Costruzioni di primitive simmetriche

Paolo D'Arco
pdarco@unisa.it

Università di Salerno

Elementi di Crittografia

Contenuti

In teoria

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

Costruzioni pratiche di primitive a chiave simmetrica

Non possono essere provate sicure a partire da nessuna assunzione piú debole ma sono basate su principi di progettazione che a volte possono essere **giustificati** dall'analisi teorica.

Queste costruzioni hanno resistito per molti anni al pubblico scrutinio. É ragionevole assumere che siano sicure. Non c'é differenza fondamentale ma **qualitativa** tra assumere che:

- La **Fattorizzazione** é difficile
  - sembra un requisito piú debole
  - piú naturale, studiata per secoli prima della applicazioni in crittografia
- **AES** é una permutazione pseudocasuale
  - assunzione piú forte
  - meno naturale e meno studiata

Costruzioni pratiche di primitive a chiave simmetrica

Obiettivi delle prossime lezioni

- presentare alcuni **principi** di progettazione usati nella costruzione delle moderne primitive crittografiche
- introdurre **costruzioni** popolari ed ampiamente usate

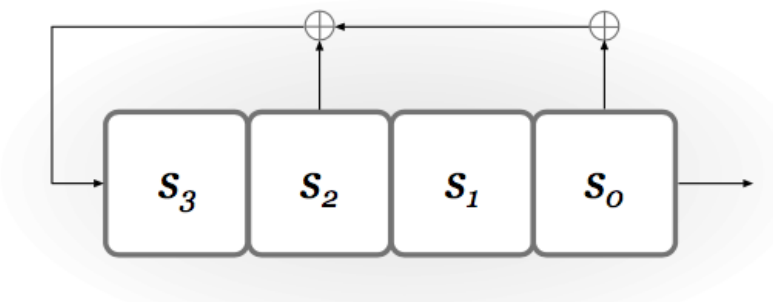
Stream cipher

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

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:

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

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

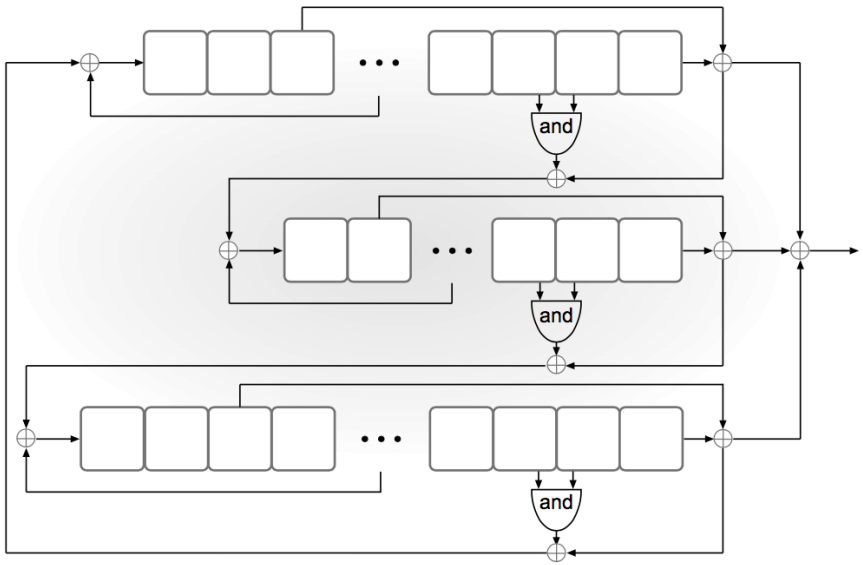
Gli *LFSR* non é richiesto che abbiano lo stesso grado.

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).

É stata notevole la resistenza alla crittoanalisi mostrata nonostante la sua semplicitá ed efficienza. Tuttavia, gli attacchi sviluppati nel tempo ne hanno ridotto i margini di sicurezza.

RC4

I *FSR* hanno ottime performance in hardware ma sono lenti nelle *implementazioni software*. *RC4*, 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.

Un attacco statistico semplice contro *RC4*

Mostra che *RC4* é "leggermente sbilanciato verso zero."

Siano S_t, lo stato di S dopo t iterazioni di *GetBits*, ed S_0 lo stato iniziale.

