Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Matematica

FUNZIONI HASH E SICUREZZA CRITTOGRAFICA

Tesi di Laurea in Crittografia

Relatore:
Chiar.mo Prof.
Davide Aliffi

Presentata da:
Camilla Valmorra

Sessione II
Anno Accademico 2012/13

Indice

Introduzione

La crittografia, dal greco *kryptos*, nascosto, e *graphein*, scrivere, è la scienza che si occupa dello studio delle scritture segrete. A partire degli anni '70 è diventata una branca della matematica e dell'informatica grazie anche all'utilizzo di tecniche di teoria dei numeri.
Nell'era dell'informazione la necessità di metodi sofisticati per la protezione dei dati è cresciuta sempre più e con essa anche i servizi elettronici.
Lo scambio su internet di informazioni riservate, come i numeri di carta di credito, è già una pratica comune, perciò proteggere i dati e i sistemi elettronici è cruciale.
Un problema molto comune è quello dell'autenticazione e dell'integrità dei documenti informatici, per questo si utilizza un cifrario a chiave pubblica e si cifra il documento con la propria chiave segreta in modo tale che chiunque possa verificarne l'autenticità utilizzando la chiave pubblica dell'utente firmatario. Per l'autenticazione di documenti di grandi dimensioni nasce però il problema del tempo di calcolo che con un algoritmo a chiave pubblica può essere oneroso. Entrano così in gioco le funzioni hash con le quali si può autenticare solo un riassunto del documento senza bisogno di autenticarlo tutto.
Hash, dall'inglese *to hash* sminuzzare, in americano indica il "battuto" di verdure e per estensione indica un composto eterogeneo cui viene data una forma incerta: *To make a hash of something*, vuol dire infatti creare confusione.
Le funzioni hash vengono utilizzate per generare una sorta di riassunto del documento; prendono in ingresso un messaggio di lunghezza variabile e producono in uscita un digest di messaggio di lunghezza fissa. Questo digest è strettamente legato al messaggio in entrata poichè ogni messaggio genera un digest univoco. Infatti, anche se si considerano due messaggi che differiscono tra loro solo per un carattere, le loro immagini hash saranno diverse.
Queste funzioni hash sono unidirezionali, cioè conoscendo il digest deve essere computazionalmente impossibile risalire al messaggio originale; ad esempio, è per questo che, quando si dimentica una password, non è possibile recu-

perarla ma solo cambiarla: infatti di una password viene memorizzata solo l'immagine hash. Inoltre deve essere impossibile trovare una collisione, ovvero due messaggi diversi che producono il medesimo digest.
Quest'ultima richiesta è molto forte, infatti sono stati trovati molti esempi di collisioni per alcune tra le più popolari funzioni hash.
Nel 1993 il NIST (National Institute of Standard and Tecnology) ha introdotto una funzione hash chiamata SHA-0, che però non fu adottata in molte applicazioni a causa della sua debolezza; questa fu successivamente modificata e corretta, nel 1995, con il nome di SHA-1.
SHA-1 è stata quella più usata in assoluto fra tutte le funzioni hash (insieme ad MD5) ed è stata impiegata in molte applicazioni e protocolli. Si tratta di una funzione hash standardizzata e libera, non protetta da copyright. Nel 2005 furono trovati dei difetti nella sicurezza di SHA-1, dovuti a possibili debolezze matematiche, e quindi si è preferito usare una sua variante, SHA-2 che è algoritmicamente equivalente alla precedente. Sebbene non siano ancora stati trovati attacchi efficienti, nel 2007 il NIST ha indetto una gara per la ricerca di una valida alternativa. Il 2 Ottobre 2012 è stata selezionata la funzione hash vincente, Keccak, ora conosciuta come SHA-3, una nuova hash standard ideata da un team di analisti italiani e belgi (Guido Bertoni, Joan Daemen, Michael Peeters e Gilles Van Assche).

Capitolo 1

Funzioni hash

1.1 Definizioni e proprietà

Definizione 1.1. Una *funzione hash* è una funzione (non iniettiva) che mappa una stringa di lunghezza arbitraria,il messaggio, in una stringa di lunghezza fissata detta digest.

$$h\colon \Sigma^* \longrightarrow \Sigma^n \quad , \quad \begin{array}{l} n\in \mathbb{N} \\ \Sigma\text{:=alfabeto} \end{array}$$

Osservazione 1.
Le funzioni hash possono essere generate da *funzioni di compressione*, cioè da funzioni che mappano stringhe di lunghezza fissata in stringhe di lunghezza fissata ma inferiore.

$$h\colon \Sigma^m \longrightarrow \Sigma^n \quad , \quad \begin{array}{l} m, n\in \mathbb{N}, \text{ con } m > n \\ \Sigma\text{:=alfabeto} \end{array}$$

Le funzioni hash e le funzioni di compressione giocano un ruolo molto importante in Crittografia, in particolare si richiede che abbiano le seguenti *proprietà crittografiche*:

1. dato un messaggio m, il digest di messaggio $h(m)$ deve poter essere calcolato molto rapidamente;

2. dato y, è computazionalmente impossibile trovare un m' tale che $h(m') = y$, cioè h è una *funzione unidirezionale* o *resistente alle controimmagini*. È perciò una funzione che non può essere invertita;

3. è computazionalmente impossibile trovare due messaggi m_1 e m_2, con $m_1 \neq m_2$, tali che $h(m_1)=h(m_2)$. In questo caso la funzione h è detta *fortemente resistente alle collisioni.*

La richiesta (3) a volte si può indebolire richiedendo che h sia *debolmente resistente alle collisioni*, cioè che dato x, sia computazionalmente impossibile trovare un $x' \neq x$ con $h(x') = h(x)$.
Questa proprietà è anche chiamata *resistenza alla seconda controimmagine.*

Esempio 1. Molte funzioni hash crittografiche possono essere descritte attraverso una semplice funzione che possiede molte delle proprietà fondamentali delle funzioni usate in pratica.
Si parte con un messaggio m di lunghezza L arbitraria e si spezza 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 $m = [m_1, m_2, ..., m_l]$, dove $l = [L/n]$ e l'ultimo blocco m_l è eventualmente riempito con degli zeri, in modo da avere n bit. Ora si può considerare la matrice che ha come righe m_1, ..., m_l.

$$\begin{bmatrix} m_{11} & m_{12} & ... & m_{1n} \\ m_{21} & m_{22} & ... & m_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ m_{l1} & m_{l2} & ... & m_{ln} \end{bmatrix}$$

Sommando per colonna modulo 2 si ottiene il vettore riga

$$h(m) = \begin{pmatrix} c_1 & c_2 & ... & c_n \end{pmatrix}.$$

Questa funzione hash prende in input un messaggio di lunghezza arbitraria e produce in output un digest di n bit. Tuttavia non è considerata crittograficamente sicura, poichè è facile trovare due messaggi che hanno lo stesso valore hash.
Quelle usate in pratica, di solito, utilizzano molte altre operazioni sui bit, in modo che sia più difficile trovare collisioni.

Osservazione 2. Le funzioni hash sono una famiglia di algoritmi che possono elaborare una qualunque mole di bit e che soddisfano i seguenti requisiti:

- l'algoritmo restituisce il *digest*, una stringa di numeri e lettere, a partire da un qualsiasi flusso di bit di qualsiasi dimensione (può essere un file ma anche una stringa);

- la stringa di output è univoca per ogni documento (se l'algoritmo è resistente alle collisioni) e ne è un identificatore, perciò l'algoritmo è utilizzabile per la firma digitale, insieme ad un sistema di crittografia a chiave pubblica;

- l'algoritmo non è invertibile, ossia non è possibile ricostruire il documento originale a partire dalla stringa che viene restituita in output, ovvero è una funzione *unidirezionale*.

Osservazione 3. *Principio dei cassetti*
Il principio dei cassetti, detto anche "*Legge del buco della piccionaia*", afferma che se n+k oggetti sono messi in n cassetti, allora almeno un cassetto deve contenere più di un oggetto. Un'altro modo di vedere il principio è che una piccionaia con m caselle può contenere al più m piccioni, se non se ne vogliono mettere più di uno in nessuna casella, un ulteriore volatile dovrà necessariamente condividere la casella con un suo simile.
Formalmente, il principio afferma che se A e B sono due insiemi finiti e B ha cardinalità strettamente minore di A, allora non esiste alcuna funzione iniettiva da A a B.

