Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea in Matematica

FUNZIONI HASH CRITTOGRAFICHE E AUTENTICAZIONE

Tesi di Laurea in Crittografia

Relatore:
Prof. Davide Aliffi

Presentata da:
Alessio Boccacci

Parole chiave: Hash integrità autenticazione md5 hmac

II Sessione
2009/10

Alla mia buona volontà,
si fa per dire...

Indice

Introduzione

La parola crittografia deriva dall'unione di due parole greche: kryptós che significa nascosto, e graphía che significa scrittura; da secoli tutti i popoli hanno la necessità di poter comunicare in segretezza: cioè in modo che nessuno, eccetto il diretto interessato, riesca a comprendere il messaggio, anche se intercettato.

Il problema può sembrare a molti risolto introducendo un metodo di cifratura del messaggio; infatti un testo cifrato non potrà essere decifrato, e quindi non potrà essere leggibile, da nessuno eccetto il ricevente che conosce i segreti necessari.
Purtroppo questo non basta a risolvere tutti i problemi legati alla comunicazione in un canale pubblico, è necessario introdurre il concetto di integrità e di autenticità del messaggio.
Un messaggio è integro quando arriva al destinatario così come è stato inviato, senza subire modifiche.
Un messaggio è autentico quando è stato effettivamente inviato dal mittente.
Risulta evidente che la cifratura non garantisce l'integrità e l'autenticità del messaggio ed è altrettanto evidente la pericolosità dello scambiarsi messaggi per i quali non si possano provare tali proprietà.
Supponiamo infatti che Alice stia mandando un messaggio cifrato a Bob, se Oscar intercetta il messaggio e lo modifica inoltrandolo poi a Bob, quest'ultimo non è in grado di capire che il messaggio è stato manomesso.
Oscar può anche mandare un messaggio a Bob facendo finta di essere Alice;

Bob non è in grado di capire che il messaggio non è stato inviato da Alice.
In altre parole si cerca di fare qualcosa di simile alla firma in calce ad un documento cartaceo, ma più sicuro. Per questo nasce il concetto di firma digitale: ogni messaggio va firmato prima di essere inviato, solo in questo modo il ricevente è in grado di verificare l'autenticità e l'integrità.

In questa tesi ci occuperemo di definire in maniera rigorosa tutti gli strumenti matematici necessari alla costruzione di alcuni algoritmi di autenticazione, dei quali dimostreremo, in alcuni casi, la sicurezza del loro utilizzo.
Nonostante riuscire a dimostrare matematicamente la sicurezza di un algoritmo sia un risultato notevole (anche se spesso bisogna mettersi in ipotesi un po' restrittive), non sempre è possibile utilizzare tali risultati nelle applicazioni pratiche.
Spesso un algoritmo sicuro non è un algoritmo efficiente, in altri casi quando si implementa un algoritmo sicuro in un altro più generale questo potrebbe risultare non più sicuro (risulta quindi necessario ridimostrare la sicurezza).
Inoltre, l'approccio matematico, come in questa tesi, non è sempre utilizzato nel campo della crittografia; spesso si preferisce creare algoritmi complicati e rifarsi ai test per dimostrarne la sicurezza.

Capitolo 1

Funzioni Hash

1.1 Definizioni e proprietà

Prima di entrare nel merito dell'argomento trattato occorre dare alcune notazioni: con Σ indicheremo un alfabeto, ossia un insieme finito di simboli, con Σ^* indicheremo l'insieme delle stringhe composte dai simboli di Σ di qualsiasi lunghezza (compresa quella nulla), con Σ^n l'insieme delle stringhe di lunghezza n.

Definizione 1.1 (Funzione Hash)**.** Sia Σ un alfabeto, h è detta funzione hash se:

$$h : \Sigma^* \longrightarrow \Sigma^n, n \text{ fissato}$$

$$x \longmapsto h(x)$$

Osservazione 1. Le funzioni hash non sono iniettive.

Dimostrazione. Sia h una funzione hash, allora $h : \Sigma^* \longrightarrow \Sigma^n, n$ fissato, $\mathsf{card}\Sigma^n = (\mathsf{card}\Sigma)^n < \mathsf{card}\Sigma^* = \infty$, quindi per il principio dei cassetti [1] non esiste alcuna funzione iniettiva da Σ^* a Σ^n. □

[1] Il principio dei cassetti afferma che se m oggetti sono messi in n cassetti ($m > n$), allora almeno un cassetto deve contenere più di un oggetto. Formalmente, equivale a dire che, se A e B sono due insiemi, e B ha cardinalità strettamente minore di A, allora non esiste alcuna funzione iniettiva da A a B.

