Capitolo 8

Funzioni hash

8.1 Funzioni hash

Una componente fondamentale di molti algoritmi crittografici è data dalle funzioni hash. Le funzioni hash che soddisfano certe proprietà di non invertibilità, possono essere usate per rendere molti algoritmi più efficienti. In questo capitolo, verranno presentate le principali proprietà delle funzioni hash e verranno discussi gli attacchi contro di esse. Verrà anche discusso brevemente il modello dell'oracolo casuale, un metodo di analisi della sicurezza degli algoritmi che usano le funzioni hash. Più avanti, nel Capitolo 9, si mostrerà l'uso delle funzioni hash nella firma digitale degli algoritmi e, nel Capitolo 10, il ruolo che esse giocano nei protocolli di sicurezza e in molte altre situazioni.
Una **funzione funzione hash crittografica** h ha come input un messaggio di lunghezza arbitraria e produce come output un **digest di messaggio** (*message digest*) di lunghezza fissata (per esempio di 160 bit come nella Figura 8.1). Inoltre devono essere soddisfatte le seguenti proprietà.

1. Dato un messaggio m, il digest di messaggio $h(m)$ può essere calcolato molto rapidamente.

2. Dato y, è computazionalmente intrattabile trovare un m' tale che $h(m') = y$. In altre parole, h è una **funzione unidirezionale** (*one-way function*) o **resistente alle controimmagini** (*preimage resistant*). Si noti che se y è il digest di un qualche messaggio, non si sta cercando di trovare questo messaggio. Si sta solo cercando un qualche m' tale che $h(m') = y$.

3. È computazionalmente intrattabile trovare due messaggi m_1 e m_2 tali che $h(m_1) = h(m_2)$ (in questo caso, la funzione h è detta **fortemente resistente alle collisioni** (*strongly collision-free*)).

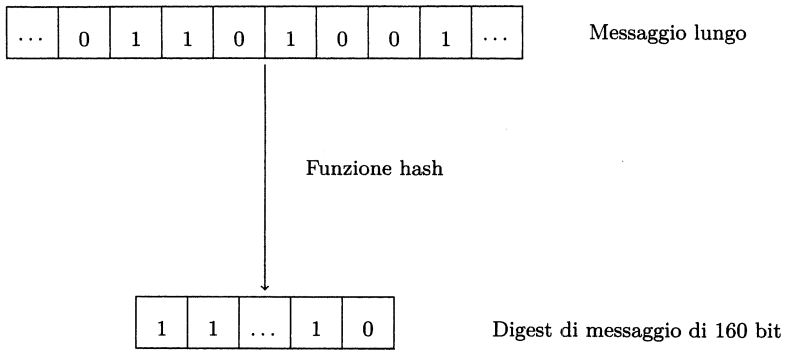

Figura 8.1 Una funzione hash.

- Poiché l'insieme dei possibili messaggi è molto più grande dell'insieme dei possibili messaggi digest, dovrebbero esserci sempre molti esempi di messaggi m_1 e m_2 con $h(m_1) = h(m_2)$. La richiesta (3) dice che dovrebbe essere difficile trovare questi esempi. In particolare, se Bob genera un messaggio m e il suo hash $h(m)$, Alice vuole essere ragionevolmente sicura che Bob non conosca un altro messaggio m' con $h(m') = h(m)$, neanche se m ed m' fossero due stringhe casuali prive di senso.
- A volte, nella pratica, si può indebolire (3) richiedendo che H sia **debolmente resistente alle collisioni** (*weakly collision-free*). Questo significa che, dato x, è computazionalmente intrattabile trovare un $x' \neq x$ con $H(x') = H(x)$. Questa proprietà è anche chiamata **resistenza alla seconda controimmagine** (*second preimage resistance*).

La richiesta (3) è la più difficile da soddisfare. Infatti, nel 2004, Wang, Feng, Lai e Yu (si veda [Wang et al.]) hanno trovato molti esempi di collisioni per le popolari funzioni hash MD4, MD5, HAVAL-128 e RIPEMD. Le collisioni di MD5 sono state usate da Ondrej Mikle per produrre due documenti differenti e dotati di senso con lo stesso hash, mentre l'articolo [Lenstra et al.] mostra come produrre esempi di certificati X.509 (si veda il Paragrafo 10.5) con lo stesso hash MD5 (si veda anche l'Esercizio 11). Questo significa che una firma digitale valida (si veda il Capitolo 9) su un certificato, è valida anche per l'altro certificato, rendendo impossibile la determinazione del certificato che è stato legittimamente firmato da un'Autorità di Certificazione. Inoltre, nel 2005, Wang, Yin e Yu [Wang et al. 2] predissero che si sarebbero potute trovare delle collisioni per la funzione hash SHA-1 con circa 2^{69} calcoli, che è molto meglio dei 2^{80} calcoli richiesti dall'attacco del compleanno (si veda il Paragrafo 8.4). Trovarono anche delle collisioni in una versione ridotta di SHA-1 con soli 60 round. Queste debolezze sono motivo di preoccupazione nell'uso di questi algoritmi hash e stanno portando alla ricerca di qualcosa che possa sostituirli.

Le funzioni hash vengono usate principalmente nelle firme digitali. Poiché la lunghezza di una firma digitale è spesso lunga almeno quanto il documento che deve essere firmato, è molto più efficiente firmare l'hash di un documento piuttosto che l'intero documento. Tutto questo verrà discusso nel Capitolo 9.

Le funzioni hash possono anche essere utilizzate per controllare l'integrità dei dati. Il problema dell'integrità dei dati si presenta sostanzialmente in due situazioni. La prima si verifica quando i dati (cifrati o meno) vengono trasmessi a un'altra persona su un canale di comunicazione disturbato che introduce errori nei dati. La seconda si verifica quando un osservatore modifica in qualche modo la trasmissione prima che questa arrivi al ricevente. In entrambi i casi, i dati sono stati alterati.

Per esempio, Alice potrebbe inviare a Bob lunghi messaggi relativi a transazioni finanziarie con Eva, cifrandoli in blocchi. Se Eva riesce a dedurre che il decimo blocco di ogni messaggio indica la somma di denaro che deve essere depositata sul suo conto, allora può facilmente sostituire il decimo blocco da un messaggio a un altro e aumentare il deposito.

Oppure Alice potrebbe inviare a Bob un messaggio formato da vari blocchi di dati e uno di questi blocchi potrebbe perdersi durante la trasmissione. In questo caso, Bob potrebbe non accorgersi nemmeno che manca un blocco.

Ecco il modo in cui possono essere usate le funzioni hash. Supponiamo che si invii $(m,h(m))$ attraverso il canale di comunicazione e che si riceva (M,H). Per controllare se si sono verificati degli errori, il ricevente calcola $h(M)$ e controlla se è uguale a H. Se si verifica qualche errore, è probabile che $h(M) \neq H$, per la proprietà di resistenza alle collisioni di h.

Esempio. Sia n un intero grande. Sia $h(m) = m \pmod n$, considerato come un intero tra 0 e $n-1$. Questa funzione chiaramente soddisfa (1), ma non (2) e (3). Infatti, dato y e posto $m = y$, si ha $h(m) = y$. Così h non è unidirezionale. Analogamente, comunque scelti due m_1 e m_2 congruenti modulo n, si ha $h(m_1) = h(m_2)$. Quindi h non è fortemente resistente alle collisioni. ■

Esempio. La funzione hash che presenteremo in questo esempio, a volte chiamata funzione hash logaritmo discreto, è dovuta a Chaum, van Heijst e Pfitzmann [Chaum et al.]. Essa soddisfa le richieste (2) e (3), ma è troppo lenta per essere usata in pratica. Nonostante questo, è utile per illustrare le idee di fondo delle funzioni hash.

Come prima cosa si sceglie un primo grande p tale che anche $q = (p-1)/2$ sia primo (si veda l'Esercizio 7 del Capitolo 9). Poi si scelgono due radici primitive α e β per p. Poiché α è una radice primitiva, esiste un a tale che $\alpha^a \equiv \beta \pmod p$. Qui però assumeremo che a non sia noto (trovare a, quando non è dato, significa risolvere un problema di logaritmo discreto, che è ritenuto difficile).

La funzione hash h manda gli interi modulo q^2 in interi modulo p. Quindi il digest di messaggio contiene circa la metà dei bit del messaggio. Non si tratta di una riduzione così drastica della grandezza del messaggio come è normalmente richiesto in pratica, ma è sufficiente per i nostri propositi.

Scritto $m = x_0 + x_1 q$ con $0 \leq x_0, x_1 \leq q-1$, si definisce

$$h(m) \equiv \alpha^{x_0} \beta^{x_1} \pmod p.$$

La seguente proposizione mostra che la funzione h è probabilmente fortemente resistente alle collisioni.