Proposizione 1.1.1. Non esiste una corrispondenza biunivoca tra il valore hash e il testo.

Cenno di dimostrazione
Dato che i testi possibili, con dimensione finita maggiore dell'hash, sono più degli hash possibili, per il *Principio dei cassetti*, ad almeno un hash corrisponderanno più testi.

Osservazione 4. Quando due testi producono lo stesso hash, si parla di *collisione*; la qualità di una funzione hash è misurata direttamente in base alla difficoltà nell'individuare due testi che generino una collisione.
Per sconsigliare l'utilizzo di algoritmi di hashing in passato considerati sicuri, è stato infatti sufficiente che un singolo gruppo di ricercatori riuscisse a generare una collisione. Questo è avvenuto ad esempio per gli algoritmi SNEFRU, MD2, MD4 ed MD5.
Un hash crittograficamente sicuro non dovrebbe permettere di risalire, in un tempo confrontabile con l'utilizzo dell'hash stesso, ad un testo che possa generarlo. Le funzioni hash ritenute più resistenti richiedono attualmente un tempo di calcolo per la ricerca di una collisione superiore alla durata dell'universo (immaginando di impiegare tutte le capacità computazionali ora disponibili in una ricerca per forza bruta).

Definizione 1.2. Due tipi di funzioni hash sono: *Modification Detection Code* (MDC) e *Message Authentication Code* (MAC).
Lo scopo degli MDC è di garantire la protezione d'integrità dei dati, cioè garantire che il messaggio m di cui si possiede il digest sia proprio quello che si sta leggendo.
Nel caso in cui si abbia a che fare con dei MAC, allora gli scopi sono non solo la protezione di integrità ma anche l'autenticazione dell'autore del messaggio. In quest'ultimo caso si utilizza una chiave condivisa k tra autore e destinatario, così che $d = h^k(m)$, dove d è il digest e m il messaggio.
Possibili attacchi ad un MAC, possono essere:

1. calcolare la chiave k a partire da un numero sufficiente di digest d_i;
2. conoscendo un sufficiente numero di $d_i = h^k(m_i)$, calcolare il valore corretto di $d_j = h^k(m_j)$, senza conoscere la chiave;
3. trovare due messaggi m e m', diversi tra loro, tali per cui $h^k(m) = h^k(m') = d$

Negli algoritmi MDC lo scopo è solitamente fare in modo che trovare due messaggi che collidono significhi calcolare almeno $2^{n/2}$ hash, dove n è la lunghezza dell'hash stesso, come si può vedere utilizzando il cosiddetto "attacco del compleanno" (si veda [2]).

L'attacco del compleanno può essere usato per trovare collisioni per le funzioni hash se il digest non è di lunghezza sufficientemente grande.

Esempio 2. *Attacco del compleanno*
Sia h una funzione hash con un output di n bit. Ci sono $N = 2^n$ output possibili. Si costruisce una lista $h(x)$ per $r = \sqrt{N} = 2^{n/2}$ scelte casuali di x, così ci si trova nella situazione di $r \approx \sqrt{N}$ persone con N possibili compleanni e quindi c'è una buona probabilità di avere due valori x_1 e x_2 con lo stesso digest, cioè una collisione. Se si considera una lista più lunga, la probabilità che ci sia una corrispondenza diventa molto alta.

Osservazione 5. Si può vedere che se gli output della funzione sono distribuiti in modo non uniforme, allora una collisione può essere trovata molto più velocemente. Il bilanciamento di una funzione hash quantifica la resistenza della funzione agli attacchi del compleanno e permette di stimare la vulnerabilità di funzioni hash come ad esempio SHA.
Le firme digitali possono essere vulnerabili ad un attacco del compleanno: un messaggio m è generalmente firmato prima calcolando $f(m)$, dove f è una funzione hash, poi usando una chiave segreta per firmare $f(m)$. Si possono

qundi creare tanti documenti simili tutti corretti e tutti con un digest diverso e uno, invece, modificato, che ha lo stesso digest di uno corretto. Con questi due digest uguali è possibile far firmare documenti falsi mostrando solo quelli corretti senza che nessun controllo se ne accorga.

Esempio 3. Supponiamo che Oscar voglia imbrogliare Luca inducendolo a firmare un contratto fraudolento.
Oscar prepara un contratto m corretto ed uno m' fraudolento, poi trova un certo numero di punti del contratto per cui m può essere modificato senza cambiarne il significato, ad esempio inserendo delle virgole, delle linee vuote, uno o due spazi dopo una frase, sostituendo dei sinonimi ecc. Combinando queste modifiche, Oscar può creare un notevole numero di varianti di m (dell'ordine di 2^k, dove k è il numero delle modifiche) che sono in pratica tutti contratti corretti. In maniera simile crea anche un gran numero di varianti del contratto fraudolento m'. Alla fine applica la funzione hash a tutte queste varianti finchè non ne trova una del contratto corretto e una di quello fraudolento che hanno lo stesso digest, cioè tali che $f(m) = f(m')$. A questo punto, Oscar presenta a Luca la versione corretta per la firma. Dopo che Luca lo ha firmato, Oscar prende la firma e la attacca al contratto fraudolento. Adesso la firma prova che Luca ha firmato il contratto fraudolento. La figura seguente mostra come si possa creare un gran numero di documenti con ugual significato ma con digest diverso.

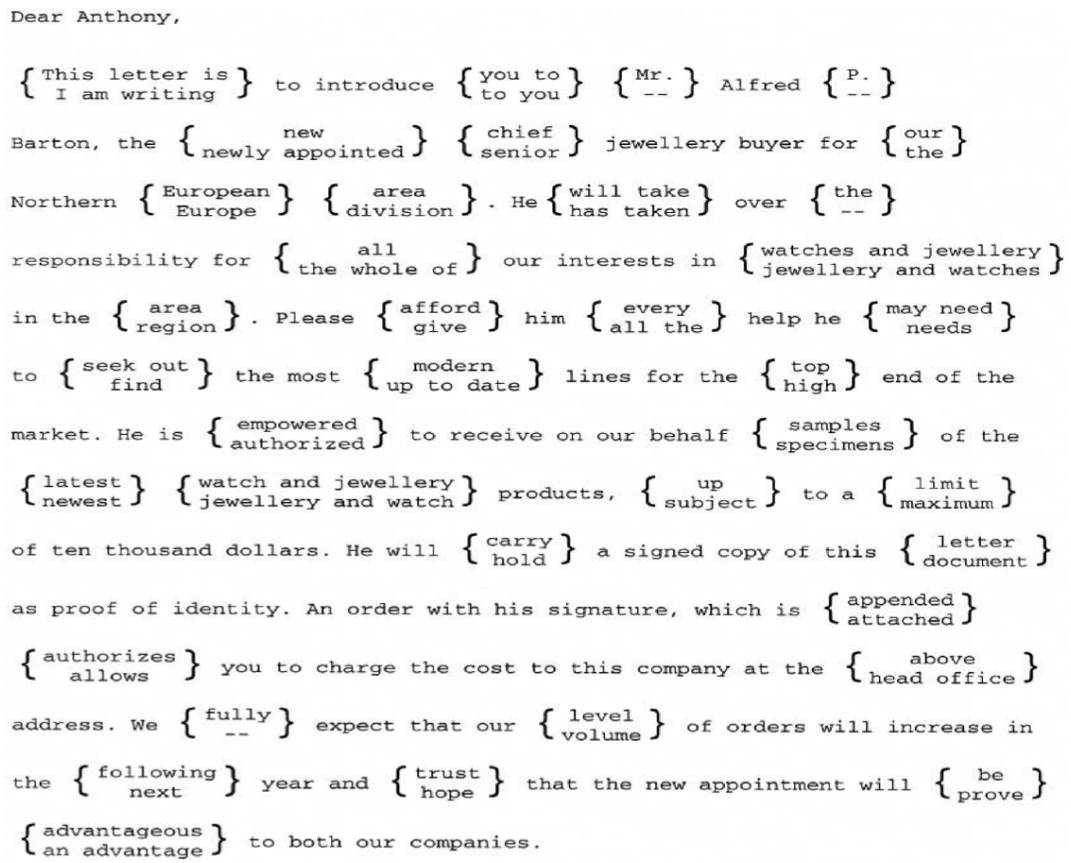

Figura 1.1: Varianti di uno stesso documento con digest diverso

Proposizione 1.1.2. Per evitare questo attacco la lunghezza dell'output di una funzione hash utilizzata in uno schema di firma digitale, deve essere grande abbastanza in modo che l'attacco del compleanno divenga computazionalmente impossibile. In genere si è nell'ordine del doppio dei bit necessari per prevenire un classico attacco a forza bruta.