Una funzione hash risulta essere nient'altro che una funzione che comprime qualsiasi stringa in una stringa di lunghezza fissata; una tale funzione non è chiaramente iniettiva, per questo è utile definire alcune proprietà di cui una funzione hash può godere che ci saranno molto utili per le nostre applicazioni.

Definizione 1.2 (Unidirezionalità)**.** Una funzione hash è detta unidirezionale se $\forall x \in \Sigma^*$ "è facile" calcolare $h(x)$ e se $\forall y \in \Sigma^n$ "non è possibile" calcolare $x : h(x) = y$

Con "è facile" intendiamo che esiste un algoritmo di complessità polinomiale, con "non è possibile" che non esiste.

Definizione 1.3 (Collisione)**.** Sia h una funzione hash; la coppia (x, x') è detta una collisione per h se $x \neq x'$ e $h(x) = h(x')$.

Proposizione 1.1.1. *Sia h una funzione hash, allora esiste una collisione (x, x').*

Dimostrazione. viene banalmente dalla osservazione 1. □

Definizione 1.4 (Resistenza debole alle collisioni)**.** Sia h una funzione hash, è detta resistente debolmente alle collisioni se, dato un $x \in \Sigma^n$, "non è possibile" trovare una collisione (x, x').

Proposizione 1.1.2. *Sia h una funzione hash; se h è resistente fortemente alle collisioni allora h è resistente debolmente alle collisioni.*

Dimostrazione. Se h è fortemente resistente alle collisioni, allora non è possibile trovare una collisione, in particolare non sarà possibile, dato $x \in \Sigma^n$, trovare una collisione (x, x'), e quindi h è resistente debolmente alle collisioni. □

Proposizione 1.1.3. *Sia h una funzione hash; se è resistente debolmente alle collisioni allora h è unidirezionale.*

Dimostrazione. Supponiamo ora per assurdo che h non sia unidirezionale; allora esiste un algoritmo di complessità polinomiale di inversione per h. Fissato x e calcolato $h(x)$, si può invertire $h(x)$ ottenendo x'; se $x \neq x'$, dato che $h(x) = h(x')$, si è trovata una collisione. □

D'ora in poi quando parleremo di resistenza alle collisioni, intenderemo resistenza forte alle collisioni.

Osservazione 2. Esistono funzioni hash unidirezionali non resistenti alle collisioni.

Dimostrazione. Un esempio di funzione hash unidirezionale non resistente alle collisione è il seguente:
$h : \Sigma^* \to \Sigma^m$, $h(x) = x^2 \mod n$, dove $n = p \cdot q$, con p e q primi;
$\Sigma = \{0, 1\}$; $\Sigma^* = \{x | x$ è l'equivalente, scritto in cifre binarie, di $x' \in \mathbb{Z}\}$, quindi un bit contiene l'informazione del segno; m è il numero di cifre binarie di n.
h è unidirezionale[2] ma è facile trovare una collisione $\forall x \in \Sigma^*$. Basta prendere l'equivalente $x' \in \mathbb{Z}$ di x, porre $y' := -x'$ e prendere y, l'equivalente di y' in Σ^*. Risulta che (x, y) è una collisione per h (infatti $h(x) = h(y)$) e quindi h non è resistente alle collisioni. □

D'ora in poi quando parleremo di resistenza alle collisioni, intenderemo resistenza forte.
Una funzione hash resistente alle collisioni è più semplicemente detta funzione hash crittografica, infatti la resistenza alla collisioni è una proprietà necessaria per poter applicare le funzioni hash nel campo della crittografia e, più nel dettaglio, per le autenticazioni.
Facciamo un esempio di quale potrebbe essere un suo uso.

[2]h è unidirezionale, infatti, è ovviamente facile farne l'immagine; mentre farne la preimmagine equivale a risolvere il problema dell'estrazione di radici quadrate di x modulo n, che è un noto problema di algebra equivalente alla fattorizzazione di $n = p \cdot q$, per risolvere la quale non sono stati trovati algoritmi polinomiali. Questo principio sta alla base della sicurezza del sistema di Rabin.

Esempio 1.1. Supponiamo che un sito abbia un sistema di autenticazione tramite password; quando un utente si vuole identificare digita la password e la invia al sito che la confronta con quella salvata nel database, se coincidono allora l'utente risulta identificato.
Questo procedimento non è necessariamente sicuro, perché chiunque riesca ad ottenere accesso al database sarà in grado di conosce la password dell'utente.
In casi come questi è utile una funzione hash crittografica: nel database verrà salvata l'immagine della password tramite la funzione, quando l'utente digita la password il sito ne fa l'immagine e la confronta con l'immagine presente nel database.
Se un hacker riesce ad accedere al database può ottenere le immagini delle password ma non riesce a trovare le password stesse perché le funzioni hash crittografiche sono unidirezionali (proposizione 1.1.3). Inoltre è importante notare che una qualsiasi collisione può sostituire la password, così se la funzione non fosse resistente alle collisioni sarebbe possibile accedere al sito anche senza conoscere la vera password.