Proposizione. *Se si conoscono due messaggi $m \neq m'$ con $h(m) = h(m')$, allora si può determinare il logaritmo discreto $a = L_\alpha(\beta)$.*

Dimostrazione. Siano $m \neq m'$ due messaggi con $h(m) = h(m')$. Se $m = x_0 + x_1 q$ e $m' = x_0' + x_1' q$, allora

$$\alpha^{x_0}\beta^{x_1} \equiv \alpha^{x_0'}\beta^{x_1'} \pmod{p}.$$

Poiché $\beta \equiv \alpha^a \pmod{p}$, questa congruenza può essere riscritta come

$$\alpha^{a(x_1 - x_1') - (x_0' - x_0)} \equiv 1 \pmod{p}.$$

Essendo α una radice primitiva modulo p, si ha che $\alpha^k \equiv 1 \pmod{p}$ se e solo se $k \equiv 0 \pmod{p-1}$. Nel nostro caso, questo significa che

$$a(x_1 - x_1') \equiv x_0' - x_0 \pmod{p-1}.$$

Se $d = \text{MCD}(x_1 - x_1', p-1)$, allora la congruenza precedente possiede esattamente d soluzioni (si veda il Paragrafo 3.3), che possono essere trovate rapidamente. Per la scelta di p, i soli fattori di $p-1$ sono 1, 2, q, $p-1$. Poiché $0 \leq x_1, x_1' \leq q-1$, si ha che $-(q-1) \leq x_1 - x_1' \leq q-1$. Quindi, se $x_1 - x_1' \neq 0$, allora si ha un multiplo non nullo di d di valore assoluto minore di q. Questo significa che $d \neq q, p-1$, cosicché $d = 1$ o 2. Quindi, ci sono al più due possibilità per a. Poiché, calcolando α^a per ogni possibilità, solo una di esse dà β, si ottiene a.
D'altra parte, se $x_1 - x_1' = 0$, allora deve essere $x_0' - x_0 \equiv 0 \pmod{p-1}$. Poiché $-(q-1) \leq x_0' - x_0 \leq q-1$, si deve avere $x_0' = x_0$. Quindi $m = m'$, contro le ipotesi. □

Ora è facile mostrare che h è resistente alle controimmagini. Supponiamo di avere un algoritmo g che parte da un digest di messaggio y e trova rapidamente un m con $h(m) = y$. In questo caso, si trovano facilmente $m_1 \neq m_2$ con $h(m_1) = h(m_2)$. Si sceglie a caso m, si calcola $y = h(m)$ e poi $g(y)$. Poiché h trasforma q^2 messaggi in $p-1 = 2q$ digest, ci sono molti messaggi m' con $h(m') = h(m)$. Pertanto la probabilità che $m' = m$ è molto bassa. Nel caso in cui $m' = m$, si prova con un altro m scelto a caso. Con pochi tentativi si dovrebbe trovare una collisione, cioé due messaggi $m_1 \neq m_2$ con $h(m_1) = h(m_2)$. La proposizione precedente dice che allora si può risolvere un problema di logaritmo discreto. Di conseguenza è poco probabile che tale algoritmo g esista.

8.2 Un semplice esempio di hash

Esistono molte famiglie di funzioni hash. La funzione hash logaritmo discreto descritta poco fa è troppo lenta per essere utile in pratica. Un motivo risiede nel fatto che utilizza l'elevamento a potenza modulare, che rende il suo costo computazionale quasi uguali a quelle di RSA o ElGamal. Anche se l'elevamento a potenza modulare è veloce, non è sufficientemente veloce per gli enormi input che vengono usati in certe situazioni. Le funzioni hash descritte in questo e nel prossimo paragrafo coinvolgono solo operazioni elementari sui bit e quindi possono essere eseguite molto più rapidamente delle procedure come l'elevamento a potenza modulare.

Verrà ora descritta l'idea di fondo di molte funzioni hash crittografiche attraverso una semplice funzione hash che possiede molte delle proprietà fondamentali delle funzioni hash usate in pratica. Questa funzione hash non ha robustezza di livello industriale e non dovrebbe mai essere usata in alcun sistema.
Se si parte con un messaggio m di lunghezza L arbitraria, allora si può spezzare m in blocchi di n bit, dove n è molto più piccolo di L. Indicando con m_j questi blocchi di n bit, si ha $m = [m_1, m_2, \cdots, m_l]$, dove $l = \lceil L/n \rceil$ e l'ultimo blocco m_l è riempito con degli zeri, in modo da avere n bit.
Ora si può considerare la matrice formata dai blocchi

$$m_j = [m_{j1}, m_{j2}, m_{j3}, \ldots, m_{jn}]$$

scritti come vettori riga, dove ogni m_{ji} è un bit. L'hash $h(m)$ è formato da n bit ed è definito in modo che l'i-esimo bit sia la somma XOR degli elementi della i-esima colonna di questa matrice, ossia $h_i = m_{1i} \oplus m_{2i} \oplus \cdots \oplus m_{li}$. Tutto questo può essere visualizzato nel modo seguente

$$\begin{bmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{l1} & m_{l2} & \cdots & m_{ln} \end{bmatrix}$$
$$\Downarrow \quad \Downarrow \quad \Downarrow \quad \Downarrow$$
$$\oplus \quad \oplus \quad \oplus \quad \oplus$$
$$\Downarrow \quad \Downarrow \quad \Downarrow \quad \Downarrow$$
$$[\ c_1 \quad c_2 \quad \cdots \quad c_n\ ] \ = \ h(m).$$

Questa funzione hash prende in input un messaggio di lunghezza arbitraria e produce in output un digest di messaggio di n bit. Tuttavia, non è considerata crittograficamente sicura, poiché è facile trovare due messaggi che hanno lo stesso valore hash (Esercizio 10).
Le funzioni hash crittografiche usate in pratica di solito utilizzano molte altre operazioni sui bit, in modo che sia più difficile trovare collisioni. Il Paragrafo 8.3 contiene vari esempi di queste operazioni.
Un'operazione usata spesso è la rotazione dei bit, come già si è visto in DES. L'operazione di rotazione a sinistra

$$m \hookleftarrow y$$

consiste, per definizione, nel far scorrere m a sinistra di y bit, in modo che gli y bit più a sinistra vengano portati negli y bit più a destra.
La semplice funzione hash definita qui sopra può essere modificata chiedendo che il blocco m_j sia ruotato a sinistra di $j-1$ posti, in modo da ottenere un nuovo blocco $m_j' = m_j \hookleftarrow j-1$. Si può poi considerare la matrice formata dai vettori riga m_j' e definire una nuova semplice funzione hash sommando XOR gli elementi di ogni

colonna:

$$\begin{bmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{22} & m_{23} & \cdots & m_{21} \\ m_{33} & m_{34} & \cdots & m_{32} \\ \vdots & \vdots & \ddots & \vdots \\ m_{ll} & m_{l,l+1} & \cdots & m_{l,l-1} \end{bmatrix}$$
$$\begin{matrix} \Downarrow & \Downarrow & \Downarrow & \Downarrow \\ \oplus & \oplus & \oplus & \oplus \\ \Downarrow & \Downarrow & \Downarrow & \Downarrow \end{matrix}$$
$$\begin{bmatrix} c_1 & c_2 & \cdots & c_n \end{bmatrix} = h(m).$$

Questa nuova funzione hash che coinvolge le rotazioni rende un po' più difficile trovare delle collisioni, anche se non lo rende impossibile (Esercizio 10). Nella costruzione di una funzione hash crittografica non bastano le rotazioni, ma occorrono molti altri stratagemmi. Nel prossimo paragrafo, verrà descritto un esempio di funzione hash usata in pratica che sfrutta le tecniche di questo paragrafo e molti altri modi di rimescolare i bit.

8.3 Algoritmo SHA

Vediamo ora cosa comporta la costruzione di una funzione hash crittografica reale. A differenza dei cifrari a blocchi, dove ci sono molti cifrari a blocchi tra cui scegliere, si hanno solo poche funzioni hash disponibili. Le più importanti sono l'algoritmo SHA (*Secure Hash Algorithm*, SHA-1), la famiglia MD (*Message Digest*) e l'algoritmo RIPEMD-160.

La famiglia MD ha una storia interessante. L'algoritmo MD originale non fu mai pubblicato. Il primo algoritmo MD a essere pubblicato fu MD2, seguito da MD4 e da MD5. In MD2 e in MD4 furono trovate delle debolezze e Ron Rivest propose MD5 come un miglioramento di MD4. Ma dopo aver trovato delle collisioni per MD5, anche la sua robustezza è meno certa.

Per questo motivo, si è scelto di discutere SHA-1 al posto della famiglia MD. La discussione che segue è piuttosto tecnica e ha solo lo scopo di dare un'idea di ciò che accade all'interno di una funzione hash.