1.2 Applicazioni

La lunghezza dei valori di hash varia a seconda degli algoritmi utilizzati, il valore più comunemente adottato è di 128 bit, che offre una buona affidabilità in uno spazio relativamente ridotto, tuttavia possono essere usate anche hash di dimensione maggiore.
Le funzioni hash sono una sorta di "*impronta digitale*" di lunghezza fissa di un messaggio, sono quindi utili in diversi settori.

Osservazione 6. Le funzioni hash possono essere utilizzate per il *controllo degli errori*: in questo caso la funzione viene utilizzata per identificare eventuali errori di trasmissioni dei dati.

Osservazione 7. Vengono utilizzate per garantire che il messaggio non sia stato modificato da un eventuale attaccante. Infatti, l'esecuzione dell'algoritmo su un testo anche minimamente modificato fornisce un digest completamente differente rispetto a quello calcolato sul testo originale, rivelando così la modifica.
Ad esempio viene utilizzato negli IDS (Intrusion Detection Systems) e negli antivirus per controllare l'integrità dei file critici di un sistema operativo.

Osservazione 8. Nella maggior parte dei casi, quando un sistema informatico richiede la memorizzazione di una password, questa non viene salvata in chiaro per motivi di sicurezza, ma viene memorizzato l'hash della password.

Esempio 4. In Windows, Linux e MacOs le password degli utenti sono salvate sotto forma di hash, quindi quando un utente scrive la propria password per accedere al sistema, ne viene calcolato il valore hash e confrontato con quello memorizzato nel sistema.

Osservazione 9. La crittografia a chiave pubblica è computazionalmente molto onerosa quindi solitamente viene usata solo per trasmettere la chiave di una crittografia simmetrica. Per le firme digitali sussiste un problema analogo. Si è risolto questo problema firmando solo un hash del documento, evitando così di firmarlo tutto. Questo ha anche il vantaggio di lasciare il documento in chiaro, leggibile anche a chi non dispone degli strumenti per verificare la firma digitale.

1.3 Trasformazione di Merkle-Damgård

Un metodo per costruire funzioni hash crittografiche resistenti alle collisioni è la trasformazione di Merkle-Damgård. Questa costruzione fu descritta da Ralph Merkle nella propria tesi di dottorato nel 1979.
Ralph Merkle e Ivan Damgård dimostrarono, separatamente, che questa struttura effettivamente funziona, e cioè che, con un appropriato schema e con una funzione di compressione resistente alle collisioni, la funzione hash che si ottiene è anch'essa resistente alle collisioni. La trasformazione consiste nel costruire una funzione hash a partire da una funzione di compressione unidirezionale resistente alle collisioni. Questo metodo rende possibile una conversione da una qualsiasi funzione hash di lunghezza fissata (ossia da una funzione di compressione), a funzioni hash vere e proprie, mantenendo la proprietà di essere resistenti alle collisioni.
La trasformazione di Merkle-Damgård è interessante anche da un punto di vista teorico in quanto mostra che la compressione di stringhe di lunghezza qualsiasi non è più difficile della compressione di stringhe di lunghezza fissata. Sia data una funzione hash resistente alle collisioni, di lunghezza fissata, che comprime il proprio input della metà; se la lunghezza dell'input è $l'(n)$=$2l(n)$, l'output ha lunghezza $l(n)$.

Definizione 1.3. *Costruzione di Merkle-Damgård*
Si denota con h una funzione hash di lunghezza fissata resistente alle collisioni, anche chiamata funzione di compressione, e la si usa per costruire una funzione hash H resistente alle collisioni che manda input di lunghezza qualsiasi in output di lunghezza fissata, $l(n)$.
Per trovare l'output si esegue il seguente processo:

- H: si pone in input una chiave s (non segreta) e una stringa x$\in${0,1} di lunghezza $L<2^{l(n)}$ e si iterano i passaggi seguenti:

  1. sia $B := \left\lceil \frac{L}{l} \right\rceil$, si completa x con degli zeri in modo da far sì che la sua lunghezza sia un multiplo di l. Si rappresenta il risultato come sequenza di blocchi x_1, ..., x_B di l-bit e si pone x_{B+1}=L, dove L è codificato usando esattamene l bit;
  2. sia z_0=0^l (Initialization Vector: vettore di l zeri);
  3. per i=1, ...,B+1 si calcola z_i=$h^s(z_{i-1}||x_i)$;
  4. l'output di H è z_{B+1}.

Si è limitata la lunghezza L di x a $2^{l(n)}$-1 così che possa essere codificata come un intero di lunghezza $l(n)$.

Il valore di z_0 è arbitrario e può essere sostituito con una qualsiasi costante.

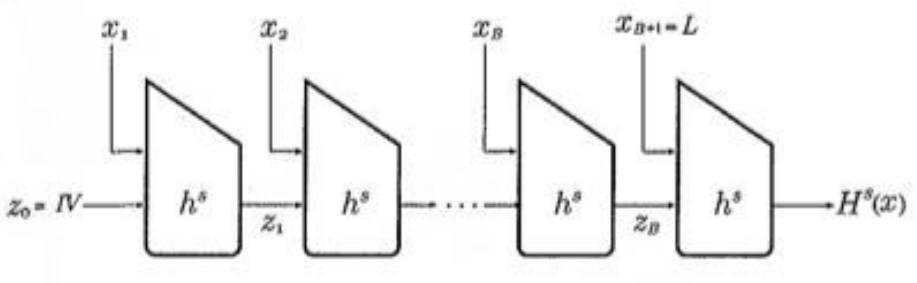

Figura 1.2: Trasformazione di Merkle-Damgård

Osservazione 10. La sicurezza della trasformazione di Merkle-Damgård segue dal fatto che, se due stringhe diverse x e x' collidono in H^s, allora ci devono essere valori intermedi distinti $z_{i-1}||x_i$ e $z'_{i-1}||x'_i$ nel calcolo di $H^s(x)$ e di $H^s(x')$, rispettivamente, tali che $h^s(z_{i-1}||x_i)$=$h^s(z'_{i-1}||x'_i)$.
Si può avere una collisione in H^s solo se c'è una collisione in h^s.

Teorema 1.3.1. *Se h è una funzione hash di lunghezza fissata, resistente alle collisioni, allora la funzione H, costruita con la Trasformazione di Merkle-Damgård, è una funzione hash resistente alle collisioni.*

Dimostrazione. Si mostra che per ogni s, una collisione in H^s porta ad una collisione in h^s. Siano x e x' due stringhe diverse, rispettivamente di lunghezza L e L' tali che $H^s(x) = H^s(x')$. Siano x_1, ..., x_B i B blocchi di x, e siano x'_1, ..., $x'_{B'}$ i B' blocchi di x'; ricordiamo che x_{B+1}=L e $x'_{B'+1}$=L'. Ci sono due casi da considerare:

1. $L \neq L'$: in questo caso l'ultimo passo del calcolo di $H^s(x)$ è $z_{B+1} := h^s(z_B||L)$ e l'ultimo passo del calcolo di $H^s(x')$ è $z'_{B+1} := h^s(z'_{B'}||L')$. Fino a quando si verifica $H^s(x) = H^s(x')$, si ha che $h^s(z_B||L) = h^s(z'_{B'}||L')$. Tuttavia L≠L', quindi $z_B||L$ e $z'_{B'}||L'$ sono due stringhe diverse che collidono per h^s.

2. $L = L'$: significa che $B = B'$ e $x_{B+1} = x'_{B'+1}$. Siano z_i e $z_{i'}$ i valori hash intermedi di x e x' durante il calcolo di $H^s(x)$ e di $H^s(x')$, rispettivamente. Finchè $x \neq x'$ ma $|x| = |x'|$, deve esistere almeno un indice i, con 1≤ i ≤B, tale che x_i≠x'_i.

Sia $i^* \leq B+1$, allora $z_B||x_{B+1}$ e $z'_B||x'_{B+1}$ sono due stringhe diverse che collidono per h^s perchè

$$h^s(z_B||x_{B+1}) = z_{B+1} = H^s(x) = H^s(x') = z'_{B+1} = h^s(z'_B||x'_{B+1}).$$

Se $i^* \leq B$, allora $z_{i^*} = z'_{i^*}$. Quindi, di nuovo, si avrà che $z_{i^*-1}||x_{i^*}$ e $z'_{i^*-1}||x'_{i^*}$ sono due stringhe differenti che collidono per h^s.

