Università Ca' Foscari - Venezia
Approfondimento del corso di Sicurezza

Blowfish

Un cifrario a blocchi di Bruce Schneier

Andrea Marolla
Anno: 2008/2009

Indice generale

1. Introduzione

Blowfish è un cifrario a blocchi a chiave segreta simmetrica. Nasce nel 1993 dal suo autore Bruce Schneier.

Tale cifrario nasce con l'idea di trovare un sostituto valido a DES, un algoritmo di cifratura simmetrica che ha fatto storia e che ha bisogno di un degno sostituto che sia sicuro dai diversi tipi di attacco possibili ([1]).

Il cifrario, come indicato in [1], è libero da qualsiasi tipo di brevetto, l'algoritmo è di pubblico dominio e completamente disponibile al sito del suo autore ([2]). Questo è un fatto che distingue maggiormente Blowfish da altri cifrari a chiavi simmetrica. Poter infatti, avere pieno accesso all'algoritmo di cifratura (e decifratura) permette di studiare approfonditamente ogni tipo di attacco possibile e, nel caso che si rilevino problemi di questo tipo, si potranno proporre liberamente delle soluzioni.

1.1 Principali caratteristiche

Blowfish si basa sulla rete di Feistel ed itera una semplice funzione di cifratura 16 volte.

Interessanti caratteristiche del cifrario sono le seguenti:

- utilizza blocchi di 64 bits;
- la dimensione della chiave può arrivare fino a 448 bits;
- la cifratura prevede una prima fase di espansione della chiave ed una seconda fase di cifratura dei dati;
- fa uso di sottochiavi ed S-Boxes che sono dipendenti dalla chiave segreta utilizzata per la cifratura;
- è molto veloce. La seguente tabella comparativa, presentata in [3], confronta le velocità dei seguenti algoritmi di cifratura:

Algoritmo	Clock cycles per round	# of round	# of clock cycles per byte encrypted
Blowfish	9	16	18
RC5	12	16	23
DES	18	16	45
IDEA	50	8	50
Triple-DES	18	48	108

Illustrazione 1: Prestazione algoritmi (rielaborazione di una immagine di fonte: [3])

Nella progettazione del cifrario, come indicato in [1], sono state tenute in considerazione i seguenti aspetti di design:

- invece di manipolare singoli bit come fa DES sarebbe bene manipolare i dati in blocchi più grossi (preferibilmente di 32 bits);

- utilizzare blocchi di 64bit (ma essere scalabile anche per blocchi di 128bit);
- permettere una lunghezza di chiave variabile fino ad almeno 256bit;
- utilizzo di operazioni semplici che risultino efficienti sui microprocessori (XOR, addizioni, tabelle, moltiplicazioni modulo...) e non utilizzare shift di lunghezza variabile, permutazioni bit-a-bit e jump incondizionati;
- poter essere implementato su processori a 8-bits con un minimo di 24 byte di RAM (in aggiunta alla quantità di RAM richiesta per memorizzare la chiave) ed 1 kbyte di ROM;
- permettere chiavi precomputabili. In sistemi con grande memoria, queste sottochiavi possono essere precomputate per realizzare le operazioni successive in modo veloce. Senza precomputazione le operazioni saranno più lente ma comunque possibili;
- permettere un numero variabile di iterazioni;
- non avere, se possibile, chiavi deboli;
- utilizzare sottochiavi che sono un hash “one-way” della chiave di partenza;
- usare un design semplice da capire (utilizzo delle reti di Feistel).

Tali aspetti permettono a Blowfish di essere un cifrario sicuro e facilmente utilizzabile.

2. Cifrari a blocchi e reti di Feistel

Molti dei cifrari utilizzati al giorno d'oggi, come descritto dettagliatamente in [4], sono cifrari a blocchi del tipo “Iterated Cipher”, che utilizzano cioè una “round function” ed un key schedule”. La cifratura e la decifratura dei testi avviene tramite una serie di rounds simili.

Spesso tali cifrari fanno uso di una sequenza di operazioni di *permutazione* e di *sostituzione*. Lo scopo di tali operazioni di permutazione e sostituzione è quello di introdurre *confusione* (cioè di rendere complessa la relazione tra il testo in chiaro ed il testo cifrato) e *diffusione* (cioè di distribuire la struttura statistica del testo in chiaro per evitare attacchi statistici) nel testo cifrato, come descritto approfonditamente da C. Shannon nel suo articolo [5]. Tali proprietà sono fondamentali ed un buon cifrario le deve assolutamente implementare.

Un particolare tipo di “iterated cipher” è il cosiddetto cifrario di Feistel. Si tratta di un cifrario a blocchi composto da una particolare struttura sviluppata dal crittologo Feistel dell'IBM, da cui ne deriva il nome, nota appunto come rete di Feistel ([6]). Divenne molto popolare quando fu utilizzata in DES. Anche Blowfish basa il suo funzionamento su questo tipo di struttura.

Come descritto in [4] e [7], un cifrario di Feistel è definito nel seguente modo: è un “iterated cipher” che mappa un testo di dimensione 2t bits $P=(L_0, R_0)$ nei blocchi di t-bits L_0 e R_0 su un testo cifrato $C=(L_r, R_r)$ con un processo di r-rounds, dove il valore $r \geqslant 1$.

Il round i-esimo, con $1 \leqslant i \leqslant r$, mappa $(L_{i-1}, R_{i-1}) \xrightarrow{K_i} (L_i, R_i)$ nel seguente modo:

- $L_i = R_{i-1}$;
- $R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$ (dove ogni K_i deriva dalla chiave principale K).

Un round di un cifrario di Feistel è sempre invertibile, infatti:

- $L_{i-1} = R_i \oplus f(L_i, K_i)$;
- $R_{i-1} = L_i$.

Vantaggi sull'utilizzo di questo tipo di cifrario sono, tra gli altri: quello di avere operazioni di cifratura e decifratura molto simili e quello permette di avere cifrari con un design facilmente comprensibile.

Come indicato in [6], cifrari che usano la rete di Feistel (o versioni modificate) sono ad esempio i seguenti: Blowfish, Camellia, CAST-128, DES, FEAL, Lucifer, Twofish...

3. Funzionamento di Blowfish

Blowfish, come già detto, è un cifrario a chiave simmetrica con una dimensione di blocchi di 64bit ed una lunghezza della chiave fino a 448bit.

