Rivest
- Nel 1993 (Der Boer e Bosselaers) pseudo-collisione e nel 1996 (Dobbertin) prima collisione
- Nel 2004 un team cinese (Wang e altri) di crittoanalisti ha presentato un metodo efficiente per trovare collisioni
- L'attacco é stato migliorato diverse volte successivamente e oggi bastano pochi minuti su PC
- Possono essere anche trovate collisioni significative
- Non dovrebbe essere piú usata

Secure hash algorithm SHA

SHA-0, SHA-1 e SHA-2, standardizzate dal NIST

SHA-1

- Introdotta nel 1995. Ha 160 bit di output
- Nessuna collisione esplicita é stata ancora trovata. Tuttavia, alcune analisi suggeriscono che presto collisioni potranno essere trovate con meno di 2^{80} passi.

News: trovate

SHA-2 (due versioni)

- SHA-256 con 256 bit di output
- SHA-512 con 512 bit di output

Secure hash algorithm SHA

Tutte le funzioni SHA (SHA-1, SHA-256 e SHA-512) sono realizzate utilizzando lo stesso schema di progettazione

- viene progettata una funzione di compressione utilizzando la costruzione di Davies-Meyer ad un cifrario a blocchi
- il dominio viene esteso usando la trasformazione di Merkle-Damgard.
- il cifrario a blocchi, identificato retroattivamente nell'analisi della costruzione e denotato con i nomi SHACAL-1 (per SHA-1) e SHACAL-2 (per SHA-2), **non** é usato per la cifratura ma progettato esplicitamente per la costruzione

Il NIST ha annunciato una nuova competizione pubblica nel 2007 per una nuova funzione

Il 5 agosto del 2015 la funzione SHA-3 é stata rilasciata come ultima della famiglia SHA

SHA-3 (Keccak)

É completamente differente da tutti gli algoritmi SHA precedenti. Supporta output di 256 e 512 bit. SHA-3:

- utilizza una permutazione f senza chiave con una lunghezza di blocco molto grande, i.e., 1600 bit
- non utilizza la trasformazione di Merkle-Damgard
- sfrutta un approccio nuovo, denominato **sponge construction**, per gestire input di lunghezza arbitraria.
- puó essere analizzata nel **modello della permutazione casuale**
  - tutte le parti hanno accesso ad un oracolo per una permutazione casuale $f : \{0,1\}^{\ell} \rightarrow \{0,1\}^{\ell}$
  - é un modello piú debole dell'ideal cipher model (fissando k si ottiene questo)

Sponge construction

Opera in due fasi: assorbimento (sponge) e spremitura (squeeze).

Nella prima, l'input $P = P_0 \dots P_{n-1}$, viene assorbito nello stato, r bit alla volta.

Nella seconda, l'output Z viene prodotto a seguito di trasformazioni di stato, r bit alla volta.