Segue che qualsiasi collisione nella funzione hash H^s, porta ad una collisione nella funzione hash a lunghezza fissata h^s.

Capitolo 2

Algoritmi SHA

Con il termine SHA (Secure Hash Algorithm), si indica una famiglia di cinque diverse funzioni crittografiche hash sviluppate a partire dal 1993 dall'NSA (National Security Agency) e pubblicate dal NIST (National Institute of Standards and Technology) nel 1995 come standard federale del governo degli Stati Uniti d'America. Come ogni algoritmo hash, l'SHA, partendo da un messaggio di lunghezza variabile, produce un *message digest* di lunghezza fissa.
La versione originale, spesso denominata SHA-0, conteneva una debolezza che fu in seguito scoperta dalla NSA e che portò ad un documento di revisione dello standard, l'SHA-1.
Gli algoritmi della famiglia SHA sono denominati SHA-1, SHA-224, SHA-256, SHA-384 e SHA-512: le ultime 4 varianti sono spesso indicate genericamente con SHA-2, per distinguerle dal primo.
SHA-1 produce un digest del messaggio di soli 160 bit, mentre gli altri producono digest di lunghezza in bit pari al numero indicato nella sigla. L'SHA-1 è l'algoritmo più diffuso della famiglia ed è utilizzato in numerose applicazioni e protocolli; la sua sicurezza è stata in parte compromessa dai crittoanalisti. Sebbene non siano ancora noti attacchi alle varianti SHA-2, esse hanno un algoritmo simile a quello di SHA-1 per cui sono stati intrapresi sforzi per sviluppare algoritmi di hashing alternativi e più sicuri.
Il 2 novembre 2007 venne annunciato nel *Federal Register* un concorso per la realizzazione di una nuova funzione SHA-3. Il 2 ottobre 2012 è stato proclamato vincitore della competizione pubblica l'algoritmo *Keccak*, ideato da un gruppo di crittografi italiani e belgi.

2.1 SHA-1

L'SHA-1 produce un hash di 160 bit usando una procedura iterativa. Il messaggio originale m viene spezzato in un insieme di blocchi di lunghezza fissata: m=$[m_1, m_2, ..., m_l]$, se necessario, al messaggio vengono concatenati alcuni bit di riempimento, in modo da completare 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 ottenunto nel round precedente: cioè, partendo da un valore iniziale X_0, si definisce X_j=$h'(X_{j-1}, m_j)$; l'X_l finale è il *digest* del messaggio.
Per costruire una funzione *hash* è necessaria una buona funzione di compressione; questa dovrebbe essere costruita in modo che ogni bit di input influenzi il maggior numero possibile di bit di output.
SHA-1 prende il messaggio originale e lo completa 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 ad un multiplo intero di 512 bit, meno 64 bit. Dopo la concatenazione degli 1 e degli 0, viene concatenata la rappresentazione a 64 bit della lunghezza T del messaggio. Così se il messaggio è di T bit, allora la concatenazione genera un messaggio formato da L=$[\mathrm{T}/512] + 1$ 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 ad uno.

Esempio 5. Se il messaggio originale è di 2800 bit, si aggiungono un "1" e 207 "0", in modo da ottenere un nuovo messaggio di lunghezza $3008 = 6*512-64$. Poichè $2800 = 101011110000_2$ in binario, si concatenano ancora 0 seguiti da 101011110000 in modo da ottenere un messaggio di lunghezza 3072. Questo viene spezzato in sei blocchi di lunghezza 512, $m_1, m_2, ..., m_6$ $(L = 6)$.

Osservazione 11. Prima di descrivere l'algoritmo hash, occorre definire alcune 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, o massimo bit a bit;

3. X$\oplus$Y : somma modulo 2 bit a bit;

4. $\neg$X : cambia gli 0 in 1 e gli 1 in 0;

5. X+Y : somma di X e Y modulo 2^{32}, dove X e Y sono considerati come interi modulo 2^{32};

6. X↩r : scorrimento circolare di X a sinistra di r posizioni (dove la parte iniziale diventa la parte finale).

Occorrono anche le funzioni, definite per $t = 0, 1, ..., 79$;

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

e le costanti $K_0, ..., K_{79}$ così definite

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

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

$$\begin{gathered} 0{=}0000, \quad 1{=}0001, \quad 2{=}0010 \quad ..., \quad 9{=}1001, \\ \mathrm{A}{=}1010, \quad \mathrm{B}{=}1011, \quad ..., \quad \mathrm{F}{=}1111 \end{gathered}$$

per cui ogni costante è una stringa di 32 bit.

2.1.1 Algoritmo

Definizione 2.1. L'algoritmo SHA-1 si può riassumere nei seguenti quattro punti fondamentali:

1. si considera un messaggio m e lo si concatena con dei bit in modo da ottenere un messaggio y della forma $y{=}m_1||m_2||...||m_L$, dove ogni m_i è formato da 512 bit;

2. si inizializzano H_0=67452301, H_1=EFCDAB89, H_2=98BADCFE, H_3=10325476, H_4=C3D2E1F0;

3. per i=1, ..., $L-1$, si eseguono i seguenti passaggi:

   (a) $m_i{=}W_0||W_1||...||W_{15}$, dove ogni W_j è formato da 32 bit.

   (b) Per $t = 16, ..., 79$ siano

   $$W_t{=}(W_{t-3}\oplus W_{t-8}\oplus W_{t-14}\oplus W_{t-16})\hookleftarrow 1.$$

(c) Si pone A=H_0, B=H_1, C=H_2, D=H_3, E=H_4;

(d) Per t=0, ..., 79 si eseguono i seguenti passi in successione:

$$\text{T=(A}\hookleftarrow 5)+ f_t\text{(B,C,D)+E+}W_t\text{+}K_t$$
$$\text{E=D,} \quad \text{D=C,} \quad \text{C=(B}\hookleftarrow 30), \quad \text{B=A,} \quad \text{A=T.}$$

(e) Si pone

$$H_0\text{=}H_0\text{+A,} \quad H_1\text{=}H_1\text{+B,} \quad H_2\text{=}H_2\text{+C,} \quad H_3\text{=}H_3\text{+D,}$$
$$H_4\text{=}H_4\text{+E.}$$

4. In output viene fornito il valore hash $H_0||H_1||H_2||H_3||H_4$ di 160 bit.

Osservazione 12. Il cuore dell'algoritmo è il punto 3, sono tutte operazioni semplici e molto veloci. Si noti che la procedura fondamentale è iterata il numero di volte necessario per riassumere l'intero messaggio, ossia L volte. Questa procedura iterativa rende l'algoritmo molto efficiente in termini di lettura ed elaborazione del messaggio.

Osservazione 13. 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 nel modo descritto al punto 2 dell'algoritmo.
Una volta elaborato il blocco m_1 del messaggio, il registro X_1 viene aggiornato per generare un registro X_2 e così via.
L'algoritmo ripete le operazioni su ogni blocco m_j da 512 bit, per ognuno di essi il registro X_j viene copiato nei sottoregistri A, B, C, D, E. Il primo blocco di messaggio, m_1, viene tagliato e rimescolato per ottenere $W_0, ..., W_{79}$, questi sono dati in pasto ad una sequenza di quattro round, corrispondenti ai quattro intervalli $0 \leq t \leq 19$, $20 \leq t \leq 39$, $40 \leq t \leq 59$, $60 \leq t \leq 79$. Ogni round prende come input il valore corrente del registro X_0 e dei blocchi W_t per quell'intervallo e opera su di essi per venti iterazioni, cioè il contatore t scorre lungo i venti 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 stesse 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 del 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 così per ogni blocco 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 seguente è rappresentata l'operazione della funzione di compressione h' sul j-esimo blocco di messaggio, m_j, usando il registro X_j. Dopo aver completato tutti gli L blocchi di messaggio, l'output finale è il digest di messaggio di 160 bit.