Seguendo la descrizione presente in [1], si osserva che l'algoritmo consiste in due parti:

1. una prima parte di espansione della chiave che si occupa di generare delle sottochiavi da utilizzare nella fase di cifratura;
2. una seconda parte di cifratura dei dati.

Utilizza anch'esso una versione della rete di Feistel per la cifratura/decifratura dei testi e si compone di 16 cicli. Ogni round consiste in:

- una operazione di permutazione dipendente dalla chiave;
- una operazione di sostituzione dipendente sia dalla chiave che dai dati.

Le operazioni che utilizza sono solamente XORs e addizioni su words di 32 bits.

Un'altra interessante caratteristica è che utilizza S-BOX grandi e dipendenti dalla chiave, per rendere più complessi i possibili attacchi.

3.1 Fase di generazione delle sottochiavi

Blowfish usa un gran numero di sottochiavi che devono essere precalcolate per poter effettuare operazioni di cifratura e decifratura dei dati ([1]).

A partire dalla chiave simmetrica, in questa fase, si ricavano:

– il P-array composto da 18 sottochiavi di 32 bits (P1, P2, ..., P18);

– 4 S-BOX, ognuna delle quali costituita da 256 sottochiavi di 32bit

(S1,0, S1,1, ..., S1,255; S2,0, S2,1, ..., S2,255; ..., S4,0,S4,1, ..., S4,255).

Le sottochiavi vengono generate attraverso il seguente procedimento:

1. si inizializzano il P-array e le quattro S-BOX con una stringa fissa inizializzata con valori esadecimali di π;
2. si esegue l'operazione di XOR tra:
   - P1 e i primi 32 bits della chiave (cioè i bits che vanno da 0 a 31);
   - P2 e i “secondi” 32 bits della chiave (cioè i bits che vanno da 32 a 63);
   - ... si continua in questo modo per tutti i bits della chiave (che può essere lunga fino a 448 bits, dunque fino a P14) e reiterando se necessario per completare tutti i P_i;
3. si codifica una stringa di 0 con Blowfish usando le sottochiavi ottenute nei passaggi precedenti;
4. si sostituiscono i valori di P1 e P2 con l'output appena ottenuto al passo 3;
5. si cifra l'output ottenuto al passo 3, usando Blowfish con le chiavi modificate;

6. si sostituiscono i valori di P3 e P4 con l'output appena ottenuto al passo 5;
7. ... si continua con questo processo sostituendo tutte le entry del P-array e, successivamente, tutte le entry delle quattro S-BOX.

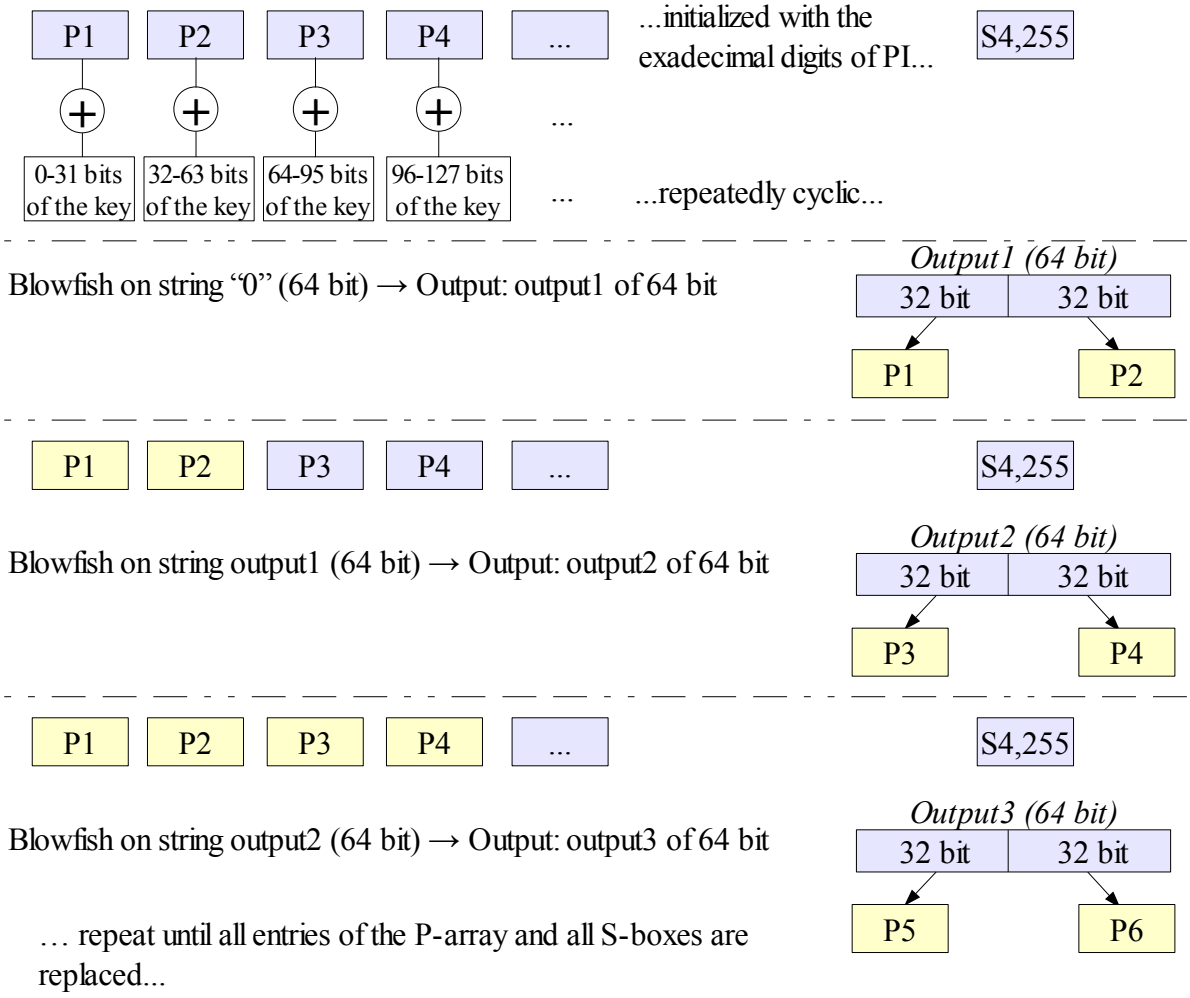

Illustrazione 2: Generazione del P-array e delle S-boxes

E' interessante notare che sono necessarie 521 iterazioni totali per generare la totalità delle sottochiavi, visto che ne servono 9 per generare il P-array e 128*4 per le S-BOX. Ciò significa che l'algoritmo risulta essere molto veloce a patto di utilizzare la stessa chiave ([8]), altrimenti è necessario ogni volta rieffettuare le 521 iterazioni.

