Tecniche di Progettazione Automatica **Anno Acc. 1995-96**
PROF. ALFREDO DE SANTIS

NUOVI CIFRARI

a cura di Francesco Peluso, Giovanni Perillo, Paola Politi *29 Ottobre 1996*

Indice

1 Introduzione

Gli studiosi di crittografia reputano di dover definire un nuovo standard di codifica in quanto il DES [16], che venne adottato come standard nella sua versione definitiva del 1977 ed attualmente ancora utilizzato, sta ultimando il suo ciclo di vita.
Il DES ha una chiave di 56 bit la quale è risultata vulnerabile all'attacco di forza bruta [22]. Inoltre i recenti attacchi tramite la crittoanalisi differenziale [1] e la crittoanalisi lineare [10] indicano che esso è vulnerabile anche ad altri tipi di attacchi.
Molti degli altri algoritmi non ancora violati, come KHUFU [11, 12], REDOC II [2, 20, 23] e IDEA [7, 8, 9], sono protetti da brevetti [18]. GOST [6], un algoritmo del governo Sovietico, è specificato senza le S-Box. Il governo U.S. si sta muovendo verso algoritmi segreti, così come l'algoritmo Skipjack nei chips Clipper e Clapstone [17].
Nasce quindi la necessità d'avere un algoritmo di codifica sicuro, senza brevetto, di pubblico dominio e liberamente utilizzato da chiunque. Per questi motivi è necessario sviluppare dei nuovi possibili algoritmi di codifica. Questi devono essere sottoposti ad un minuzioso esame di crittoanalisi. La speranza è che uno o più algoritmi candidati superino tale esame e possano eventualmente diventare un nuovo standard.
Questo documento prende in considerazione i requisiti che un algoritmo di codifica standard deve avere. Tenuto conto che può non essere possibile soddisfare tutte le richieste con un unico algoritmo, può essere possibile soddisfarle con una famiglia d'algoritmi basati sugli stessi principi crittografici.

1.1 Area di applicazione

Un algoritmo di codifica standard deve essere efficiente nella codifica in differenti applicazioni:

- *Codifica di massa.* L'algoritmo deve essere efficiente nella codifica di file dati o di flussi continui di dati.
- *Generazione random dei bit.* L'algoritmo dovrebbe essere efficiente nella produzione di singoli bit random.
- *Codifica di pacchetti.* L'algoritmo dovrebbe essere efficiente nella codifica di pacchetti di dati (un pacchetto ATM ha un campo dati di 48 byte). Esso dovrebbe essere implementato in un'applicazione dove pacchetti successivi possono essere codificati o decodificati con chiavi differenti.
- *Hashing.* L'algoritmo dovrebbe essere efficiente nel calcolo di una funzione hash one-way.

1.2 Piattaforme

Un algoritmo di codifica standard deve essere implementabile su differenti piattaforme, ciascuna con i propri requisiti. Questi includono:

- *Special hardware.* L'algoritmo dovrebbe essere efficientemente implementabile in custom VLSI hardware.

- *Large processors.* Per le applicazioni più veloci sarà sempre usato un hardware specifico ma le implementazioni software sono più comuni. L'algoritmo, quindi, dovrebbe essere efficiente su microprocessori a 32 bit con una cache dati-programmi di 4 kbyte.
- *Small processors.* Dovrebbe essere possibile implementare l'algoritmo su smart cards, anche se in maniera inefficiente. Le richieste per gli small processors sono molto difficili da soddisfare in quanto su queste piattaforme le limitazioni della RAM e della ROM risultano essere consistenti. Allora in queste piccole macchine l'efficienza diventa molto importante; se bisogna scegliere, conviene indirizzarsi sui large processors piuttosto che sugli small processors.

1.3 Requisiti addizionali

I requisiti aggiuntivi, che vengono di seguito specificati, dovrebbero essere imposti in un algoritmo di codifica standard.

- L'algoritmo dovrebbe essere semplice da codificare. Infatti l'esperienza fatta col DES [19] ha mostrato che i programmatori spesso faranno errori di implementazione se l'algoritmo risulta complicato. L'algoritmo, quindi, dovrebbe essere robusto contro questi errori.
- L'algoritmo dovrebbe avere uno spazio delle chiavi, permettendo così ad ogni stringa di bit random della lunghezza richiesta, di essere una possibile chiave. Queste stringhe non dovrebbero essere delle chiavi deboli.
- L'algoritmo dovrebbe facilitare la gestione delle chiavi nelle implementazioni software. Di solito le implementazioni software del DES usano tecniche di gestione delle chiavi inefficienti. In particolare, la password che l'utente digita diventa la chiave; questo significa che sebbene il DES ha un ipotetico spazio delle chiavi di 2^{56}, esso è limitato alle chiavi costruite con i 95 caratteri stampabili ASCII. Inoltre, nel DES, sono più probabili chiavi che corrispondono a parole di senso compiuto o quasi.
- L'algoritmo dovrebbe essere facilmente modificabile per adeguarlo a diversi livelli di sicurezza eventualmente richiesti, siano essi di minima o di massima sicurezza.
- Tutte le operazioni dovrebbero manipolare dati in blocchi di byte. Quando è possibile, i dati dovrebbero essere manipolati in blocchi di 32 bit.

2 Blowfish

Bruce Schneier nel 1994 [29] ha proposto un nuovo cifrario a blocchi con chiave segreta, chiamato Blowfish.
Il Blowfish non soddisfa tutti i requisiti discussi prima, per un nuovo standard crittografico, esso è adattabile solo per applicazioni dove la chiave non cambia spesso, come una "comunication link" o una codifica di un "automatic file". Esso è significativamente più veloce del DES quando viene implementato su microprocessori a 32 bit con una grossa cache dati, come il Pentium e il PowerPC.

2.1 Decisioni di progetto

Basandosi sui parametri sopra descritti, in fase di progetto si sono prese le seguenti decisioni riguardanti l'algoritmo. L'algoritmo dovrebbe:

- manipolare dati in grossi blocchi, preferibilmente di taglia di 32 bit (e non in singoli bit come nel DES).
- gestire anche blocchi di taglia di 64 bit e di 128 bit.
- avere una chiave a lunghezza variabile, da un minimo di 32 bit fino ad almeno 256 bit.
- usare operazioni semplici che risultino efficienti su microprocessori come, OR-esclusivo, addizioni, uso di tabelle, moltiplicazioni modulari. Non dovrebbe usare shift di lunghezza variabile, permutazioni bit a bit e jump condizionati.
- essere implementabile su processori a 8 bit con un minimo di 24 byte di RAM in aggiunta alla RAM richiesta per memorizzare la chiave e 1 kilobyte di ROM.
- impiegare sottochiavi precomputabili. In sistemi a grande memoria, queste sottochiavi possono essere precomputate per realizzare, successivamente, operazioni in modo veloce. Non precomputando le sottochiavi si avranno operazioni più lente, ma dovrebbe essere comunque possibile codificare i dati anche senza una precomputazione di esse.
- consistere di un numero variabili di iterazioni. Per applicazioni con una chiave di piccola taglia il trade-off tra la complessità di un attacco di forza bruta e un attacco differenziale produce un grosso numero di iterazioni superflue. Quindi dovrebbe essere possibile ridurre il numero di iterazioni senza perdita di sicurezza.
- non avere, se possibile, chiavi deboli; se questo non è possibile, la proporzione di chiavi deboli dovrebbe essere abbastanza piccola in modo da rendere improbabile la scelta a caso di una di esse. Inoltre, ogni chiave debole dovrebbe essere esplicitamente conosciuta in modo che possa non essere considerata durante il processo di generazione delle chiavi.
- usare sottochiavi che sono una hash one-way della chiave. Questo permetterebbe di usare frasi lunghe per la chiave senza perdita di sicurezza.
- non avere una struttura lineare, come ad esempio la proprietà di complementazione del DES, che riduce la complessità di ricerca esaustiva [4].

- usare un progetto che risulti semplice da capire. Questo renderà facile l'analisi e accrescerà la confidenza con l'algoritmo. In pratica, questo significa che l'algoritmo sarà un cifrario a blocchi Feistel iterato [21].

Molte di queste decisioni di progetto non sono nuove. Tutti i cifrari a blocchi, da Lucifer [5, 21] in poi, sono cifrari Feistel e tutti hanno uno spazio delle chiavi, con la possibile eccezione di alcune chiavi deboli. FEAL [13, 14, 15] e Khufu [11] usano un numero di iterazioni variabile. Khufu [11] ha un grande numero di sottochiavi che sono funzioni one-way della chiave. RC2 [18] ha una lunghezza variabile della chiave. GOST [6] usa una parola di lunghezza 32 bit e un blocco di 62 bit. MMB [2] usa un parola di lunghezza 32 bit e un blocco di 64 bit.

2.2 Building blocks

È stato dimostrato che ci sono diversi "building blocks" che producono dei codici efficienti. Molti di questi possono essere implementati efficientemente su microprocessori a 32 bit.

- *S-Box grandi.* S-Box grandi sono più resistenti alla crittoanalisi differenziale. Un algoritmo con una parola lunga 32 bit può usare S-Box a 32 bit. Khufu e REDOC III usano entrambi 256 ingressi, quindi S-Box di 32 bit [11, 20].

- *S-Box dipendenti dalle chiavi.* Mentre le S-Box fissate debbono essere progettate per resistere alla crittoanalisi differenziale e lineare, le S-Box dipendenti dalla chiave sono molto più resistenti a questi tipi di attacchi. Esse sono usate nell'algoritmo Khufu [11]; mentre S-Box variabili, che potrebbero essere eventualmente dipendenti dalla chiave, sono usate in GOST [6].

- *Combinazione di operazioni di differenti gruppi algebrici.* Il cifrario IDEA ha introdotto questo concetto, combinando XOR mod 2^{16}, addizioni mod 2^{16}, e moltiplicazione mod $2^{16}+1$ [17]. Il cifrario MMB usa una parola a 32 bit, e combina XOR mod 2^{32} con moltiplicazioni mod $2^{32}-1$ [12].

- *Permutazioni dipendenti dalla chiave.* Le permutazioni iniziali e finali del DES sono fissate e sono state giudicate crittograficamente inutili. Khufu esegue uno XOR tra il blocco testo e la chiave all'inizio e alla fine dell'algoritmo [11].

2.3 Descrizione dell'algoritmo

Il Blowfish è un cifrario a blocchi con chiave segreta a lunghezza variabile. Esso è una Feistel network che itera una semplice funzione di codifica 16 volte. La taglia del blocco è di 64 bit e la chiave può avere una lunghezza fino a 448 bit. Sebbene c'è una fase di inizializzazione complessa eseguita prima di ogni codifica, l'attuale codifica di dati è molto efficiente sui large microprocessor.
L'algoritmo consiste di due parti: una parte di espansione della chiave e una parte di cifratura dei dati.
L'espansione converte una chiave di al più 448 bit in alcuni array di sottochiavi per un totale di 4168 byte.
La cifratura dei dati si ottiene attraverso una Feistel network di 16 round. Ogni round consiste di una permutazione dipendente dalla chiave e da una sostituzione dipendente dalla chiave e

dai dati.
Tutte le operazioni sono XOR e addizioni di parole di 32 bit. Le sole operazioni addizionali sono quattro letture per round in array di dati indicizzati.

2.3.1 Sottochiavi

Blowfish usa un grande numero di sottochiavi. Queste chiavi devono essere precomputate prima di ogni cifratura o decifratura dei dati.

1. Il P-array consiste di 18 sottochiavi di 32 bit:
$$P_1, P_2, \ldots, P_{18}.$$
2. Ci sono quattro S-Box di 32 bit di 256 ingressi ciascuna:
$$\begin{gathered} S_{1,0},\ S_{1,1}\ \ldots, S_{1,255}; \\ S_{2,0},\ S_{2,1}\ \ldots, S_{2,255}; \\ S_{3,0},\ S_{3,1}\ \ldots, S_{3,255}; \\ S_{4,0},\ S_{4,1}\ \ldots, S_{4,255}. \end{gathered}$$

Il P-array necessita di 18×4 byte e le 4 S-Box necessitano di $(256 \times 4) \times 4$ byte, per un totale di 4168 byte.
Il metodo esatto usato per calcolare queste sottochiavi sarà descritto nel seguito.

2.3.2 Codifica

Il Blowfish è una Feistel network composta da 16 round, essa è mostrata in Figura 1, dove con il simbolo $\oplus$ è denotato lo XOR bit a bit.
L'input è un elemento di 64 bit denotato con x.

1. *Dividi* x *in due blocchi da* 32 *bit ciascuno* : $x_L,\ x_R$
2. **for** $i = 1$ **to** 15
3. $\quad x_L = x_L \oplus P_i$
4. $\quad x_R = F(x_L) \oplus x_R$
5. $\quad$ **Swap** x_L *e* x_R
6. $x_L = x_L \oplus P_{16}$
7. $x_R = F(x_L) \oplus x_R$
8. $x_R = x_R \oplus P_{17}$
9. $x_L = x_L \oplus P_{18}$
10. $Ciphertext = x_L x_R$

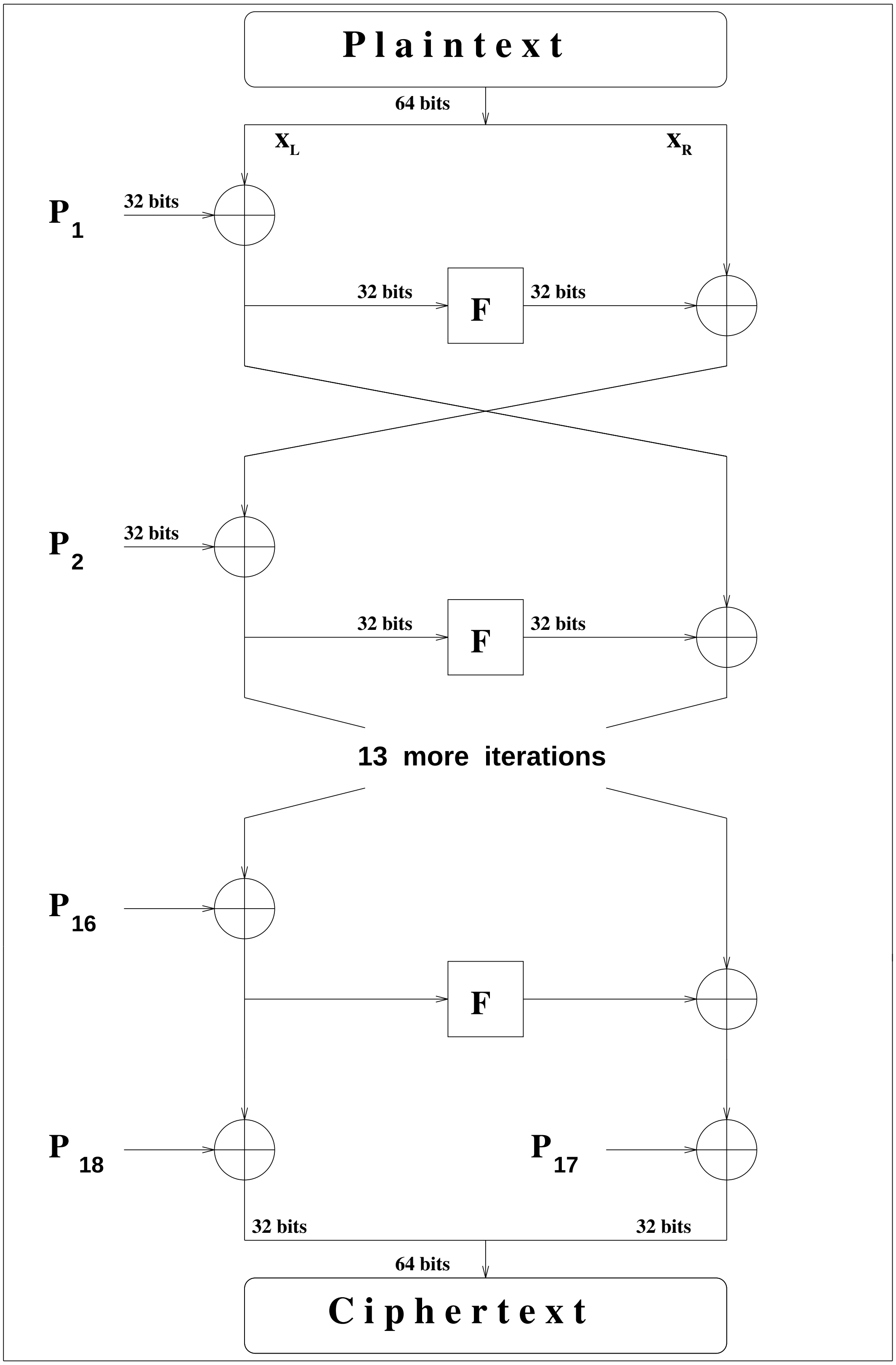

Figura 1: Blocco del diagramma di Blowfish

La funzione F:

1. *Dividi* x_L *in quattro blocchi di 8 bit* : a, b, c, d
2. $F(x_L) = ((S_{1,a} + S_{2,b} \bmod 2^{32}) \oplus S_{3,c}) + S_{4,d} \bmod 2^{32}$

In Figura 2 è mostrata la funzione F, dove con il simbolo $\boxplus$ è denotata l'addizione mod 2^{32}. La decodifica funziona allo stesso modo della codifica, eccetto l'uso in ordine inverso delle sottochiavi $P_1, P_2, \ldots, P_{18}$. L'implementazione di Blowfish che richiede alte velocità dovebbre realizzare il ciclo e assicurare che le sottochiavi siano immagazzinate nella cache.