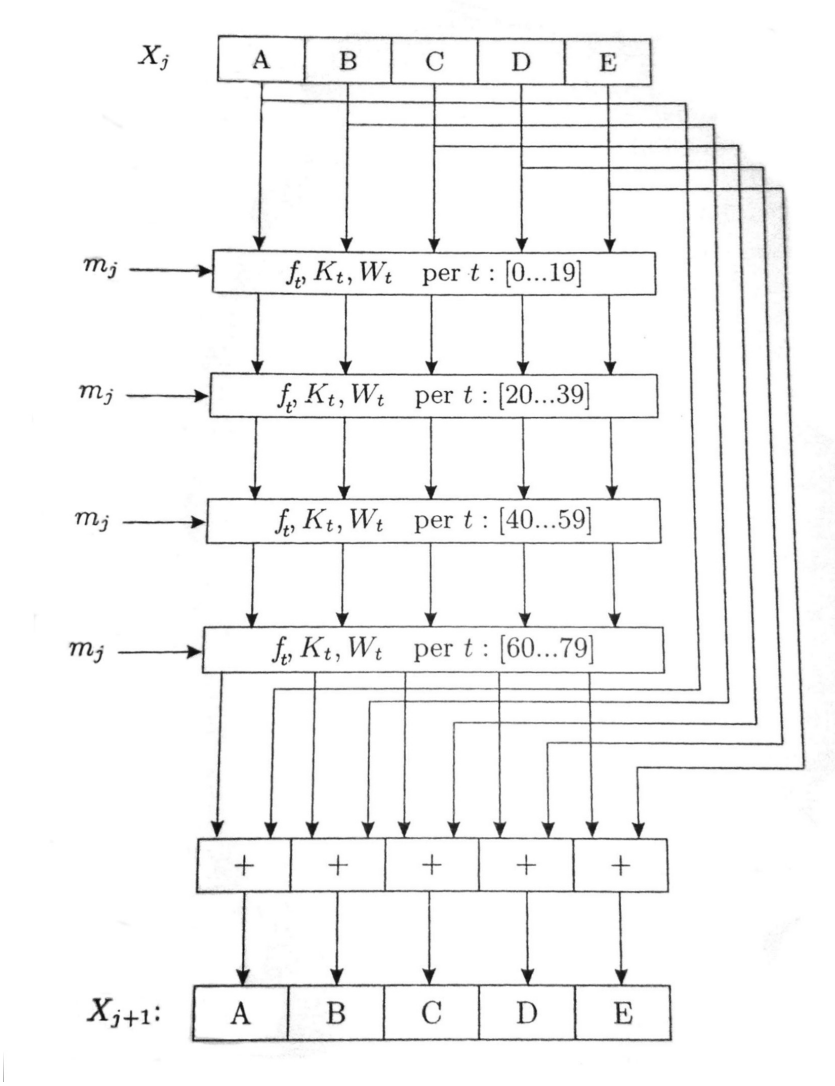

Figura 2.1: Operazioni svolte da SHA-1 su un singolo blocco di messaggio m_j

Il blocco fondamentale dell'algoritmo è l'insieme di operazioni che hanno luogo sui sottoregistri nel passo 3d, queste operazione sono rappresentate nella figura seguente.

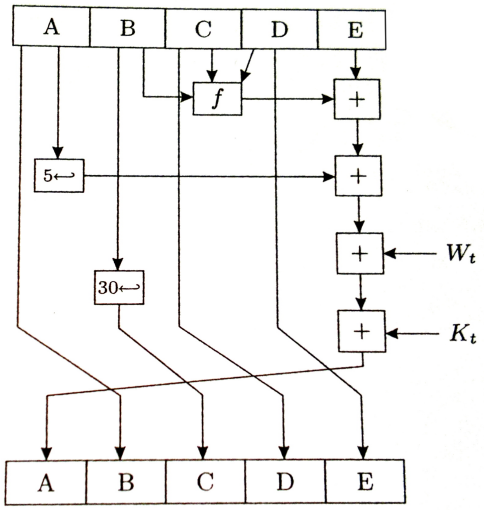

Figura 2.2: Operazioni che hanno luogo su ognuno dei sottoregistri in SHA-1

SHA-1 usa anche complicate operazioni di rimescolamento che sono eseguite da f_t e dalle costanti K_t.

Esempio 6. Vediamo ora alcuni esempi di *digest* generati da SHA-1 di alcune frasi:

SHA-1("Il cuore dell'algoritmo è il punto centrale") =
6106bc2b99fea092f6a596332d9319ee6dc63dae
SHA-1("Il cuore dell'algoritmo è il punt*i* centrale") =
dc68a87f792ff736c5acb010e38e0be2fb90cd65

Si può notare come, cambiando una sola lettera, cambi completamente l'hash della frase a causa di una reazione a catena nota come effetto valanga. La stringa nulla è rappresentata da:

SHA-1(" ") = da39a3ee5e6b4b0d3255bfef95601890afd80709

Osservazione 14. SHA-1 fu inizialmente usato anche in Microsoft Office 2007. Prima che un documento di Word, Excel e Power Point, protetto da una password venga aperto, la password viene convertita 50000 volte con chiave a 128 bit. Questo rallenta considerevolmente la velocità di un possibile

attacco a forza bruta e di conseguenza anche la probabilità di aprire un documento protetto da password. Le conversioni di SHA-1 vengono usate solo quando viene scelta la password per aprire il documento, in quanto questo è l'unico tipo di password che ne permette la cifratura. In Microsoft Office 2010 il numero di conversioni è stato raddoppiato e se ne effettuano 100000.

Nel 2005 furono trovate debolezze matematiche nell'algoritmo di SHA-1. Ci fu quindi bisogno di un nuovo algoritmo standard, l'SHA-2 con algoritmo molto simile ad SHA-1, ma resistente agli attacchi che avevano reso debole il precedente.

2.2 SHA-2

SHA-2 è formato da una famiglia di quattro funzioni hash: SHA-224, SHA-256, SHA-384, SHA-512. Sono state sviluppate dall'*NSA* e pubblicate dal NIST nel 2001 come *Standard Federale dei processi informatici degli Stati Uniti*. SHA-2 include cambiamenti sostanziali rispetto a SHA-1; le quattro funzioni hash che compongono SHA-2 hanno output rispettivamente di 224, 256, 284 e 512 bit. Gli ultimi tre algoritmi furono ufficializzati come standard nel 2002, mentre l'SHA-224 fu introdotto nel 2004. SHA-256 e SHA-512 lavorano rispettivamente con dati di 32 e 64 bit, utilizzano un numero diverso di rotazioni e di costanti addizionali, ma la loro struttura è praticamente uguale. Gli algoritmi SHA-224 e SHA-384 sono solo versioni troncate dei due precedenti, con hash calcolati con diversi valori iniziali.
Tutte queste varianti sono state brevettate dal governo statunitense ma rilasciate con licenza libera in modo che chiunque possa accedervi sia per lo studio sia per l'utilizzo.
Gli algoritmi SHA-2, a differenza di SHA-1, non hanno ricevuto molta attenzione dai crittoanalisti, per cui la loro sicurezza in campo crittografico non è stata del tutto provata; nel 2003 Gilbert e Handschuh hanno studiato queste nuove varianti e non hanno trovato vulnerabilità.
La sicurezza dell'algoritmo dipende interamente dalla capacità di produrre un valore diverso per ogni specifico input. Quando una funzione hash produce lo stesso *digest* per due diversi dati, allora si dice che c'è una collisione, perciò maggiore è la resistenza alle collisioni e più efficiente sarà l'algoritmo (si veda esempio 3).
Nonostante SHA-2 sia presumibilmente più sicura di SHA-1, non viene utilizzata maggiormente; questo probabilmente è dovuto al fatto che SHA-2 non è supportata da sistemi operativi come Windows XP con service pack 2 e anche più recenti. La mancanza di urgenza nell'inserire la compatibilità con SHA-2 potrebbe essere stata causata dal desiderio di attendere la standardizzazione di SHA-3.
SHA-256 viene anche usato per autenticare i pacchetti software di *Linux Debian*, e per maggior sicurezza ora i fornitori di Unix e Linux stanno iniziando ad usare 256 e 512 bit per l'hashing delle password. SHA-2 comprende gli algoritmi di sicurezza richiesti dalla legge del Governo degli Stati Uniti per certe applicazioni quali l'uso con altri algoritmi crittografici e protocolli per la protezione di informazioni riservate. SHA-1 è stato fortemente sconsigliato per l'uso governativo, il *NIST* affermò anche che dal 2010 tutte le agenzie federali avrebbero dovuto smettere di usare SHA-1 per le applicazioni che richiedono la resistenza alle collisioni e che avrebbero dovuto usare la famiglia SHA-2.

