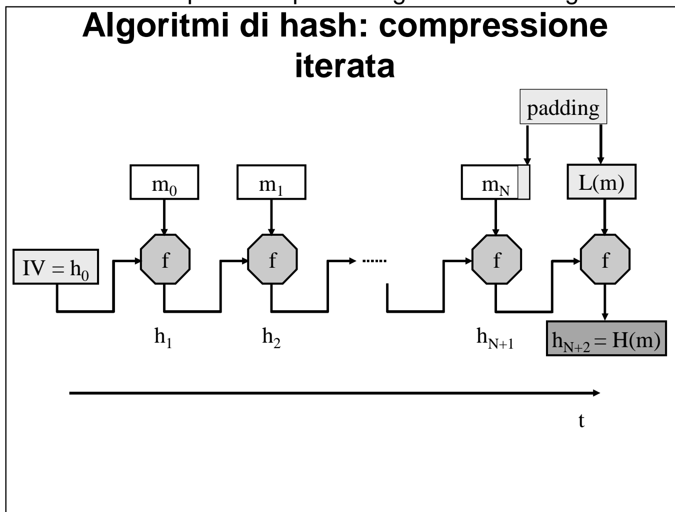
Algoritmi di hash

La funzione **hash sicura** è un'importante **primitiva crittografica**. Abbiamo visto, infatti, che il calcolo di un'impronta è chiamato in causa nell'attestazione di integrità, nell'attestazione di origine e nella generazione di numeri pseudocasuali; altri interessanti impieghi riguardano l'identificazione (v. 2.2.3 e 2.2.4). Quattro proprietà costituiscono un prerequisito per ogni algoritmo che realizza una funzione hash crittografica.

- R18 (efficienza): "*il calcolo di* **H(x)** *è computazionalmente facile per ogni* **x**".
- R19 (robustezza debole alle collisioni): "*per ogni* **x** *è infattibile trovare un* **y ≠ x** *tale che* **H(y) = H(x)**".
- R20 (resistenza forte alle collisioni): "*è infattibile trovare una qualsiasi coppia* **x, y** *tale che* **H(y) = H(x)**".
- R21 (unidirezionalità): "*per ogni* **h** *è infattibile trovare un* **x** *tale che* **H(x) = h**".

1 - Efficienza

Occorre poter comprimere agevolmente stringhe binarie arbitrariamente lunghe.



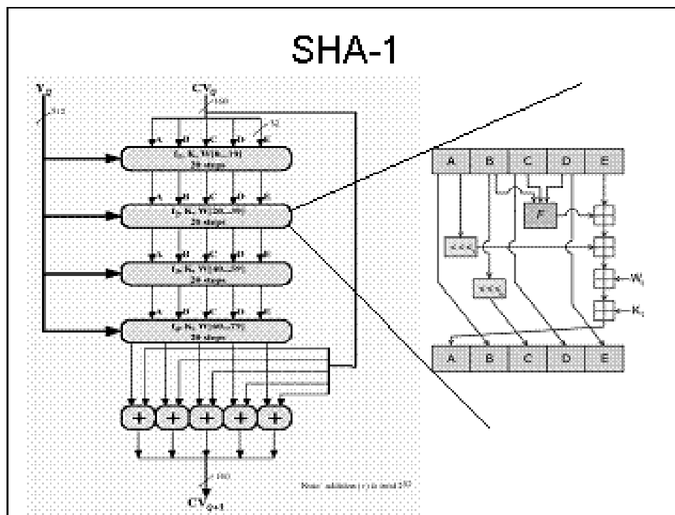
A tal fine, tutti gli algoritmi di *hash* si ispirano ad un principio ormai ben consolidato, detto della **compressione iterata**[8].

Ogni iterazione impiega una funzione di compressione **f**, **resistente alle collisioni** e **facile** da calcolare, che genera **n** bit d'uscita a partire da due dati d'ingresso, uno di **r** bit, con **r** maggiore di **n**, ed uno di **n** bit.

Il messaggio **m** è suddiviso in **blocchi** $\mathbf{m_0}$, $\mathbf{m_1}$, .., $\mathbf{m_N}$, tutti di lunghezza **r**; se l'ultimo blocco è più piccolo, si completa con degli "zeri" (*padding*). In ogni caso si aggiunge poi un ulteriore blocco che indica la lunghezza di **m**.

L'algoritmo comprime un blocco alla volta assumendo come stato interno il risultato del passo precedente: $\mathbf{h_i = f(m_{i-1}, h_{i-1})}$.