L'algoritmo SHA fu sviluppato dalla National Security Agency (NSA) e dato al National Institute of Standards and Technology (NIST). La versione originale, spesso indicata con SHA-0, fu pubblicata nel 1993 come Federal Information Processing Standard (FIPS 180). SHA-0 conteneva una debolezza che fu in seguito scoperta dalla NSA e che portò a un documento di revisione dello standard (FIPS 180-1), rilasciato nel 1995, che descrive la versione migliorata SHA-1. Quest'ultimo è ora l'algoritmo hash raccomandato dal NIST.[1]

[1] Nel 2001, il NIST ha pubblicato quattro funzioni hash facenti parte della famiglia SHA, caratterizzate da un digest più lungo. Queste varianti sono note come SHA-224, SHA-256, SHA-348 e SHA-512. In modo simile a quanto avvenuto per AES, nel 2007 il NIST ha sollecitato proposte per il nuovo algoritmo hash che assumerà il nome di SHA-3. La valutazione dei 63 candidati dovrebbe concludersi nel 2012 con la proclamazione del vincitore. (*N.d.Rev.*)

SHA-1 produce un hash di 160 bit e si basa sulle stesse idee alla base di MD4 e MD5. Queste funzioni hash usano una procedura iterativa. Come in precedenza, il messaggio originale m viene spezzato in un insieme di blocchi di lunghezza fissata: $m = [m_1, m_2, \cdots, m_l]$. Se necessario, al messaggio vengono concatenati alcuni bit di riempimento, in modo da riempire interamente anche l'ultimo blocco. I blocchi del messaggio vengono poi elaborati mediante una successione di round che usano una funzione di compressione h', che combina il blocco corrente con il risultato ottenuto nel round precedente. In altre parole, si parte con un valore iniziale X_0 e si definisce $X_j = h'(X_{j-1}, m_j)$. L'X_l finale è il digest di messaggio.

Il trucco per costruire una funzione hash è definire una buona funzione di compressione. Questa funzione di compressione dovrebbe essere costruita in modo che ogni bit di input influenzi il maggior numero di bit di output. Una delle principali differenze tra SHA-1 e la famiglia MD è che per SHA-1 i bit di input sono usati più spesso durante l'esecuzione della funzione hash che per MD4 o MD5. Questo approccio più conservativo rende l'architettura di SHA-1 più sicura sia di MD4 sia di MD5, anche se la rende un po' più lenta.

SHA-1 inizia prendendo il messaggio originale e completandolo con un bit 1 seguito da una successione di bit 0. I bit 0 vengono aggiunti in modo che il nuovo messaggio abbia una lunghezza pari a un multiplo intero di 512 bit, meno 64 bit. Dopo la concatenazione degli 1 e degli 0, viene concatenata la rappresentazione di 64 bit della lunghezza T del messaggio. Così, se il messaggio è di T bit, allora la concatenazione genera un messaggio formato da $L = \lfloor T/512 \rfloor + 1$ blocchi di 512 bit. Il messaggio concatenato viene spezzato in L blocchi m_1, m_2, ..., m_L. L'algoritmo hash prende in input questi blocchi uno a uno.

Per esempio, se il messaggio originale è di 2800 bit, si aggiungono un "1" e 207 "0", in modo da ottenere un nuovo messaggio di lunghezza $3008 = 6 \cdot 512 - 64$. Poiché $2800 = 101011110000_2$ in binario, si concatenano cinquantadue 0 seguiti da 101011110000 in modo da ottenere un messaggio di lunghezza 3072. Questo viene spezzato in sei blocchi di lunghezza 512.

Prima di descrivere l'algoritmo hash, occorre definire le seguenti operazioni su stringhe di 32 bit.

1. $X \wedge Y$ = AND logico bit a bit, ossia moltiplicazione modulo 2 bit a bit, o minimo bit a bit.
2. $X \vee Y$ = OR logico bit a bit, ossia massimo bit a bit.
3. $X \oplus Y$ = somma modulo 2 bit a bit.
4. $\neg X$ cambia gli 1 in 0 e gli 0 in 1.
5. $X + Y$ = somma di X e Y modulo 2^{32}, dove X e Y sono considerati come interi modulo 2^{32}.
6. $X \hookleftarrow r$ = scorrimento di X a sinistra di r posizioni (dove la parte iniziale diventa la parte finale).

Occorrono anche le funzioni

$$f_t(B,C,D) = \begin{cases} (B \wedge C) \vee ((\neg B) \wedge D) & \text{se } 0 \le t \le 19 \\ B \oplus C \oplus D & \text{se } 20 \le t \le 39 \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & \text{se } 40 \le t \le 59 \\ B \oplus C \oplus D & \text{se } 60 \le t \le 79 \end{cases}$$

e le costanti $K_0, \ldots, K_{79}$ definite ponendo

$$K_t = \begin{cases} \texttt{5A827999} & \text{se } 0 \le t \le 19 \\ \texttt{6ED9EBA1} & \text{se } 20 \le t \le 39 \\ \texttt{8F1BBCDC} & \text{se } 40 \le t \le 59 \\ \texttt{CA62C1D6} & \text{se } 60 \le t \le 79. \end{cases}$$

I valori di queste costanti sono scritti in *notazione esadecimale*. Ogni cifra o lettera rappresenta una stringa di 4 bit:

$$0 = 0000, \quad 1 = 0001, \quad 2 = 0010, \quad \ldots, \quad 9 = 1001,$$

$$A = 1010, \quad B = 1011, \quad \ldots, \quad F = 1111.$$

Per esempio, `BA1` è uguale a $11 \cdot 16^2 + 10 \cdot 16^1 + 1 = 2977$.
L'algoritmo SHA-1 è riassunto nella tabella della prossima pagina. Il cuore dell'algoritmo è il punto (3), che viene presentato nella Figura 8.2. Tutte le operazioni coinvolte nell'algoritmo SHA-1 sono elementari e molto veloci. Si osservi che la procedura fondamentale è iterata il numero di volte necessario per riassumere l'intero messaggio. Questa procedura iterativa rende l'algoritmo molto efficiente in termini di lettura ed elaborazione del messaggio.
Esaminiamo ora l'algoritmo passo a passo. SHA-1 parte generando un registro iniziale X_0 di 160 bit formato da cinque sottoregistri H_0, H_1, H_2, H_3, H_4 di 32 bit. Questi sottoregistri sono inizializzati come segue:

$$\begin{aligned} H_0 &= 67452301 \\ H_1 &= EFCDAB89 \\ H_2 &= 98BADCFE \\ H_3 &= 10325476 \\ H_4 &= C3D2E1F0. \end{aligned}$$

Dopo che viene elaborato il blocco m_j del messaggio, il registro X_j viene aggiornato per generare un registro X_{j+1}.
SHA-1 ripete le operazioni su ognuno dei blocchi di messaggio m_j da 512 bit. Per ogni blocco m_j, il registro X_j viene copiato nei sottoregistri A, B, C, D, E. Il primo blocco di messaggio, m_0, viene tagliato e rimescolato per ottenere $W_0, \ldots, W_{79}$. Questi sono dati in pasto a una sequenza di quattro round, corrispondenti ai quattro intervalli $0 \le t \le 19$, $20 \le t \le 39$, $40 \le t \le 59$ e $60 \le t \le 79$. Ogni round prende come input il valore corrente del registro X_0 e dei blocchi W_t per quell'intervallo e opera su di

L'algoritmo SHA-1

1. Partire con un messaggio m. Concatenare dei bit, come specificato nel testo, in modo da ottenere un messaggio y della forma $y = m_1 \| m_2 \| \cdots \| m_L$, dove ogni m_i è formato da 512 bit.
2. Inizializzare $H_0 = 67452301$, $H_1 = EFCDAB89$, $H_2 = 98BADCFE$, $H_3 = 10325476$, $H_4 = C3D2E1F0$.
3. Per $i = 0, \ldots, L-1$, eseguire le seguenti istruzioni.
   (a) Scrivere $m_i = W_0 \| W_1 \| \cdots \| W_{15}$, dove ogni W_j è formato da 32 bit.
   (b) Per $t = 16, \ldots, 79$, porre
   $$W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \hookleftarrow 1$$
   (c) Porre $A = H_0$, $B = H_1$, $C = H_2$, $D = H_3$, $E = H_4$.
   (d) Per $t = 0, \ldots, 79$, eseguire i seguenti passi in successione:
   $$T = (A \hookleftarrow 5) + f_t(B,C,D) + E + W_t + K_t,$$
   $$E = D, \quad D = C, \quad C = (B \hookleftarrow 30), \quad B = A, \quad A = T.$$
   (e) Porre $H_0 = H_0 + A$, $H_1 = H_1 + B$, $H_2 = H_2 + C$, $H_3 = H_3 + D$, $H_4 = H_4 + E$.
4. Fornire in output $H_0 \| H_1 \| H_2 \| H_3 \| H_4$. Questo è il valore hash di 160 bit.

essi per 20 iterazioni (ossia il contatore t scorre lungo i 20 valori dell'intervallo). Ogni iterazione usa la costante K_t e l'operazione $f_t(B,C,D)$ del round (costante e operazione restano le medesime per tutte le iterazioni di quel round). Uno dopo l'altro, ogni round aggiorna (A,B,C,D,E). Terminato il quarto round, quando $t = 79$, i sottoregistri (A,B,C,D,E) di output sono sommati ai sottoregistri $(H_0, H_1, H_2, H_3, H_4)$ di input per produrre 160 bit di output che diventano il successivo registro X_1, che verrà copiato in (A, B, C, D, E) nell'elaborazione del successivo blocco di messaggio m_1. Il registro X_1 di output può essere considerato come l'output della funzione di compressione h' quando ha come input X_0 e m_0, ossia $X_1 = h'(X_0, m_0)$.
Si continua in questo modo per ognuno dei blocchi di messaggio m_j di 512 bit, usando il precedente registro X_j in output come input per calcolare il successivo registro X_{j+1} di output, ossia $X_{j+1} = h'(X_j, m_j)$. Nella Figura 8.2 è rappresentata l'operazione della funzione di compressione h' sul j-esimo blocco di messaggio, m_j, usando il registro