2.3 SHA-3

Poichè negli ultimi anni molti attacchi ad algoritmi hash sono andati a buon fine, nel novembre 2007 il $NIST$ decise di indire una competizione a livello internazionale ad accesso libero per implementare una nuova versione dell'algoritmo SHA che fornisse un maggior livello di resistenza agli attacchi rispetto alle precedenti versioni. A fine ottobre 2008 avevano già ricevuto 64 algoritmi da tutto il mondo, arrivarono all'ultimo round solo 5 di essi: *Blake*, *Grostl*, JH, $Keccak$ e *Skein*. Il $NIST$ ospitò tre conferenze sugli algoritmi per ottenere feedback pubblici e, basandosi sui commenti e sulle review dei candidati, il 2 ottobre 2012 proclamò l'algoritmo $Keccak$ vincitore del contest.
$Kaccak$ è stato creato da Guido Bertoni, Joan Daemen, Gilles Van Assche e Michael Peeters, un team di crittografi belgi e italiani.

Osservazione 15. Il $NIST$ scelse $Keccak$ per il suo design elegante, gli ampi margini di sicurezza, la buona performance, l'eccellente efficienza nell'implementazione hardware e la sua flessibilità.
$Keccak$ usa una nuova costruzione detta "*a spugna*", basata su permutazioni fisse che possono essere facilmente regolate e che possono generare *digest* lunghi o corti in base alle necessità. I creatori avevano anche inserito nell'algortmo la possibilità di generare crittografia autenticata.

Definizione 2.2. Le $funzioni$ "*spugna*″ sono una generalizzazione del concetto di funzioni crittografiche con output infiniti, la loro costruzione richiama permutazioni random. Queste funzioni possono eseguire operazioni crittografiche quasi simmetriche: dal ricavare un *digest*, alla generazione di un numero pseudorandom, all'autenticazione crittografica.

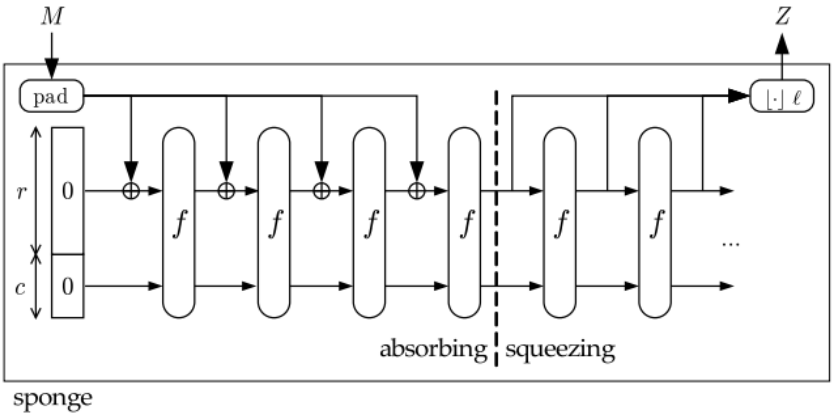

Figura 2.3: Struttura di una funzione *spugna*

Nelle funzioni "*spugna*" l'input viene "*assorbito*" ad una velocità fissata e l'output viene rilasciato ("*squeezed*") alla stessa velocità. Hanno una struttura più generale di una funzione hash, con input di lunghezza variabile e output di lunghezza arbitraria basata su una permutazione (f) di lunghezza fissa che opera su un fissato numero di bit (b),che è anche l'ampiezza dello "*state*". Questa permutazione viene eseguita bit a bit e opera su $b{=}r + c$, dove r indica la velocità e c la capacità.
Il messaggio in input viene suddiviso in blocchi da r bit (se necessario vengono completati con degli 0), poi la funzione *spugna* assorbe i blocchi e rilascia l'output. Nella fase di assorbimento, ad ogni blocco dato in input, viene applicata la somma bit a bit ai primi r bit dello "*state*" e poi l'intero vettore di stato viene permutato da f. Una volta processati tutti i blocchi del messaggio, la funzione *spugna* passa alla fase di "rilascio". In questa fase i primi r bit dello "state" vengono rilasciati come blocchi di output e tra un rilascio e l'altro viene nuovamente applicata f all'intero vettore di stato. Il numero di blocchi in output viene scelto arbitrariamente. Gli ultimi c bit dello "state" non dipendono mai direttamente dai blocchi di input e non vengono mai rilasciati come output nella fase di rilascio, ma vengono modificati solo dalla permutazione f.

Osservazione 16. Nella costruzione dell'algoritmo con le funzioni *spugna*, $Keccak$ richiama una delle sette permutazioni denominate $Keccak-f[b]$, con $b{=}25$, 50, 100, 200, 400, 800, 1600. Per il contest per SHA-3, il team propose la permutazione più grande, $Keccak - f[1600]$. Possono comunque essere usate anche permutazioni più piccole in alcuni ambienti. Ogni permutazione consiste nell'iterazione di una funzione semplice, simile alla cifratura di un blocco, senza l'utilizzo di una chiave. La scelta di operazioni bit a bit è limita-

ta alle seguenti operazioni: *disgiunzione esclusiva* (XOR), moltiplicazione modulo 2 bit a bit (AND), negazione logica (NOT) e rotazioni.

Osservazione 17. In $Keccak$ la funzione principale è una permutazione $Keccak - f[b]$ scelta fra le sette sopra citate. Lo "*state*" è organizzato in matrici di ampiezza 5x5, dove ogni array ha lunghezza w=1, 2, 4, 8, 16, 32, 64. Quando viene implementato in un processore a 64 bit, una riga di $Keccak - f[1600]$ può essere rappresentata come un'istruzione da 64 bit per una CPU.
Se si applica la costruzione a "*spugna*" alla funzione $Keccak - f[r + c]$ e si applica uno specifico completamento del messaggio di input ,si ottiene la "$funzione\ spugna$" $Keccak[r, c]$, con c che rappresenta la capacità e r la velocità.
Lo scopo è quindi quello di costruire una permutazione f che non abbia proprietà sfruttabili per la crittoanalisi. Per costruirne una resistente hanno pensato di realizzarla come una permutazione iterata, come un blocco cifrato con una sequenza di round identici che consistono ognuno di semplici passi. Nonostante ciò, non ci sono chiavi, ci sono costanti per ogni round invece di chiavi e le permutazioni inverse possono non essere efficienti. Inoltre hanno fatto sì che non ci fossero grandi probabilità di correlazioni tra input e output e che le permutazioni fossero immuni alla crittoanalisi integrale, agli attacchi algebrici e agli attacchi di spostamento e di sfruttamento di simmetria.

Esempio 7. SHA-3 in $Keccak - f[r, c]$ utilizza r=1088 e c=512 con un'ampiezza di permutazione di (b=$r + c$) b=1600 ed una sicurezza pari a 256.
Un'applicazione un po' più "*leggera*" può essere fatta con r=40 e c=160 perciò con un'ampiezza di permutazione di 200 e una sicurezza di 80, la stessa di SHA-1.

Osservazione 18. $Keccak$ può essere usato per le firme digitali, per preservare l'integrità dei dati, per l'identificazione dei dati (negli antivirus online o nel peer-2-peer), oppure anche per creare hash random o per conservare e verificare password.

Capitolo 3

Resistenza alle collisioni e sicurezza

Se si suppone che tra r persone ci siano N 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, cioè ci si aspetta una collisione (o k-collisione). Si è notato che la natura iterativa della maggior parte degli algoritmi hash li rende meno resistenti alle collisioni di quanto ci si possa aspettare. Ciò è stato messo in evidenza da *Joux* [8], che diede alcune implicazioni riguardo le proprietà delle funzioni hash concatenate.

3.1 Collisioni

Ci sono sostanziali differenze tra le funzioni hash standard e quelle resistenti alle collisioni; nelle prime, il bisogno di minimizzare le collisioni nel settaggio della struttura dei dati, diventa una richiesta obbligatoria per evitare collisioni nell'impostazione della crittografia. Inoltre per le strutture dati si può assumere che gli input siano scelti indipendentemente dalla funzione hash e senza alcun intento di causare collisioni. Per la crittografia invece si ha a che fare con un avversario che potrebbe scegliere degli elementi che dipendono dalla funzione hash, con l'*intento* di causare collisioni. Ciò significa che le richieste che vengono fatte per le funzioni hash usate in crittografia sono molto più restrittive di quelle che si fanno per le analoghe fatte per le strutture dati e quindi le funzioni hash resistenti alle collisioni sono molto più complicate da costruire.