Il dato $\mathbf{h_0}$ è una **costante** detta vettore d'inizializzazione; l'ultimo **h** è l'impronta di **m**.



Discutere in dettaglio come sono fatti gli algoritmi che realizzano oggi funzioni hash sicure è argomento che esula dai limiti di questo corso[9]. Per avere almeno un'idea è sufficiente fare riferimento alla figura a lato.

In SHA-1 il messaggio da comprimere è suddiviso in blocchi da 512 bit (16 parole da 32 bit); lo stato interno è di 160 bit (5 parole di 32 bit denominate A,B,C,D,E); al termine delle iterazioni è restituita un'impronta di 160 bit.

Una volta che il blocco del messaggio è stato espanso in 80 parole da 32 bit (in figura W_i per i = 0,1,.., 79), la funzione di compressione le elabora una alla volta in 4 *round* formati ciascuno da 20 *step*.

In ogni step i dati a 32 bit sono sottoposti ad operazioni logiche, rotazioni, trasposizioni ed addizioni modulo 2^{32}. Al termine del quarto round i valori di stato sono sommati modulo 2^{32} con quelli impiegati nel primo round.

E' interessante prendere subito nota che il precisare la lunghezza del messaggio nell'ultimo blocco è un'indispensabile "patch" per una vulnerabilità presente in tutti gli algoritmi a compressione iterata.

[8] R.Merkle "Secrecy, Authentication and Public Key Systems", *UMI Research Press*, Ann Arbor Michigan 1978; "One Way Hash functions and DES", *Proceedings of CRYPTO '89*; Springger-Verlag

[9] A chi lo vuole approfondire suggeriamo la consultazione di [9], cap. 12

Per giustificare questa tecnica consideriamo una sorgente che, dopo aver concordato un segreto **s** con la destinazione, le invia un messaggio **m** ed il suo "autenticatore" **H(s||m)**, calcolato però senza aver aggiunto alla fine di **m** l'indicazione di quanto è lungo.

L'intruso può in questo caso condurre un attacco di *length extension*: individua un'estensione **m'** di **m** di suo interesse (forma cioè un messaggio **m||m'**) e calcola **H(m')**, ponendo attenzione a fornire all'algoritmo come stato iniziale non la costante $\mathbf{h_0}$, ma il dato **H(s||m)** che ha intercettato.

L'impronta **H(m')** così ottenuta è per costruzione uguale a quella di **H(s||(m||m')** e quindi il messaggio forgiato dall'intruso sarà giudicato dalla destinazione come generato dalla sorgente. Inserire la lunghezza del messaggio in coda al messaggio rende questo attacco più difficile, ma non impossibile.

Invertire la posizione di **s** e di **m** non risolve il problema dell'autenticazione.

Supponiamo che l'intruso riesca a trovare un messaggio **m*** in collisione con **m**.

Con questa ipotesi si ha **H(m||s) = H(m*||s)**: lo stato interno raggiunto dall'algoritmo **H** dopo la compressione delle prime componenti è, infatti, lo stesso e l'uguaglianza non si modificherà con la successiva compressione di **s**, **qualsiasi esso sia**. In conclusione all'intruso è sufficiente inviare il messaggio **m*||H(m||s)** per spacciarsi per uno dei due corrispondenti.

Esistono altri difetti, più o meno piccoli, che impediscono agli attuali algoritmi di presentare esattamente il comportamento aleatorio della funzione hash ideale. Il loro impiego come meccanismi per la sicurezza richiede dunque l'adozione di accorgimenti particolari.

Nell'attesa, forse vana, di un algoritmo di hash perfetto, i Crittografi suggeriscono di applicare al messaggio **due successive compressioni**; naturalmente ciò va a scapito dell'efficienza. Ferguson e Schneier ([10], pag. 92) propongono di calcolare l'impronta come **H(H(m))**, o come **H(H(m)||m**). Analoga soluzione è adottata nello standard Internet **HMAC** che esamineremo nel cap. 4.

2 – Resistenza debole alle collisioni

L'impiego della funzione hash nella verifica dell'integrità di un messaggio **m** presuppone che nessuno sappia trovare un messaggio **m***, diverso da **m**, ma tale che sia **H(m) = H(m*)**.

Messaggi di questo tipo esistono sicuramente ed occorre quindi valutare attentamente la complessità computazionale del seguente attacco: si sceglie a caso **m***, si calcola **H(m*)** e si confronta il risultato con **H(m)**, ripetendo il procedimento fino a quando non si ha successo.