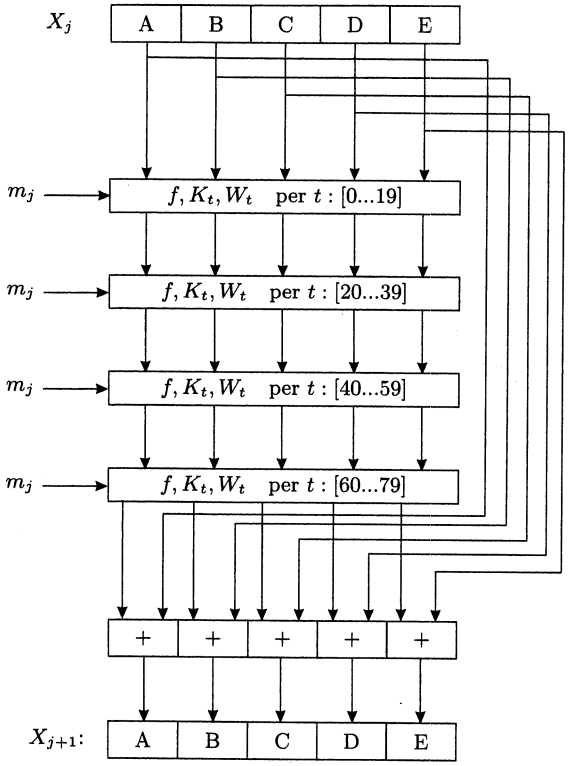

Figura 8.2 Le operazioni eseguite da SHA-1 su un singolo blocco di messaggio m_j.

X_j. Dopo aver completato tutti gli L blocchi di messaggio, l'output finale è il digest di messaggio di 160 bit.
Il blocco fondamentale dell'algoritmo è l'insieme di operazioni che hanno luogo sui sottoregistri nel passo (3d). Queste operazioni sono rappresentate nella Figura 8.3. Esse prendono i sottoregistri e operano su di essi usando rotazioni e somme XOR, analogamente al metodo descritto nel Paragrafo 8.2. Tuttavia, SHA-1 usa anche complicate operazioni di rimescolamento, che sono eseguite da f_t e dalle costanti K_t. Per maggiori dettagli su queste e altre funzioni hash e per la teoria coinvolta nella loro costruzione, si vedano [Stinson], [Schneier] e [Menezes et al.].

8.4 Attacchi del compleanno

Se in una stanza ci sono 23 persone, la probabilità che due di esse festeggino il compleanno nello stesso giorno è poco più del 50%. Se ce ne sono 30, la probabilità è circa del 70%. Questo fatto, che può sembrare sorprendente, è chiamato **paradosso**

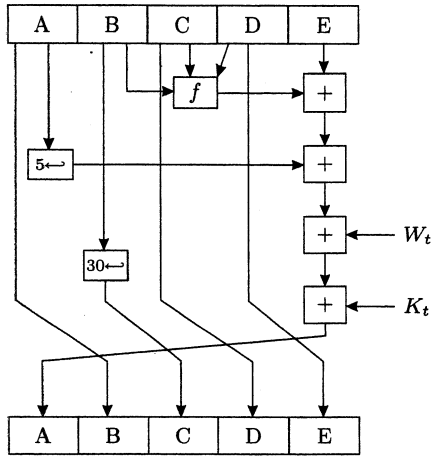

Figura 8.3 Le operazioni che hanno luogo su ognuno dei sottoregistri in SHA-1.

del compleanno. Tuttavia esso è vero e ora vedremo perché, ignorando gli anni bisestili e supponendo che i compleanni siano tutti ugualmente probabili (altrimenti, le probabilità sarebbero leggermente più alte).
Consideriamo il caso di 23 persone. Calcoleremo la probabilità che i compleanni cadano tutti in giorni diversi. Per fare questo, disponiamo le persone in fila. Poiché il compleanno della prima persona cadrà in un qualche giorno dell'anno, la probabilità che il compleanno della seconda persona cada in un giorno diverso è $(1 - 1/365)$. Se i compleanni delle prime due persone cadono in due giorni diversi, allora la probabilità che il compleanno della terza persona cada in un giorno diverso dai primi due è $(1 - 2/365)$. Quindi, la probabilità che il compleanno di tutte e tre le persone cada in giorni diversi è $(1 - 1/365)(1 - 2/365)$. Continuando in questo modo, si ha che la probabilità che i compleanni delle 23 le persone cadano tutte in giorni diversi risulta essere

$$\left(1 - \frac{1}{365}\right)\left(1 - \frac{2}{365}\right)\cdots\left(1 - \frac{22}{365}\right) = 0{,}493.$$

Quindi, la probabilità che almeno due compleanni cadano nello stesso giorno è

$$1 - 0{,}493 = 0{,}507.$$

Consideriamo, per esempio, il caso di 40 persone. Supporremo che tra le prime 30 persone non ce ne siano due con lo stesso compleanno, altrimenti avremmo già trovato la coppia cercata. Poiché 30 giorni di compleanno sono già stati scelti, per ciascuna delle altre 10 persone si ha circa il 10% di probabilità che un giorno di compleanno scelto a caso coincida con uno dei primi 30. Quindi non dovrebbe essere troppo sorprendente trovare una corrispondenza. Infatti, la probabilità che ci sia una corrispondenza tra 40 persone è dell'89%.

Più in generale, supponiamo che ci siano N oggetti, con N grande, e che ci siano r persone ognuna delle quali sceglie un oggetto (con ripetizioni, nel senso che più persone possono scegliere lo stesso oggetto). Allora

$$\text{Prob(esiste una corrispondenza)} \approx 1 - e^{-r^2/2N}.$$

Si tenga presente che questa è solo un'approssimazione che vale per N grande. Per n piccolo è meglio usare il prodotto dato sopra e ottenere una risposta esatta. La derivazione di questa approssimazione è data nell'Esercizio 5. Scegliendo $r^2/2N = \ln 2$, si trova che se $r \approx 1{,}177\sqrt{N}$, allora la probabilità che almeno due persone scelgano lo stesso oggetto è del 50%.

Per riassumere, se ci sono N possibilità e si ha una lista di lunghezza $\sqrt{N}$, allora c'è una buona probabilità di avere una corrispondenza. Se si vuole aumentare la probabilità di avere una corrispondenza, si può scegliere una lista di lunghezza $2\sqrt{N}$ o $5\sqrt{N}$. È sufficiente una lunghezza pari a una costante per $\sqrt{N}$ (invece di qualcosa come N). Se, per esempio, si hanno 40 targhe che finiscono tutte con un numero di 3 cifre, qual è la probabilità che due targhe finiscano con le stesse 3 cifre? Si ha $N = 1000$ (il numero dei possibili numeri di 3 cifre) e $r = 40$ (il numero di targhe in considerazione). Poiché

$$\frac{r^2}{2N} = 0{,}8,$$

la probabilità approssimata di avere una corrispondenza è

$$1 - e^{-0{,}8} = 0{,}551,$$

che è più del 50%. Questa è soltanto un'approssimazione. La risposta esatta si ottiene calcolando

$$1 - \left(1 - \frac{1}{1000}\right)\left(1 - \frac{2}{1000}\right)\cdots\left(1 - \frac{39}{1000}\right) = 0{,}546.$$

Ma qual è la probabilità che almeno una di queste 40 targhe abbia le stesse ultime 3 cifre uguali a una data targa? Ogni targa ha probabilità $1 - 1/1000$ di non coincidere con quella data. Quindi la probabilità che non ci sia corrispondenza con nessuna delle 40 targhe è $(1-1/1000)^{40} = 0{,}961$. La ragione per cui il paradosso del compleanno funziona è che non si sta semplicemente cercando una corrispondenza tra una targa fissata e le altre targhe. Si sta cercando una corrispondenza tra due qualsiasi targhe dell'insieme in considerazione e questo porta ad avere molte più possibilità di corrispondenze.

Le applicazioni di queste idee alla crittologia richiedono lo studio di un problema leggermente diverso. Consideriamo due stanze, ognuna delle quali con 30 persone. Qual è la probabilità che qualcuno che si trova nella prima stanza abbia lo stesso compleanno di qualcuno che si trova nella seconda stanza? Più in generale, se ci sono N oggetti e due gruppi di r persone, ognuna delle quali sceglie un oggetto (con ripetizioni), qual è la probabilità che una persona del primo gruppo scelga lo stesso oggetto scelto da una persona del secondo gruppo? Se $\lambda = r^2/N$, allora la probabilità che ci sia una corrispondenza è circa $1 - e^{-\lambda}$. La probabilità di esattamente i corrispondenze è circa $(\lambda^i e^{-\lambda}/i!)$. Un'analisi di questo problema, e di alcune sue generalizzazioni, si trova in [Girault et al.].