Definizione 3.1. Come accennato in precedenza, si chiama *collisione*, in una funzione hash H, una coppia di input distinti x e x' tali che $H(x)=H(x')$. In questo caso si dice anche che x e x' collidono in H.

Osservazione 19. Solitamente si è interessati a funzioni H che hanno dominio infinito e immagine finita, in questo caso devono esistere collisioni per il *principio dei cassetti*, e la richiesta sarà dunque solo quella che le collisioni debbano essere computazionalmente difficili da trovare.
A volte invece si considerano funzioni H in cui sia il dominio che il codominio sono finiti, in questo caso si è interessati alle funzioni che comprimono l'input. Se la compressione non è richiesta, allora non ha senso la richiesta di resistenza alle collisioni. Ad esempio le funzioni identiche sono banalmente resistenti alle collisioni.

Osservazione 20. Formalmente si avrà a che fare con una $famiglia$ di funzioni hash (H) indicizzata da una "*chiave*" s, che hanno in input due dati, la chiave e la stringa x, in output invece si ha una stringa $H_s(x)=H(s,x)$. L'esigenza è che debba essere difficile trovare collisioni H_s per una chiave s generata in maniera casuale. La chiave non è una chiave crittografica e non viene mantenuta segreta.

3.2 Resistenza alle collisioni

Se una funzione hash producesse output casuali di n bit, quindi $N=2^n$ sarebbero i valori possibili, e si calcolassero i valori della funzione hash $r=2^{n(k-1)/k}$, ci si potrebbe aspettare di ottenere una collisione. Tuttavia spesso si possono ottenere collisioni molto più facilmente come conseguenza del fatto che le funzioni hash sono implementate con algoritmi iterativi (si veda Osservazione 1, Capitolo 1).

Definizione 3.2. Una $funzione$ hash H si dice *resistente alle collisioni* se è impossibile per qualsiasi algoritmo polinomiale probabilistico trovare collisioni in H.

Esempio 8. Per testare la resistenza di una funzione hash si può eseguire il seguente esperimento:

1. viene generata una chiave s;
2. all'avversario A viene data la chiave s e l'avversario genera due output x e x' (se la funzione hash è una funzione a lunghezza fissata per input di lunghezza $l'(n)$, allora si richiederà che x, $x' \in \{0,1\}^{l'(n)}$);

3. l'output sarà 1 se e solo se $x \neq x'$ e $H_s(x) = H_s(x')$.

In questo caso diciamo che A ha trovato una collisione.

Osservazione 21. La definizione di resistenza alle collisioni indica perciò che nessun avversario efficiente possa trovare una collisione nel modo dell'esempio sopracitato, eccetto che con una probabilità insignificante. È quindi una richiesta forte di sicurezza quella della resistenza alle collisioni ed è difficile da ottenere, tuttavia in alcune applicazioni basta fare affidamento su richieste meno ardue. Quando si considerano funzioni hash crittografiche sono solitamente da considerare tre livelli di sicurezza:

1. resistenza alle collisioni;

2. resistenza alla seconda controimmagine (una funzione hash è tale se dati s e x è impossibile per un avversario, in un tempo polinomiale probabilistico, trovare $x' \neq x$ tali che $H_s(x')=H_s(x)$);

3. resistenza alla controimmagine (cioè, dati s e $y=H_s(x)$, ma non x stesso, per un x scelto a caso, è impossibile per un avversario, in un tempo polinomiale probabilistico, trovare un valore x' tale che $H_s(x')=y$. Significa che H_s è una funzione unidirezionale).

Ogni funzione che è resistente alle collisioni è anche resistente alla seconda controimmagine. Intuitivamente se x è noto, l'avversario può trovare un $x' \neq x$ per il quale si verifica che $H_s(x')=H_s(x)$, cioè può trovare una collisione. Allo stesso modo, qualsiasi funzione resistente alla seconda controimmagine è anche resistente alla controimmagine. Questo è dovuto al fatto che se fosse possibile invertire y e trovare un x' tale che $H_s(x')=y$, allora sarebbe possibile, considerando un x dato, calcolare $y:=H_s(x)$ e poi invertire y per ottenere x': con grosse probabilità si avrebbe $x \neq x'$ e in questo caso si sarebbe trovata una seconda controimmagine.

Osservazione 22. Una differenza sostanziale tra le funzioni hash resistenti alle collisioni usate nella pratica e quelle descritte in precedenza sta nel fatto che, generalmente, si usano funzioni senza chiave. Questo significa che viene definita una funzione hash H fissata e nessuna chiave (s) viene utilizzata insieme ad essa. Solo da un punto di vista teorico è necessario inserire una chiave.
Il massimo che si può richiedere ad una funzione hash senza chiave è che non esista un algoritmo che richieda un ragionevole intervallo di tempo e trovi una collisione in H.
In pratica questa sorta di garanzia di sicurezza è sufficiente.

Le funzioni hash con la chiave hanno dei vantaggi: se viene trovata una collisione (anche impiegando grandi risorse in molti anni), allora H non è più a tutti gli effetti una funzione resistente alle collisioni, e deve essere rimpiazzata; invece se H è una funzione con chiave, allora una collisione per H_s trovata con un attacco a forza bruta non necessariamente renderebbe più facile trovare una collisione in $H_{s'}$ per una nuova chiave s'. Perciò, finchè la chiave viene rigenerata ogni volta, H può venire riutilizzata.
Tuttavia non si può sperare di provare la resistenza alle collisioni per le funzioni hash usate nella pratica, in particolare perchè sono senza chiave. Ciò non significa però che le condizioni di sicurezza non siano valide tutte quando queste funzioni vengono usate. La sicurezza che dipende dalla resistenza alle collisioni dimostra che, se la costruzione, con certe condizioni, può essere "forzata" da qualche avversario in un tempo polinomiale, allora una collisione si può trovare nella funzione hash nello stesso tempo polinomiale.

Osservazione 23. Per le costruzioni pratiche, l'attacco del compleanno dà un limite inferiore alla lunghezza dell'output di una funzione hash che deve essere rispettato per poter avere un accettabile livello di sicurezza. Se le funzioni hash fossero resistenti alle collisioni contro avversari che agiscono in tempi dell'ordine di 2^l, allora la lunghezza dell'output della funzione dovrebbe essere di almeno $2l$ bit.
Quindi una buona funzione hash resistente alle collisioni in pratica dovrebbe avere un output di lunghezza almeno 160 bit, ciò significa che l'attacco del compleanno impiegherebbe un tempo pari a 2^{80}, impossibile da raggiungere al giorno d'oggi.

Esempio 9. Due funzioni hash famose sono $MD5$ e $SHA-1$. A causa di attacchi recenti però, $MD5$ non è più sicura e non dovrebbe essere usata in nessuna applicazione che richiede resistenza alle collisioni.
Entrambe le funzioni per prima cosa definiscono delle funzioni di compressione che comprimono gli input in stringhe di lunghezza fissata (queste funzioni di compressione sono funzioni resistenti alle collisioni), poi la trasformazione di $Merkle-Damgård$ (o una molto simile) viene applicata alla funzione di compressione in modo da ottenere una funzione hash resistente alle collisioni per input con lunghezza qualsiasi.
In molte funzioni hash, come ad esempio in SHA-1, c'è una funzione di compressione h' che opera su input di lunghezza fissata. Inoltre c'è un valore iniziale IV fissato. Il messaggio viene completato per ottenere il formato desiderato e poi si eseguono le seguenti istruzioni:

1. si spezza il messaggio M in blocchi $M_1, M_2, ..., M_l$;
2. si pone H_0 uguale al valore iniziale IV;

3. per i=1, 2, ..., l , si pone H_i=$f(H_{i-1}, M_i)$;
4. $H(M)$=H_l.