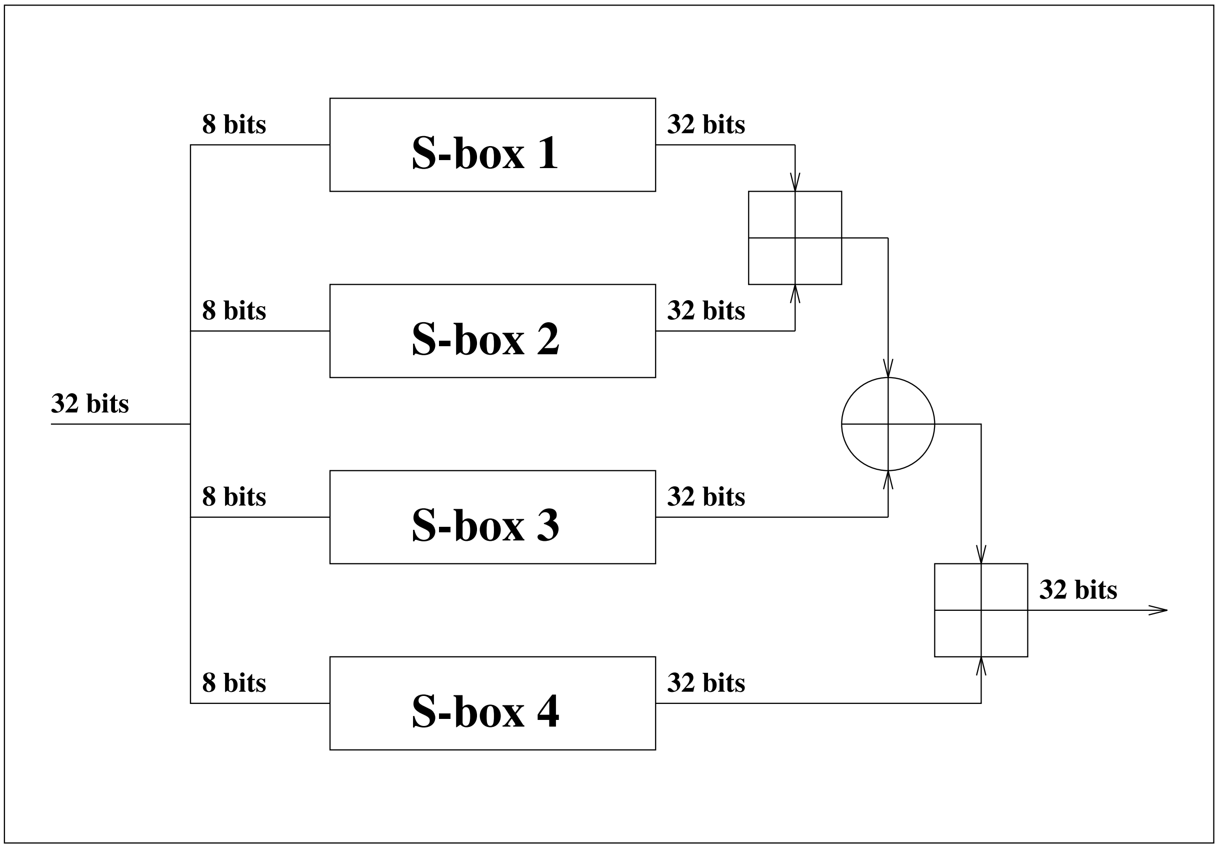

Figura 2: Funzione F

2.3.3 Generazione di sottochiavi e S-Box

Le sottochiavi sono calcolate usando l'algoritmo Blowfish. Il metodo è il seguente:

1. Inizializzare anzitutto nell'ordine il P-array e a seguire le quattro S-Box con una stringa fissata. Questa stringa consiste dei primi digit esadecimali di π, meno l'iniziale 3. Per esempio:

$$\begin{aligned} P_1 &= 0x243f6a88 \\ P_2 &= 0x85a308d3 \\ P_3 &= 0x13198a2e \\ P_4 &= 0x03707344 \end{aligned}$$

2. Fare lo XOR di P_1 con i primi 32 bit della chiave, poi lo XOR di P_2 con i secondi 32 bit della chiave, e così via per tutti i bit della chiave (eventualmente fino a P_{14}). Ripetere il ciclo attraverso i bit della chiave sino a che non sia stato eseguito uno XOR dell'intero P-array con i bit della chiave. (Per ogni chiave corta, c'è almeno una chiave equivalente più lunga; per es. se A è una chiave di 64 bit, allora AA, AAA, etc sono chiavi equivalenti più lunghe.).

3. Codificare una stringa tutti zero con l'algoritmo Blowfish, usando le sottochiavi P_i descritte nei passi 1 e 2.

4. Sostituire P_1 e P_2 con l'output del passo 3.

5. Codificare l'output del passo 3 usando l'algoritmo Blowfish avente le prime due sottochiavi m odificate.

6. Sostituire P_3 e P_4 con l'output del passo 5.

7. Continuare il processo sostituendo con l'output continuamente codificato dall'algoritmo Blowfish tutte le entrate del P-array ed a seguire tutte le entrate delle quattro S-Box in ordine.

Sono richieste un totale di 521 iterazioni per generare tutte le sottochiavi necessarie. Si osserva che nel processo di generazione delle sottochiavi viene generata una coppia di sottochiavi alla volta, quindi 18/2 iterazioni sono necessarie per generare il P-array e 256/2 iterazioni sono necessarie per generare ciascuna delle quattro S-Box.
È possibile immagazzinare le sottochiavi piuttosto che eseguire questo processo di derivazione quando richiesto, in quanto ciò causa un aumento del tempo di computazione della codifica.

2.4 Mini Blowfish

Le seguenti versioni mini di Blowfish sono definite soltanto per la crittoanalisi; esse non sono suggerite per reali implementazioni. Blowfish-32 ha il blocco di 32 bit e le entrate dell'array delle sottochiavi di 16 bit (ciascuna S-Box avrà $(\frac{16}{4})^2 = 16$ ingressi). Blowfish-16 ha il blocco di 16 bit e le entrate dell'array delle sottochiavi di 8 bit (ciascuna S-Box avrà $(\frac{8}{4})^2 = 4$ ingressi).

2.5 Analisi delle scelte

- Alla base della filosofia del Blowfish vi è la semplicità del progetto che rende l'algoritmo semplice da comprendere e semplice da implementare. Si spera che il progetto organizzato mediante l'uso di una Feistel network (una semplice S-box substitution e una semplice P-box substitution) non conterrà alcun errore.

- Il blocco di 64 bit permette una parola di 32 bit che in questo modo è anche compatibile con la taglia dei blocchi di algoritmi esistenti.
  Blowfish permette di aumentare in maniera agevole la taglia del blocco a 128 bit o di diminuirla a più piccole dimensioni. Le diverse versioni del mini-Blowfish possono essere significativamente più semplici da crittoanalizzare della versione completa.

- Le operazioni fondamentali sono state scelte avendo come obiettivo principale la velocità. XOR, ADD, MOV sono efficienti per una cache su architetture Intel e Motorola. Tutte le sottochiavi si adattano nella cache di un 80486, 68040, Pentium e PowerPc.

- La Feistel network che costituisce il corpo del Blowfish è progettata per essere molto semplice pur conservando le desiderabili proprietà crittografiche della struttura.
  In Figura 3 è descritto il Round i, dove $R_{n,i}$ sono funzioni reversibili del testo e della chiave, N_i è una funzione non reversibile del testo e della chiave.
  Per velocità e semplicità di calcolo è stato scelto lo XOR come funzione reversibile.

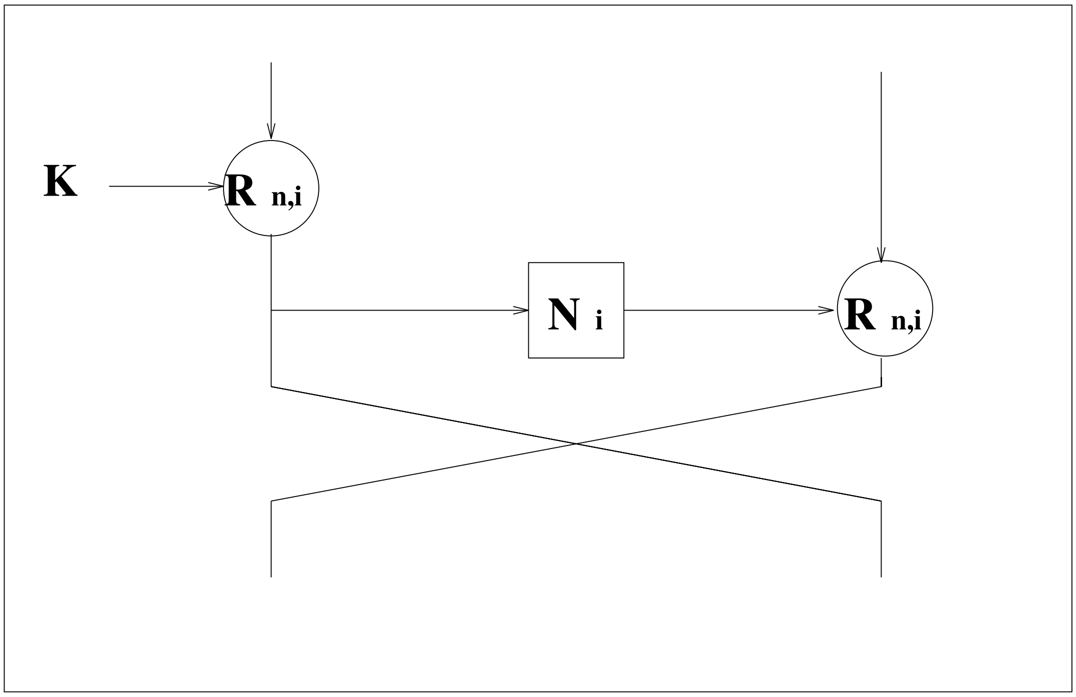

Figura 3: Round i di Blowfish

- È stata considerata la possibilità di usare una funzione reversibile più complicata, con moltiplicazioni modulari e rotazioni. Ma queste operazioni accrescerebbero grandemente il tempo di esecuzione dell'algoritmo e dal momento che la primaria risorsa della sicurezza dell'algoritmo è la funzione F si è deciso di preservare da complicazioni il tempo di esecuzione di tale funzione.

- La funzione non reversibile F dà al Blowfish il migliore effetto a valanga possibile per una Feistel network: ogni bit di testo nella metà sinistra del round ha influenza su ogni bit di testo nella metà destra. Inoltre, poichè ogni bit della sottochiave è influenzato da ogni bit della chiave, la funzione ha anche un perfetto effetto a valanga, dopo ogni round, tra la chiave e la metà destra del testo.

- È stata considerata la possibilità di aggiungere una funzione reversibile mixata, più complicata dello XOR, prima del primo e dopo l'ultimo round. Questo potrebbe confondere ulteriormente i valori di entrata nella Feistel network. Si è scartata questa eventuale aggiunta come una complicazione del tempo di computazione con un non chiaro beneficio crittografico.

2.5.1 S-BOX

- La funzione non reversibile è progettata per essere efficace, veloce e semplice. Idealmente si ricerca una singola S-box con 2^{32} parole di 32 bit, ma questo sarebbe difficile da realizzare. L'eventuale scelta di 256 entrate alle S-Box era un compromesso tra questi tre obiettivi. Il piccolo numero di bit utilizzato per l'entrate delle S-box può avere delle debolezze rispetto alla crittoanalisi lineare, ma queste debolezze sono nascoste sia combinando l'output delle 4 S-Box sia rendendole dipendenti dalla chiave.

- Sono state usate infatti 4 differenti S-Box invece di una, principalmente per evitare simmetrie quando differenti byte dell'input alla funzione F sono una permutazione bit a bit di un'altro input di 32 bit.
  Invece di avere 4 S-Box differenti si sarebbe potuto usare una stessa S-Box ed eseguire una permutazione non banale di ciascuno dei singoli 4 differenti output; il progetto delle 4 S-Box è più veloce e più semplice da programmare e sembra essere più sicuro.

- La funzione che combina gli output delle 4 S-Box è molto veloce.
  Una funzione più semplice sarebbe potuta essere lo XOR dei 4 valori, ma unendo addizione mod 2^{32} e XOR si combinano due differenti gruppi algebrici senza istruzioni aggiuntive. L'alternanza di addizione e XOR termina con un'operazione di addizione perchè uno XOR combina il risultato finale con x_R.

- Qualora fosse stata scelta una stessa S-Box ripetuta, come esposto sopra, sarebbe necessaria una funzione combinante gli output delle S-Box per eliminare le simmetrie. Si è considerato l'uso in Blowfish di una più complessa funzione combinante i 4 output delle S-Box (usando moltiplicazioni modulari, rotazioni, etc) ma non la si è scelta perchè la complicazione aggiunta non sembrava necessaria.

- Le S-Box dipendenti dalla chiave sono protette contro la crittoanalisi differenziale e lineare. Quindi, poichè la struttura delle S-Box è completamente nascosta ai crittoanalisti, questi tipi di attacchi necessitano di più tempo.
  Sarebbe anche possibile rimpiazzare queste S-Box variabili con 4 S-Box fissate progettate per essere resistenti a questi tipi di attacchi; ma le S-Box dipendenti dalla chiave sono più facili da implementare e meno suscettibili ad argomentazioni di proprietà "nascoste". Inoltre queste S-Box possono essere create su richiesta riducendo così la necessità di dover memorizzare grosse strutture dati.

- Ogni bit di x_L é usato come input ad una sola S-Box.
  Nel DES molti bit sono usati come input a due S-box, questo rafforza considerevolmente l'algoritmo contro attacchi differenziali. Ma si ritiene che questa complicazione addizionale non sia necessaria utilizzando S-Box dipendenti dalla chiave. Inoltre S-Box più grandi potrebbero togliere un più considerevole spazio di memoria.

2.5.2 Funzione F

- La funzione F non dipende dall'iterazione. Si è presa in considerazione la possibilità di aggiungere questa dipendenza, ma si è ritenuto che non abbia alcun merito crittografico. Il P-array substitution può essere considerato parte di questa funzione ed esso è già dipendente dall'iterazione.

- L'algoritmo presenta 16 round, questo numero determina la taglia del P-array e quindi il processo di generazione delle sottochiavi. 16 iterazioni permettono una chiave lunga fino a 448 bit. I progettisti del Blowfish a seguito dell'accumulo di ulteriori dati crittoanalitici attendono di avere la capacitá di ridurre questo numero in modo da accellerare il processo dell'algoritmo.

2.5.3 Processo di generazione di sottochiavi

- Nella progetto dell'algoritmo ci sono due modi fondamentali per assicurare che la chiave sia lunga abbastanza da garantire un particolare livello di sicurezza.