3.2 Fase di cifratura

La struttura di Blowfish si basa sulla rete di Feistel e consiste di 16 round.

A partire dal testo in input X (di 64bit) si procede secondo il seguente algoritmo:

```
Dividi x in due parti di 32 bits xL, XR
FOR i=1 to 16:
      xL = xL ⊕ Pi  (dove Pi è la i-esima chiave del P-array)
      xR = f(xL) ⊕ xR
      Swap xL and xR
      next i
Swap xL and xR (utile per annullare l'ultimo Swap)
xR = xR ⊕ P17
xL = xL ⊕ P18
Ricombina xL e xR
```

Illustrazione 3: Algoritmo Blowfish (fonte: [1])

La funzione f utilizzata nell'algoritmo è la seguente:

Si suddivide xL in 4 blocchi di 8 bits: a, b, c, d

$F(xL) = ((S1,a + S2,b \bmod 2^{32}) \oplus S3,c) + S4,d \bmod 2^{32}$

Illustrazione 4: Funzione f di Blowfish (fonte: [1])

I diagrammi presenti nella successiva pagina rappresentano sinteticamente il funzionamento dell'algoritmo di Blowfish (Illustrazione 4) e della funzione f (Illustrazione 5).

Si è detto che un lato positivo della rete di Feistel è che la cifratura e decifratura sono operazioni molto simili. Visto che Blowfish si basa proprio su questa struttura ha anche questa caratteristica. Infatti la decifratura è esattamente uguale alla cifratura, solo che le P1, P2, ... P18 vengono usate nell'ordine inverso ([1]).

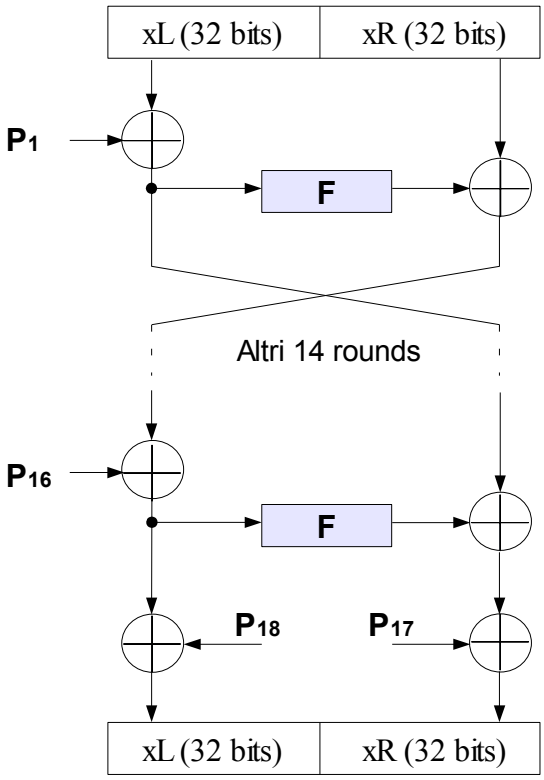

Illustrazione 5: Schema dell'algoritmo Blowfish

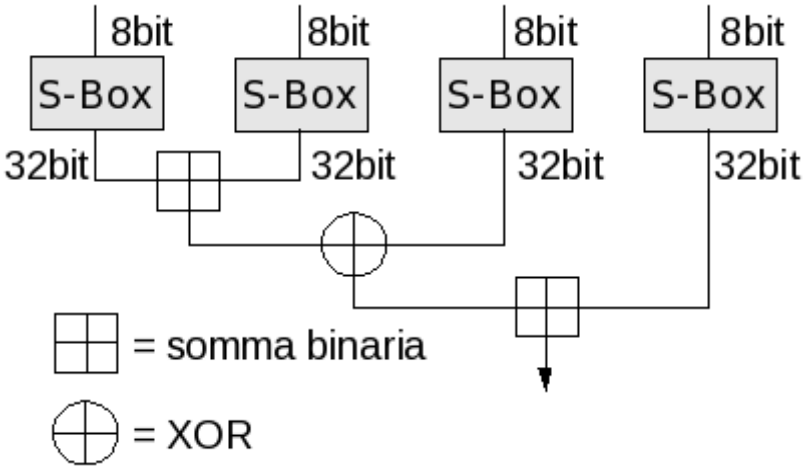

Illustrazione 6: Funzione f di Blowfish

4. Crittoanalisi e debolezze di Blowfish

Blowfish, come già accennato, si differenzia da altri cifrari a chiave simmetrica, tra le altre cose, per la sua totale libertà da brevetti, ma soprattutto per il fatto che il suo algoritmo è completamente di pubblico dominio. Come già detto, nel 1994 fu pubblicato l'articolo che descriveva questo nuovo cifrario, da parte del suo autore Bruce Schneier. Con la pubblicazione dell'algoritmo il magazine di software “Dr. Dobb's Journal”, come descritto in [1], ha indetto un “contest” (una sfida), con un premio di $1000, per la migliore crittoanalisi di Blowfish che avrebbe ricevuto entro Aprile 1995 (cioè un anno dopo la pubblicazione dell'algoritmo).

Naturalmente, un tipo di attacco sempre possibile in un qualsiasi cifrario è quello di tipo “brute force” (forza bruta), dove cioè si prova a decifrare il testo utilizzando tutte le possibili chiavi. Se la dimensione della chiave è adeguata, un attacco di questo tipo non è possibile poichè richiederebbe troppo tempo per essere attuato (ad esempio una chiave di 64 bits, che al giorno d'oggi non è più considerata sicura perché troppo piccola, richiederebbe 2^{64} prove di combinazioni diverse), si tratta dunque di studiare altri tipi di attacchi o debolezze di Blowfish.