Le funzioni hash hanno un utilizzo pratico anche fuori dall'ambito della sicurezza, in questo caso non è necessario richiedere proprietà crittografiche.

Esempio 1.2. Spesso, scaricando archivi (soprattutto se di grosse dimensioni) dal web, ci si ritrova dentro oltre ai file che stiamo cercando un file con estensione SFV. Questo file tra le altre cose contiene l'immagine di ciascun file dell'archivio tramite una funzione hash. Grazie a questa informazione è possibile verificare se i file sono integri (spesso in rete si possono ottenere dati corrotti o addirittura mancanti). Come? Si fa l'immagine dei file e si confronta con quanto scritto nel file SFV, se le informazioni coincidono allora i file sono integri.
Anche in questo caso, è necessario supporre che una modifica al file comporti sempre (o almeno con probabilità molto elevata) un cambiamento nel valore della funzione hash, ossia che la probabilità di una collisione sia piccola.

1.2 Riformulazioni algoritmiche

In questo capitolo cercheremo di costruire un algoritmo hash di cui si possa dimostrare la sicurezza in campo informatico, cioè che non esista alcun algoritmo di complessità polinomiale che permetta con probabilità significativa di trovare collisioni.
D'ora in poi, come si è soliti fare nel linguaggio informatico, l'alfabeto di riferimento sarà $\{0,1\}$. Inoltre lavoreremo con famiglie di funzioni hash indicizzate da chiavi s di lunghezza finita n, cioè $\{H^s\}_{s\in\{0,1\}^n}$ dove la chiave s non è segreta; in questo modo, ogni volta che utilizzeremo l'algoritmo, verrà generata una nuova chiave, e quindi verrà anche utilizzata una nuova funzione hash (nella sezione 1.5 verrà spiegato meglio il perché di questa scelta).
Per il seguito sarà utile riformulare la definizione di funzione hash come segue:

Definizione 1.5 (funzione hash)**.** Una funzione Hash è una coppia di algoritmi (Gen, H) che soddisfa le seguenti condizioni:

- Gen è un algoritmo probabilistico che, dato in input $n \in \mathbb{N}$, detto parametro di sicurezza, restituisce in output s scelto in $\{0,1\}^n$;
- esiste un algoritmo l, di complessità polinomiale, tale che H prende in input la chiave s e la stringa $x \in \{0,1\}^*$ e restituisce in output la stringa $H^s(x) \in \{0,1\}^{l(n)}$.

Se H^s è definita su $\{0,1\}^{l'(n)}$ con $l'(n) \geq l(n)$ allora (Gen, H) è detta una funzione hash con input di lunghezza fissata.

Chiamiamo Π la funzione hash (Gen, H) con parametro di sicurezza n, e consideriamo un esperimento $\mathsf{Hash}-\mathsf{coll}_{\mathcal{A},\Pi}(n)$ definito dalle seguenti condizioni:

- l'algoritmo Gen dà in output la chiave s;
- l'avversario $\mathcal{A}$ conosce s e trova x, x' tramite un suo algoritmo;
- se $x \neq x'$ e $H^s(x) = H^s(x')$ allora $\mathsf{Hash}-\mathsf{coll}_{\mathcal{A},\Pi}(n) = 1$ e in questo caso diciamo che $\mathcal{A}$ ha trovato una collisione.

Alla luce della nuova definizione di funzione hash e di questo esperimento risulta naturale ridefinire in modo più preciso il concetto di resistenza alle collisioni:

Definizione 1.6 (Resistenza alle collisioni)**.** Una funzione hash $\Pi = (\mathsf{Gen}, H)$ è resistente alle collisioni se per ogni avversario $\mathcal{A}$ che utilizza algoritmi probabilistici polinomiali esiste una funzione trascurabile[3] negl tale che

$$Pr[\mathsf{Hash} - \mathsf{coll}_{\mathcal{A},\Pi}(n) = 1] \leq \mathsf{negl}(n)$$