  1. Una è di progettare accuratamente l'algoritmo affinchè sia presentata la completa entropia della chiave e non ci sia una migliore strada per crittoanalizzare l'algoritmo se non con un attacco di forza bruta.
  2. L'altra è di progettare un algoritmo con una chiave di così tanti bit che gli attacchi che riducono l'effettiva lunghezza della chiave di diversi bit sia irrilevante.

  Si è scelto quest'ultimo modo perchè Blowfish è destinato a grossi microprocessori aventi grande quantità di memoria.

- Il processo di generazione di sottochiavi è destinato a preservare l'intera entropia della chiave ed a distribuirla uniformemente attraverso le sottochiavi. Esso è anche progettato in modo che durante la costruzione definitiva delle sottochiavi, nella fase di inizializzazione, l'insieme delle sottochiavi sia distribuito in modo random.
  Si sono scelti i numeri decimali di π per l'iniziale tavola delle sottochiavi per due ragioni: perchè essi sono una sequenza di bit random non messa in relazione con l'algoritmo e perchè essi possono anche essere memorizzati come parte dell'algoritmo o derivati (ossia calcolati) quando necessario.
  La scelta di π non è vincolata, qualunque stringa di bit random sarà sufficiente (come il numero decimale e, RAND tables, output di un generatore di numeri random). Ma se la stringa iniziale non è random (per esempio, testi ASCII, con molti bit di ogni byte a zero) questa non casualità si propagherà durante l'algoritmo.

- Il processo di generazione delle sottochiavi provoca un cambiamento graduale del P-array in quanto viene generata una coppia di sottochiavi per volta. Ciò è fatto per proteggere da alcuni attacchi tale processo che utilizza sottochiavi fissate e conosciute. Il limite di 448 bit sulla grandezza della chiave assicura che ogni bit di ciascuna sottochiave dipenda in ogni bit dalla chiave.

- Viene eseguito uno XOR tra i bit della chiave ripetuti ed i numeri decimali di π nel P-array iniziale per prevenire il seguente potenziale attacco: si assume che la chiave si estende a tutto il P-array non ripetendo i bit della stessa ma aggiungendo tutti i bit zero.
  Consideriamo chiavi di 64 bit. Un'attaccante allora potrebbe trovare due chiavi che differiscono solo nei valori dei 64 bit su cui è stato eseguito lo XOR con P_1 e P_2 le quali producono, usando le sottochiavi iniziali conosciute, lo stesso valore cifrato.
  Se così fosse, l'attaccante potrebbe trovare due chiavi che producono le stesse sottochiavi. Questo è un attacco molto allettante per un inefficiente generatore di chiavi.

- Per prevenire questo tipo di attacco, si è stabilito l'iniziale valore del testo in chiaro nel processo di generazione delle sottochiavi. Non vi è nulla di speciale riguardo la stringa di tutti zero ma è importante che tale valore sia fissato.

- L'algoritmo di generazione di sottochiavi non assume che i bit della chiave siano random. Anche chiavi i cui bit sono altamente correlati, come una stringa di caratteri alfanumerici ASCII con i bit di ogni byte settati a zero, produrrà sottochiavi random. Per produrre sottochiavi con la stessa entropia, è richiesta una chiave alfanumerica più lunga.

- Il tempo utilizzato nel processo di generazione di sottochiavi aggiunge una considerevole complessità ad un attacco a forza bruta. Le sottochiavi generate da una macchina cracking a forza bruta sono troppe lunghe per essere immagazzinate, come richiesto, su un nastro a massa. Sono necessarie un totale di 512 iterazioni dell'algoritmo di cifratura per testare una singola chiave perchè alle 9 iterazioni per inizializzare il P-array si aggiungono le 2^9 iterazioni per inizializzare le 4 S-Box e a queste si deve aggiungere l'iterazione necessaria per la codifica stessa. (Si ricorda che nel processo di generazione delle sottochiavi viene generata una coppia di sottochiavi alla volta. Il P-array è costituito da $P_1, \ldots, P_{18}$ quindi sono necessarie 18/2 iterazioni dell'algoritmo, poi si devono calcolare le 2^8 entrate di ciascuna delle 4 S-Box quindi sono necessarie $2^8 \times 4/2$ iterazioni, a queste si deve sommare l'iterazione necessaria per la codifica. Si richiedono:

$$(18/2) + (2^8 \times 4)/2 + 1 = 522 \text{ iterazioni.}$$

2.6 Possibili semplificazioni

Si sono analizzate diverse possibili semplificazioni mirate a diminuire le richieste di memoria ed il tempo di esecuzione. Esse sono accennate di seguito:

- *Meno S-Box e più piccole.* Può essere ridotto il numero delle S-Box da 4 ad 1. Inoltre può essere possibile sovrapporre le entrate in una singola S-Box.
  Consideriamo una singola S-Box costituita da 259 byte (vedi Figura 4).

  L'output relativo all'entrata 0 della S-Box consiste dei byte S[0], S[1], S[2], S[3],
  l'output relativo all'entrata 1 della S-Box consiste dei byte S[1], S[2], S[3], S[4],

  $$\vdots$$

  $$\vdots$$

  l'output relativo all'entrata 255 della S-Box consiste dei byte S[255], S[256], S[257], S[258].

La su esposta semplificazione ridurrebbe la memoria richiesta dalle 4 S-Box da 4096 byte a 1024 byte e successivamente ridurrebbe ulteriormente la richiesta di memoria per una singola S-Box da 1024 byte a 259 byte (cioè 256+3 byte).

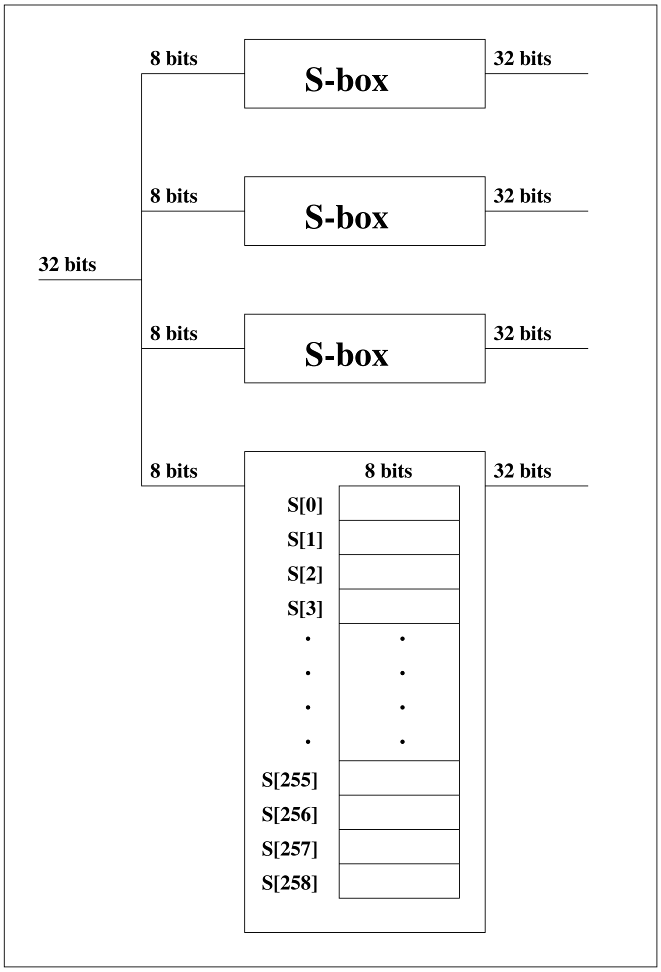

Figura 4: Esempio di semplificazione

- *Meno iterazioni* Sicuramente si può ridurre il numero di iterazioni da 16 ad 8 senza compromettere la sicurezza dell'algoritmo. Il numero delle iterazioni richieste per la sicurezza può dipendere dalla lunghezza della chiave. È da notare che, con la corrente procedura di generazione delle sottochiavi, un algoritmo ad 8 iterazioni non può accettare che una chiave lunga al più 192 bit.
  ($32 \cdot 14$=448) per 16 round, mentre ($32 \cdot 6$=192) per 8 round.

- *Calcolo diretto delle sottochiavi* Il corrente metodo di calcolo delle sottochiavi richiede che tutte le sottochiavi siano calcolate prima di ogni cifratura dei dati. Infatti, è impossibile calcolare l'ultima sottochiave dell'ultima S-Box senza calcolare ogni sottochiave che la precede. Potrebbe essere preferibile un metodo alternativo per il calcolo delle sottochiavi: quello dove ogni sottochiave può essere calcolata indipendentemente da ogni altra.

2.7 Conclusione

Si suppone che il modo più efficiente per rompere il Blowfish sia attraverso una ricerca esaustiva dello spazio delle chiavi. L'algoritmo è ancora in fase evolutiva quindi può essere soggetto a modifiche. Attraverso i risultati degli attacchi crittoanalitici, anche alle versioni di mini Blowfish, si spera di migliorare la qualità di quest'algoritmo di codifica. Blowfish non è brevettato e rimarrà così in tutte le nazioni. L'algoritmo è di pubblico dominio e può essere usato liberamente da chiunque.

3 SAFER (Secure And Fast Encryption Routine)

3.1 Cenni storici

Il SAFER è un algoritmo di codifica a blocchi orientato a byte con chiave segreta che fu presentato [24], per la prima volta, con il nome di SAFER K-64 durante i lavori del workshop tenutosi alla Cambridge University nel dicembre del 1993. Fu commissionato dalla Cylink Corporation (Sunnyvale, CA, USA) come un cifrario senza brevetto. La lunghezza della sua chiave era di 64 bit ed il numero di iterazioni andavano da un minimo consigliato di 6 ad un massimo di 10.
L'anno seguente, lo stesso autore J.L.Massey [25] presentò una nuova versione del SAFER con una chiave di 128 bit, chiamato SAFER K-128. Quando le due metà della chiave scelta dall'utente erano identiche, le sottochiavi prodotte da questo processo coincidevano con quelle prodotte dalla schedulazione delle sottochiavi per SAFER K-64.
Nel settembre del '95 Massey adottò un processo di schedulazione delle chiavi rafforzato per ovviare alle debolezze evidenziate dagli studi di Knudsen [26, 27] e Murphy. Il primo aveva escogitato un attacco che produceva collisioni per il SAFER a sei round, dopo aver scelto solo circa 2^{24} coppie plaintext-chiave, circa 256 volte più veloce di quando si tenta a caso. Lo stesso Knudsen propose le contromisure a questo e ad altri possibili attacchi rinunciando a tutti i diritti di proprietá, questo portò Massey ad adottare una nuova schedulazione delle chiavi ed a cambiare il numero di round consigliati, che per SAFER SK-64 sono 8, e vanno da un minimo di 6 round ad un massimo di 10 round, mentre per SAFER SK-128 non è cambiato nulla. Con SK vengono indicate le nuove versioni del SAFER.

3.2 Introduzione

Il SAFER è un cifrario non brevettato, quindi utilizzabile liberamente da chiunque. È un algoritmo di codifica a blocchi con chiave segreta orientato a byte, in quanto nel processo di codifica e di decodifica sono usate soltanto operazioni sui byte.
La lunghezza del blocco del plaintext, del ciphertext e della chiave selezionata dall'utente è di 8 byte (64 bit).
Il SAFER è un cifrario iterativo nel senso che la codifica è realizzata applicando la stessa trasformazione per r round, dopo di che effettua una trasformazione output. Il numero di round consigliato è 8, anche se si può scegliere un minimo di 6, ma se si desidera una maggiore sicurezza si può usare un r più grande (non più di 10).
Ciascun round usa due sottochiavi di 8 byte (64 bit) ottenute da un processo di schedulazione della chiave segreta selezionata dall'utente (anch'essa di 8 byte). La trasformazione output usa un'altra sottochiave di 8 byte, determinata sempre dallo stesso processo di schedulazione.
Una caratteristica inusuale del SAFER è che, in contrasto con molti cifrari iterati a blocchi proposti recentemente, la fase di codifica e di decodifica sono leggermente differenti.
Nella schedulazione delle chiavi vengono effettuati degli shift circolari al livello di bit, tale schedulazione avviene solamente una volta fino a che non si cambia la chiave.
Per avere la sicurezza con un processo così semplice, SAFER utilizza due nuovi concetti crittografici: la **PHT** e le **Key Biases**.
La prima è una *trasformazione lineare non ortodossa* chiamata **Pseudo Hadamard Trasformation (PHT)**, che permette al cifrario di ottenere rapidamente la diffusione desiderata di

piccoli cambiamenti nel plaintext o nella chiave, che arriva fino al ciphertext. Il problema della diffusione di un cifrario a blocchi è molto sentito, infatti si cerca di ottenerla selezionando attentamente il tipo di permutazione da adottare basandosi su test statistici solidi. La PHT assicura la diffusione necessaria, infatti sembra essere migliore di quelle che forniscono altri cifrari.
La seconda prevede l'uso di **Key Biases** aggiuntive, in modo tale da eliminare il problema delle weak key che costituisce un limite per molti cifrari a blocchi.
Il SAFER comprende una procedura ricorsiva semplice da implementare che genera casualmente queste *key biases* ed usa nei processi di codifica e di decodifica solamente operazioni sui byte che sono particolarmente utili in applicazioni come smart card dove la capacità del processore è molto limitata.

3.3 SAFER SK-64

3.3.1 Codifica

La struttura di codifica del cifrario SAFER SK-64 è mostrato in Figura 5.
L'algoritmo di codifica consiste in r round di trasformazioni identiche che sono applicate in sequenza al plaintext, seguite da una trasformazione output che produce il ciphertext finale. Ciascun round è controllato da due sottochiavi di 8 byte e la trasformazione output è controllata da una sottochiave di 8 byte. Queste $2r+1$ sottochiavi sono tutte derivate dalla chiave selezionata dall'utente di 8 byte in un modo che sarà spiegato dopo. Il plaintext, il ciphertext e tutte le sottochiavi sono lunghe 8 byte (64 bit).
I dettagli della struttura dei round di codifica del SAFER è mostrata in Figura 6. I byte sono ordinati da sinistra, cioè , il byte 1 è il byte più significativo.
Il primo passo dentro l'i-esimo round è la *Mixed XOR/Byte-Addition Operation* dell'input al round con la sottochiave K_{2i-1}. Tale operazione consiste in uno XOR bit a bit dei byte 1, 4, 5, 8 insieme con l'addizione byte a byte (addizione mod 256) dei byte 2, 3, 6, 7 dell'input al round i con la corrispondente sottochiave K_{2i-1}. Gli 8 byte del risultato sono allora passati attraverso un *nonlinear layer* dove sono sottoposti ad una delle due differenti trasformazioni “highly nonlinear”:

1. l'operazione labellata con $45^{(\ )}$ in Figura 6 indica che se il byte in input è l'intero j allora il byte in output è $45^j \bmod 257$ (si assume che l'output è 0 quando il risultato modulare è 256 e questo si ha per $j = 128$). Il motivo per cui è stata scelta questo tipo di trasformazione è che 257 è primo, l'aritmetica modulo 257 è l'aritmetica del campo finito GF(257) e l'elemento 45 è un elemento primitivo di questo campo.

2. l'operazione labellata con $\log_{45}$ in Figura 6 indica che se il byte in input è l'intero j allora il byte output è $\log_{45}(j)$ (si assume che l'output è 128 se l'input è $j = 0$). Le caratteristiche di non linearità di questo mapping sono simili a quelle descritte per l'esponentizzazione.

L'output di queste otto trasformazioni non lineari è combinato con la sottochiave K_{2i} in una operazione che consiste dell'addizione byte a byte (addizione mod 256) dei byte 1, 4, 5, 8 e dello XOR bit a bit dei byte 2, 3, 6, 7 dell'output delle trasformazioni non lineari con la sottochiave K_{2i}. Questa operazione è detta *Mixed Byte-Addition/XOR operation*.
L'output della *Mixed Byte-Addition/XOR operation* passa attraverso una *three level linear layer of boxes* che sono labellate con **2-PHT** in Figura 6.