Ancora, se ci sono N possibilità e si hanno due liste di lunghezza $\sqrt{N}$, allora c'è una buona probabilità di avere una corrispondenza. Inoltre, se si vuole aumentare la possibilità di avere una corrispondenza, si possono considerare liste di lunghezza $2\sqrt{N}$ o $5\sqrt{N}$. È sufficiente una lunghezza pari a una costante per $\sqrt{N}$ (invece di qualcosa confrontabile con N).

Per esempio, se si sceglie $N = 365$ e $r = 30$, allora

$$\lambda = 30^2/365 = 2{,}466.$$

Poiché $1 - e^{-\lambda} = 0{,}915$, si ha circa il 91,5% di probabilità che qualcuno del primo gruppo di 30 persone abbia lo stesso compleanno di qualcuno del secondo gruppo di 30 persone.

- L'attacco del compleanno può essere usato per trovare collisioni per funzioni hash se l'output della funzione hash non è sufficientemente grande. Sia h una funzione hash con un output di n bit. Ci sono $N = 2^n$ output possibili. Si costruisca una lista $h(x)$ per circa $r = \sqrt{N} = 2^{n/2}$ scelte casuali di x. Allora ci si trova nella situazione di $r \approx \sqrt{N}$ "persone" con N possibili "compleanni" e quindi c'è una buona possibilità di avere due valori x_1 e x_2 con lo stesso valore hash. Se si considera una lista più lunga, formata per esempio da $r = 10 \cdot 2^{n/2}$ valori di x, la probabilità che ci sia una corrispondenza diventa molto alta.
- Analogamente, supponiamo di avere due insiemi S e T di input. Se si calcola $h(s)$ per circa $\sqrt{N}$ elementi $s \in S$ scelti a caso e $h(t)$ per circa $\sqrt{N}$ elementi $t \in T$ scelti a caso, allora ci si aspetta che qualche valore di $h(s)$ sia uguale a qualche valore di $h(t)$. Questa situazione si ritrova in un attacco agli schemi di firma nel Capitolo 9, dove S sarà un insieme di documenti validi e T sarà un insieme di documenti fraudolenti.

Se l'output della funzione hash è di circa $n = 60$ bit, i precedenti attacchi hanno un'alta probabilità di successo. È necessario costruire liste di lunghezza approssimativamente $2^{n/2} = 2^{30} \approx 10^9$ e immagazzinarle. Questo è possibile su quasi tutti i computer. Tuttavia, se la funzione hash produce output di 128 bit, allora le liste hanno una lunghezza di circa $2^{64} \approx 10^{19}$, che è troppo grande, sia in termini di tempo che di memoria.

8.4.1 Un attacco del compleanno al logaritmo discreto

Se p è un primo grande, si può valutare $L_\alpha(\beta)$, ossia si può risolvere $\alpha^x \equiv \beta \pmod p$, con probabilità alta mediante un attacco del compleanno.

Si costruiscono due liste, entrambe di lunghezza circa $\sqrt{p}$:

1. la prima lista contiene i numeri $\alpha^k \pmod p$ per circa $\sqrt{p}$ valori di k scelti a caso,
2. la seconda lista contiene i numeri $\beta\alpha^{-\ell} \pmod p$ per circa $\sqrt{p}$ valori di ℓ scelti a caso.

C'è una buona probabilità che esista una corrispondenza tra qualche elemento della prima lista e qualche elemento della seconda lista. Se è così, si ha

$$\alpha^k \equiv \beta\alpha^{-\ell}, \qquad \text{ossia} \quad \alpha^{k+\ell} \equiv \beta \pmod p.$$

Quindi, $x \equiv k + \ell \pmod{p-1}$ è il logaritmo discreto cercato.

Confrontiamo questo metodo con il metodo *Baby Step, Giant Step* (BSGS) descritto nel Paragrafo 7.2. Entrambi i metodi hanno un tempo di esecuzione e un'occupazione di memoria proporzionali a $\sqrt{p}$. Tuttavia, l'algoritmo BSGS è *deterministico*, ossia è garantito che produca una risposta, mentre l'algoritmo del compleanno è probabilistico, ossia è probabile (ma non garantito) che produca una risposta. Inoltre, l'algoritmo BSGS possiede un vantaggio computazionale. Il calcolo di un elemento di una lista da uno precedente richiede una moltiplicazione (per α o per α^{-N}). Nell'algoritmo del compleanno, l'esponente k è scelto a caso e quindi α^k deve essere calcolato ogni volta e questo rende l'algoritmo più lento. Quindi l'algoritmo BSGS è migliore rispetto al metodo del compleanno.

8.5 Multicollisioni

In questo paragrafo si mostrerà che la natura iterativa della maggior parte degli algoritmi hash li rende meno resistenti di quanto ci si possa aspettare alle multicollisioni, ossia input $x_1, \ldots, x_n$ tutti con lo stesso valore hash. Questo fu messo in evidenza da Joux [Joux], che diede anche alcune implicazioni riguardanti le proprietà delle funzioni hash concatenate, che verranno discusse più avanti.

Si supponga che ci siano r persone e N possibili compleanni. Si può dimostrare che, se $r \approx N^{(k-1)/k}$, allora esiste una buona probabilità che almeno k persone abbiano lo stesso compleanno. In altre parole, ci si aspetta una k-collisione. Se l'output di una funzione hash è casuale, allora ci si aspetta che questa stima valga per k-collisioni di valori della funzione hash. Ossia, se una funzione hash produce output di n bit, quindi $N = 2^n$ possibili valori, e se si calcolano $r = 2^{n(k-1)/k}$ valori della funzione hash, allora ci si aspetta una k-collisione. Tuttavia, come mostreremo in seguito, spesso si possono ottenere collisioni molto più facilmente.

In molte funzioni hash, come per esempio SHA-1, c'è una funzione di compressione f che opera sugli input di una lunghezza fissata. Inoltre, c'è un valore iniziale IV fissato. Il messaggio è completato per ottenere il formato desiderato. Poi si eseguono le seguenti istruzioni.

1. Spezzare il messaggio M in blocchi $M_1, M_2, \ldots, M_\ell$.
2. Porre H_0 uguale al valore iniziale IV.
3. Per $i = 1, 2, \ldots, \ell$, porre $H_i = f(H_{i-1}, M_i)$.
4. Porre $H(M) = H_\ell$.

In SHA-1, la funzione di compressione è rappresentata nella Figura 8.3 (nelle pagine precedenti). Per ogni iterazione, prende un input $A||B||C||D||E$ di 160 bit dall'iterazione precedente insieme a un blocco di messaggio m_i di lunghezza 512 e fornisce in output una nuova stringa $A||B||C||D||E$ di lunghezza 160.

Si supponga che l'output della funzione f, e quindi anche della funzione hash H, sia formato da n bit. Un attacco del compleanno può trovare, in circa $2^{n/2}$ passi, due blocchi m_0 e m_0' tali che $f(H_0,m_0) = f(H_0,m_0')$. Sia $h_1 = f(H_0,m_0)$. Un secondo attacco del compleanno trova due blocchi m_1 e m_1' con $f(h_1,m_1) = f(h_1,m_1')$. Continuando in questo modo, si pone

$$h_i = f(h_{i-1}, m_{i-1})$$

e si usa un attacco del compleanno per trovare m_i e m_i' con

$$f(h_i,m_i) = f(h_i,m_i').$$

Questo processo continua finché non si hanno t coppie di blocchi m_0, m_0', m_1, m_1', $\ldots$, m_{t-1}, m_{t-1}', dove t è un intero da determinare in seguito.

Ognuno dei 2^t messaggi

$$\begin{gathered}
m_0||m_1||\cdots||m_{t-1} \\
m_0'||m_1||\cdots||m_{t-1} \\
m_0||m_1'||\cdots||m_{t-1} \\
m_0'||m_1'||\cdots||m_{t-1} \\
\cdots\cdots\cdots \\
m_0'||m_1||\cdots||m_{t-1}' \\
m_0||m_1'||\cdots||m_{t-1}' \\
m_0'||m_1'||\cdots||m_{t-1}'
\end{gathered}$$

(tutte le possibili combinazioni tra gli m_i e gli m_i') ha lo stesso valore hash, a causa della natura iterativa dell'algoritmo hash. Ogni volta che si calcola $h_i = f(m, h_{i-1})$, se $m = m_{i-1}$ o $m = m_{i-1}'$ allora si ottiene lo stesso valore h_i. Quindi l'output della funzione f durante ogni passo dell'algoritmo hash è indipendente dal fatto che si usi un m_{i-1} o un m_{i-1}'. Pertanto l'output finale dell'algoritmo hash è lo stesso per tutti i messaggi. Si ha così una 2^t-collisione.