1.3 L'attacco del compleanno

Prima di costruire una funzione hash resistente alle collisioni è utile dare un esempio di attacco capace di trovare collisioni.
Supponiamo di avere $H : \{0,1\}^* \to \{0,1\}^l$, scegliamo arbitrariamente q distinti elementi di $\{0,1\}^*$, $x_1, \ldots, x_q$, ne facciamo le immagini tramite H, $y_i = H(x_i)\ \forall i = 1, \ldots, q$, e infine controlliamo se $y_i = y_j$ per qualche $i \neq j$ con $i, j \in \{1, \ldots q\}$. Se troviamo collisioni il nostro attacco ha funzionato.
Se $q = 2^l + 1$ troveremo sicuramente una collisione poiché $\mathsf{card}\{0,1\}^l = 2^l$.
Quando q è più piccolo di $2^l + 1$ allora può essere utile calcolare la probabilità con cui si può trovare una collisione. Calcolare questa probabilità è simile a trovare qual è la probabilità che 2 studenti siano nati nello stesso giorno in una classe di q alunni; per questo motivo il metodo appena enunciato viene chiamato attacco del compleanno: sfrutta il cosiddetto paradosso dei compleanni.
Il paradosso consiste nel fatto, non paradossale, ma solo poco intuitivo, che basta prendere un gruppo di 23 persone per avere una probabilità di circa il 51% che almeno due siano nate lo stesso giorno, 50 persone per avere

[3]Una funzione f è detta trascurabile se, per ogni polinomio p, esiste N tale che, per ogni $n > N$, valga

$$f(n) < \frac{1}{p(n)}.$$

addirittura il 97% di probabilità.
Per calcolare questa probabilità in una classe di q alunni basta calcolare il complementare della probabilità che tutti gli alunni siano nati in giorni diversi, cioè:

$$1 - \left(\frac{1}{365^q} \frac{365!}{(365-q)!} \right).$$

Proposizione 1.3.1. *Nelle notazioni precedenti, se* $q \geq \frac{1+\sqrt{1+2^{l+3}\log 2}}{2}$*, allora la probabilità di trovare una collisione è maggiore di* $\frac{1}{2}$*.*

Dimostrazione.

$$P(y_1 \neq y_2) = 1 - P(y_1 = y_2) = 1 - \sum_{i=1}^{2^l} \frac{1}{2^l} \cdot \frac{1}{2^l} = 1 - \frac{1}{2^l}$$

$$P(y_3 \neq y_2 \cap y_3 \neq y_1 | y_1 \neq y_2) = 1 - P(y_3 = y_2 \cup y_3 = y_1 | y_1 \neq y_2) = 1 - \frac{2}{2^l}$$

iterando il procedimento risulta:

$$P(y_k \neq y_{k-1} \cap \cdots \cap y_k \neq y_1 | y_1 \neq y_2 \neq \cdots \neq y_{k-1}) = 1 - \frac{k-1}{2^l}$$

da cui si può calcolare la probabilità di non trovare collisioni:
$P(y_1 \neq y_2 \neq \cdots \neq y_q) = P(y_2 \neq y_1) \cdot P(y_3 \neq y_1 \cap y_3 \neq y_2 | y_1 \neq y_2) \cdot \cdots \cdot P(y_q \neq y_{q-1} \cap \cdots \cap y_q \neq y_1 | y_1 \neq y_2 \neq \cdots \neq y_{q-1}) = \left(1 - \frac{1}{2^l}\right) \cdot \left(1 - \frac{2}{2^l}\right) \cdot \cdots \cdot \left(1 - \frac{q}{2^l}\right) =$

$$= \prod_{i=1}^{q-1} \left(1 - \frac{i}{2^l}\right).$$

Denotiamo con E la probabilità di avere almeno una collisione, $1 - \mathsf{E}$ è la probabilità di non avere collisioni. Ora consideriamo lo sviluppo in serie di Taylor arrestato al primo ordine della funzione e^{-x}, cioè $e^{-x} = 1 - x + o(x^2)$, $x \to 0$, risulta $e^{-\frac{i}{2^l}} \leq 1 - \frac{i}{2^l}$, quindi (denotando con E la probabilità di

avere almeno una collisione, $1 - \mathsf{E}$ risulta quindi la probabilità di non avere collisioni):

$$1 - \mathsf{E} \leq \prod_{i=1}^{q-1} e^{-\frac{i}{2^l}} = e^{-\sum_{i=1}^{q-1} \frac{i}{2^l}} = e^{-\frac{1}{2^l}\sum_{i=1}^{q-1} i} = e^{-\frac{q(q-1)}{2\cdot 2^l}}.$$

Risulta:

$$\mathsf{E} \geq 1 - e^{-\frac{q(q-1)}{2\cdot 2^l}}.$$

Sostituendo l'ipotesi, $q \geq \frac{1+\sqrt{1+2^{l+3}\log 2}}{2}$, otteniamo $\mathsf{E} \geq \frac{1}{2}$. □

Vogliamo ora ricavare dei dati numerici per q; poniamo $q := 2^m$, questa tabella mostra il valore di m al variare di l:

l	50	100	150	200
m	25.24	50.24	75.24	100.24

Dalla lettura di questa tabella risulta che $m \approx l/2$, da cui la seguente:

Osservazione 3. Se $q \approx 2^{l/2}$, allora la probabilità di trovare una collisione è circa $\frac{1}{2}$.