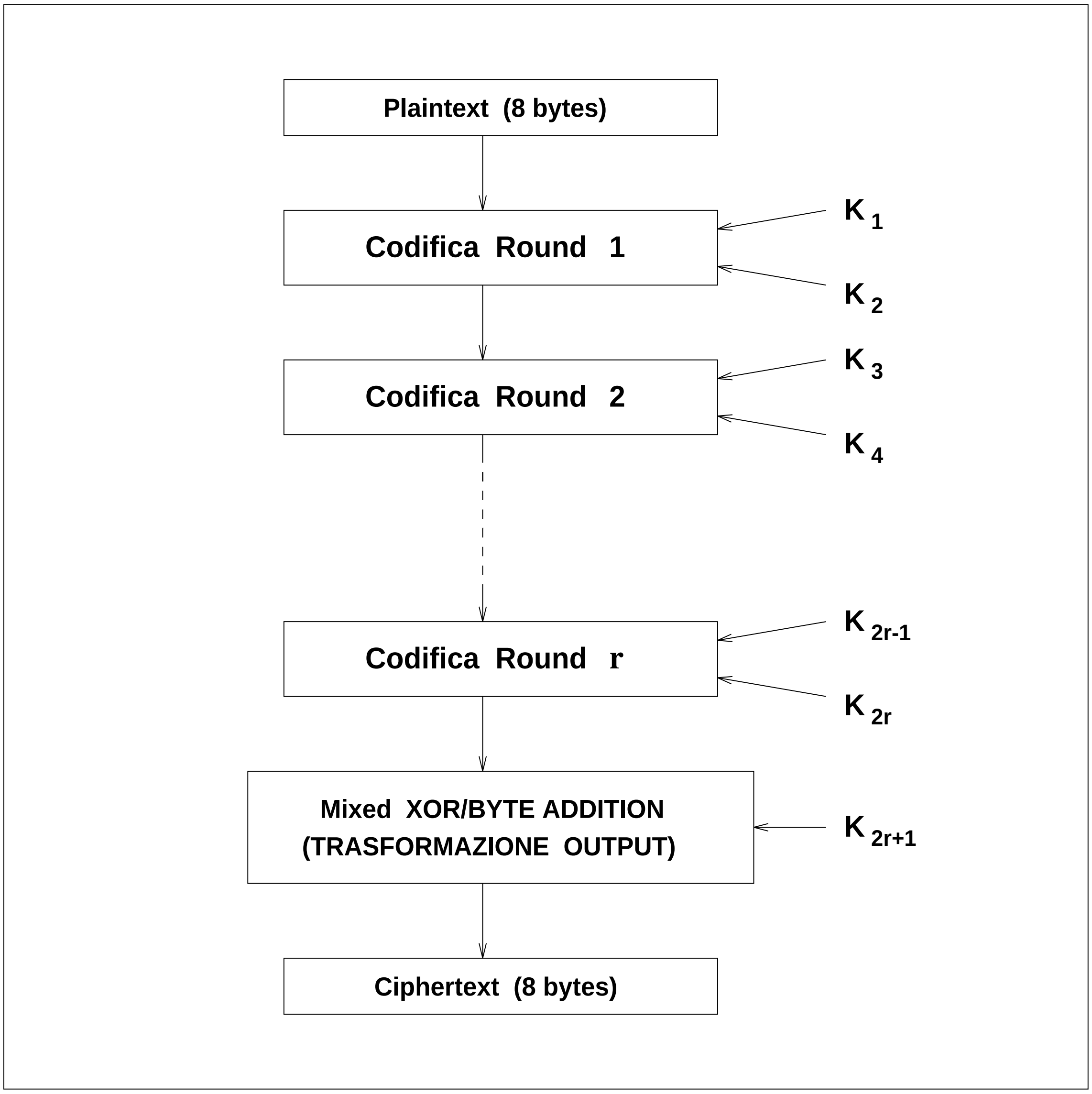

Figura 5: Codifica del SAFER SK-64

Se i due byte in input a un 2-PHT sono (a_1,a_2) dove a_1 è il byte più significativo, allora i due byte di output sono (b_1,b_2) dove

$$\begin{cases} b_1 = 2a_1 + a_2 \\ b_2 = a_1 + a_2 \end{cases} \tag{1}$$

e dove l'aritmetica è modulo 256.
Tra i livelli del *three level linear layer* è applicata la *decimation-by-2 permutation*, la funzione che crea la diffusione. L'output di questo *three level linear layer* costituisce l'output del round. In ultimo viene realizzata una *trasformazione output* mediante una *Mixed XOR/Byte-Addition operation* tra l'ultima sottochiave K_{2r+1} con i corrispondenti byte dell'output degli r round. Esempi di codifica sono presentati nell'Appendice D.

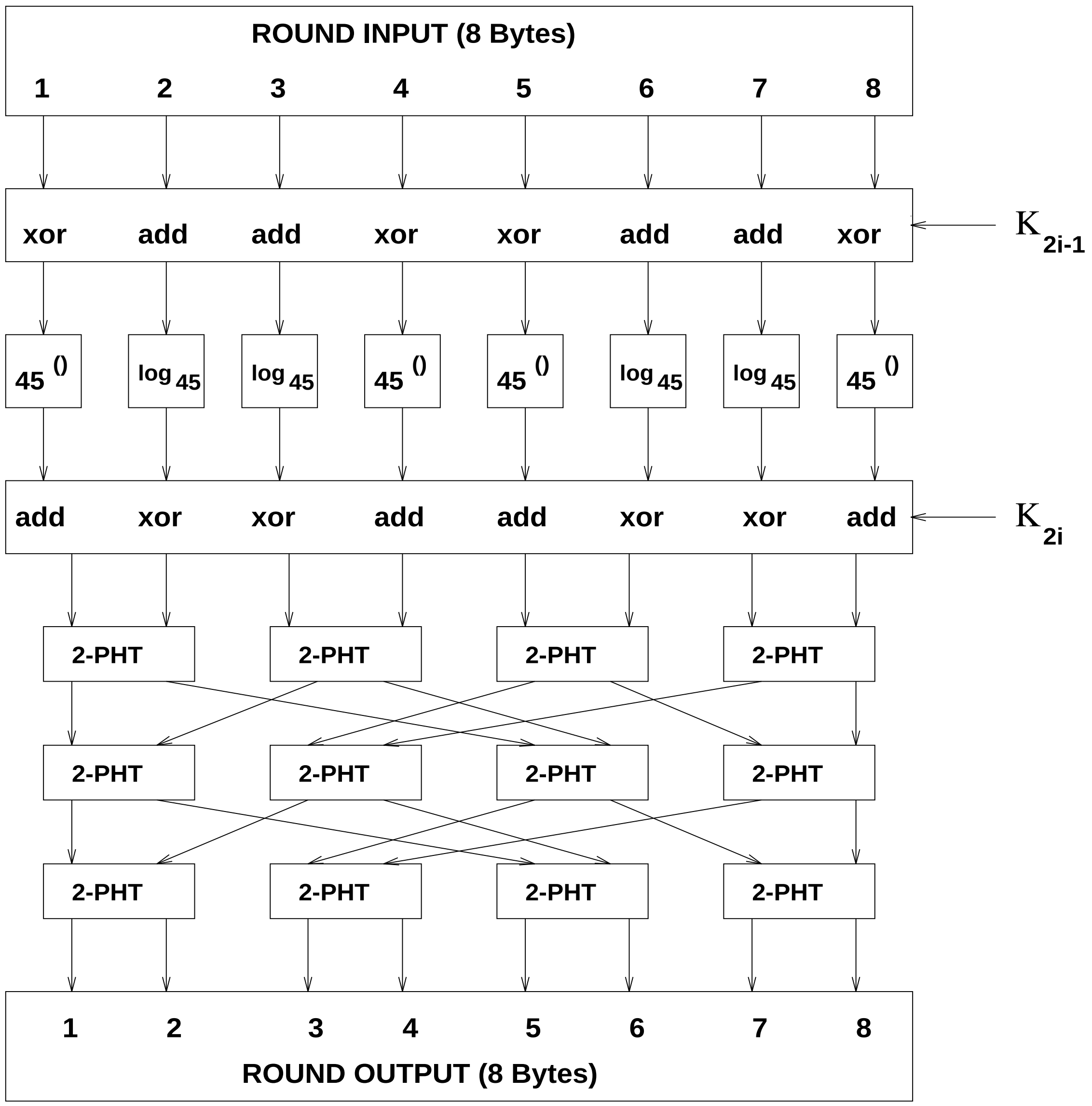

Figura 6: Round i della codifica del SAFER SK-64

3.3.2 Decodifica

Una particolare caratteristica del SAFER è che i round di decodifica differiscono dai round di codifica così che un codificatore non può essere convertito a decodificatore semplicemente invertendo le chiavi schedulate.
La struttura della decodifica è mostrata in Figura 7.
L'algoritmo di decodifica consiste in una *trasformazione input* che viene applicata al blocco ciphertext ed è seguita da r round di trasformazioni identiche.
La trasformazione input consiste in una *Mixed XOR/Byte-Subtraction* della sottochiave K_{2r+1} con il blocco ciphertext.
Tale operazione consiste nello XOR bit a bit dei byte 1, 4, 5, 8 insieme alla sottrazione byte a byte (sottrazione mod 256) dei byte 2, 3, 6, 7 del ciphertext e della sottochiave K_{2r+1}.
La struttura dettagliata dei round di decodifica è mostrata in Figura 8.
L'i-esimo round di decodifica comincia passando l'input del round attraverso il *three level inverse linear layer.*

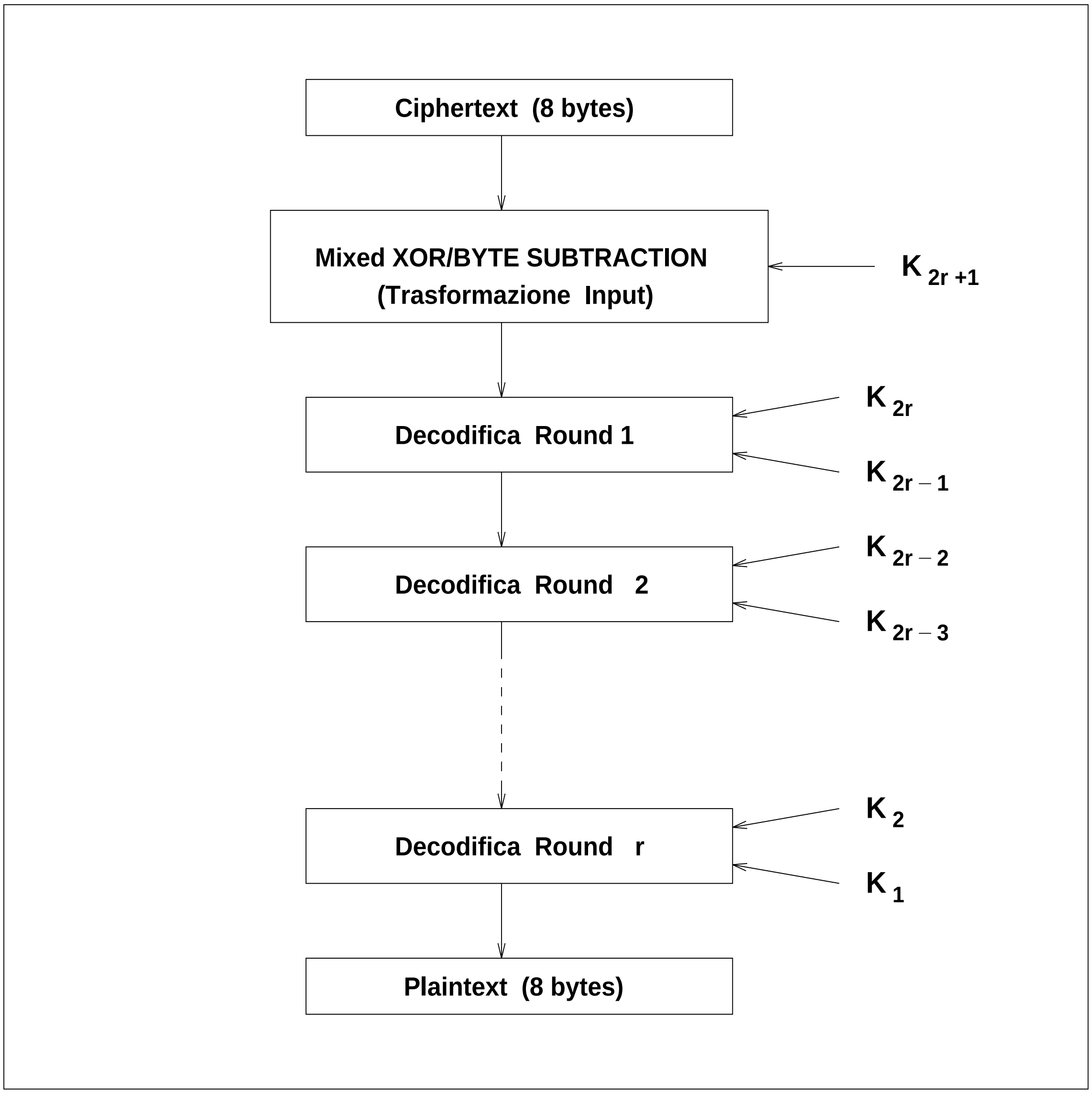

Figura 7: Decodifica del SAFER SK-64

È semplice controllare dalla (1) che l'inversa di PHT (IPHT) è data da

$$\begin{cases} b_1 &= \quad a_1 - a_2 \\ b_2 &= -b_1 + a_2 \end{cases} \tag{2}$$

IL *fan-out* delle due permutazioni tra i livelli del *three level inverse linear layer* è l'inversa della *decimate-by-2 permutation* usato nei round di codifica.
Il passo successivo è la *Mixed Byte-Subtraction/XOR* dell'output del *three level inverse linear layer* con la sottochiave $K_{2r+2-2i}$. Tale operazione consiste nella sottrazione byte a byte (sottrazione mod 256) dei byte 1, 4, 5, 8 insieme con lo XOR bit a bit dei byte 2, 3, 6, 7 dell'output del *three level inverse linear layer* con la sottochiave $K_{2r+2-2i}$.
Nel passo successivo del round di decodifica gli 8 byte sono passati attraverso un *inverse nonlinear layer*, che differisce dal *nonlinear layer* del round di codifica per lo scambio delle locazioni delle due differenti trasformazioni "highly nonlinear". Nell'ultimo passo dell'i-esimo round di decodifica l'output di queste 8 trasformazioni non lineari è combinato con la sottochiave

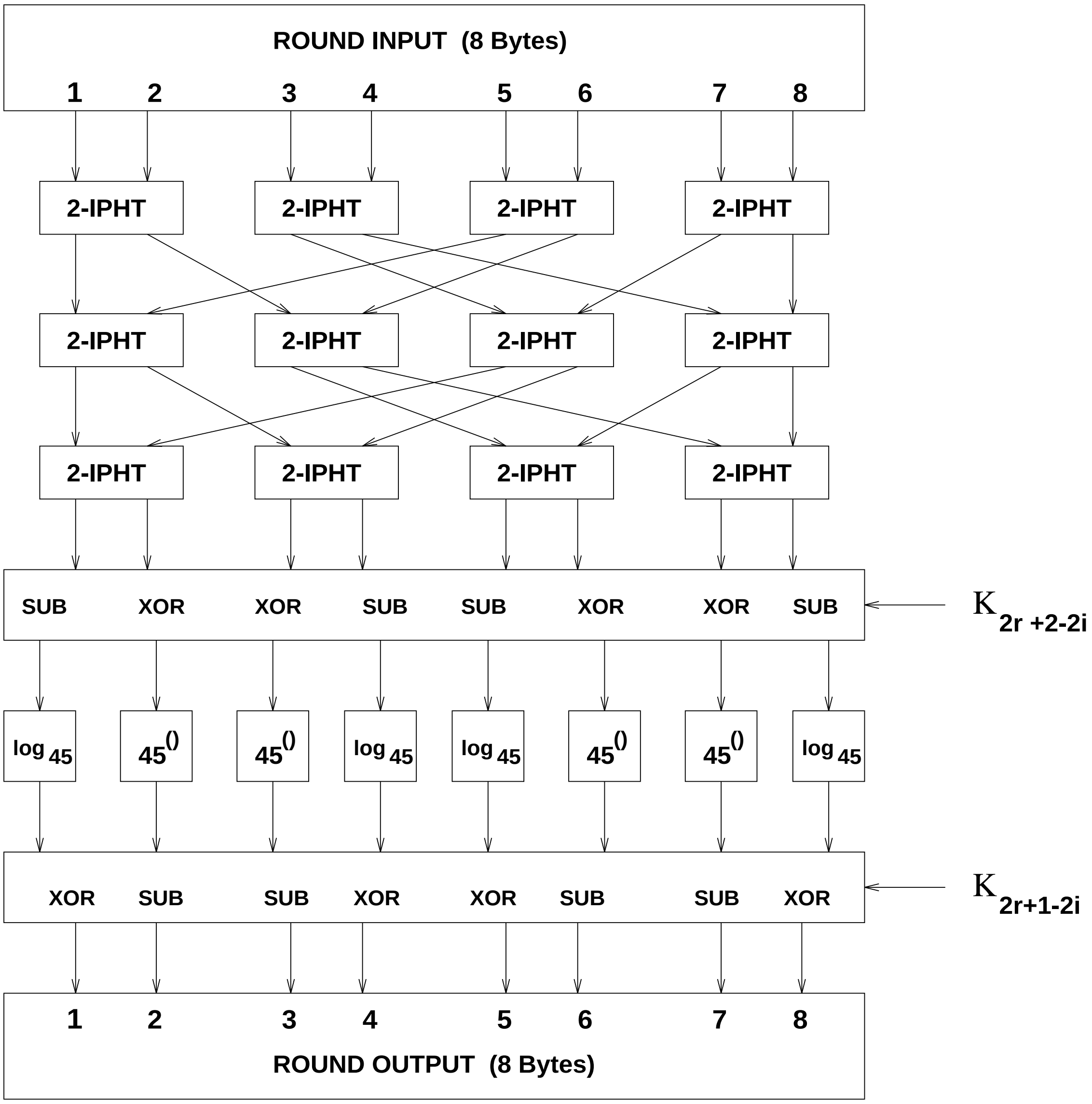

Figura 8: Round i della decodifica del SAFER SK-64

$K_{2r+1-2i}$ mediante l'operazione detta *Mixed XOR/Byte-Subtraction*.