Nel Settembre 1995, Bruce Schneier ha pubblicato un secondo articolo, [9], dove riassumeva le caratteristiche ed il funzionamento del cifrario e dove parlava dei tentativi di crittoanalisi inviati al “Dr. Dobb's Journal”. In tutto, di attacchi, ne furono inviati 5 ma è importante sottolineare che nessuno di questi è stato in grado di crittoanalizzare la versione completa dell'algoritmo Blowfish (composta da 16 rounds).

Come descritto in [9], Jon Kelsey è riuscito a sviluppare un attacco capace di rompere una versione modificata di Blowfish funzionante con 3 round, ma senza riuscire però ad estendere tale attacco ai 16 round. Vikramijit Singh Chhabra ha cercato dei metodi per implementare una macchina che permetta una ricerca efficiente della chiave tramite brute-force.

I tentativi di crittoanalisi più interessanti furono però quelli inviati da Serge Vaundenay, pubblicati in [8], poiché evidenzia la presenza di sottochiavi che possono essere considerate deboli.

4.1 Crittoanalisi di Serge Vaundenay

Serge Vaundenay, nel suo articolo ([8]), presenta tre diversi tipi di attacchi a Blowfish. I primi due attacchi prevedono la conoscenza della parte della chiave che descrive la funzione f (cioè tutte le sottochiavi delle S-Boxes, ma non le sottochiavi P1, P2, …, P18), mentre il terzo non prevede alcun tipo di conoscenza e riesce solamente a dire se la chiave usata ha una particolare struttura poco sicura, cioè se è “debole”.

La cosa più interessante dunque è che nel suo articolo, Serge Vaundenay riesce dunque ad evidenziare la presenza di chiavi deboli in Blowfish.

Una chiave è debole (nota anche con il termine di “weak key”) in Blowfish se almeno una delle 4 S-BOX ha una collisione ([8]). Per chiarire il concetto si può supporre ad esempio che questo accada per la S-BOX S1. Si è visto che ogni S-Box è composta da 256 byte. Dire che esiste una collisione su tale S-Box significa che su S1 esistono 2 byte diversi, chiamiamoli `a` e `a'`, per cui `S1,a=S1,a'` (ad esempio potrebbe

succedere che S1,2 = S1,8).

Sfruttando la presenza di una chiave debole e conoscendo le entries delle S-BOXes è possibile, attraverso un attacco “chosen plaintext”, descritto dettagliatamente in [8], ottenere tutte le $P_i, \forall i = 1,...,10$ in una versione di Blowfish a 8 round, con un numero di coppie di “chosen plaintext” pari a 2^{23} (su 16 round sono invece necessarie 3×2^{51} coppie).

Come indicato in [8], la probabilità di avere una “weak key”, cioè che vi sia una collisione su una delle S-BOXes è 2^{-15} , ciò significa che una chiave ogni 2^{15} mediamente è debole.

Considerando sempre una versione di Blowfish ad 8-round, è possibile verificare se la chiave usata per cifrare un testo è debole, senza conoscere nulla delle entries delle S-BOXes, attraverso un attacco “chosen plaintext”, in cui sono necessarie 2^{22} chosen plaintext ([8]).

Se la chiave non è debole, conoscendo le entries delle S-BOXes, è comunque possibile un attacco sempre di tipo “chosen plaintext” sfruttando possibili collisioni nella funzione f. Nella versione di Blowfish ad 8 round è possibile crittoanalizzare tutte le $P_i, \forall i = 1,...,10$, ponendosi in questo caso, con 2^{48} chosen plaintexts.

Schneier, nel suo articolo [9], suggerisce, se si è preoccupati di avere a che fare con una “weak key”, di effettuare l'espansione della chiave e controllare se vi sono entries uguali nelle S-BOXes, ma non lo reputa necessario.

4.2 Debolezza nelle sottochiavi del terzo e quarto round

Dieter Schmidt nel suo [10], ha mostrato che le sottochiavi del terzo e quarto round che vengono generate non dipendono dai primi 64bit della chiave.

Questa “debolezza” in Blowfish deriva da come è implementato l'algoritmo di generazione delle sottochiavi, a partire dalla chiave principale, che è stato descritto nella sezione 3.1.

Si è visto infatti che:

1. per prima cosa P1, …, P18 vengono inizializzati con valori esadecimali di π;
2. si esegue l'operazione di XOR per ogni Pi con i corrispettivi 32 bits della chiave `[32*(i-1)....32*(i)]` (per ogni i=1,...,18) fino al possibile, reiterando in caso di bisogno per riassegnare tutti i P1....P18;
3. si cifra con Blowfish e le sottochiavi P1,...,P18 appena generate la stringa 0 ottenendo blocco X di 64 bits (visto che il testo 0 di partenza è di 64 bits) cifrato;
4. si sostituiscono i valori di P1 e P2 con l'output appena ottenuto, e lo si fa nel seguente modo ([10]): P1 = XL (primi 32 bits di X) e P2 = XR (secondi 32 bits di X);
5. si cifra l'output della fase 3 con Blowfish e le nuove sottochiavi;
6. si assegnano a P3 = XL e P4 = XR del nuovo output ottenuto dalla fase 5;
7. si continua in questo modo fino a generare tutti i Pi e tutte le entry delle S-Box.

Il problema sta proprio nel punto (4) appena elencato. In [10] infatti si fa notare che nell'articolo [1] non è esplicitato chiaramente che a P1 sono assegnati i primi 32 bits di X e a P2 i secondi 32 bits, ma si dice più genericamente che a P1 e P2 sono assegnati i bits di X, ma analizzando il codice sorgente si osserva che l'assegnamento è proprio quello descritto al punto (4).

Il motivo per il quale questo tipo di suddivisione porta ad avere le sottochiavi P3 e P4 a non essere dipendenti dai primi 64bit della chiave è il seguente: poniamoci nella fase di generazione delle chiavi:

- Alla fine della prima iterazione (quella che definisce i reali valori di P1 = XL e P2 = XR del testo 0 cifrato) si ha appunto che P1 = XL e P2 = XR.
- Quando inizia la seconda iterazione (quella che assegnerà i nuovi valori definitivi di P3 e P4, cifrando il nuovo testo in output ottenuto dalla fase 3), le operazioni che vengono eseguite nel primo round sono le seguenti:
  - XL = XL $\oplus$ P1 = XL $\oplus$ XL = 0 (dunque XL diventa = 0 !!!);
  - XR = XR $\oplus$ F(0) $\oplus$ P2 = XR $\oplus$ F(0) $\oplus$ XR = F(0);
- al secondo round della seconda iterazione XL diventa:
  - XL = 0 $\oplus$ F(F(0)) = F(F(0));
- dunque dal terzo round della seconda iterazione sia XL che XR non dipendono più dalla chiave di partenza!