Trattando S_0 come una permutazione *uniforme* di $\{0, \ldots, 255\}$ risulta:

$$Pr[S_0[2] = 0 \wedge X = S_0[1] \neq 2] = \frac{1}{256} \cdot (1 - \frac{1}{255}) \approx \frac{1}{256}.$$

Supponiamo sia cosí.

Nella prima iterazione di *GetBits*, $i := 1$ ed il valore $j := S_0[i] = S_0[1] = X$.

Quindi, $S_0[1]$ ed $S_0[X]$ vengono scambiati $\Rightarrow S_1[X] = S_0[1] = X$

Nella seconda iterazione di *GetBits*, $i := 2$, mentre j diventa

$$j + S_1[i] = X + S_1[2] = X + S_0[2] = X$$

Un attacco statistico semplice contro *RC4*

Poi $S_1[2]$ ed $S_1[X]$ vengono scambiati, e quindi

$$S_2[X] = S_1[2] = S_0[2] = 0 \quad \text{e } S_2[2] = S_1[X] = X.$$

Infine, il valore di S_2 in posizione
$S_2[i] + S_2[j] = S_2[2] + S_2[X] = 0 + X = X$ viene dato in output.

Ma questo valore é esattamente $S_2[X] = 0$.

D'altra parte, quando $S_0[2] \neq 0$, il secondo byte di output é uniformemente distribuito. Pertanto, risulta $Pr[$il secondo byte di output sia $0]$

$$\begin{aligned} &= 1 \cdot Pr[S_0[2] = 0 \wedge X = S_0[1] \neq 2] + \frac{1}{256} \cdot Pr[S_0[2] \neq 0 \wedge X = S_0[1] \neq 2] \\ &\approx 1 \cdot \frac{1}{256} + \frac{1}{256} \cdot (1 - Pr[S_0[2] = 0 \wedge X = S_0[1] \neq 2]) \\ &\approx \frac{1}{256} + \frac{1}{256} = \frac{2}{256} \end{aligned}$$

(2 volte quanto ci si aspetterebbe se fosse una perm. uniforme)

Un attacco utilizzando il vettore *IV*

IV viene usato nello standard di cifratura *WEP*

Il cuore dell'attacco sta nella possibilitá di estendere la conoscenza dei primi n byte di k alla conoscenza dei primi $n+1$ *byte*

Nota: se *IV* viene anteposto alla chiave k', allora $k = IV||k'$

$\Rightarrow$ i primi byte di k sono noti.

Supponiamo *IV* sia lungo 3 byte.

Adv aspetta fino a quando *IV* é di una determinata forma. Una buona per l'attacco é

$$IV = (3, 255, X), \text{ con } X \text{ byte generico.}$$

$\Rightarrow$ $k[0] = 3, k[1] = 255$ e $k[2] = X$.

Un attacco utilizzando il vettore *IV*

É possibile verificare che, dopo le prime 4 iterazioni del secondo ciclo di $Init(\cdot)$, risultano:

$$S[0] = 3, \qquad S[1] = 0, \qquad \text{e } S[3] = X + 6 + k[3].$$

Nelle successive 252 iterazioni, $S[0], S[1]$ ed $S[3]$ non vengono modificati fino a quando $j \notin \{0, 1, 3\}$.

Se j assume valori in accordo alla distribuzione uniforme, allora la

$$Pr[j \notin \{0, 1, 3\}] = (\frac{253}{256})^{252} \approx 0,05$$

$\Rightarrow$ il 5% delle volte, $S[0], S[1]$ ed $S[3]$ **non** vengono piú modificati.

Pertanto, il primo byte che *GetBits* dá in output sará $S[3] = X + 6 + k[3]$

$\Rightarrow$ $k[3]$ viene rivelato.

Un attacco utilizzando il vettore *IV*

Quindi *Adv* sa che il 5% delle volte il primo byte di output é correlato a $k[3]$.

Indovinare a caso $k[3]$ ha prob. di successo $\frac{1}{256} \approx 0,4\%$ delle volte, molto meno del 5%

$\Rightarrow$ Collezionando un numero sufficientemente grande di campioni del primo byte di output per diversi *IV* di inizializzazione della forma giusta, *Adv* ottiene una stima molto accurata di $k[3]$

Questo attacco puó quindi essere usato per recuperare la chiave.