3.4 Analisi delle scelte

SAFER fu progettato in accordo con i principi di confusione e di diffusione di Shannon [28] così da ottenere sicurezza nei cifrari con chiave segreta.
Quando in un round la sottochiave non è tutta zero nella fase di codifica, la sua combinazione mediante la *Mixed XOR/Byte-Addition* (oppure *Mixed Byte-Addition/XOR*) con il segnale dentro il round agisce come una combinazione non lineare rispettando le successive trasformazioni nella *nonlinear layer* e nelle *linear layer*. Questo da al cifrario la *confusione* richiesta e lo si vede dal legame complesso che si ha tra le statistiche del plaintext e quelle del ciphertext (purchè quei piccoli cambiamenti si diffondano velocemente attraverso il cifrario). La garanzia di questa *diffusione* in SAFER è infatti data dallo sviluppo di una nuova trasformazione lineare non ortodossa (PHT). La trasformazione di Hadamard tradizionale (HT) (chiamata qualche

volta anche "*Walsh transform*" o "*Walsh-Hadamard transform*") ha al posto delle equazioni (1) le equazioni

$$\begin{cases} b_1 = a_1 + a_2 \\ b_2 = a_1 - a_2 \end{cases} \tag{3}$$

Si vede come il determinante della matrice dei coefficienti è -2 che rende queste equazioni non invertibili per l'aritmetica sui byte (aritmetica mod 256) dove $-2 = 254$ non ha un inverso moltiplicativo. Ha anche lo sgradevole effetto di richiedere una moltiplicazione per $\frac{1}{2}$ nella trasformazione inversa in quei sistemi numerici dove 2 ha un inverso moltiplicativo.
Scegliendo equazioni la cui matrice dei coefficienti ha determinante uguale ad 1 evitiamo entrambi questi problemi, infatti possiamo usare la normale aritmetica sui byte e non c'è uno *scale factor* sgradito.
La PHT garantisce una diffusione completa dentro nel *three level linear layer* (vedi Appendice A). La garanzia di diffusione completa dentro un *layer* non viene realizzata pienamente quando si considerano cambiamenti di singoli bit sui byte input. Perchè a causa del fattore 2 delle equazioni (1), alcuni bit dell'input saranno diffusi solo in 4 byte (o 2 byte, o 1 byte) dell'input di una *linear layer*, ma il loro effetto è immediatamente diffuso a tutti gli 8 byte nella successiva *linear layer*. Questo lo si può vedere nell'Appendice B. Se abbiamo (1,0,0,0,0,0,0,0) la PHT è (8,4,4,2,4,2,2,1) quindi abbiamo una buona diffusione. Mentre dato che $2 \cdot 128 = 0 \bmod 256$ allora da (128,0,0,0,0,0,0,0) che contiene un unico bit non zero, avrà la PHT (0,0,0,0,0,0,0,128) che non mostra diffusione. Mentre da (0,0,0,0,0,0,0,128) si ottiene la PHT (128,128,128,128,128,128,128,128) che mostra la diffusione completa sui byte output. Si nota come (128,0,0,0,0,0,0,0) sia l'unico vettore che non mostra diffusione sotto un'applicazione di PHT.
Non si conoscono altri cifrari con una diffusione garantita così rapida. Questa diffusione rapida è la principale ragione per cui 8 round di codifica sono abbastanza per rendere SAFER resistente agli attacchi.

3.5 Schedulazione delle chiavi

La schedulazione delle chiavi, che è la procedura che genera le sottochiavi K_2 K_3 $\cdots, K_{2r+1}$ dalla sottochiave K_1 (a cui è stato assegnato il valore della chiave selezionata dall'utente), è illustrata in Figura 9.
Il processo di generazione delle sottochiavi consiste in $2r$ round durante i quali verranno utilizzate $2r$ *key biases* (da 64 bit ciascuna). Esse hanno la proprietà di assicurare che ogni sottochiave risultante sia generata in maniera random (ed in particolare che non più di un round possa essere tutto zero).
Indichiamo con $b[i,j]$ il j-esimo byte della *key bias* B_i il cui valore è dato dalla seguente relazione:

$$b[i,j] = [45^{45^{(9i+j)} \mathrm{mod} 257}] \mathrm{mod} 257 \tag{4}$$

Una tabella che da il preciso valore delle *key biases* per SAFER SK-64 è data nell'Appendice C. L'esame di questa tabella mostra che la sequenza risultante di *key biases* è casuale.
L'utilizzo di queste *key biases*, che è una novità nei cifrari iterati ci assicura di non incorrere in *weak keys* come per esempio nel DES, dove ogni chiave generata può essere tutta zero.
La Figura 9 mostra come K, la chiave selezionata dall'utente a 64 bit (8 byte) con l'aggiunta

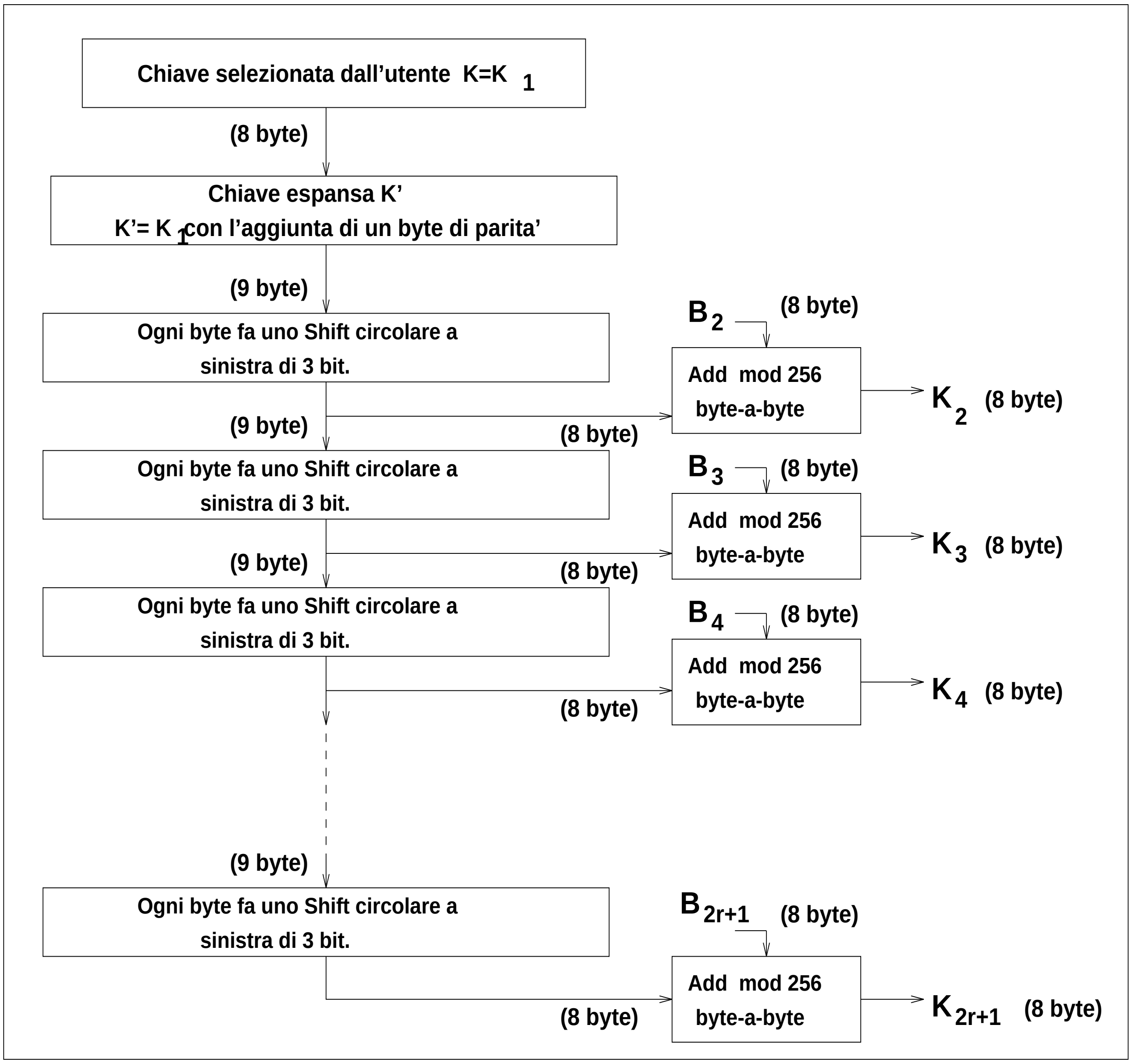

Figura 9: Schedulazione delle chiavi in SAFER SK-64

di un nono byte di paritá, indicato in Figura 9 con K' è usata per generare le sottochiavi (64 bit) K_2, $K_3, \cdots, K_{2r+1}$ che sono richieste per realizzare gli r round di codifica.
In ogni round si effettuano shift circolari successivi a partire dalla chiave K'. Lo shift in questione è uno shift circolare byte-a-byte di 3 bit a sinistra.
Le *key biases* vengono addizionate (l'addizione è modulo 256) con 8 dei 9 byte shiftati circolarmente di K' (gli 8 byte selezionati nel primo round sono i byte 1,....,8, poi nel secondo round i byte 2,....,9, poi 3,...,9,1 e così via).
Idealmente si desidera che la sequenza delle sottochiavi $K_1, \cdots, K_{2r+1}$ sia una sequenza di sottochiavi indipendenti e scelte in maniera uniformemente random. Naturalmente questo non può essere realizzato in senso stretto perchè tutte le sottochiavi in questa sequenza sono determinate interamente dalla prima sottochiave K_1 selezionata dall'utente. Realizzare questa indipendenza, anche se complicata, è l'obiettivo della schedulazione delle chiavi che così non può essere sfruttata da un attaccante.

3.6 SAFER SK-128

Il SAFER SK-128 ha la struttura dei round di codifica e di decodifica, le trasformazioni output e input e le *key biases* identiche a quelle del SAFER SK-64; le uniche differenze stanno nel processo di generazione delle sottochiavi e nel numero di round.
Le $2r + 1$ sottochiavi di 64 bit sono derivate dalla chiave selezionata dall'utente di 128 bit in accordo alla schedulazione delle chiavi mostrata in Figura 10, mentre sono consigliati 10 round fino ad un massimo di 12 round.

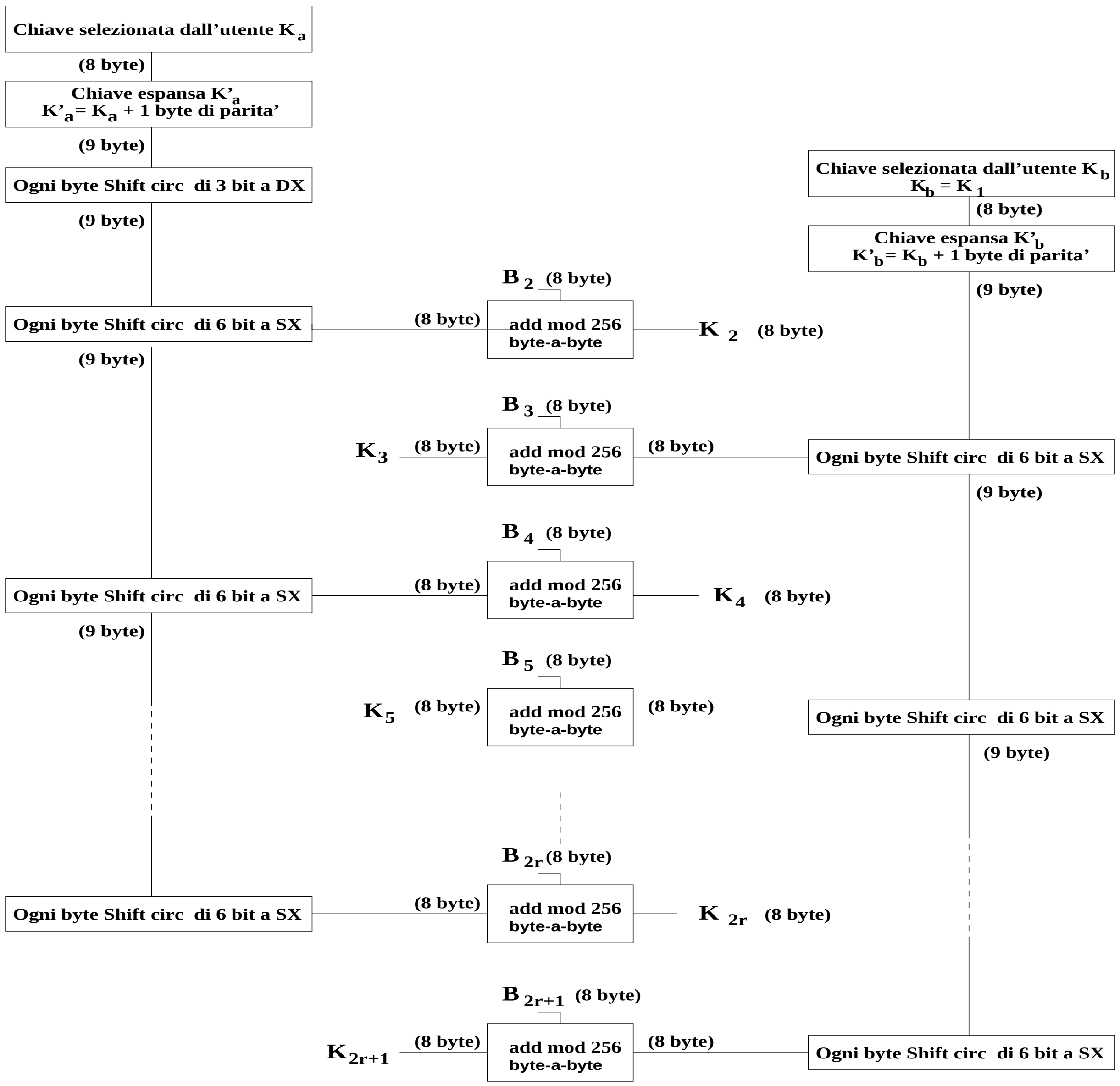

Figura 10: Schedulazione della chiave in SAFER SK-128

Il programma della schedulazione delle chiavi di SAFER SK-128 è stato realizzato da un Gruppo Speciale di Progetto del Ministero degli Interni di Singapore, che ha dato all'autore del SAFER l'onere di presentarlo alla comunità crittografica rinunciando ad ogni diritto di proprietà . L'idea fu gradita da Massey che infatti l'ha adottata.