In SHA-1, la funzione di compressione è rappresentata in Figura 2.2 con funzione di compressione h'=f.
Per ogni iterazione, prende un input $A||B||C||D||E$ di 160 bit dall'iterazione precedente insieme ad un blocco di messaggio m_i di lunghezza 512 e fornisce in output una nuova stringa $A||B||C||D||E$ di lunghezza 160.
Se si suppone 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 $f(h_0, m_0)$=$f(h_0, m'_0)$.
Un secondo attacco del compleanno trova due blocchi m_1 e m'_1 con $f(h_1, m_1)$=$f(h_1, m'_1)$. Si pone poi

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

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 finchè non si hanno t coppie di blocchi $m_0, m'_0, m_1, m'_1, ..., m_{t-1}, m'_{t-1}$, dove t è un intero da determinare.
Ognuno dei 2^t messaggi (|| indica la concatenazione di stringhe)

$m_0||m_1||...||m_{t-1}$
$m'_0||m_1||...||m_{t-1}$
$m_0||m'_1||...||m_{t-1}$
$m'_0||m'_1||...||m_{t-1}$
.....
$m'_0||m_1||...||m'_{t-1}$
$m_0||m'_1||...||m'_{t-1}$
$m'_0||m'_1||...||m'_{t-1}$

(tutte le possibili combinazioni tra gli m_i e gli m'_i) ha lo stesso valore hash a causa della natura iterativa dell'algoritmo.
Ogni volta che si calcola $f(m, h_{i-1})$, 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 $t2^{n/2}$ passi e ha un tempo di esecuzione atteso di circa una costante per $tn2^{n/2}$.

Esempio 10. Utilizzando l'esempio precedente, se si ha t=2, allora per trovare quattro messaggi con lo stesso valore hash viene impiegato circa il doppio del tempo che verrebbe impiegato per trovarne solo due. Se l'output della funzione hash fosse realmente casuale, invece che prodotto da un algoritmo iterativo, la procedura precedente 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.

Osservazione 24. Una conseguenza interessante del rischio di facili collisioni in funzioni hash iterative è l'inutilità dei tentativi di migliorare le funzioni concatenando i loro output. Prima di *Joux* si pensava che la concatenazione di due funzioni hash H_1 e H_2

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

fosse una funzione hash significativamente più robusta delle singole funzioni H_1 e H_2. Questo avrebbe permesso di usare funzioni hash deboli per costruirne di più robuste. Tuttavia ora sembra che questo non sia vero.
Supponiamo che l'output di H_i sia di n_i bit, per i=1, 2, e che H_1 venga calcolata con un algoritmo iterativo. Non si fanno ipotesi su H_2. Allora 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. Poichè questi messaggi hanno tutti lo stesso valore in H_1, si ha una collisione per $H_1||H_2$. Il tempo per trovare questa collisione 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 $2^{(n_2+n_2)/2}$ che un attacco del compleanno standard impiegherebbe sulla funzione hash concatenata.

Osservazione 25. Per ottenere la stima $n_2n_1 2^{n_1/2} + n_2 2^{n_2/2}$ per il tempo di esecuzione si devono usare $\frac{1}{2}n_2n_1 2^{n_1/2}$ 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 valuta 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 da H_2. Questo dà il termine $n_2 2^{n_2/2}$ nel tempo di esecuzione stimato.
Alla luce di tutto ciò si ha che la lunghezza dell'output di $MD5$ è di 128 bit e quella di $SHA-1$ è di 160 bit. Inoltre, il fatto che la lunghezza dell'output di $SHA-1$ sia maggiore di quella di $MD5$, fa sì che il generico attacco

del compleanno risulti più difficoltoso da mettere in atto; se per $MD5$ un attacco del compleanno richiederebbe circa $2^{128/2}$=2^{64} calcoli, per $SHA-1$ ne servirebbero $2^{160/2}$=2^{80}.

3.3 Sicurezza

Le vulnerabilità nelle funzioni hash sono oggetto di intensa attività di ricerca per far in modo di allertare con sufficiente anticipo gli enti responsabili e per prevenire l'introduzione di debolezze negli standard. La *crittoanalisi differenziale* studia come le differenze introdotte negli input si ripercuotano negli stadi intermedi e negli ouput, l'obiettivo è quello di individuare collisioni.
Nel 2004, un gruppo di crittoanalisti cinesi presentarono un attacco a $MD5$ (algoritmo utilizzato prima di $SHA-1$) e una serie di funzioni hash correlate. La loro tecnica per trovare collisioni permetteva di controllare, seppur in minima parte, le collisioni individuate. Con la potenza di calcolo odierna, si riuscirebbe a trovare una collisione in $MD5$ di una password in poche ore. Nel 2005, lo stesso gruppo di cinesi che aveva presentato un attacco efficiente ad $MD5$, ha fatto quietamente circolare uno studio che descrive i loro risultati: collisioni in $SHA-1$ in 2^{69} operazioni hash (molto meno del numero di operazioni richieste in un attacco a forza bruta (2^{80}) basato sulla lunghezza hash). Questo attacco va ad aggiungersi ai precedenti attacchi ai danni di $SHA-0$ e $MD5$. Uno dei ricercatori ha affermato che, come punti deboli per poter effettuare l'attacco, hanno utilizzato il fatto che la fase di pre-elaborazione non era abbastanza complicata e che alcune operazioni matematiche nei primi 20 round avevano inaspettati problemi di sicurezza.

La vulnerabilità di $SHA-1$ è proprio quella che ha permesso gli attacchi del virus *Flame*, attivo dal 2010 e scoperto solo nel 2012, il più complesso virus di spionaggio e furto dati mai scoperto. *Flame* era in grado di attivare il microfono del computer e registrare le conversazioni, tracciare i tasti premuti, salvare schermate, esaminare il traffico o comunicare via Bluetooth con dispositivi vicini. Sono stati identificati sistemi infetti in Africa e Medio Oriente, in particolare in Iran, Israele e Sudan, il che ha rafforzato l'ipotesi di un'arma di spionaggio nelle mani di un qualche governo (anche se poi qualche traccia è stata rilevata anche in Ungheria, Austria, Russia, Hong Kong ed Emirati Arabi).
Questi attacchi hanno fatto sì che ci si orientasse verso funzioni hash più forti, con lunghezze di output maggiori, quindi meno suscettibili agli attacchi finora conosciuti. Da notare a questo riguardo è la famiglia di funzioni $SHA-2$, essa estende $SHA-1$ e include funzioni hash con lunghezze output

di 256 e 512 bit.
Nel 2010 il $NIST$ ha pubblicato un articolo in cui raccomandava di non utilizzare più $SHA-1$, bensì $SHA-2$ a partire dalla fine dell'anno corrente, ma siccome in $Microsoft\ Windows\ XP$ non era supportato $SHA-2$, la Casa di Redmond rese disponibile per i suoi utenti il $Service\ Pack$ 3 che aggiungeva il supporto $SHA-256$, $SHA-384$ e $SHA-512$ (non $SHA-224$). In $Microsoft\ Windows\ Vista$ invece gli algoritmi di $SHA-2$ sono nativamente implementati nel sistema operativo.
A differenza di una decina di anni fa, gli algoritmi hash più diffusi, quelli della famiglia $SHA-2$, sono abbastanza sicuri per ora (il $NIST$ confida nella sicurezza di $SHA-2$, oggi ampiamente utilizzato), nonostante ciò però, siccome le conoscenze sono migliorate e con esse anche la velocità dei processori, è necessario disporre di un nuovo algoritmo più robusto, $SHA-3$ (o $Keccak$), che è stato ritenuto un buon complemento della famiglia di algoritmi $SHA-2$. Un vantaggio di $Keccak$ è dato dalla sua differenza nel progetto e nelle proprietà di implementazione rispetto a $SHA-2$, infatti, sembra estremamente improbabile che un singolo nuovo attacco crittoanalitico possa minacciare entrambi gli algoritmi. Le significative differenze nelle proprietà di implementazione dei due algoritmi offrono ai progettisti di nuovi protocolli e applicazioni una maggiore libertà di scelta.

Bibliografia

[1] J. Katz & Y. Lindell, Introduction to Modern Cryptography, 2007, Chapman & Hall/CRC Press.

[2] W. Trappe, L. C. Washington, Crittografia con elementi di teoria dei codici, 2009, Pearson-Pentice Hall.

[3] J. Buchmann, Introduction to Cryptography, 2004, Springer.

[4] Cryptographic hash algorithm competition
http://csrc.nist.gov/groups/ST/hash/sha-3/index.html

[5] The Keccak sponge function family
http://keccak.noekeon.org/

[6] Keccak and the SHA-3 Standardization
http://csrc.nist.gov/groups/ST/hash/sha-3/documents/Keccak-slides-at-NIST.pdf

[7] Wikipedia/Portale:Crittografia

[8] A. Joux, Multicollisions in iterated hash functions. Application to cascaded constructions, Advances in Cryptology - CRYPTO 2004, Lecture Notes in Computer Science 3152, 2004, Springer, pp. 306-316