Molto piú importante del precedente, che indica una lieve debolezza strutturale dell'algoritmo *RC4*.

Cifrari a blocchi

Un cifrario a blocchi é una permutazione con chiave efficientemente calcolabile, cioé $F : \{0,1\}^n \times \{0,1\}^\ell \rightarrow \{0,1\}^\ell$ 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).

Cifrari a blocchi

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 complessitá $2^{n/2} \Rightarrow$ cifrario buono
- scenario concreto: $n = 256$, attacco con complessitá $2^{128} \Rightarrow$ 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

Cifrari a blocchi

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

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

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

$$\Downarrow$$

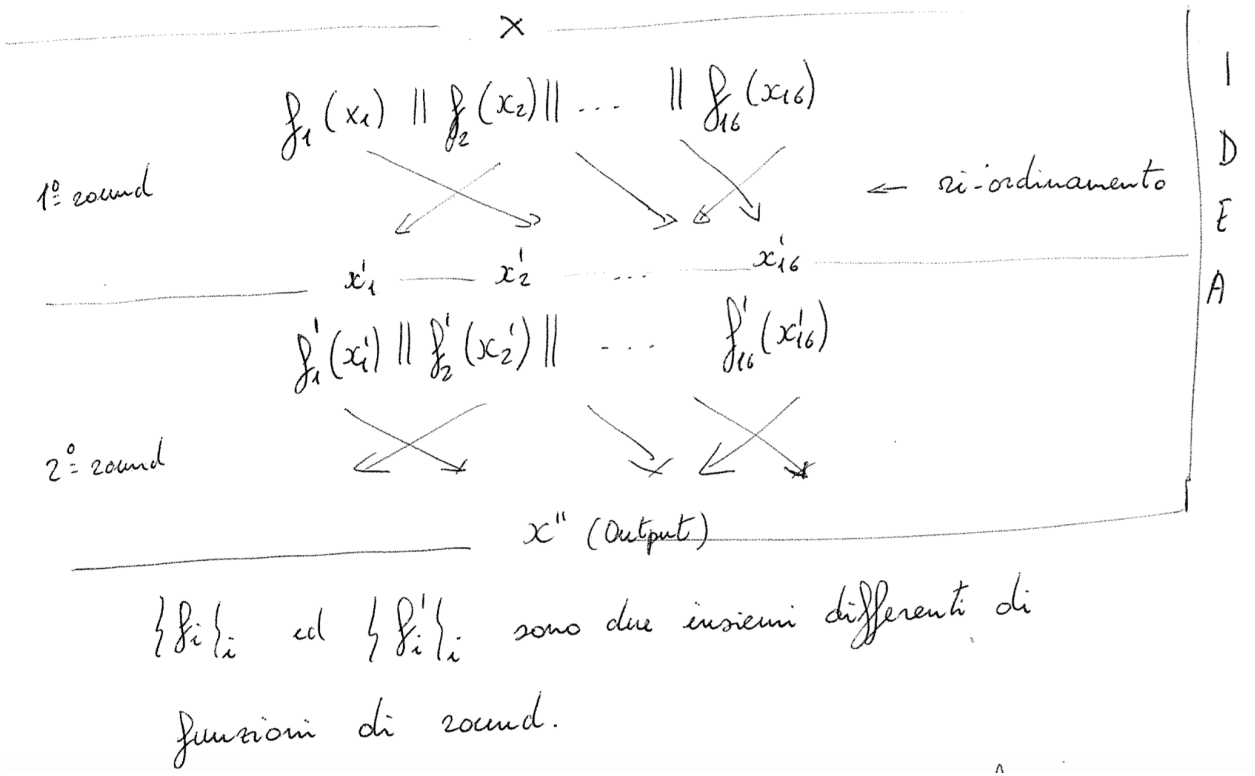
La confusione é *locale* al byte.

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 confusione e della diffusione.