La metà sinistra e destra della chiave selezionata dall'utente a 128 bit sono denotate rispettivamente con K_a e K_b.
Il modo di selezionare 8 byte dai 9 byte della semi-chiave sinistra e destra è identico al precedente, in particolare si selezionano i byte 1,2,...,8; 3,4,....,9,1 e così via della chiave shiftata K_a e i byte 2,3,...,9; 4,5,...9,1,2 e così via della chiave shiftata K_b. Invece la taglia dello shift circolare cambia. Per K_a il primo shift circolare è di 3 bit a destra mentre i seguenti sono di 6 bit a sinistra. Per K_b gli shift circolari sono tutti di 6 bit a sinistra.
Analogo al SAFER SK-64 è il modo di generarsi le *key biases*. Le *key biases* pari vengono addizionate (l'addizione è mod 256) con 8 dei 9 byte shiftati circolarmente della K_a, mentre le *key biases* dispari vengono addizionate con 8 dei 9 byte shiftati circolarmente della K_b (vedi Figura 10).
Esempi di codifica sono presentati nell'Appendice E.

3.7 Conclusioni

Il motivo della popolarità del SAFER è legato principalmente al carattere di non proprietà e al fatto che può essere usato liberamente da chiunque; inoltre altro fattore importante è che dentro i processi di codifica e di decodifica tutte le operazioni compiute sono fatte su byte. Questo rende il SAFER estremamente attraente per l'implementazione su smart card con processori di 8 bit. Infatti un prototipo di implementazione del SAFER è risultata essere circa 2,5 volte più veloce di un'implementazione ottimizzata completa su smart card del DES.
Si è visto come il SAFER ottiene una buona diffusione e una buona confusione, che sono le due caratteristiche basilari che contribuiscono alla sicurezza del cifrario a blocchi.
La migliore misura di sicurezza disponibile oggi per un cifrario iterativo a blocchi è la sua resistenza all'attacco della crittoanalisi differenziale [1]. È facile vedere che per l'appropriata definizione di differenza tra una coppia di blocchi plaintext (o una coppia di blocchi ciphertext) il SAFER è un Cifrario di Markov[7]. Questo semplifica molto la sua analisi sulla resistenza alla crittoanalisi differenziale. Il SAFER è sicuro contro la crittoanalisi differenziale, unico neo è il numero (grosso) di round necessari per avere sicurezza.

3.8 Appendici

Appendice A

Vettore input e vettore output al *three level linear layer* nella fase di codifica.
Gli ultimi tre esempi sono esplicitati più in particolare nell'Appendice B.

VETTORE INPUT	0	0	0	0	0	0	0	1
VETTORE OUTPUT	1	1	1	1	1	1	1	1
VETTORE INPUT	0	0	0	0	0	0	1	0
VETTORE OUTPUT	2	2	1	1	2	2	1	1
VETTORE INPUT	0	0	0	0	0	1	0	0
VETTORE OUTPUT	2	2	2	2	1	1	1	1
VETTORE INPUT	0	0	0	0	1	0	0	0
VETTORE OUTPUT	4	4	2	2	2	2	1	1
VETTORE INPUT	0	0	0	1	0	0	0	0
VETTORE OUTPUT	2	1	2	1	2	1	2	1
VETTORE INPUT	0	0	1	0	0	0	0	0
VETTORE OUTPUT	4	2	2	1	4	2	2	1
VETTORE INPUT	0	1	0	0	0	0	0	0
VETTORE OUTPUT	4	2	4	2	2	1	2	1
VETTORE INPUT	1	0	0	0	0	0	0	0
VETTORE OUTPUT	8	4	4	2	4	2	2	1
VETTORE INPUT	128	0	0	0	0	0	0	0
VETTORE OUTPUT	0	0	0	0	0	0	0	128
VETTORE INPUT	0	0	0	0	0	0	0	128
VETTORE OUTPUT	128	128	128	128	128	128	128	128

Appendice B

3 esempi di come avviene la *diffusione* nel *three level linear layer*:

	a_1 a_2	a_1 a_2	a_1 a_2	a_1 a_2
input	0 0	0 0	0 0	0 1
b_1:	0	0	0	1
b_2:	0	0	0	1
	0 0	0 1	0 0	0 1
b_1:	0	1	0	1
b_2:	0	1	0	1
	0 1	0 1	0 1	0 1
b_1:	1	1	1	1
b_2:	1	1	1	1
	1 1	1 1	1 1	1 1

	a_1 a_2	a_1 a_2	a_1 a_2	a_1 a_2
input	1 0	0 0	0 0	0 0
b_1:	2	0	0	0
b_2:	1	0	0	0
	2 0	0 0	1 0	0 0
b_1:	4	0	2	0
b_2:	2	0	1	0
	4 0	2 0	2 0	1 0
b_1:	8	4	4	2
b_2:	4	2	2	1
	8 4	4 2	4 2	2 1

	a_1 a_2	a_1 a_2	a_1 a_2	a_1 a_2
input	128 0	0 0	0 0	0 0
b_1:	0	0	0	0
b_2:	128	0	0	0
	0 0	0 0	128 0	0 0
b_1:	0	0	0	0
b_2:	0	0	128	0
	0 0	0 0	0 0	128 0
b_1:	0	0	0	0
b_2:	0	0	0	128
	0 0	0 0	0 0	0 128

Appendice C

Tabella delle *key biases* per il SAFER:

(In questa tabella sono elencate le *key biases* da B_2 a B_{21}, sebbene, quando si usano 8 round per il SAFER, le *key biases* richieste sono soltanto quelle che vanno da B_2 a B_{17}.)

Bias	B_2	22	115	59	30	142	112	189	134
Bias	B_3	71	126	36	86	241	119	136	70
Bias	B_4	177	186	163	183	16	10	197	55
Bias	B_5	201	90	40	172	100	165	236	171
Bias	B_6	198	103	149	88	13	248	154	246
Bias	B_7	102	220	5	61	211	138	195	216
Bias	B_8	106	233	54	73	67	191	235	212
Bias	B_9	155	104	160	101	93	87	146	31
Bias	B_{10}	113	92	187	34	193	190	123	188
Bias	B_{11}	99	148	95	42	97	184	52	50
Bias	B_{12}	253	251	23	64	230	81	29	65
Bias	B_{13}	143	41	221	4	128	222	231	49
Bias	B_{14}	127	1	162	247	57	218	111	35
Bias	B_{15}	254	58	208	28	209	48	62	18
Bias	B_{16}	205	15	224	168	175	130	89	44
Bias	B_{17}	125	173	178	239	194	135	206	117
Bias	B_{18}	19	2	144	79	46	114	51	133
Bias	B_{19}	141	207	169	129	226	196	39	47
Bias	B_{20}	122	159	82	225	21	56	43	252
Bias	B_{21}	66	199	8	228	9	85	94	140

Appendice D

Esempi di codifica con SAFER SK-64.

(cioè con la schedulazione delle chiavi a 64 bit rafforzata).

PLAINTEXT	1	2	3	4	5	6	7	8
Key	0	0	0	0	0	0	0	1
dopo round 1	131	177	53	27	130	249	141	121
dopo round 2	68	73	32	102	134	54	206	57
dopo round 3	248	213	217	11	23	68	0	243
dopo round 4	194	62	109	79	24	18	13	84
dopo round 5	153	156	246	172	40	72	173	39
dopo round 6	154	242	34	6	61	35	216	28
CRYPTOGRAM	21	27	255	2	173	17	191	45

PLAINTEXT	1	2	3	4	5	6	7	8
Key	1	2	3	4	5	6	7	8
dopo round 1	223	98	177	100	46	234	13	210
dopo round 2	182	246	230	93	158	14	48	89
dopo round 3	45	234	128	149	40	101	10	134
dopo round 4	30	17	249	236	158	120	69	100
dopo round 5	1	200	182	241	0	127	152	162
dopo round 6	144	85	94	214	5	38	65	150
CRYPTOGRAM	95	206	155	162	5	132	56	199

Appendice E

Esempi di codifica con SAFER SK-128.

(cioè con la schedulazione delle chiavi a 128 bit rafforzata).

PLAINTEXT	1	2	3	4	5	6	7	8
Key K_a	0	0	0	0	0	0	0	1
Key K_b	0	0	0	0	0	0	0	1
dopo round 1	131	177	53	27	130	249	141	121
dopo round 2	68	73	32	102	134	54	206	57
dopo round 3	248	213	217	11	23	68	0	243
dopo round 4	194	62	109	79	24	18	13	84
dopo round 5	153	156	246	172	40	72	173	39
dopo round 6	154	242	34	6	61	35	216	28
dopo round 7	100	31	172	67	44	75	133	219
dopo round 8	78	226	239	135	210	83	93	72
dopo round 9	72	64	46	195	163	159	243	114
dopo round 10	3	133	76	190	191	52	220	123
CRYPTOGRAM	65	76	84	90	182	153	74	247

PLAINTEXT	1	2	3	4	5	6	7	8
Key K_a	1	2	3	4	5	6	7	8
Key K_b	0	0	0	0	0	0	0	0
dopo round 1	64	214	74	216	103	222	26	54
dopo round 2	61	14	68	15	46	111	124	80
dopo round 3	197	124	96	59	255	24	2	30
dopo round 4	63	59	214	103	236	166	153	24
dopo round 5	66	254	26	45	152	223	5	122
dopo round 6	89	47	58	105	161	38	135	45
dopo round 7	19	202	174	44	57	206	52	25
dopo round 8	78	179	113	208	169	26	121	22
dopo round 9	53	17	81	215	120	37	206	246
dopo round 10	189	177	9	0	186	82	208	253
CRYPTOGRAM	255	120	17	228	179	167	46	113

PLAINTEXT	1	2	3	4	5	6	7	8
Key K_a	0	0	0	0	0	0	0	0
Key K_b	1	2	3	4	5	6	7	8
dopo round 1	95	186	209	220	166	66	213	10
dopo round 2	200	65	189	120	96	135	42	166
dopo round 3	64	199	43	166	132	171	31	40
dopo round 4	199	167	76	189	145	158	241	19
dopo round 5	71	55	184	212	108	198	77	108
dopo round 6	173	197	139	11	17	48	97	59
dopo round 7	17	51	142	4	170	7	207	124
dopo round 8	62	205	253	225	167	179	228	202
dopo round 9	133	168	127	138	193	243	34	226
dopo round 10	59	194	69	220	220	231	123	148
CRYPTOGRAM	73	201	157	152	165	188	89	8

4 TEA (Tiny Encryption Algorithm)

Anche l'algoritmo TEA appartiene a quell'ultima generazione di sistemi crittografici la cui progettazione è stata guidata dal proposito di attenersi alle richieste dei futuri standard della crittografia odierna. Inoltre gli autori del TEA, Wheeler e Needham [35], hanno strutturato l'algoritmo in modo da rendere facile ogni sua implementazione nella maggior parte dei linguaggi eseguibili sui computer moderni. Nel seguito viene riportato un frammento di una possibile implementazione dell'algoritmo realizzata in linguaggio C. Questo programma accetta parole di 32 bit e fa uso di una breve fase d'inizializzazione e di una piccola iterazione la quale dovrebbe accrescere ulteriormente la sicurezza della cifratura dei dati.

ROUTINE DI CODIFICA

```
                /*Dati immagazzinati in V[0] e V[1] e chiave in K[0],...,K[3]*/

void code ( long *V,long*k )
{
unsigned long Y=V[0], Z=V[1], sum=0,                        /*Inizializzazione*/
              delta=0x9e3779b9,         /*Una tabella costante per la chiave*/
              n=32 ;
while (n>0)   {                                       /*Ciclo base di partenza*/
 sum += delta
   Y += ((Z<<4)+K[0])^(Z+sum)^((Z>>5)+K[1]);
   Z += ((Y<<4)+K[2])^(Y+sum)^((Y>>5)+K[3]);
   n--;
              }                                                   /*Fine ciclo*/
V[0]=Y; V[1]=Z;
}
```

4.1 Descrizione e scelte dell'algoritmo

Nonostante la routine vista precedentemente sia di tipo Feistel, si nota che al posto di uno XOR, vengono utilizzati come operatori reversibili, l'addizione e successivamente la sottrazione. Difatti lo XOR combinato con l'ADD serve esclusivamente per ottenere una non linearitá, mentre uno shift duale causa un mixaggio ripetuto di tutti i bit della chiave con quelli dei dati. La chiave immagazzinata in K[0],...,K[3], è un insieme di 128 bit, ciò dovrebbe essere sufficiente affinchè semplici attacchi di forza bruta non risultino efficaci. La schedulazione della chiave viene effettuata tramite l'addizione della costante `delta` con la Z non shiftata, in questo modo la chiave K[0],...,K[3] rimane inalterata. In altre versioni dell'algoritmo testate dagli autori, la costante `delta` veniva sommata direttamente alle K[0],...,K[3], ma la versione vista precedentemente sembra essere più semplice e più efficace. La costante `delta` deriva dal numero aureo e il suo valore è

$$\texttt{delta} = (\sqrt{5} - 1) \cdot 2^{31}.$$

Per come è strutturato il programma, in ogni ciclo viene usato un multiplo differente di `delta`, ciò viene fatto per evitare l'utilizzo di una singola costante da modificare continuamente in maniera tediosa. C'è l'intuizione che l'algoritmo suddescritto non sia molto sensibile al valore di `delta`, la scelta di questi infatti, è stata motivata dall'unico scopo di evitare un pessimo valore per gli specifici usi in questione. Visto il valore assegnato inizialmente a `delta`, si riscontra che quest'ultima assume un valore dispari sia con un troncamento, sia con un arrotondamento per eccesso (infatti $(\sqrt{5}-1)\cdot 2^{31} = 2.654435e{+}9$). In questo modo si garantisce il cambiamento di tutti i digit della somma durante l'aggiornamento della variabile `sum` nei 32 cicli del TEA. In versioni dell'algoritmo precedenti a quella suddescritta, gli autori del TEA testarono la possibilità di utilizzare, al posto dell'operatore ADD, la moltiplicazione per ottenere il mixaggio dei bit della chiave con quelli dei dati. A questo tipo d'implementazione però , subbentrò l'onere di dover aggiungere uno shift ad ogni ciclo, inoltre rispetto all'implementazione dell'algoritmo vista prima, tale versione risultò due volte più lenta. Fra i vari test effettuati dai progettisti dell'algoritmo, fu valutato anche l'uso di una tavola *look up* all'interno del TEA, ma ciò finora non sembra aver apportato nessuna miglioria, anzi l'uso di questo tipo di tavola comporta un certo appesantimento. Infatti se K[Z & 3] (elemento generico di una tavola *look up* composta da parti della chiave K di 128 bit e parti del messaggio) fosse usato al posto di K[0], subbentrerebbe una probabilità di $(3/4)^{32}$ che un elemento non venga assolutamente utilizzato (valore ottenuto considerando che ogni elemento viene selezionato fra quattro elementi in 32 cicli differenti), mentre la probabilità di ritardo sull'uso specifico di un singolo elemento è ancora più alta. Per questi motivi e per i notevoli tempi di inizializzazione apportati dall'uso di questo tipo di tavola, l'utilizzo di quest'ultima non fu mai preso in considerazione.

4.2 Analisi dell'algoritmo

Prima di procedere alla pubblicazione dell'algoritmo vista prima, Wheeler e Needham, hanno valutato un numero considerevole di algoritmi e quello visto in precedenza non è risultato essere nè il più veloce nè il più breve, ma è stato ritenuto essere il miglior compromesso per quanto riguarda la sicurezza, la facilità d'implementazione, la mancanza di tavole specifiche e per le ragionevoli prestazioni descritte prima. Nei vari test effettuati dai reviewer è stato provato anche un cycle test con 34 bit per determinare la lunghezza dei cicli stessi e con una versione più potente di questo tipo di test, è stata calcolata la lunghezza del ciclo di una funzione differenziale del tipo

$$d(x) = f(x \oplus 2^p) \oplus f(x)$$

la quale è in grado di testare la resistenza di alcuni tipi di crittoanalisi differenziale [1], ma finora nessuno di questi test ha riscontrato pesanti debolezze del TEA. L'algoritmo presentato è abbastanza semplice, il che permette una sua semplice traduzione in un ampio numero di linguaggi differenti o in linguaggio assembler, purchè lo XOR venga eseguito in una sola operazione, inoltre è abbastanza breve per poterlo caricare dalla memoria o per copiarlo direttamente e usa una sequenza di operazioni su parole piuttosto che perdere la potenza del computer usando operazioni agenti su byte o su blocchi di 4 bit. Soltanto su di un linguaggio privo di shift o di XOR la codifica dell'algoritmo risulta difficile. Anche l'implementazione hardware non è complessa, infatti la sua complessità è dello stesso ordine del DES [16], nonostante la chiave del TEA sia due volte più lunga. Viene ora illustrata di seguito l'implementazione in linguaggio C della routine di decodifica la quale è strutturalmente identica a quella di codifica.