Per fare qualche conto è utile impiegare il modello ideale della funzione hash: "**una funzione hash sicura, sottoposta ad una sequenza di ingressi scelti a caso, fornisce una sequenza di impronte descrivibile con una variabile aleatoria che assume valori nell'intervallo $0 \div 2^n-1$ con distribuzione uniforme di probabilità**".

Con questa ipotesi, la probabilità di successo di un solo tentativo è 2^{-n}, quella di insuccesso $1 - 2^{-n}$.

Dopo **k** tentativi la probabilità di successo è data dal teorema binomiale:

$$P_1(2^n, k) = 1 - (1 - 2^{-n})^k = k \times 2^{-n} - k \times (k-1) \times 2^{-2n}/2 + k \times (k-1) \times (k-2) \times 2^{-3n}/6 - .. \text{ ecc.}$$

Se **n** è grande, 2^{-n} è molto piccolo e ci si può dunque arrestare al primo termine:

$$P_1(2^n, k) > k \times 2^{-n}$$

Fissata una desiderata probabilità di successo, il numero **k** di prove da fare è proporzionale a 2^n. L'algoritmo d'attacco è dunque esponenziale nella dimensione **n** dell'impronta: **O(exp(n))**. Per fronteggiarlo, la dimensione dell'impronta è progressivamente aumentata.

ESEMPI – Fino ad una decina di anni fa l'algoritmo di hash più usato è stato **MD5** (R. Rivest,1991), che impiega 512 bit di blocco e 64 passi di elaborazione per calcolare 128 bit d'uscita.

Successivamente, a seguito dell'individuazione di vulnerabilità in MD5 e comunque per aumentare la robustezza a fronte di attacchi *birthday*, è stato impiegato **SHA-1** (NIST, 1994, revisione del precedente standard SHA): la funzione di compressione usa 512 bit di blocco e 80 passi per calcolare un'impronta di 160 bit.

In ambito europeo ha trovato impiego anche **RIPEMD-160** (1996, revisione di RIPEM-128 del progetto europeo RIPE): la funzione di compressione usa 512 bit di blocco e 160 passi (o meglio 80 con parallelismo 2) per calcolare 160 bit d'uscita.

Nel 2005 è stato evidenziato che anche questi algoritmi sono attaccabili più facilmente di quanto previsto. L'attenzione degli sviluppatori di applicazioni sicure si è quindi spostata su **Tiger** (Anderson e Biham, 1996), che ha un'impronta di 192 bit, e su **SHA-256, -384, -512** (2002, NIST), strutturalmente simili a SHA-1, ma ancora più robusti alle collisioni avendo rispettivamente 256, 384 e 512 bit d'uscita. In Europa si sta attualmente perfezionando e valutando **Whirlpool**, che fornisce impronte di 512 bit.

3 – Resistenza forte alle collisioni

Il controllo dell'autenticità di un messaggio mediante **firma digitale** impone di studiare un secondo contesto in cui l'attaccante può scegliere a caso anche **m**. Un impostore può, infatti, avvalersi di un famoso paradosso statistico.

Birthday paradox – *Nell'ipotesi che le date di nascita sono equiprobabili, è sufficiente scegliere a caso 253 persone per avere una probabilità maggiore di 0,5 che una di queste compie gli anni in un giorno prefissato. Sono invece sufficienti 23 persone se si vuole che due o più di queste compiano gli anni nello stesso giorno.*

Vediamo le conseguenze del paradosso nel caso di una funzione hash ideale con **n** bit d'uscita.

Vogliamo calcolare la probabilità $\mathbf{P_2(2^n, k)}$ di ottenere almeno due valori d'uscita identici immettendo in ingresso, per **k** volte, un valore scelto a caso.

Le sequenze così generabili sono $\mathbf{(2^n)^k}$; quelle con valori tutti diversi sono $\mathbf{2^n! / (2^n - k)!}$.

La probabilità di avere una stringa con almeno due simboli identici è dunque:

$$\mathbf{P_2(2^n, k) = 1 - 2^n! / ((2^n)^k \times (2^n - k)!) = 1 - (1-1/2^n) \times (1-2/2^n) \times .. . \times (1-(k-1)/ 2^n)}$$

Per avere una formulazione più chiaramente interpretabile, ricorriamo alla disuguaglianza $\mathbf{(1-x) \leq e^{-x}}$, che è valida per $x \geq 0$ e che risulta tra l'altro una buona approssimazione di (1-x, quando x ha valori piccoli.