Chiaramente questo non è un problema grave da compromettere la sicurezza dell'algoritmo, visto che va a toccare solamente 2 delle 18 sottochiavi, ma è comunque un piccolo “difetto” da tenere in considerazione.

4.3 Conclusioni sulla sicurezza del cifrario

Da quanto detto, si può concludere, che ad oggi, non esistono, o comunque non sono stati resi noti, attacchi alla versione completa di Blowfish, che comprende tutti e 16 i rounds.

Gli aspetti di progetto, elencati nell'articolo che presenta l'algoritmo ([1]), sono stati dunque in buona parte rispettati, anche se purtroppo si è visto che esistono delle chiavi deboli (come descritto dettagliatamente in [8]) e che le sottochiavi del terzo e quarto round che vengono generate non dipendono dai primi 64 bits della chiave (come descritto dettagliatamente in [10]), difetti che naturalmente si volevano evitare. In ogni caso, queste piccole “debolezze” non sono state sufficienti per consentire di effettuare una crittoanalisi della versione completa (16 rounds) di Blowfish, dunque il cifrario può essere considerato sicuro.

5. Applicazioni di Blowfish

Blowfish è applicato a moltissimi casi reali. Nel suo sito, Bruce Schneier ([2]) presenta una lista di oltre 150 applicazioni che lo utilizzano, tra i quali è possibile citare: OpenBSD, PasswordSafe, Truecrypt... Gode inoltre di una certa notorietà, a prova di ciò si può anche far notare che è stato citato in un episodio del famoso serial televisivo “24” americano ([2]).

Naturalmente i motivi del suo utilizzo pratico trovano riscontro nelle sue qualità:

- prima di tutto sicurezza della cifratura;
- velocità di cifratura se la chiave usata per cifrare non cambia;
- ma anche “lentezza” in caso cifratura con chiavi diverse.

Una delle sue caratteristiche di punta è infatti la sua velocità di codifica, a patto di non cambiare la chiave, poiché in tal caso bisogna rigenerare le sottochiavi e le S-BOX, operazione che richiede del tempo e lo porta invece ad essere classificato come un cifrario lento ([11]). Grazie proprio a questo fatto, una variante di Blowfish è utilizzata in OpenBSD come metodo di hashing per le password, in modo da evitare attacchi di tipo “forza bruta”. Per provare ogni possibile chiave infatti, si devono generare ogni volta tutte le sottochiavi, dunque l'attacco diventa inapplicabile.

Qui si seguito si approfondisce l'applicazione di Blowfish in OpenBSD come metodo di hashing delle password, seguendo la descrizione dettagliata presentata in [12].

5.1 bcrypt in OpenBSD

In OpenBSD, sistema operativo open famoso per la sua particolare attenzione riguardo la sicurezza Blowfish viene utilizzato per la funzione hash `bcrypt`, usata per tenere nascoste le password di sistema nel file delle password.

5.1.1 File delle password

Un utente che vuole avviare una sessione di sistema solitamente inserisce la sua “UserID” e la sua “Password” per effettuare il login.

Per verificare che la password inserita sia quella corretta, il sistema mantiene un file delle password, che contiene appunto le diverse password, cifrate opportunamente, dei diversi utenti del sistema.

In UNIX, le password vengono memorizzate nel file delle password, con una funzione hash che ha l'importante caratteristica di essere “one-way” (ovvero una funzione hash h: X → Y dove dato un $y \in Y$ è computazionalmente intrattabile trovare $x \in X$ tale che $h(x) = y$ ([4])). Tale funzione hash è: `crypt`.

L'idea è dunque quella di mantenere questo file, dove per ogni utente è memorizzato il relativo hash della password. Quando l'utente invia il proprio “userID” e la propria “password” per effettuare il login:

1. il sistema controlla che il nome utente “userID” sia presente nel file delle password;

2. calcola l'hash della “password”;
3. controlla che tale hash calcolato sia uguale a quello presente nel suo file delle password salvato in corrispondenza dell'utente “userID”.

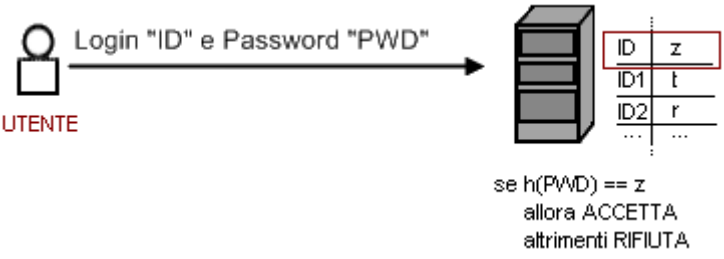

Illustrazione 7: Login al sistema

Naturalmente è anche importante che la funzione hash usata sia anche second preimage resistant (ovvero che dato x, deve essere intrattabile trovare $x' \neq x$ tale che h(x)=h(x'), [4]), o ancora meglio (strong) collision resistant (cioè che deve essere intrattabile trovare x, x' diversi tali che h(x)=h(x'), [4]).

In OpenBSD l'amministratore può scegliere lo schema di hashing delle password agendo sul file “passwd.conf” ([12]). Per differenziare tra hash di password che usano diversi algoritmi, ogni hash di password che utilizza un algoritmo diverso da crypt standard ha un prefisso identificativo diverso (MD5-crypt: 1; bcrypt: $2a$) ([12]).

5.1.2 Attacchi al file delle password

Per cercare di scoprire la password di un utente, si può procedere con attacchi di tipo “forza bruta”, cioè attacchi dove l'attaccante S prova tutte le possibili password. Si può anche procedere con un attacco che prova tutte le password più comuni contenute in un dizionario di hashes di passwords precomputato, partendo dall'idea che solitamente gli utenti del sistema utilizzano passwords comuni ([7]), che riescono cioè a ricordare facilmente (esempi possono essere: $tefano1, peter83, H1lary78...).