ROUTINE DI DECODIFICA

```
void decode ( long*V, long*K )                        {
    unsigned long n = 32, sum, Y = V[0], Z=V[1]
    delta = 0x9e3779b9;
sum = delta<<5;
                                                                /*inizio ciclo*/
while ( n--0)                                     {
    Z-= (( Y<<4 ) + K[2])^( Y + sum )^(( Y>>5 ) + K[3]);
    Y-= (( Z<<4) + K[0] )^( Z + sum )^(( Z>>5 ) + K[1]);
    sum-= delta;                                  }
                                                               /*fine ciclo*/
V[0] = Y; V[1]= Z;
                                                      }
```

4.3 Possibilitá d'impiego del TEA

Questo tipo di algoritmo può rimpiazzare il software d'implementazione del DES. Infatti, anche se la velocità non sia stata lo scopo principale nella creazione di questo algoritmo (32 cicli con 64 round), in un'implementazione il TEA è risultato tre volte più veloce di un buon software d'implementazione DES (16 round). Tutte le modalità d'uso del DES possono essere replicate dal TEA, inoltre si possono variare facilmente sia i cicli di calcolo, sia la stessa chiave. Una possibile miglioria potrebbe essere quella di copiare K[0],...,K[3], in a, b, c, d, prima che l'iterazione di indicizzazione sia portata fuori dal loop, ciò dovrebbe ridurre il tempo di esecuzione di circa 1/6. Un'altra modifica può essere quella di strutturare l'implementazione dell'algoritmo come una coppia di macro, in questo modo le chiamate a capo verrebbero rimosse. Ci si aspetta che con l'incremento del numero di iterazioni si possa accrescere il grado di sicurezza.

5 SEAL (Software-optimized Encryption ALgorithm)

5.1 Introduzione

Spesso sorge l'esigenza di voler effettuare una cifratura di messaggi ad un tasso di dati abbastanza alto e questi è un requisito solitamente conseguito solo con l'aiuto di un supporto hardware crittografico. Sfortunatamente, l'hardware crittografico celere è soventemente assente e la confidenzialità con i dati viene sacrificata in quanto il costo del software crittografico è stimato troppo costoso. Il costo computazionale del software crittografico è una funzione dell'algoritmo di base e della qualità dell'implementazione dell'algoritmo stesso. Pur non tenendo conto dell'implementazione, un algoritmo crittografico strutturato per girare bene in hardware non sarà eseguibile allo stesso modo su un supporto software. L'algoritmo crittografico del DES [16] non è un'eccezione, nonostante questi sia il metodo più diffuso per la cifratura simmetrica. Per cui visti quali sono i propositi generali dei computer di oggi, quello che risulta essere necessario per un buon sistema di cifratura, è un algoritmo capace di essere eseguito celermente e soprattutto essere software-oriented (cioè eseguibile interamente via software). Con questi propositi è stato progettato SEAL[36] (Software Encryption Algorithm, conosciuto come SEAL 1.0). Questo algoritmo è inteso essere adoperato come uno stream cipher, su un moderno processore a 32-bit SEAL può cifrare messaggi ad un tasso di circa 5 istruzioni per byte. Al confronto, l'algoritmo DES è 10-30 volte più dispendioso. Addirittura un Codice a Ridondanza Ciclica (CRC) è più dispendioso. Il cifrario è una funzione pseudorandom, la quale sotto il controllo di una chiave (dapprima preprocessata in una tavola interna) espande un indice corto in una stringa pseudorandom molto più lunga. Questa stringa può essere adoperata come un one-time pad, ovvero per ottenere il testo in chiaro partendo da tale stringa basterà inserire quest'ultima come input allo stesso algoritmo di codifica.
Le potenzialità degli algoritmi crittografici software furono già introdotte da Merkle nel 1991, egli infatti propose tre algoritmi software-oriented [31]. Fra gli algoritmi proposti da Merkle troviamo Khufu, il quale in spirito è molto simile a SEAL. In contrasto a Khufu, SEAL non è un cifrario a blocchi, il suo uso invece è inteso come uno stream cipher. Questa è una differenza la quale ha aiutato gli autori, Rogaway e Don Coppersmith, a spingere sull'ulteriore velocità a vantaggio di SEAL piuttosto di quella che Merkle aveva fatto nella sua più aggressiva istanza di Khufu (inferiore numero di rounds). Un primo cifrario a blocchi software-oriented precedente a Khufu è FEAL (Fast Data Encryption Algorithm) [34]. Ma questo algoritmo (e varianti di esso) non è stato testato essere particolarmente sicuro (vedi [1] per la storia e attacchi) e neppure del tutto veloce. Gli algoritmi RC2 e RC4 sono proprietà RSA (algoritmi disegnati da Rivest [33]) e l'ultimo menzionato è uno stream cipher del quale si dice che sia celere, ma finora non è stata effettuata nessuna esposizione di questo algoritmo.

5.2 Caratteristiche di SEAL

Caratteristiche della chiave e scelte di progetti di SEAL sono spiegati di seguito.

5.2.1 Preprocessing della chiave

Nelle applicazioni classiche nelle quali è richiesta una cifratura veloce dei dati via software, si riscontra che la cifratura dei dati può avvenire durante il corso di una sessione di comunicazione a un remoto partner, o durante il corso di una login session a una macchina particolare. In ogni caso la chiave a che protegge la sessione è precisata nella sessione d'inizializzazione.

Tipicamente questa sessione viene ultimata in pochi microsecondi e non è un'operazione a tempo critico. E' quindi accettabile spendere qualche microsecondo per la mappattura della chiave (nel cifrario in questione è di 160 bit), piuttosto che utilizzare una meno concisa rappresentazione della trasformazione crittografica di questa chiave e il cifrario SEAL è dotato di questa caratteristica. Il primo compito è quello di inizializzare le tavole T, R e S, le quali dipendono tutte dalla chiave a, infatti la chiave viene utilizzata unicamente per la definizione di queste tre tavole. L'inizializzazione di queste tavole viene effettuata utilizzando una funzione $G_a(i)$, la quale è specificata per una stringa a di 160 bit e per un intero $0 \leq i \leq 2^{32}$, considerato come una stringa a 32 bit il cui valore come numero binario senza segno è i. Questa funzione è basata direttamente sull'SHA ed definita direttamente nelle sezioni 5-7 di [32]; la definizione viene qui ripetuta solo per facilità di riferimento.

SHA

Dapprima vengono fatte le seguenti definizioni.

Per $0 \leq t \leq 19$, poni $K_t = 0x5a827999$ e $f_t(B,C,D) = (B \wedge C) \vee (\overline{B} \wedge D)$
Per $20 \leq t \leq 39$, poni $K_t = 0x6ed9eba1$ e $f_t(B,C,D) = B \oplus C \oplus D$
Per $40 \leq t \leq 59$, poni $K_t = 0x8f1bbcdc$ e $f_t(B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
Per $60 \leq t \leq 79$, poni $K_t = 0xca62c1d6$ e $f_t(B,C,D) = B \oplus C \oplus D$.

Input $(a,\, i)$
La stringa a di 160 bit è frammentata in 5 parole di 32 bit, $A = H_0H_1H_2H_3H_4$
e M_1 a 512 bit è posto a $i \parallel 0^{480}$ e dopo processato da :

1. Dividi M_1 in 16 parole W_0, $W_1, ...W_{15}$ dove W_0 è la parola sinistra più significativa, cosicchè
$$W_0 = i, \;\; W_1 = W_2 = ... = W_{15}.$$
2. For t = 16 to 79 let $W_t = W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}$.
3. let $A = H_0$, $B = H_1$, $C = H_2$, $D = H_3$, $E = H_4$.
4. For t = 0 to 79 do
TEMP = $A \ggg 27 + f_t(B,C,D) + E + W_t + K_t$; $E = D$; $D = C$;
$C = B \ggg 2$; $B = A$; A=TEMP;
5. $H_0 = H_0 + A$; $H_1 = H_1 + B$; $H_2 = H_2 + C$; $H_3 = H_3 + D$; $H_4 = H_4 + E$;
Dopo il processing di M_1 il valore di G_a(i) e la stringa di 160 bit $H_0H_1H_2H_3H_4$.

Non appena è terminata la computazione della chiave a la funzione G viene reindicizzata per costruire una funzione Γ le cui immagini sono parole di 32 bit invece di 160 bit l'una. La funzione Γ è definita da $\Gamma_a(i) = H^i_{i\bmod 5}$ dove $H^i_0H^i_1H^i_2H^i_3H^i_4 = G_a(\lfloor i/5 \rfloor)$. In questo modo una tabella di valori Γ è esattamente una tabella di valori G letta da sinistra a destra e da top a bottom.

A questo punto vengono definite le tavole T, R e S:

$$\begin{array}{lll} T[i] = \Gamma_a(i) & \text{per ogni} & 0 \leq i < 512, \\ S[j] = \Gamma_a(0x1000 + j) & \text{per ogni} & 0 \leq j < 256, \\ R[k] = \Gamma_a(0x2000 + k) & \text{per ogni} & 0 \leq k < 4\lceil (L-1)/8192 \rceil. \end{array}$$

Si nota che la computazione di queste tavole richiede il calcolo della funzione di compressione dell'SHA su di un numero variabile di blocchi - 131 volte per ottenere un output di 512 byte di lunghezza, 207 volte per un output massimale di 64 Kbyte -.

5.2.2 Incremento lunghezza, Funzione pseudorandom e Variabile output

La funzione SEAL è un tipo di obiettivo crittografico soprannominato" funzione pseudorandom." Tali obiettivi vengono prima descritti in [30]. Diversamente da molte funzioni pseudorandom, SEAL è un incrementatore di lunghezza. Sotto il controllo di una chiave a di 160 bit, la funzione SEAL mappa una stringa n di 32 bit in una stringa SEAL$_a(n)$ di L-bit. Il numero L può essere costruito sia grande che piccolo in base alle necessità delle specifiche applicazioni. Una chiave a' di lunghezza arbitraria può indicizzare SEAL semplicemente definendo a=SHA[a']. Una funzione pseudorandom può essere adoperata per fare un buono stream cipher, nel quale la cifratura di un messaggio dipende non solo dalla chiave a e dal messaggio X ma anche dalla posizione del messaggio nel flusso di dati. Questa posizione spesso è una sequenza di numeri, presente già nelle applicazioni che stiano adoperando il metodo crittografico. Altre volte n può indicare l'indirizzo fisico di un pezzo di dato. In qualsiasi caso la cifratura di una stringa nella posizione n è data da $(n, X \oplus SEAL_a(n))$, supponendo che si sia posto L = $|X|$ bit di output di SEAL$_a(n)$.

5.2.3 Table-driven cipher

La decisione di fare un cifrario che utilizzasse una grossa tavola (come Khufu o il software DES), al posto di un programma a linee dirette di operazioni logiche (come MD5 o SHA), fu presa perchè si intuì che l'approccio al *table driven* conduceva a un cifrario più celere e più facile da progettare. Con l'algoritmo della tavola guidata si dovrebbe ottenere una più rapida diffusione e ci sarebbero piccole tentazioni a produrre un cifrario la cui efficiente implementazione necessiti di codice automodificante. Durante la progettazione sorse il seguente problema: in vista di un'iterazione al primo livello di cache i server potessero volere immagazzinare in un secondo livello cache la rappresentazione della trasformazione cifrata di dieci client, fu deciso quindi che non si dovesse essere troppo espansivi con la taglia delle tavole da usare, per cui la scelta fu di usare tavole di circa 3 Kbyte.

procedure$_a(n, \ell, A, B, C, D, n1, n2, n3, n4)$
$A \leftarrow n \oplus R[4\ell]$;
$B \leftarrow n \ggg 8 \oplus R[4\ell + 1]$;
$C \leftarrow n \ggg 16 \oplus R[4\ell + 2]$;
$D \leftarrow n \ggg 24 \oplus R[4\ell + 2]$;
for$j \leftarrow 1$ **to** 2 **do**
    $P \leftarrow A\&0x7fc$; $B \leftarrow B + T[P/4]$; $A \leftarrow A \ggg 9$;
    $P \leftarrow B\&0x7fc$; $C \leftarrow C + T[P/4]$; $B \leftarrow B \ggg 9$;
    $P \leftarrow C\&0x7fc$; $D \leftarrow D + T[P/4]$; $C \leftarrow C \ggg 9$;
$(n_1, n_2, n_3, n_4) \leftarrow (D, B, A, C)$;
$P \leftarrow A\&0x7fc$; $B \leftarrow B + T[P/4]$; $A \leftarrow A \ggg 9$;
$P \leftarrow B\&0x7fc$; $C \leftarrow C + T[P/4]$; $B \leftarrow B \ggg 9$;
$P \leftarrow C\&0x7fc$; $D \leftarrow D + T[P/4]$; $C \leftarrow C \ggg 9$;
$P \leftarrow D\&0x7fc$; $A \leftarrow A + T[P/4]$; $D \leftarrow D \ggg 9$;

Figura 1. Inizializzazione di $(A, B, C, D, n_1, n_2, n_3, n_4)$ utilizzando (n, ℓ). Questa inizializzazione dipende dalle tabelle T e R, le quali derivano direttamente dalla chiave A.

5.3 Descrizione dell'algoritmo SEAL

5.3.1 Notazioni

Una stringa di 32 bit viene definita "word", mentre la definizione di "byte" corrisponde ad una stringa di 8 bit. La stringa vuota è denotata con λ. I bit di una stringa x di lunghezza t saranno numerati con $x_0 x_1 ... x_{t-1}$. I numeri vengono scritti in esadecimale preceduti da "$0x$" e utilizzando i simboli "a"-"f" per rappresentare rispettivamente i numeri decimali da 10 a 15. Con $y \ggg t$ viene denotato lo shift circolare destro della parola y di t bit; ad esempio, l' i-esimo bit di $y \ggg t$ è $y_{(i-t) \bmod 32}$. Con i simboli "$\wedge$" ,"$\vee$" e "$\oplus$" si denota AND, OR, XOR; con $\overline{A}$ il complemento di A. Con $A + B$ viene denotata la somma, ignorando il riporto, degli interi senza segno A e B; ciò naturalmente è la somma mod 2^{32} dei numeri A e B. Con "$\|$" si denota l'operatore di concatenazione, mentre con odd (.) si intende un predicato che risulta essere vero se e solo se il suo argomento è un numero dispari.

5.3.2 La Funzione pseudorandom

Dato il numero L, le tavole T, R e S determinate dalla chiave a e dati i 32 bit indicizzati con n, l'algoritmo di Figura 2 estende n ad una stringa pseudorandom di L-bit. L'algoritmo usa una routine "Inizializza" per mappare n e ℓ alle parole $A, B, C, D, n_1, n_2, n_3, n_4$. Tale procedura è data in Figura 1. Il loop esterno di Figura 2 viene stoppato dalla linea 10 quando un numero sufficiente di bit output è stato generato. Va notato inoltre, che tutte le divisioni per quattro fatte nella descrizione dell'algoritmo non devono essere realmente valutate come divisioni o come shift, il lavoro effettuato è semplicemente una indicizzazione all'interno della tavola per unità di byte, invece che per unità di word. Ciò su qualche piattaforma è molto efficiente, allo stesso modo di quanto lo possa essere su qualche macchina dotata di indirizzamento byte.