Riprendiamo ora il nostro discorso; supponendo che una valutazione di H si faccia in un tempo costante, allora una collisione si può trovare in $\mathcal{O}(l \cdot 2^{l/2})$. In questo caso, se vogliamo che l'avversario $\mathcal{A}$ non riesca a trovare una collisione nel tempo T, è necessario che l sia almeno $2\log T$ bit. Notiamo che a livello computazionale, considerare un attacco con $2^{l/2}$ operazioni oppure uno con $2^l + 1$ non fa nessuna differenza; in ogni caso se $l(n) = \mathcal{O}(\log(n))$, entrambi gli attacchi avranno complessità polinomiale. Nella pratica, però, utilizzare l'attacco del compleanno fa un'enorme differenza: se $l = 128$ bit allora l'attacco può avere successo con 2^{64} valutazioni (che si riescono a fare in tempi ragionevoli) a differenza delle 2^{128} valutazioni.
Da quanto detto finora deduciamo la seguente proposizione:

Osservazione 4. Sia Π una funzione hash, se è resistente alle collisioni allora

$$l(n) = o(log(n)), n \to \infty$$

.

Dimostrazione. Supponiamo che $l(n)$ non sia un $o(log(n))$, allora $l(n) = \mathcal{O}(\log(n))$, risulta che un qualsiasi attacco è polinomiale, e quindi Π non è resistente alle collisioni. □

Osservazione 5. Non vale il viceversa della osservazione 4.

L'attacco del compleanno ha un grosso problema che lo rende inapplicabile: necessita di troppa memoria. L'algoritmo prima descritto dice che, se $l = 128$, dobbiamo fare 2^{64} valutazioni di H per poi confrontarle tra di loro, per fare ciò dobbiamo avere occupati nella memoria del nostro calcolatore $l \cdot 2^{64}$ bit che, senza considerare l, sono 2^{61} bytes che in altre parole sono più un miliardo di gigabytes.
E' possibile costruire un algoritmo con complessità e probabilità di successo simili all'attacco del compleanno, ma con necessità di memoria molto bassa, come segue: si prende un x_0 casuale e, si costruisce una successione, ponendo $x_i = H(x_{i-1})$, $\forall i \geq 1$. Per k generico risulta $x_k = H^k(x_0)$, in particolare $x_{2i} = H^{2i}(x_0) = H(H^{2i-1}(x_0)) = H(H(H^{2(i-1)}(x_0))) = H(H(x_{2(i-1)}))$. Ad ogni passo si confrontano x_i e x_{2i}, se questi sono uguali allora x_{i-1} e $H(x_{2(i-1)})$ sono una collisione.
Un'altra caratteristica dell'attacco del compleanno è che l'avversario riesce facilmente a trovare una collisione di senso compiuto. Supponiamo, ad esempio, che x sia una e-mail che parla di un certo argomento; l'avversario vuole trovare una collisione, cioè un x' tale che $x \neq x'$ e $H(x) = H(x')$, con la particolarità che questa collisione sia una precisa mail scelta dall'avversario, non una stringa casuale.
Come è possibile ciò? Basta che l'avversario scelga un testo preciso, prenda 64 termini contenuti in esso, e per ognuno trovi un sinonimo; ora è in grado di scrivere 2^{64} testi tutti diversi, ma tutti con lo stesso significato, con buona probabilità che uno di questi avrà la stessa valutazione di x mediante H.