Questa procedura impiega circa $t\,2^{n/2}$ passi e ha un tempo di esecuzione atteso di circa una costante per $tn\,2^{n/2}$ (si veda l'Esercizio 6). Se, per esempio, $t = 2$, allora per trovare quattro messaggi con lo stesso valore hash impiega solo circa il doppio del tempo che impiega per trovare due messaggi con lo stesso hash. Se l'output della funzione hash fosse realmente casuale, invece che prodotto da un algoritmo iterativo, la procedura di prima non funzionerebbe. Il tempo atteso per trovare quattro messaggi con lo stesso hash sarebbe allora circa $2^{3n/4}$, che è molto maggiore del tempo richiesto per trovare due messaggi collidenti. Quindi è più facile trovare delle collisioni con un algoritmo hash iterativo.

Una conseguenza interessante della precedente discussione riguarda i tentativi di migliorare le funzioni hash concatenando i loro output. Prima che apparisse [Joux], si pensava che la concatenazione

$$H(M) = H_1(M)||H_2(M)$$

di due funzioni hash H_1 e H_2 fosse una funzione hash significativamente più robusta delle singole funzioni H_1 e H_2 prese singolarmente. Questo avrebbe permesso di usare funzioni hash deboli per costruirne di più robuste. Tuttavia, ora sembra che questo non sia vero. Si supponga che l'output di H_i sia di n_i bit, per $i = 1, 2$, e che H_1 venga

calcolata mediante un algoritmo iterativo, come nella precedente discussione. Non si fanno ipotesi su H_2. Si può anche assumere che sia un oracolo casuale, nel senso del Paragrafo 8.6. In un tempo di circa $\frac{1}{2}n_2n_1 2^{n_1/2}$, si possono trovare $2^{n_2/2}$ messaggi che hanno tutti lo stesso valore hash per H_1. Poi si calcola il valore di H_2 per ognuno di questi $2^{n_2/2}$ messaggi. Per il paradosso del compleanno, ci si aspetta di trovare una corrispondenza tra questi valori di H_2. Poiché questi messaggi hanno tutti lo stesso valore in H_1, si ha una collisione per $H_1||H_2$. Quindi, in un tempo proporzionale a $n_2n_1 2^{n_1/2} + n_2 2^{n_2/2}$ (spiegheremo questa stima tra breve), ci si aspetta di essere in grado di trovare una collisione per $H_1||H_2$. Questo tempo non è molto maggiore di quello impiegato da un attacco del compleanno per trovare una collisione per l'hash più lungo tra H_1 e H_2 ed è molto minore del tempo $2^{(n_1+n_2)/2}$ che un attacco del compleanno standard impiegherebbe sulla funzione hash concatenata.
Come si ottiene la stima $n_2n_1 2^{n_1/2} + n_2 2^{n_2/2}$ per il tempo di esecuzione? Sono stati usati $\frac{1}{2}n_2n_1 2^{n_1/2}$ passi per ottenere i $2^{n_2/2}$ messaggi con lo stesso valore in H_1. Ognuno di questi messaggi è formato da n_2 blocchi di lunghezza fissata. Poi si è valutato H_2 su ognuno di questi messaggi. Per quasi ogni funzione hash, il tempo di valutazione è proporzionale alla lunghezza dell'input. Quindi, il tempo di valutazione è proporzionale a n_2 per ognuno dei $2^{n_2/2}$ messaggi inviati ad H_2. Questo dà il termine $n_2 2^{n_2/2}$ nel tempo di esecuzione stimato.

8.6 Modello dell'oracolo casuale

Idealmente, una funzione hash è indistinguibile da una funzione casuale. Il modello dell'oracolo casuale, introdotto nel 1993 da Bellare e Rogaway [Bellare-Rogaway], fornisce un metodo per analizzare la sicurezza degli algoritmi crittografici che usano funzioni hash trattando le funzioni hash come oracoli casuali.
Un oracolo casuale funziona nel modo seguente. Chiunque può dare un input all'oracolo, che produce un output di lunghezza fissata. Se l'input è già stato inserito da qualcun altro in precedenza, allora l'oracolo fornisce in output lo stesso valore dato nell'occasione precedente. Se l'input non è mai stato dato in precedenza all'oracolo, allora l'oracolo dà un output scelto a caso. Per esempio, può lanciare n monete oneste e usare il risultato per produrre un output di n bit.
Per ragioni pratiche, un oracolo casuale non può essere usato nella maggior parte degli algoritmi crittografici. Tuttavia, il supporre che una funzione hash si comporti come un oracolo casuale permette di analizzare la sicurezza di molti crittosistemi che usano funzioni hash.
Tale ipotesi è già stata fatta nel Paragrafo 8.4. Quando si calcola la probabilità che un attacco del compleanno trovi collisioni per una funzione hash, si assume che l'output della funzione hash sia distribuito casualmente e uniformemente tra tutte le possibilità. Se così non fosse, ossia se esistessero dei valori della funzione hash che tendessero a presentarsi più frequentemente degli altri, allora la probabilità di trovare collisioni sarebbe un po' più alta (si consideri, per esempio, il caso estremo di un funzione hash davvero cattiva che ha una probabilità altissima di fornire in output un certo valore). Quindi la nostra stima per la probabilità di avere collisioni vale in realtà solo in una situazione idealizzata. In pratica, l'uso di funzioni hash reali probabilmente produce solo poche collisioni in più.

Mostreremo ora il modo in cui si usa il modello dell'oracolo casuale per analizzare la sicurezza di un crittosistema. Poiché il testo cifrato è molto più lungo del testo in chiaro, il sistema che verrà descritto non è così efficiente come i metodi del tipo OAEP (si veda il Paragrafo 6.2). Tuttavia, questo sistema è un buon esempio d'uso del modello dell'oracolo casuale.
Sia f una funzione unidirezionale che Bob sa come invertire. Per esempio, $f(x) = x^e \pmod n$, dove (e,n) è la chiave pubblica RSA di Bob. Sia H una funzione hash. Per cifrare un messaggio m, che si assume della stessa lunghezza dell'output di H, Alice sceglie a caso un intero r modulo n e pone come testo cifrato

$$(y_1, y_2) = (f(r), H(r) \oplus m).$$

Quando Bob riceve (y_1, y_2), calcola

$$r = f^{-1}(y_1), \qquad m = H(r) \oplus y_2.$$

Si vede facilmente che questa decifrazione produce il messaggio originale m.
Si consideri ora il seguente problema. Ad Alice vengono mostrati due testi in chiaro m_1 e m_2 e un testo cifrato, ma non le viene detto a quale testo in chiaro corrisponde il testo cifrato dato. Il suo compito è di indovinare qual è. Se non può farlo con una probabilità significativamente maggiore del 50%, allora si dirà che il crittosistema possiede la **proprietà di indistinguibilità del testo cifrato**.
Si supponga che la funzione hash sia un oracolo casuale. Mostreremo che se Alice può avere successo con una probabilità significativamente maggiore del 50%, allora può invertire f con una probabilità significativamente maggiore di zero. Quindi se f è realmente una funzione unidirezionale, il crittosistema possiede la proprietà di indistinguibilità del testo cifrato.
Si supponga ora che Alice abbia un testo cifrato (y_1, y_2) e due testi in chiaro m_1 e m_2. Le è concesso fare una serie di domande all'oracolo casuale. Ogni volta che gli invia un valore r, riceve in risposta il valore $H(r)$. Si supponga che nel tentativo di indovinare chi tra m_1 e m_2 dà (y_1, y_2), Alice richieda i valori hash di ogni elemento di un qualche insieme $L = \{r_1, r_2, \ldots, r_\ell\}$.
Quando Alice richiede il valore $H(x)$ per ogni $x \in L$, calcola $f(x)$ per il corrispondente x. Se $r \in L$, alla fine arriva a provare $x = r$ e trova che $f(r) = y_1$. In questo caso, Alice sa che questo è il valore corretto di r. Poiché riceve $H(r)$ dall'oracolo, calcola $H(r) \oplus y_2$ per ottenere il testo in chiaro, che è m_1 o m_2.
Se $r \notin L$, allora Alice non conosce il valore di $H(r)$. Poiché H è un oracolo casuale, i possibili valori di $H(r)$ sono distribuiti casualmente e uniformemente. Quindi anche i possibili valori di $H(r) \oplus m$, al variare di m, sono distribuiti casualmente e uniformemente tra tutte le possibilità. Questo significa che y_2 non dà ad Alice alcuna informazione circa il fatto che esso provenga da m_1 o da m_2. Pertanto, se $r \notin L$, Alice ha probabilità $1/2$ di indovinare il testo in chiaro corretto.
In termini di probabilità, questa procedura si scrive nel modo seguente. Se $r \notin L$, Alice indovina correttamente metà delle volte. Se $r \in L$, Alice indovina sempre correttamente. Quindi

$$\text{Prob(Alice indovina correttamente)} = \frac{1}{2}\,\text{Prob}(r \notin L) + \text{Prob}(r \in L).$$

Si supponga ora che Alice abbia probabilità almeno $\frac{1}{2} + \epsilon$ di indovinare correttamente, dove $\epsilon > 0$ è un qualche numero fissato. Poiché $\text{Prob}(r \notin L) \leq 1$ (questo è vero per ogni probabilità), si ottiene

$$\frac{1}{2} + \epsilon \leq \frac{1}{2} + \text{Prob}(r \in L)$$

e quindi

$$\text{Prob}(r \in L) \geq \epsilon.$$

Ma se $r \in L$, allora Alice scopre che $f(r) = y_1$. Quindi la probabilità che risolva $f(r) = y_1$ rispetto r è almeno ϵ.