5.3.3 Lunghezza output

Si ricorda che SEAL è inteso come un generatore di output a lunghezza variabile. Supponendo che L sia il numero di bit output desiderato, allora verrà assunto un largo limite per L, dell'ordine di $L \leq 64 \times 1024 \times 8$. L'algoritmo stoppa la generazione di bit non appena viene prodotto un L', il quale è il più piccolo multiplo di 128 che abbia una dimensione di almeno L-bit.

function$_a(n)$

$y = \lambda;$

for $\ell \leftarrow 0$ **to** ∞ **do**

  $_a(n, \ell, A, B, C, D, n_1, n_2, n_3, n_4)$

  for $i \leftarrow$ **to** 64 **do**

    $P \leftarrow A \,\&\, 0x7fc;\ B \leftarrow B + T[P/4];\ A \leftarrow A \ggg 9;\ B \leftarrow B \oplus A;$

    $Q \leftarrow B \,\&\, 0x7fc;\ C \leftarrow C \oplus T[Q/4];\ B \leftarrow B \ggg 9; C \leftarrow C + B;$

    $P \leftarrow (P + C)\&0x7fc;\ D \leftarrow D + T[P/4];\ C \leftarrow C \ggg 9;\ D \leftarrow D \oplus C;$

    $Q \leftarrow (Q + D)\&0x7fc;\ A \leftarrow A \oplus T[Q/4];\ D \leftarrow D \ggg 9;\ A \leftarrow A + D;$

    $P \leftarrow (P + A)\&0x7fc;\ B \leftarrow B \oplus T[P/4];\ A \leftarrow A \ggg 9;$

    $Q \leftarrow (Q + B)\&0x7fc;\ C \leftarrow C + T[Q/4];\ B \leftarrow B \ggg 9;$

    $P \leftarrow (P + C)\&0x7fc;\ D \leftarrow D \oplus T[P/4];\ C \leftarrow C \ggg 9;$

    $Q \leftarrow (Q + D)\&0x7fc;\ A \leftarrow A + T[Q/4];\ D \leftarrow D \ggg 9;$

    $y \leftarrow y \parallel B + S[4i-4] \parallel C \oplus S[4i-3] \parallel D + S[4i-2] \parallel A \oplus S[4i-1];$

    if $|y| \geq L$ **then return** $(y_0 y_1 \ldots\ldots\ldots y_{L-1});$

    if $odd(i)$ **then**$(A, C) \leftarrow (A + n_1, C + n_2);$

      else $(A, C) \leftarrow (A + n_3, C + n_4);$

Figura 2. Il cifrario mappa una stringa n di 32 bit in una stringa$_a$(n) di L-bit sotto il controllo delle tabelle T, R e S derivanti dalla chiave a.

5.4 Piattaforme di Applicazione e Performance

I veicoli di esecuzione che dovrebbero far girare bene l'algoritmo includono Intel 386^{TM}, Intel 486^{TM} e il processore Pentium^{TM} e implementazioni dell' IBM Power Architetture^{TM}. A causa delle particolari sfide che sorgono nell'ottenere un cifrario che giri bene sui 386/486 e a causa della particolare diffusione di questa famiglia di processori, si è dovuto ottimizzare il cifrario mentalmente. Nonostante le difficoltà di ottimizzazione che si riscontrano su questi veicoli, ci si aspetta che il cifrario lavori bene su qualsiasi processore moderno a 32 bit. Le limitazioni rilevanti dei 386/486/pentium sono: un piccolo insieme registri; un'architettura di istruzione a due operandi; un piccolo primo livello cache. In più dettaglio, questi chips hanno le seguenti caratteristiche. Essi hanno microprocessori CISC a 32 bit, con clock a 25-66 MHz. Hanno 8 registri specifici complessivi, e 6 registri segmento. Generalmente le istruzioni lavorano su 2 operandi (A←A op B) invece di 3 (A←B op C). Sul 486, la forma più semplice delle istruzioni add, and, mov e xor utilizzano tutte un periodo di clock; ror (ruota a destra) ne prende due. Sul 386, ciascuno di questi valori è più grande di un'unità . Il processore usa un canale di istruzione a 4 fasi. Come conseguenza, se il registro base per il calcolo di un indirizzo corrisponde al registro destinazione dell'istruzione precedente, un ciclo extra verrà speso. Sul 486, l'uso di registro segmento non-default può addizionare un ciclo addizionale. Il 486 ha una cache di 8 Kbyte su chip sia per i dati che per le istruzioni; il processore Pentium ha due cache da 8 Kbyte, una per i dati e una per le istruzioni. Il processore Pentium ha due canali (bus) denominati U e V, uno dei quali ha un insieme di istruzioni molto ristretto. Non fu una meta designata che il cifrario dovesse esibire una struttura di dipendenza di istruzioni che ci permettesse di riempire sempre entrambi i canali. Con un'implementazione Assembler, con L=512 $\times$ 8 bit e la chiave gia mappata in una tabella, SEAL cifra ad un tasso di 3.6 Mbytes/second su di un IBM PS/2^{TM} dotato di un microprocessore 486 e clock 25 MHz. Un PC 486 con 50 MHz esegue lo stesso lavoro ad un tasso di 7.2 Mbytes/second. Con un'implementazione in linguaggio C, SEAL esegue la stessa cifratura ad un tasso di 4.5 Mbytes/second su un RISC System/6000^{TM} Modello 530. (Questa una macchina a 40 MHz.) Questi esperimenti computano $\oplus_{i=0}^{M}$ SEAL$_a$(i) per un grosso M.

5.5 Accorgimenti finali

Molte applicazioni le quali richiedono una cifratura veloce via software, richiedono anche un'autentificazione di messaggi veloce. Per tali applicazioni Seal può essere combinato con un codice software ottimizzato all'autentificazione messaggi (MAC). Un software efficiente MAC è soggetto a studi separati da questo contesto. Un cifrario molto simile a SEAL può essere dato per architetture a 64 bit. Le tavole dovranno essere due volte più ampie e la routine “Inizializza” leggermente variata. Come solito quando si introduce un nuovo sistema crittografico, sono ben accolti possibili attacchi.

5.6 Appendice

Esempio: In questo esempio viene mostrato il funzionamento globale del SEAL una volta dati in input una chiave a di 160 bit e il numero L di bit di output desiderati. Supponiamo che la chiave a sia

a=67452301 efcdab89 98badcfe 10325476 c3d2e1f0

e che siano desiderati 4Kbyte come output (L=32768 bit). Il primo passo compiuto dal cifrario è la costruzione delle tavole T, R e S. Avendo come input la chiave prima indicata le tavole saranno così costruite:

$R[0], R[1], ...R[15]$: (in quanto 4⌈32768-1/8192⌉=4⌈3.9⌉=4×4=16)

e55c4955	59a40fb1	495a4508	850454f4	5b0200e9	8594d576	dc10d31e	1ecdb16b
e699bcf5	e2d35068	300f8bdc	bd39f67c	c06491dc	119fd703	2db7fdf7	2fee5c42

$T[0], T[1], ..., T[511]$:

92b404e5	56588ced	6c1acd4e	bf053f68	09f73a93	e786e543	b184176b	081df5cc
da3a23e0	a2a758ea	e85e0feb	078b776d	a13956cb	9e7c092c	6970262b	d1472567
....	....	....	....	....	....	....	....
....	....	....	....	....	....	....	....
....	....	....	....	....	....	....	....
....	....	....	....	....	....	....	....
ece03c22	82031f4d	c09f5ed8	88e489f0	61c376e7	2056695a	814a313d	e48f7e71

$S[0], S[1], ...S[255]$:

68318e58	0781c371	8bf691be	8afd6893	1a421779	bdfec86e	8382b844	da3ecc73
bc20de5c	5755b8fd	a82fe7da	e2c59ebc	3c43fbd0	db005522	64ee8a5e	97553175
....	....	....	....	....	....	....	....
....	....	....	....	....	....	....	....
....	....	....	....	....	....	....	....
f1ab9dab	72c8d7c9	57222650	e41fc10a	473af4e3	6364b447	c173f302	8cbe7fbf

Supponendo che il messaggio sia n=013577af la stringa $y=\mathrm{SEAL}_a(n)$ sarà costituita da 1024 word. Alla fine del processing verrà effettuato uno XOR di tutte le word computate ottenendo così un'unica stringa Y di 32 bit, tale stringa rappresenta il testo cifrato finale. Se dalla stringa Y si volesse risalire al testo in chiaro basterà computare $\mathrm{SEAL}_a(Y)$.
Nella pagina successiva viene dato il risultato della computazione di $\mathrm{SEAL}_a(n)$.

$\mathrm{SEAL}_a(n)=y[0], y[1], ..., y[1023]$:

9c3cf4eb	56ec2dd2	cd663283	eaf8b9f7	2ab4fd07	f5bc9ebb	11429760	6555325a
e3adc16f	caf41b01	972186cf	07e119c3	35050126	c9c6a240	15f90e07	16611570
obd13fc5	b0bbb36d	7f66916f	c92ed8c2	a75d7b65	2c2e92f8	25c55f8d	c8260443
50eaee36	f57610ee	8bf691be	8afd6893	1a421779	bdfec86e	8382b844	3c39a962
da3a23e0	a2a758ea	e85e0feb	078b776d	a13956cb	9e7c092c	6970262b	d1472567
....	....	....	....	....	....	....	....
....	....	....	....	....	....	....	....
....	....	....	....	....	....	....	....
....	....	....	....	....	....	....	....
....	....	....	....	....	....	....	....
....	....	....	....	....	....	....	....
....	....	....	....	....	....	....	....
....	....	....	....	....	....	....	....
66f5822c	1a7d26b2	32227ef7	2b5cba59	e8d8e3b8	ce9a0bab	075aea70	9af456ed

d2f91c98	f7573abd	c90e510f	8bbede45	4b3db413	eb90f47b	d6c05db1	8c437e10
ac58339a	69c8a201	089c412e	9c5ea298	c1a4bdf4	3a408a0	cb1d7f46	93c3df3d
....	....	....	....	....	....	....	....
da9f4c73	c0b7e6f5	aeb46849	5dd72e83	b5770d75	e18c7336	f620ef9c	b2d84735

La stringa Y risultante dallo XOR di queste 1024 word è Y=e421eec0.

Riferimenti

[1] E. Biham and A. Shamir, *Differential Criptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.

[2] T.W. Cusick and M.C. Wood, *The REDOC-II Cryptosystem*, Advances in Cryptology CRYPTO '90 Proceedings, Springer-Verlag, 1991, pp. 545-563

[3] J. Deamen, R. Govaerts and J. Vandewalle, *Block Ciphers Based on Modular Arithmetic* Proceedings of the 3^{rd} Symposium on State and Progress of Research in Cryptography, Rome, Italy, 15-16 Feb 1993, pp. 80-89.

[4] J.H. Evertse, *Linear Structures in Blockciphers*, Advances in Cryptology EUROCRPYT '87 , Springer-Verlag, 1988, pp. 249-266.

[5] H. Feistel, *Cryptography and Computer Privacy*, Scientific American, v. 228, n. 5, May '73, pp. 15-23.

[6] GOST 28147-89, *Cryptographic Protection for Data Processing Systems*, *Cryptographic Transformation Algorithm*, Government Standard of the U.S.S.R., Inv. No. 3583, UDC 681.325.6.:006.354. (in Russian)

[7] X. Lai, J. Massey and S. Murphy, *Markov Ciphers and differential Cryptanalysis*, Advances in Cryptology EUROCRPYT '91 Processing, Springer-Verlag, 1991, pp. 17-38.

[8] J.L. Massey and X. Lai, *Device for Concerting a Digital Block and the Use Thereof*, International Patent PCT/CH91/00117, 16 May 1991.

[9] J.L. Massey and X. Lai, *Device for The Conversion of a Digital Block and Use of Same*, U.S. Patent 5,214,703, 25 May 1993.

[10] M. Matsui, *Linear Criptanalysis Method for DES Cipher*, Advances in Cryptology CRYPTO '93 Proceedings, Springer-Verlag, 1994.

[11] R.C. Merkle, *Fast Software Encription Functions*, Advances in Cryptology CRYPTO '90 Proceedings, Springer-Verlag, 1991, pp. 476-501.

[12] R.C. Merkle, *Method and Apparatus for Data Encription*, U.S. Patent 5,003,597, 26 Mar 1991.

[13] S. Miyaguchi, *The FEAL-8 Cryptosystem and Call for Attack*, Advances in Cryptology CRYPTO '89 Proceedings, Springer-Verlag, 1990, pp. 624-627.

[14] S. Miyaguchi, *Expansion of the FEAL Cipher*, NNT Review, v. 2, n. 6, Nov 1990.

[15] S. Miyaguchi, *The FEAL Cipher Family*, Advances in Cryptology CRYPTO '90 Proceedings, Springer-Verlag, 1991, pp. 627-638.

[16] National Bureau of Standards, *Data Encryption Standard*, U.S. Department of Commerce, FIPS Publication 46, Jan 1977.

[17] National Istitute of Standards and Technology, *Clipper Chip Technology*, 30 Apr 1993.

[18] RSA Laboratories, *Answers to Frequently Asked Questions About Today's Cryptography*, Revision 2.0, Rsa Data Security Inc., 5 Oct 1993.

[19] B. Schneier, *Data Guardians*, MacWorld, Feb 1993, 145-151.

[20] B. Schneier, *Applied Cryptography*, John Wiley and Sons, New York, 1994.

[21] J.L. Smith, *The Design of Lucifer, A Cryptographic Device for Data Communication*, RC 3326, White Plains: IBM Research.

[22] M.J. Weiner, *Efficient DES Key Search*, Advances in Cryptology CRYPTO '93 Proceedings, Springer-Verlag, in preparation.

[23] M.C. Wood, *Method of Cryptographically Transforming Electronic Digital Data from One Form to Another*, U.S. Patent 5,003,596, 26 Mar 1991.

[24] J.L. Massey, *SAFER K-64: A Byte-Oriented Block Ciphering Algorithm*, pp. 1-17 in *Fast Software Encryption* (Ed. R. Anderson), Proceedings of the Cambridge Security Workshop, Cambridge, U.K., Dec. 9-11, 1993, Lecture Notes in Computer Science No. 809, Heidelberg and New York: Springer, 1994.

[25] J.L. Massey, *SAFER K-64: One Year Later* pp. 212-241, presented at the K.U. Leuven Workshop on Algorithms, Leuven, Belgium, 14-16 December 1994, and to appear in *Fast Software Encryption* (Ed. B. Preneel), to be published by Springer-Verlag in the Lecture Notes in Computer Science Series, 1995.

[26] L.R. Knudsen, *A Weakness in SAFER K-64*, manuscript submitted to CRYPTO '95, Feb. 16, 1995.

[27] L.R. Knudsen, *A Key-Schedule Weakness in SAFER K-64*, presented at Crypto '96 in Santa Barbara, California.

[28] C.E. Shannon, *Communication Theory of Secrecy Systems*, *Bell System Tech. J.*, vol.28, pp. 656-715, Oct. 1949.

[29] B. Schneier, *Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)*, pp. 191-204 in *Fast Software Encryption* (Ed. R. Anderson), Proceedings of the Cambridge Security Workshop, Cambridge, U.K., Dec. 9-11, 1993, Lecture Notes in Computer Science No.809. Heidelberg and New York: Springer, 1994.

[30] O. Goldreich, S. Goldwasser and S. Micali, *How to Construct Random Functions*, in the Journal of the ACM, Vol. 33, No. 4, pp. 210-217, 1986.

[31] R. Merkle, *Fast Software Encryption Functions* in Proceedings of CRYPTO '91, Springer-Verlag, Also U.S. Patent 5,003,597, March 1991.

[32] National Istitute of Standards, FIPS Publication 180, *Secure Hash Standard*, May 1993.

[33] RSA Data Security Inc., *Promotional Materials and Personal Communications*, Redwood City, California.

[34] A. Shimizu and S. Miyaguchi, *Fast Data Encryption Algorithm FEAL*, Eurocript '87.

[35] D.J. Wheeler and R.M. Needham, *TEA, a Tiny Encryption Algorithm* pp.363-366, presented at the K.U. Leuven Workshop on Algorithms, Leuven, Belgium, 14-16 December, 1994, and to appear in *Fast Software Encryption* (Ed. B. Preneel), to be published by Springer-Verlag in the Lecture Notes in Computer Science Series, 1995.

[36] P. Rogaway and D. Coppersmith, *A Software-Optimized Encryption Algorithm*, pp. 56-63 in *Fast Software Encryption* (Ed. R. Anderson), Proceedings of the Cambridge Security Workshop, Cambridge, U.K., Dec. 9-11, 1993, Lecture Notes in Computer Science No. 809, Heidelberg and New York: Springer, 1994.