1.4 Trasformazione di Merkle-Damgård

Prima di costruire una funzione hash vera e propria è molto utile enunciare e provare la sicurezza della trasformazione di Merkle-Damgård; tale algoritmo ci permette di costruire, a partire da una funzione hash con input di lunghezza fissata, una funzione hash che mantiene alcune delle sue proprietà, in particolare vedremo che mantiene la resistenza alle collisioni.
Sia (Gen, h) una funzione hash con input di lunghezza fissata resistente alle collisioni, con $h^s : \{0,1\}^{2l(n)} \longrightarrow \{0,1\}^{l(n)}$ (si prende questo dominio per comodità, in realtà si può estendere il discorso che stiamo per affrontare a input di qualsiasi lunghezza $l'(n)$ con $l'(n) > l(n)$), dove s è la chiave generata da Gen. Costruiamo una nuova funzione hash (Gen, H) dove Gen rimane lo stesso, mentre H è la funzione che prende in input una chiave s e una stringa $x \in \{0,1\}^*$ di lunghezza $L < 2^{l(n)}$; per trovare l'output, H esegue il seguente procedimento (per semplicità $l = l(n)$):

- poniamo $B := \lceil \frac{L}{l} \rceil$;
- affianchiamo ad x la stringa $\underbrace{0 \cdots 0}_{Bl-L}$, così facendo la lunghezza di x diventa multipla di l;
- guardiamo x come insieme di stringhe $x_1, x_2, \ldots, x_B$ ciascuna di lunghezza l;
- poniamo $x_{B+1} := L$ scrivendo L in modo che anche x_{B+1} sia di lunghezza l (cioè aggiungendo zeri se necessario);
- inizializziamo $z_0 := \underbrace{0 \cdots 0}_{l}$;
- per $i = 1, \ldots, B+1$ calcoliamo $z_i := h^s(z_{i-1}||x_i)$;
- poniamo l'output della funzione $H^s(x) := z_{B+1}$.

Tale algoritmo viene chiamato *trasformazione di Merkle-Damgård*.
Nella costruzione abbiamo detto che la lunghezza dell'input x, cioè L, deve

essere minore di 2^l, questo perché altrimenti non sarebbe possibile scrivere L con l cifre binarie. Può essere inoltre utile osservare che la stringa di l zeri utilizzata per inizializzare z_0 può essere rimpiazzata da una qualsiasi costante che chiameremo IV.

Teorema 1.4.1. *Se* (Gen, h) *è una funzione hash di lunghezza fissata resistente alle collisioni, allora* (Gen, H)*, costruita con la trasformazione di Merkle-Damgård, è una funzione hash resistente alle collisioni.*

Dimostrazione. Per dimostrare il teorema cerchiamo di dimostrare che per ogni chiave s, se abbiamo una collisione (x, x') per la funzione hash H, si può trovare una collisione anche per la funzione hash h. Vediamo nel dettaglio: siano x e x' due stringhe differenti di lunghezza rispettivamente L e L' e tali che $H^s(x) = H^s(x')$, siano $x_1, \ldots, x_B$ i B blocchi che compongono la stringa x (con le eventuali aggiunte di zeri), siano $x'_1, \ldots, x'_B$ i B' blocchi che compongono la stringa x' (con le eventuali aggiunte di zeri).
Ricordando che $x_{B+1} = L$ e $x'_{B'+1} = L'$, abbiamo due casi:

1. $L \neq L'$. In questo caso risulta banale trovare una collisione per h, infatti l'ultimo passo della trasformazione di Merkle-Damgård per fare la valutazione di x tramite la funzione H^s è stato porre $z_{B+1} := h^s(z_B||L)$, per fare la valutazione di x' tramite H^s è stato porre $z'_{B'+1} := h^s(z'_{B'}||L')$; siccome (x, x') è una collisione per H allora $H^s(x) = H^s(x')$ si ha che $h^s(z_B||L) = h^s(z'_{B'}||L')$ e, siccome $L \neq L'$, $(z_B||L, z'_{B'}||L')$ è una collisione per h.

2. $L = L'$. Notiamo che se ciò accade allora abbiamo automaticamente $B = B'$ e $x_{B+1} = z'_{B'+1}$. Siano z_i e z'_i i valori calcolati dal procedimento di trasformazione di Merkle-Damgård per la valutazione di x e di x' tramite H. Se $x \neq x'$ ma $|x| = |x'|$ allora esiste almeno un indice i (con $1 \leq i \leq B$) tale che $x_i \neq x'_i$. Sia i^* il più grande degli indici tali che $z_{i^*-1}||x_{i^*} \neq z'_{i^*-1}||x'_{i^*}$. Se $i^* = B + 1$ allora $z_B||z_{B+1}$ e $z'_B||x'_{B+1}$ sono due stringhe diverse che collidono per h^s perché

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

Se $i^* \leq B$ allora $z_{i^*} = z'_{i^*}$, questo implica che $z_{i^*-1}||x_{i^*}$ e che $z'_{i^*-1}||x_{i^*}$ sono due stringhe diverse che collidono per h^s.