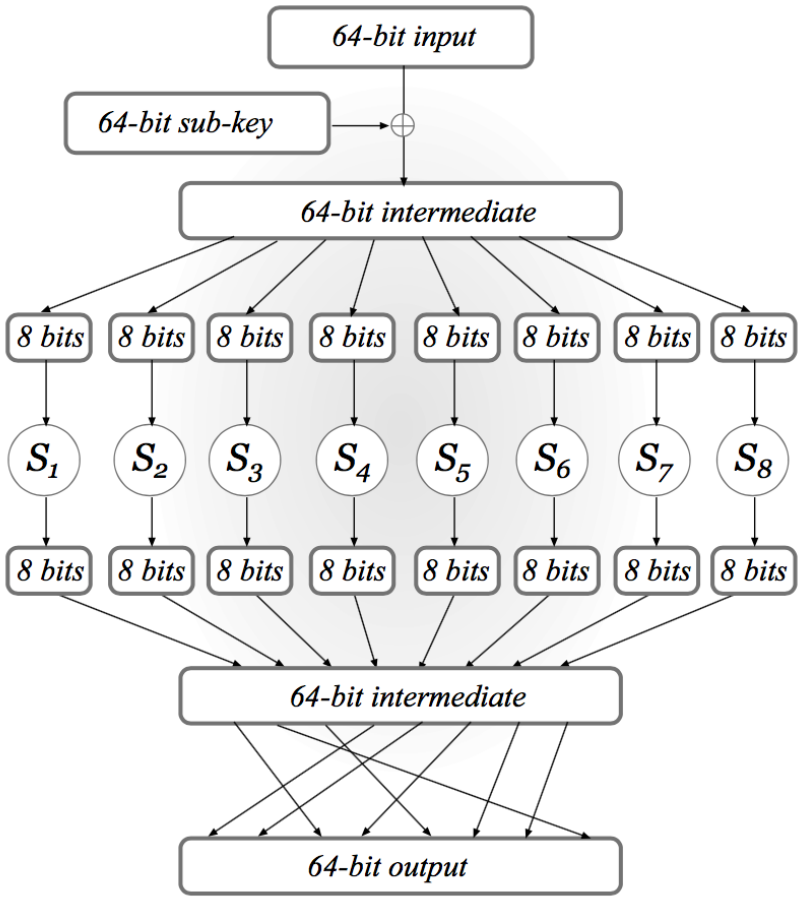
Idea di fondo:

- invece di usare una porzione della chiave k per scegliere una f_i, fissiamo una **funzione di sostituzione** pubblica S
- diremo che S é una 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
- **Mixing 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.

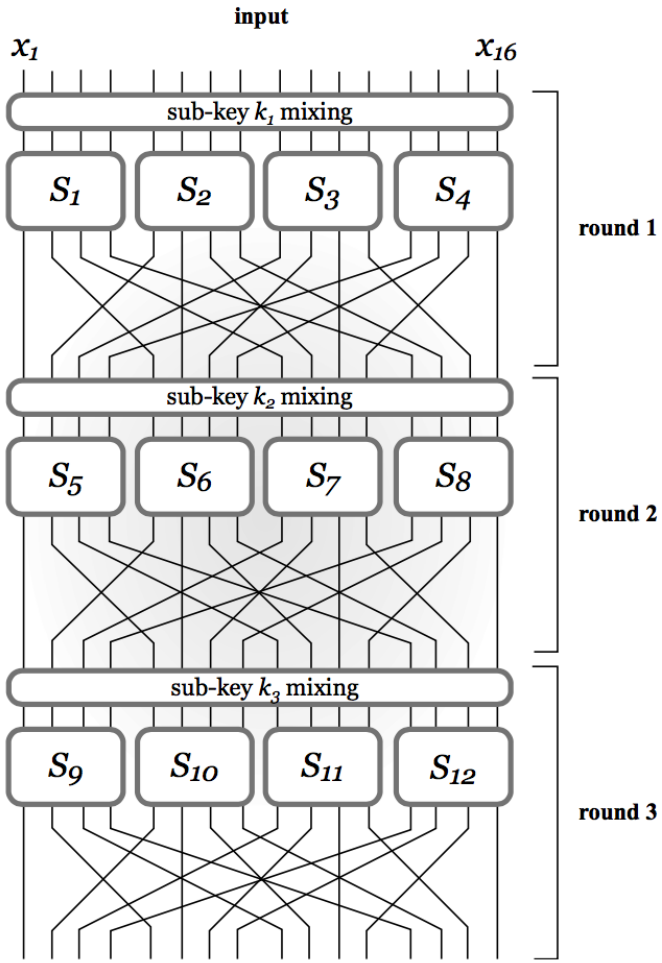
Una SPN é invertibile (data la chiave).