Gli attacchi possibili sono i seguenti:

- Attacco on-line: L'attaccante S prova tutte le possibili password per ogni utente, inviando ogni volta una richiesta di login. Naturalmente il sistema risponde con il login solamente quando i dati sono corretti. Questo tipo di attacco è evitabile ponendo un numero massimo di tentativi o comunque rallentando il tempo di verifica tra un tentativo e l'altro in modo da rendere troppo costoso l'attacco;
- Attacco off-line: l'idea è quella di ottenere in qualche modo il file delle password e poi effettuare un attacco brute-force su di esso, o comunque attraverso un attacco che si basa su un dizionario di hashes di passwords comuni. In questo secondo caso, basterebbe una semplice ricerca e confronto degli hashes. Questo è un attacco di tipo “off-line” in quanto non richiede l'invio da parte di S di “userID” e “password” ma è possibile attaccare direttamente il file delle passwords. I metodi visti per rallentare l'attacco on-line non sono dunque validi. Questo tipo di attacco è infatti più complesso da trattare.

Un metodo per complicare l'attacco al file delle password è quello di utilizzare un cosiddetto “salt” di un certo numero di bits. Ogni volta che un utente “userID” modifica

la sua “password”, il sistema gli genera casualmente un “salt”, che deve avere una dimensione in bits abbastanza grande, e va a memorizzare nel file delle password: `userID, salt, h(password, salt)` (invece che solamente `userID, h(password)`). Si memorizza dunque l'hash della password assieme al salt. Il “salt” è diverso per ogni utente visto che è generato casualmente, dunque un attaccante S non può usare una tabella di password (dizionario) singola per tutti gli utenti, visto che ogni utente ha il proprio salt (una tabella di questo tipo dovrebbe memorizzare per ogni utente le possibili passwords con tutti i possibili salt, ottenendo così dimensioni che dovrebbero essere intrattabili).

E' importante fare alcune considerazioni, approfonditamente trattate in [12] e qui di seguito riportate:

- Nel 1976, quando è stato introdotto crypt in UNIX, esso non era in grado di effettuare l'hash di più di quattro password per secondo. Dopo circa 20 anni (anni 2000), con il continuo miglioramento dell'hardware e con il software sempre più performante era possibile effettuare oltre 200000 operazioni di crypt al secondo. E' necessario dunque tener conto di questo aspetto parametrizzando il costo della funzione hash di cifratura delle password.
- Oggi esistono sistemi più sofisticati del file delle password di UNIX, ma solitamente essi comunque dipendono comunque da una password. Altri approcci che non richiedono password, come con “authentication hardware”, sono più difficili da implementare, dunque l'uso delle password gioca ancora un ruolo fondamentale ([12]) nei sistemi moderni. Il metodo del file delle password è un buon metodo solo che è necessario dunque utilizzare una funzione hash che sia difficile da attaccare.

Tali considerazioni permettono di far capire che le password sono ancora tutt'oggi un sistema molto usato e che è necessario parametrizzare il costo della funzione hash in modo da adattarla agli aggiornamenti dell'hardware. Una buona funzione per la cifratura delle password deve dunque rendere difficile il recuperare informazioni della password, ma deve anche fare in modo di rendere utilizzabile la funzione di cifratura della password per valutare la correttezza di un login.

5.1.3 crypt UNIX

In questa sezione viene descritto il “crypt” tradizionale usato in UNIX, secondo la descrizione presentata in [12].

Il crypt tradizionale è nato nel 1976, usa DES e permette di utilizzare password di un massimo di 8 caratteri. Fa uso anche di “salt” a 12 bits.

L'idea che sfrutta è la seguente:

- si genera la chiave per il DES (che deve essere di 56 bits) a partire dalla password di 8 caratteri (che viene completata con degli 0 se ha lunghezza minore, o troncata a 8 caratteri se ha lunghezza superiore). Lo si fa combinando i 7 bits “low-order” di ognuno degli 8 caratteri della password;
- si usa un “salt” generato casualmente di 12 bits in modo che la stessa password possa generare 4096 cifrature. Lo si usa nella funzione E del DES, che viene modificata per eseguire la seguente operazione: scambio dei bits *i* e

i+24 quando il bit *i* è settato nel salt;

– si cifra il testo “0” di 64 bits per 25 volte (in modo da rallentare gli attacchi) con la chiave generata.

$$h(password) = \left(DES^{salt}_{56\text{bit } password}(0)\right)(25\, iterazioni)$$

In output si ottiene il salt di 12bit concatenato con i 64bit cifrati della chiave.

Esistono diverse versioni modificate della versione di crypt appena descritta. Ad esempio MD5 crypt è una evoluzione di crypt che permette di avere una password di lunghezza illimitata ed un salt di dimensione variabile tra 12 bits e 48 bits.

Sia crypt che MD5 crypt hanno comunque costi fissi e sono soggetti ai problemi descritti nelle precedenti sezioni.

5.1.4 bcrypt

Come già detto, in OpenBSD (già dalla versione 2.1), per la funzione di cifratura delle passwords si usa (come standard) una versione modificata di Blowfish che parametrizza, in qualche modo, anche il costo della computazione. Qui di seguito vengono descritti i passi implementativi dell'algoritmo bcrypt, trattati approfonditamente in [12].

La versione dell'algoritmo `Eksblowfish`, usato in OpenBSD, è molto simile all'originale, infatti si compone sempre di 16 iterazioni dove, ad ogni iterazione si eseguono esattamente le operazioni di Blowfish:

- $R_i = L_{i-1} \oplus P_i$
- $L_i = R_{i-1} \oplus F(R_i)$

infine: $R_{17} = L_{16} \oplus P_{17}$ e $L_{17} = R_{16} \oplus P_{18}$.

Anche la funzione F è esattamente la stessa del Blowfish originale. La differenza nella versione usata in OpenBSD sta nel modo in cui vengono generati il P-array e le quattro S-Boxes. La versione originale di Blowfish inizializza tali strutture come descritto nella sezione 3.1, mentre qui utilizza la seguente funzione `EksBlowfishSetup` (descritta dettagliatamente in [12]):

```
EksBlowfishSetup(cost, salt, key)
      state ← InitState()
      state ← ExpandKey(state, salt, key)
      repeat (2^cost)
            state ← ExpandKey(state, 0, salt)
            state ← ExpandKey(state, 0, key)
      return state
```

*Illustrazione 8: EksBlowfishSetup (fonte: **[12]**)*

Come si può vedere ha 3 parametri in input:

- `cost`: parametrizza il costo della computazione;
- `salt`: parametrizza il salt (di 128 bits) da usare nella computazione;
- `key`: chiave da usare nella cifratura.