□

Capitolo 2

Autenticazione

2.1 Autenticità di un messaggio

Quando mandiamo un messaggio tramite mail, possiamo cifrarlo in modo da renderlo illeggibile per chiunque lo intercetti, tranne che per il ricevente che conosce la chiave per decifrarlo. Nonostante la cifratura risolva molti dei problemi più comuni, un eventuale avversario ha ancora a disposizione diversi modi per recare danni allo scambio di messaggi tra due interlocutori. Un avversario che intercetta il messaggio cifrato può facilmente cambiarlo e inoltrarlo verso il destinatario, il quale decifrerà un messaggio diverso da quello mandato dal mittente. In base al sistema di cifratura poi sarà più o meno facile per l'avversario cambiare il messaggio cifrato in modo da renderlo comunque di senso compiuto dopo la decifrazione. Per questi motivi viene spesso utilizzato, oltre alla cifratura, un algoritmo che garantisca l'autenticità e l'integrità del messaggio, in questo modo chi riceve il messaggio è in grado di capire se il messaggio è stato realmente scritto dal mittente oppure se è stato modificato.

2.2 MAC

Un tale algoritmo è detto Message Authentication Code (brevemente MAC) ed è così formalmente definito:

Definizione 2.1 (MAC/MAC di lunghezza fissata)**.** Π è detto MAC se è una tripla di algoritmi probabilistici polinomiali $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$, dove:

- Gen, dato in input n, genera una chiave k in $\{0,1\}^n$;
- Mac prende in input la chiave k e un messaggio $m \in \{0,1\}^*$, e restituisce in output un tag t, scriviamo $t \leftarrow \mathsf{Mac}_k(m)$;
- Vrfy è un algoritmo deterministico, che riceve la coppia (m,t) e la chiave k e dà in output un bit b. Se $b = 1$ allora il messaggio è autentico, altrimenti $b = 0$. Poniamo $\mathsf{Vrfy}_k(m,t) =: b$.

Se Mac è definito per $m \in \{0,1\}^{l(n)}$ allora Π è detto MAC di lunghezza fissata $l(n)$ e poniamo $\mathsf{Vrfy}_k(m,t) = 0 \ \forall m \notin \{0,1\}^{l(n)}$.

Intuitivamente, un MAC è sicuro se nessun avversario riesce, in tempi polinomiali, a generare un tag t valido per un messaggio m che non è già stato inviato in precedenza (ossia di cui non è già stato prodotto un tag).
Come abbiamo fatto per definire la resistenza alle collisioni, costruiamo ora un esperimento ideale in cui un avversario vuole generare un tag t valido.
Sia $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$, e n il suo parametro di sicurezzza. Un avversario $\mathcal{A}$ esegue un esperimento chiamato $\mathsf{Mac} - \mathsf{forge}_{\mathcal{A},\Pi}(n)$ definito come segue:

- viene generata una chiave k da Gen;
- l'avversario $\mathcal{A}$ ha accesso ad un oracolo[1] che fornisce un tag t per ogni messaggio m scelto da $\mathcal{A}$. Tramite un suo algoritmo, $\mathcal{A}$ genera una coppia (m,t);

[1] il termine oracolo viene qua utilizzato per esprimere il concetto di macchina (teorica, non per forza reale), in grado di dare una risposta ad un problema di qualsiasi complessità. Interpellare un oracolo costa una sola operazione e quindi non aumenta la complessità dell'esperimento.

- denotiamo con Q l'insieme di tutti i messaggi m per i quali è stato chiesto un tag in precedenza (ad esempio all'oracolo di $\mathcal{A}$).
  $\mathsf{Mac} - \mathsf{forge}_{\mathcal{A},\Pi}(n) = 1$ se $\mathsf{Vrfy}_k(m,t) = 1$ e se $m \notin Q$.

Quindi l'esperimento ha successo se $\mathcal{A}$ riesce a generare un tag t valido per m, senza ricorrere all'oracolo o riutilizzare un messaggio precedente.

Definizione 2.2 (Sicurezza di un MAC)**.** Un MAC, $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$, è sicuro se per ogni algoritmo probabilistico polinomiale eseguito da un avversario $\mathcal{A}$, esiste una funzione trascurabile negl tale che:

$$Pr(\mathsf{Mac} - \mathsf{forge}_{\mathcal{A},\Pi}(n) = 1) \leq \mathsf{negl}(n).$$

La condizione $m \notin Q$ è necessaria perché, per un avversario $\mathcal{A}$, è sempre possibile intercettare una coppia (m,t) e rimandarla tale quale; in questo modo l'avversario invia un messaggio con un tag valido. Inoltre in alcuni casi può essere molto dannoso un attacco di questo tipo, pensiamo al seguente esempio: un cliente invia alla propria banca un messaggio per chiedere di trasferire 1000 euro in un altro conto; se qualcuno intercetta il suo messaggio, lo può riprodurre uguale ad esempio per altre 9 volte, chiedendo così alla banca di trasferire 10000 euro. Un attacco di questo tipo è detto *replay attack*.