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

Effetto valanga

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
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.

Effetto valanga

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 di **un byte** di K_2 (2^8 in totale)

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

- per ognuno di essi, puó invertire il passo finale di key mixing, ottenendo l'output di una S-Box

- usando l'attacco precedente ottiene, per il byte di K_2, i valori di un byte di K_1

Per ogni ipotesi sul byte di K_2 ottiene un'unica scelta possibile per il byte di K_1. Ripete il processo per ognuna delle S-box.

Attacchi contro SPN con un numero di round ridotto

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

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

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

$\Rightarrow$ precedentemente era 2^{64}

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

Attacchi contro SPN con un numero di round ridotto

Euristicamente, un elemento in ogni lista di 2^8 coppie é consistente con (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 il byte di y' é $\frac{1}{2^8}$

$\Rightarrow$ coincide con la prob. con cui i 16 bit della chiave 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

Attacchi contro SPN con un numero di round ridotto

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 R 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

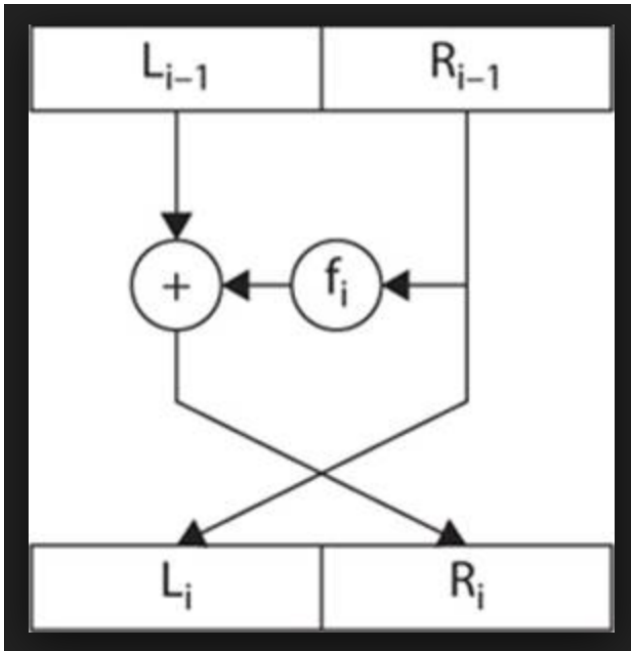
$$f_i : \{0,1\}^{\ell/2} \rightarrow \{0,1\}^{\ell/2} \qquad \text{come} \qquad f_i(R) \stackrel{def}{=} \hat{f}_i(K_i, R)$$

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:

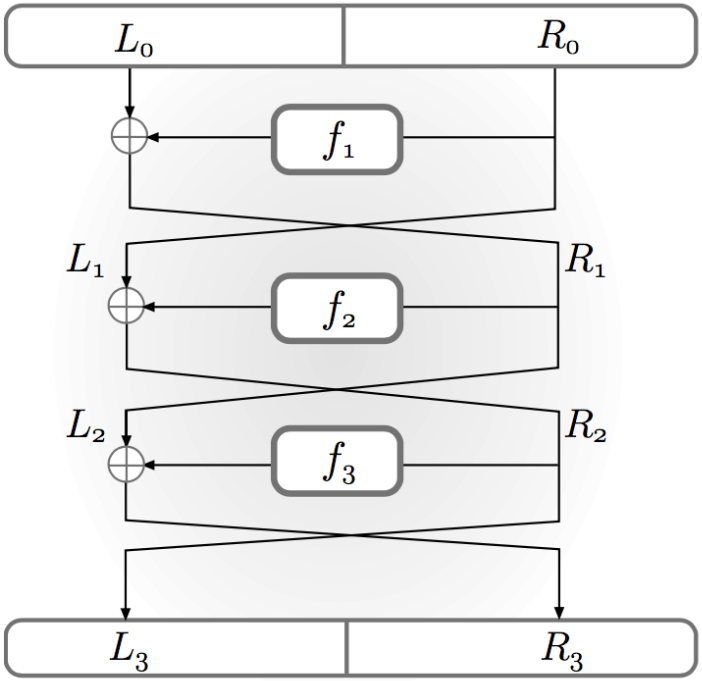
Input: L_{i-1} ed R_{i-1}.

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:

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

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