Restituisce in output il P-array e le S-boxes.

La funzione `InitState()` non fa altro che copiare i valori di π nel P-array e nelle S-boxes, come visto nell'algoritmo originale, e restituire lo stato in `state`.

La funzione `ExpandKey(state, salt, key)` si occupa di modificare il P-array e le S-boxes in base ai parametri che gli sono stati passati. Per prima cosa esegue una operazione di XOR tra tutte le sottochiavi del P-array e la chiave di cifratura:

- i primi 32 bits della chiave XOR P1;
- i secondi 32 bits della chiave XOR P2...

Il processo è ciclico, dunque quando la chiave termina si riparte con i primi 32 bits della chiave e si continua per tutti i P_i del P-array.

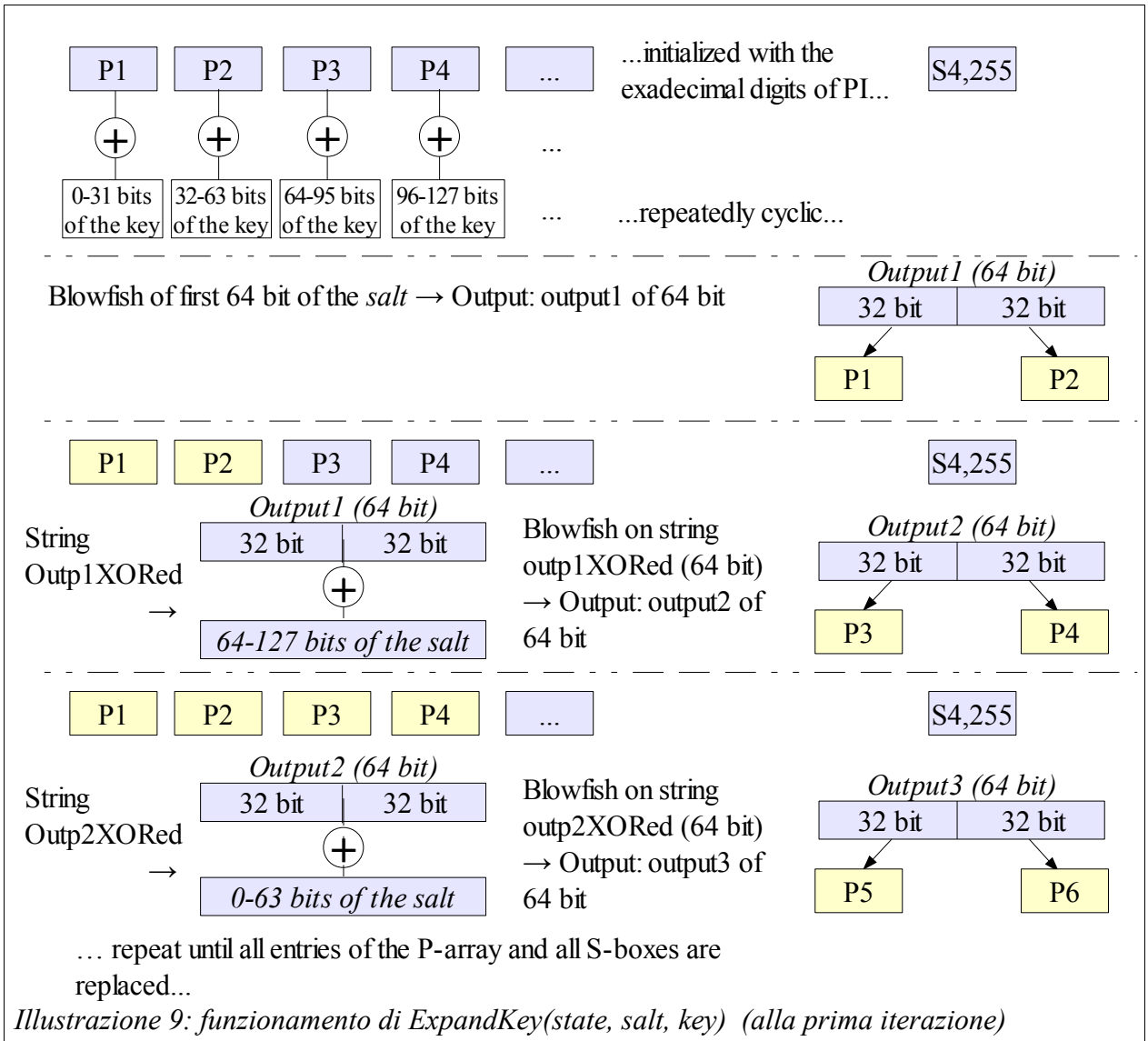

Illustrazione 9: funzionamento di ExpandKey(state, salt, key) (alla prima iterazione)

A questo punto cifra con Blowfish i primi 64 bits del salt. Il testo cifrato risultante va dunque a sostituire P1 e P2. Si esegue poi l'XOR dello stesso testo cifrato (di 64 bits) con i secondi 64 bits del salt e il risultato cifrato con il nuovo stato ottenuto al passo precedente. L'output ottenuto va a sostituire P3 e P4. Si esegue dunque l'XOR del testo cifrato con i primi 64 bits del salt, si cifra ed il risultato va a sostituire P5 e P6. Si continua in questo modo, alternando i primi ed i secondi 64 bits del salt, fino a sostituire tutto il P-array e tutte le S-Boxes, ottenendo così il nuovo stato `state` che dipende sia dalla chiave che dal salt.

A questo punto si eseguono 2^{cost} iterazioni, alternando due chiamate:

- `state ← ExpandKey(state, 0, salt)`: che usa come salt il testo 0 (di 128 bits) e come chiave il salt di 128bit;
- `state ← ExpandKey(state, 0, key)`: che usa come salt il testo 0 (di 128 bits), cioè è equivalente ad una singola iterazione dell'algoritmo di generazione delle chiavi di Blowfish standard.

Al termine di queste iterazioni si ottiene finalmente lo stato finale `state`.

A questo punto, `eksblowfish` (che usa per generare lo stato `EksBlowfishSetup`), viene usato nella funzione `bcrypt` usata per cifrare le passwords.