Se si assume che per Alice sia computazionalmente intrattabile trovare r con probabilità almeno ϵ, allora si conclude che per Alice è computazionalmente impossibile indovinare correttamente con probabilità almeno $\frac{1}{2} + \epsilon$. Quindi, se la funzione f è unidirezionale, il crittosistema possiede la proprietà di indistinguibilità del testo cifrato.

Si osservi che nella precedente argomentazione è importante assumere che i valori di H siano distribuiti casualmente e uniformemente. Se non fosse così, ossia se la funzione hash presentasse una leggera prevalenza di alcuni valori, allora Alice potrebbe avere qualche metodo per indovinare correttamente con una probabilità maggiore del 50%, magari con probabilità $\frac{1}{2} + \epsilon$. Questo porterebbe alla conclusione che $\text{Prob}(r \in L) \geq 0$, che non dà alcuna informazione. Quindi l'ipotesi che la funzione hash sia un oracolo casuale è importante.

Naturalmente, una buona funzione hash probabilmente si comporta in modo molto simile a un oracolo casuale. In questo caso, l'argomento precedente mostra che il crittosistema con una funzione hash reale dovrebbe essere abbastanza resistente ai tentativi di Alice di indovinare. Tuttavia, Canetti, Goldreich e Halevi [Canetti et al.] hanno costruito un crittosistema che è sicuro nel modello dell'oracolo casuale, ma che non è sicuro per ogni scelta concreta della funzione hash. Fortunatamente, questa costruzione non è una di quelle che vengono usate in pratica.

La procedura precedente, che riduce la sicurezza di un sistema alla risolubilità di qualche problema fondamentale, come la non invertibilità di una funzione unidirezionale, è comune nelle dimostrazioni di sicurezza. Per esempio, nel Paragrafo 7.5, certe questioni relative al crittosistema a chiave pubblica di ElGamal sono state ridotte alla risolubilità dei problemi di Diffie-Hellman.

Il Paragrafo 8.5 mostra che la maggior parte delle funzioni hash non si comporta come un oracolo casuale rispetto alle multicollisioni. Questo significa che bisogna stare attenti quando si utilizza il modello dell'oracolo casuale.

L'uso del modello dell'oracolo casuale nell'analisi di un crittosistema è piuttosto controverso. Tuttavia, molte persone pensano che esso permetta di ottenere alcune indicazioni sulla robustezza del sistema. Se un sistema non è sicuro nel modello dell'oracolo casuale, allora sicuramente non è sicuro in pratica. La controversia nasce quando si dimostra che un sistema è sicuro nel modello dell'oracolo casuale. A questo punto, cosa si può dire sulla sicurezza delle implementazioni reali? Crittografi diversi daranno risposte diverse. Tuttavia, al momento, sembra che non ci siano metodi migliori di analisi della sicurezza che siano di validità generale.

8.7 Cifratura mediante funzioni hash

Le funzioni hash crittografiche sono uno degli strumenti crittografici maggiormente usati, secondi forse solo ai cifrari a blocchi. Trovano applicazione in molte aree della sicurezza delle informazioni. Nel Capitolo 9, vedremo un'applicazione delle funzioni hash alla firma digitale, dove il fatto di comprimere la rappresentazione dei dati rende più efficiente la generazione di una firma digitale. Ora vedremo il modo in cui possono essere utilizzate come cifrari per garantire riservatezza dei dati.

Una funzione hash crittografica prende un input di lunghezza arbitraria e fornisce un output di grandezza fissata, apparentemente casuale. In particolare, se si hanno due input simili, allora i loro hash dovrebbero essere differenti. In generale, i loro hash sono molto differenti. Questa è una proprietà che le funzioni hash condividono con i buoni cifrari ed è una proprietà che permette di usare una funzione hash per eseguire una cifratura.

L'uso di una funzione hash per eseguire una cifratura è molto simile a un sistema di cifratura in cui l'output di un generatore di numeri pseudocasuali è sommato XOR con il testo in chiaro. Abbiamo visto un esempio di questo tipo quando abbiamo studiato la modalità OFB (*output feedback mode*) di un cifrario a blocchi. Analogamente a quanto faceva il cifrario a blocchi per l'OFB, la funzione hash genera un flusso di bit pseudocasuali che è sommato XOR con il testo in chiaro per generare un testo cifrato. Per far sì che una funzione hash crittografica operi come un cifrario a flusso, occorrono due componenti: una chiave condivisa tra Alice e Bob e un vettore di inizializzazione. Tralasciamo per un attimo la questione del vettore di inizializzazione e supponiamo per ora che Alice e Bob abbiano stabilito una chiave segreta condivisa K_{AB}.

Ora, Alice potrebbe generare un byte pseudocasuale x_1 prendendo il byte più a sinistra dell'hash di K_{AB}, cioé $x_1 = L_8(h(K_{AB}))$. Poi potrebbe cifrare un byte di testo in chiaro p_1 sommando XOR con il byte casuale x_1 in modo da produrre un byte di testo cifrato

$$c_1 = p_1 \oplus x_1.$$

Ma se ha più di un byte di testo in chiaro, come dovrebbe continuare? Si usa la retroazione, analogamente a quanto fatto nella modalità OFB. Il byte pseudocasuale successivo dovrebbe essere generato da $x_2 = L_8(h(K_{AB} \| x_1))$. Allora il byte di testo cifrato successivo può essere generato da

$$c_2 = p_2 \oplus x_2.$$

In generale, il byte pseudocasuale x_j viene generato da $x_j = L_8(h(K_{AB} \| x_{j-1}))$ e la cifratura avviene semplicemente sommando XOR x_j con il testo in chiaro p_j. La decifrazione è molto facile, poiché Bob deve semplicemente rigenerare i byte x_j e sommarli XOR con il testo cifrato c_j per ottenere il testo in chiaro p_j.

C'è un piccolo problema con questa procedura di cifratura e decifrazione. Cosa accade se Alice vuole cifrare un messaggio Lunedì e un messaggio differente Mercoledì? Come dovrebbe generare i byte pseudocasuali? Se esegue da capo la procedura, allora la successione pseudocasuale degli x_j di Lunedì e Mercoledì sarà la stessa e questo dovrebbe essere evitato.

Bisogna introdurre qualche fattore di casualità per assicurare che i due flussi di bit siano differenti. Così, ogni volta che Alice manda un messaggio, dovrebbe scegliere un

vettore di inizializzazione casuale che verrà indicato con x_0. Allora inizia generando $x_1 = L_8(h(K_{AB}\|x_0))$ e poi procede come prima. Ma ora deve inviare x_0 a Bob, cosa che può fare quando invia c_1. Se Eva intercetta x_0, ancora non è in grado di calcolare x_1 poiché non conosce K_{AB}. Infatti, se h è una buona funzione hash, allora x_0 non dovrebbe dare alcuna informazione su x_1.
L'idea di usare una funzione hash per avere una procedura di cifratura può essere modificata per avere una procedura di cifratura che incorpora il testo in chiaro, analogamente alla modalità CFB.

8.8 Esercizi

1. Sia p un primo e sia α un intero con $p \nmid \alpha$. Sia $h(x) \equiv \alpha^x \pmod p$. Spiegare perché $h(x)$ non è una buona funzione hash crittografica.

2. Sia $n = pq$ il prodotto di due primi grandi distinti e sia $h(x) = x^2 \pmod n$.

   (a) Perché h è resistente alle controimmagini? (Naturalmente, esistono alcuni valori, come 1, 4, 9, 16, ..., per i quali è facile trovare una controimmagine, ma in generale è difficile.)

   (b) Perché h non è fortemente resistente alle collisioni?

3. Si supponga che un messaggio m sia diviso in blocchi lunghi 160 bit: $m = M_1\|M_2\|\cdots\|M_\ell$. Sia $h(x) = M_1 \oplus M_2 \oplus \cdots \oplus M_\ell$. Quali delle proprietà (1), (2), (3) delle funzioni hash sono soddisfatte da h?

4. In una famiglia di quatto componenti, qual è la probabilità che non ci siano due persone che hanno il compleanno nello stesso mese? (Assumere che tutti i mesi abbiano uguale probabilità.)

5. In questo esercizio si ottiene la formula (8.1) che dà la probabilità di avere almeno una corrispondenza in una lista di lunghezza r quando ci sono N possibili compleanni.

   (a) Siano $f(x) = \ln(1-x) + x$ e $g(x) = \ln(1-x) + x + x^2$. Mostrare che $f'(x) \le 0$ e $g'(x) \ge 0$ per $0 \le x \le 1/2$.