Dopo qualche passaggio[10], tenendo anche conto che **k** è molto grande, si ottiene:

$$\mathbf{P_2(2^n, k) \approx 1 - exp(- 2^{-n} \times k^2/2)}$$

Esplicitando per **k** si ha:

$$\mathbf{k = sqrt(2\ ln(1/(1-P_2))) \times sqrt(2^n)}$$

ESEMPIO – Nel caso del paradosso, bisogna mettere **365** al posto di $\mathbf{2^n}$. Posto $\mathbf{P_2 = 0,5}$ si ottiene:

$$\mathbf{k = sqrt(2 \times ln(2)) \times sqrt(365) = 22,54}.$$

Una volta fissata la desiderata probabilità di successo, l'individuazione di due coincidenze qualsiasi richiede un numero di tentativi proporzionale a $\mathbf{2^{n/2}}$: l'algoritmo è quindi difficile, ma, a parità di **n**, enormemente più facile di quello che individua una collisione con un prefissato valore di hash.

Ritorniamo ora al problema di sicurezza posto dalla firma digitale.

L'impossibilità pratica di individuare una collisione (R19) ha portato a considerare l'impronta un sicuro **elemento identificativo del messaggio** da cui è stata calcolata e ad autenticare solo questa piccola stringa, a tutto vantaggio dell'efficienza. L'uso della coppia di trasformazioni **S, V** consente, infatti, al destinatario di rilevare e di scartare ogni messaggio generato dall'intruso.

C'è però ancora una minaccia: un mittente disonesto potrebbe voler disporre di **due messaggi con la stessa firma**, per poter in un secondo tempo sostenere di aver comunicato quello che gli conviene di più.

A tal fine il malintenzionato deve condurre il seguente attacco (*birthday attack*):

1. genera $\mathbf{2^{n/2}}$ messaggi uno diverso dall'altro, ma tutti ottenuti apportando lievi modifiche al primo messaggio;
2. calcola e memorizza l'hash di tutte queste versioni;
3. introduce lievi modifiche al secondo messaggio, calcola l'hash e controlla se è presente in memoria; in caso negativo ripete il passo 3: per il paradosso del giorno di compleanno può aspettarsi di trovare una coincidenza dopo $\mathbf{2^{n/2}}$ iterazioni.

ESEMPIO – Nel sito del corso è disponibile un programma che consente di condurre un attacco birthday su un'impronta di piccole dimensioni.

Si noti che il malintenzionato potrebbe essere anche il verificatore. Una volta ricevuto un messaggio autentico, con l'attacco del giorno del compleanno egli può, infatti, crearne uno di diverso significato e di uguale impronta ed escogitare un trucco (esiste e lo vedremo) per farsi firmare dal corrispondente anche questa versione.

Il rispetto di R20 richiede dunque funzioni hash crittografiche con **un numero di bit d'uscita due volte più grande del livello di sicurezza** che si vuole ottenere.

Attualmente, come si è già detto, si usano impronte di 160 bit, ma sarà opportuno passare al più presto ad algoritmi che forniscono 256 bit d'uscita.

4 – Unidirezionalità

L'**impossibilità di inversione** è richiesta alle funzioni hash, quando sono impiegate per proteggere la segretezza di un dato di cui è resa nota la sola impronta.

Un caso tipico in cui H deve essere *one-way* si ha quando un messaggio in chiaro m è autenticato da H(m||s), come abbiamo già segnalato a pag. 18.

[10] v. [9], pag.335; dal sito del Corso è scaricabile il notebook Birthday.nb sviluppato da Francesca Nocerino.

Se R21 non fosse vera, anche lo schema di firma digitale con appendice avrebbe una sottile vulnerabilità. L'intruso I sceglie un numero a caso **r** e calcola **x = V(r)**, ove **V**, non segreta, è la trasformazione di verifica della firma di un certo utente U. A questo punto I calcola **y =** $\mathbf{H^{-1}(x)}$ e si costruisce così una coppia **y, r** la cui paternità non potrebbe essere ripudiata da U. Lasciando perdere il fatto che **y** quasi sicuramente è privo di significato, se U avesse voluto autenticarlo, avrebbe proprio ottenuto come etichetta **r = s(H(y))**, ove **S** è la trasformazione che per ipotesi solo lui deve saper fare.

Il più efficace algoritmo di inversione deve dunque avere complessità **O(exp(n))**. Tale algoritmo esiste per qualsiasi funzione hash, appartiene alla categoria degli attacchi con forza bruta e richiede di provare uno dopo l'altro i possibili valori di **x**, fino a quando non si trova quello che ha l'hash desiderato.