L'algoritmo `bcrypt` usa:

- un salt di 128 bits;
- cifra un “valore magico” di 192 bits;
- e sfrutta il costoso metodo di generazione delle sottochiavi di `eksblowfish`.

L'algoritmo è qui di seguito presentato:

```
bcrypt(cost, salt, pwd)
     state ← EksBlowfishSetup(cost, salt, key)
     ctext ← “OrpheanBeholderScryDoubt”
     repeat (64)
          ctext ← EncryptECB(state, ctext)
     return Concatenate(cost, salt, ctext)
```

*Illustrazione 10: bcrypt (fonte: **[12]**)*

Come si può vedere, per prima cosa si genera `state` con la chiamata alla funzione `EksBlowfishSetup(cost, salt, key)`. La maggior parte del tempo richiesto da bcrypt viene speso in questa chiamata, fatto comprensibile per tutto quello che è stato detto finora. A questo punto viene cifrato 64 volte il testo “OrpheanBeholderScryDoubt” di 192 bits usando `eksblowfish` in modalità ECB (ovvero ogni blocco di 64 bits viene cifrato con la stessa chiave).

In output si ottiene dunque la concatenazione di `cost`, i 128 bits del `salt` e il risultato della cifratura, cioè `ctext`.

5.1.5 Sicurezza di bcrypt

La funzione bcrypt, secondo [12], garantisce tutti gli aspetti di sicurezza precedentemente discussi, cioè garantisce:

- di avere la proprietà di essere second preimage resistant;
- di essere maggiormente sicura da attacchi con dizionari grazie all'uso di un salt di grande dimensione (128 bits);
- di avere costo adattabile.

E' importante sottolineare che un salt a 128 bits garantisce una buona sicurezza da attacchi che usano dizionario, visto che costruire un dizionario tenendo in considerazione un salt a 128 bits richiederebbe uno spazio enorme. Inoltre un salt a 128 bits permette anche di evitare “salt collisions” (dove un “salt collision” si ha quando due password vengono cifrate con lo stesso salt, [12]).

Il costo adattabile permette a bcrypt di adattarsi ai potenziamenti hardware che avvengono nel corso degli anni in modo da mantenere sicuro il metodo usato. L'amministratore può specificare il parametro “cost” agendo sul file “passwd.conf”. Al momento della pubblicazione dell'articolo [12], i valori utilizzati di “cost” erano: 6 per un utente normale; 8 per un superuser.

6. Conclusioni

In questo approfondimento si è cercato di spiegare il funzionamento del cifrario a blocchi Blowfish seguendo principalmente la descrizione approfondita degli articoli del suo autore, Bruce Schneier ([1],[9]), cercando inoltre di evidenziare i suoi obiettivi di progetto.

Per prima cosa, si è cercato di presentare una brevissima introduzione ai cifrari a blocchi moderni, seguendo le descrizioni approfondite presenti principalmente in [4], [7], si è passati poi subito a Blowfish.

Dopo averne descritto il funzionamento, cioè come avviene la creazione delle sottochiavi e la compressione dei testi in chiaro, si è cercato di analizzare l'aspetto riguardante la sicurezza del cifrario, analizzando principalmente gli articoli [8],[9],[10]. Dalle analisi effettuate risulta la presenza di alcune chiavi deboli nel cifrario ([8]) che non compromettono comunque la sua sicurezza nella versione completa di 16 rounds. Inoltre viene evidenziato il fatto che le sottochiavi del terzo e quarto round non dipendono dai primi 64bit della chiave ([10]). Anche questo comunque è un aspetto che non va a compromettere la sicurezza del cifrario.

Si è poi cercato di far vedere che questo cifrario è utilizzato in molte applicazioni reali, come visibile dalla lista di applicazioni che usano Blowfish presente nel sito di Schneier [2]. Infine si è cercato di analizzare l'implementazione di Blowfish in OpenBSD come sistema di hashing delle password, seguendo l'articolo [12]. Si è cercato di dare per prima cosa una introduzione generale al problema della gestione delle password in un sistema, seguendo le descrizioni presenti in [7] e [12] e si è poi passati alla descrizione più approfondita dell'algoritmo (presentato in [12]). La funzione hash presentata risulta essere sicura da attacchi anche in considerazione di attacchi futuri, tiene cioè in considerazione gli aggiornamenti hardware che avvengono negli anni.

Concludendo si può dire che Blowfish è un buon cifrario a blocchi. Le principali caratteristiche che lo contraddistinguono da altri cifrari a blocchi, sono sicuramente la totale disponibilità del codice sorgente; la comprensibilità; il particolare metodo di generazione delle sottochiavi a partire dalla chiave primaria; la dimensione variabile della chiave e la sua velocità di cifratura a patto di usare la stessa chiave.

Come già detto, Blowfish vanta una grande quantità di applicazioni reali.

Bibliografia

[1] Bruce Schneier, Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish), Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993), Springer-Verlag, 1994
[2] Bruce Schneier, http://www.schneier.com
[3] B. Schneier, D. Whiting, Fast Software Encryption: Designing Encryption Algorithms for Optimal Software Speed on the Intel Pentium Processor, Fast Software Encryption, Fourth International Workshop Proceedings (January 1997), Springer-Verlag, 1997
[4] Douglas R. Stinson, Cryptography. Theory and Practice. Third Edition, CRC, 2006
[5] Claude E. Shannon, Communication Theory of Secrecy Systems, Bell System Technical Journal vol 28-4, 1949
[6] Wikipedia, http://it.wikipedia.org/wiki/Rete_di_Feistel (dicembre 2008)
[7] A. Menezes, P. van Oorschot and S. Vanstone, Handbook of Applied Cryptography, CRC, 1997
[8] Serge Vaudenay, On the Weak Keys of Blowfish, Laboratorie d'Informatique de l'Ecole Normale Supérieure, November 1995
[9] Bruce Schneier, The Blowfish Encryption Algorithm -- One Year Later, Dr. Dobb's Journal, September 1995
[10] Dieter Schmidt, On the Key Schedule of Blowfish, Technical report, February 2005
[11] Wikipedia, http://it.wikipedia.org/wiki/Blowfish
[12] Niels Provos, David Mazières, A Future-Adaptable Password Scheme, Proceedings of the Annual USENIX Technical Conference, 1999