   (b) Usando il fatto che $f(0) = g(0) = 0$, f è decrescente e g è crescente, mostrare che
   $$-x - x^2 \le \ln(1-x) \le -x \quad \text{per } 0 \le x \le 1/2.$$

   (c) Mostrare che se $r \le N/2$, allora
   $$-\frac{(r-1)r}{2N} - \frac{r^3}{3N^2} \le \sum_{j=1}^{r-1} \ln\left(1 - \frac{j}{N}\right) \le -\frac{(r-1)r}{2N}.$$

   Suggerimento: osservare che
   $$\sum_{j=1}^{r-1} j = \frac{(r-1)r}{2} \quad \text{e} \quad \sum_{j=1}^{r-1} j^2 = \frac{(r-1)r(2r-1)}{6} < r^3/3.$$

   (d) Sia $\lambda = r^2/(2N)$ con $\lambda \le N/8$ (questo implica che $r \le N/2$). Mostrare che
   $$e^{-\lambda}e^{c_1/\sqrt{N}} \le \prod_{j=1}^{r-1}\left(1 - \frac{j}{N}\right) \le e^{-\lambda}e^{c_2/\sqrt{N}}$$
   dove $c_1 = \sqrt{\lambda/2} - \frac{1}{3}(2\lambda)^{3/2}$ e $c_2 = \sqrt{\lambda/2}$.

   (e) Osservare che quando N è grande, $e^{c/\sqrt{N}}$ è vicino a 1. Usare questo fatto per mostrare che quando N diventa grande e λ è costante con $\lambda \le N/8$, allora si ha l'approssimazione
   $$\prod_{j=1}^{r-1}\left(1 - \frac{j}{N}\right) \approx e^{-\lambda}.$$

6. Sia $f(x)$ una funzione con output di n bit e con input molto più lunghi di n bit (questo implica che devono esistere collisioni). Con un attacco del compleanno si ha probabilità 1/2 di trovare una collisione in circa $2^{n/2}$ passi.

   (a) Se si ripete l'attacco del compleanno finché non si trova una collisione, mostrare che il numero atteso di ripetizioni è
   $$1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{8} + 4 \cdot \frac{1}{16} + \cdots = 2$$
   (la somma S che compare a primo membro può essere calcolata scrivendo $S - \frac{1}{2}S = \frac{1}{2} + (2-1)\frac{1}{4} + (3-2)\frac{1}{8} + \cdots = 1$).

   (b) Se ogni valutazione di f richiede un tempo pari a una costante per n (questa ipotesi è realistica poiché gli input che servono per trovare le collisioni possono essere scelti in modo da avere, per esempio, $2n$ bit), mostrare che il tempo atteso per trovare una collisione per la funzione f è pari a una costante per $n\,2^{n/2}$.

   (c) Mostrare che il tempo atteso per produrre i messaggi m_0, m_0', ..., m_{t-1}, m_{t-1}' del Paragrafo 8.5 è pari a una costante per $tn\,2^{n/2}$.

7. È data una funzione hash iterativa, come nel Paragrafo 8.5, tranne che per il fatto che la funzione viene leggermente modificata a ogni iterazione. Per concretezza, si assuma che l'algoritmo proceda nel modo seguente. Si ha una funzione di compressione f che opera sugli input di lunghezza fissata. Si ha anche una funzione g che produce output di lunghezza fissata e si ha un valore iniziale fissato IV. Il messaggio è completato per ottenere il formato desiderato. Si eseguono poi le seguenti istruzioni.

   1. Spezzare il messaggio M in blocchi M_1, M_2, ..., M_ℓ.
   2. Porre H_0 uguale al valore iniziale IV.
   3. Per $i = 1, 2, \ldots, \ell$, porre $H_i = f(H_{i-1}, M_i\|g(i))$.
   4. Porre $H(M) = H_\ell$.

Mostrare che il metodo del Paragrafo 8.5 può essere usato per produrre delle multicollisioni per questa funzione hash.

8. I valori iniziali K_t in SHA-1 potrebbero sembrare casuali. Ecco come vengono scelti.

(a) Calcolare $\lfloor 2^{30}\sqrt{2} \rfloor$ e scrivere il risultato in esadecimale. Il risultato dovrebbe essere K_0.

(b) Eseguire un calcolo simile sostituendo $\sqrt{2}$ con $\sqrt{3}$, $\sqrt{5}$ e $\sqrt{10}$ e confrontare con K_{20}, K_{40} e K_{60}.

9. **(a)** Sia E_K una funzione di cifratura con N possibili chiavi K, N possibili testi in chiaro e N possibili testi cifrati. Si supponga che

i. se si conosce la chiave di cifratura K, sia facile trovare la funzione di decifrazione D_K (quindi questo problema non riguarda i metodi a chiave pubblica);

ii. per ogni coppia (K_1, K_2) di chiavi, sia possibile trovare una chiave K_3 tale che $E_{K_1}(E_{K_2}(m)) = E_{K_3}(m)$ per ogni testo in chiaro m;

iii. per ogni coppia testo in chiaro–testo cifrato (m,c), esista, di solito, solo una chiave K tale che $E_K(m) = c$;

iv. si conosca una coppia testo in chiaro–testo cifrato (m,c).

Dare un attacco del compleanno che trovi la chiave K in circa $2\sqrt{N}$ passi. (*Osservazione:* questo algoritmo è molto più veloce di una ricerca a forza bruta tra tutte le chiavi K, che richiede un tempo proporzionale a N.)

(b) Mostrare che il cifrario a scorrimento (si veda il Paragrafo 2.1) soddisfa le condizioni della parte (a) e spiegare come attaccare il cifrario a scorrimento modulo 26 usando due liste di lunghezza 6. (Naturalmente, si può anche trovare la chiave semplicemente sottraendo il testo in chiaro dal testo cifrato. Lo scopo di questa parte è semplicemente quella di illustrare la parte (a).)

10. **(a)** Mostrare che nessuna delle due funzioni hash del Paragrafo 8.2 è resistente alle controimmagini. In altre parole, dato un arbitrario y (di lunghezza appropriata), mostrare come trovare un input x il cui hash è y.

(b) Trovare una collisione per ognuna delle due funzioni hash del Paragrafo 8.2.

11. Sia H una funzione hash iterativa che opera successivamente sui blocchi di input di 512 bit. In particolare, esiste una funzione di compressione h e un valore iniziale IV. L'hash di un messaggio $M_1 \| M_2$ di 1024 bit viene calcolato da $X_1 = h(IV, M_1)$ e $H(M_1 \| M_2) = h(X_1, M_2)$. Si supponga di avere trovato una collisione $h(IV,x_1) = h(IV, x_2)$ per due blocchi x_1 e x_2 da 512 bit. Si scelgano due primi distinti p_1 e p_2, ognuno approssimativamente di 240 bit. Si considerino x_1 e x_2 come numeri tra 0 e 2^{512}.

(a) Mostrare che esiste un x_0, con $0 \le x_0 < p_1p_2$, tale che

$$x_0 + 2^{512}x_1 \equiv 0 \pmod{p_1} \quad \text{e} \quad x_0 + 2^{512}x_2 \equiv 0 \pmod{p_2}.$$

(b) Mostrare che se $0 \le k < 2^{30}$, allora $q_1 = (x_0 + 2^{512}x_1 + kp_1p_2)/p_1$ è circa 2^{784} e analogamente per $q_2 = (x_0 + 2^{512}x_2 + kp_1p_2)/p_2$. (Si assuma che x_1 e x_2 siano approssimativamente 2^{512}.)

(c) Usare il teorema dei numeri primi (si veda il Paragrafo 3.1) per mostrare che la probabilità che q_1 sia primo è di circa $1/543$ e che la probabilità che q_1 e q_2 siano entrambi primi è circa $1/300000$.

(d) Mostrare che è verosimile che esista un k con $0 \le k < 2^{30}$ tale che q_1 e q_2 siano entrambi primi.

(e) Mostrare che $n_1 = p_1q_1$ e $n_2 = p_2q_2$ soddisfano $H(n_1) = H(n_2)$.

Questo metodo per produrre due moduli RSA con gli stessi valori hash si basa sul metodo di [Lenstra et al.] che usa una collisione per produrre due certificati X.509 con gli stessi hash. Il metodo presentato qui produce moduli $n = pq$ con p e q di grandezze significamente differenti (240 bit e 784 bit), ma un avversario non lo sa senza fattorizzare n. Trovare un fattore di 240 bit di un numero di 1024 bit va ben oltre la tecnologia attuale (anno 2005).

8.9 Problemi al calcolatore

1. **(a)** Se in una classe ci sono 30 studenti, qual è la probabilità che almeno due di essi festeggino il compleanno nello stesso giorno? Confrontare questo risultato con l'approssimazione data dalla formula (8.1).

(b) Quanti dovrebbero essere gli studenti per avere un 99% di possibilità che almeno due di essi siano nati nello stesso giorno? (*Suggerimento:* usare l'approssimazione per ottenere una risposta approssimata. Usare poi il prodotto, per vari numeri di studenti, finché non si trova la risposta esatta.)

(c) Quanti dovrebbero essere gli studenti per avere il 100% di probabilità che almeno due di essi siano nati nello stesso giorno?

2. Un professore espone i voti assegnati agli studenti di un corso usando le ultime quattro cifre del numero di matricola di ogni studente. In un corso di 200 studenti, qual è la probabilità che almeno due studenti abbiano le stesse quattro cifre?