2.3 MAC e funzioni hash

In un MAC il ruolo principale è sicuramente svolto dall'algoritmo Mac precedentemente definito, la sicurezza dipende prima di tutto da come è stato costruito e da cosa utilizza; si possono costruire MAC sicuri che utilizzano funzioni pseudocasuali, cifrari a blocchi o funzioni hash; noi considereremo soltanto quest'ultimo caso.
In questa parte ci occuperemo di NMAC (Nested MAC) e HMAC (Hashed MAC); questi MAC utilizzano funzioni hash di lunghezza fissata alle quali viene applicata la trasformazione di Merkle-Damgård; la costante

IV utilizzata viene scelta in $\{0,1\}^l$ e diventa un parametro delle funzioni hash che denoteremo quindi con $H^s_{IV}(x)$; vedremo come questa scelta sarà determinante.

2.3.1 NMAC

In questa sezione descriveremo l'algoritmo NMAC, prima di tutto diamo uno sguardo informale per capire quali sono le funzioni e le chiavi in gioco. Sia h una funzione hash con input in $\{0,1\}^{2n}$ e output in $\{0,1\}^n$; sia H la funzione hash con input in $\{0,1\}^*$ e output in $\{0,1\}^n$ ottenuta utilizzando la trasformazione di Merkle-Damgård su h. La h, oltre alla solita chiave s non segreta, utilizza una nuova chiave segreta, k_1; definiamo $h^s_{k_1}(x) := h^s(k_1||x)$. Di conseguenza, se applichiamo la trasformazione di Merkle-Damgård con $IV = k_1$ alla h^s, otteniamo (ricordiamo che possiamo dividere x in blocchi, $x_1||x_2||\ldots||x_B$ e che $L = |x|$ con un'eventuale aggiunta di zeri):
$z_1 = h^s(k_1||x_1) = h^s_{k_1}(x_1)$;
$z_2 = h^s(z_1||x_2)$;
...
$z_B = h^s(z_{B-1}||x_B)$;
$H^s_{k_1}(x) = z_{B+1} = h^s(L||z_B)$.
Dopo aver trovato $H^s_{k_1}(x)$ si utilizza una nuova chiave k_2 calcolando il tag $h^s_{k_2}(H^s_{k_1}(x)) = h^s(k_2||H^s_{k_1}(x))$.
Formalmente possiamo così definire un algoritmo NMAC:

Definizione 2.3 (NMAC)**.** Sia $(\widetilde{\mathsf{Gen}}, h)$ una funzione hash con input di lunghezza fissata e resistente alle collisioni; sia $(\widetilde{\mathsf{Gen}}, H)$ il risultato della trasformazione di Merkle-Damgård su $(\widetilde{\mathsf{Gen}}, h)$.
Un NMAC è un MAC, cioè una tripla $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$, tale che:

- Gen esegue $\widetilde{\mathsf{Gen}}$ ottenendo una chiave s. Sceglie inoltre due chiavi k_1 e k_2 in $\{0,1\}^n$ in maniera casuale uniforme. Dà in output (s, k_1, k_2).
- Mac prende in input le chiavi (s, k_1, k_2) e un messaggio $m \in \{0,1\}^*$. Dà in output il tag $t := h^s_{k_2}(H^s_{k_1}(m))$.

- Vrfy prende in input (s, k_1, k_2), $m \in \{0,1\}^*$ e il tag t. Genera come output 1 se $t = Mac_{s,k_1,k_2}(m)$, altrimenti 0.

Per dimostrare la sicurezza dell'NMAC (teorema 2.3.1) è utile dare questa definizione, che vale più in generale.

Definizione 2.4 (funzione hash che induce un MAC sicuro)**.** Sia $(\widetilde{\mathsf{Gen}}, h)$ una funzione hash con input di lunghezza fissata; sia $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ un MAC tale che:
Gen esegue $\widetilde{\mathsf{Gen}}$ per ottenere s e sceglie k in modo uniforme casuale in $\{0,1\}^n$; la chiave è (s,k); preso in input $m \in \{0,1\}^n$, $\mathsf{Mac}_{s,k}(m) := h^s_k(m)$; Vrfy è definito come al solito.
Se sono soddisfatte queste ipotesi, allora diciamo che $(\widetilde{\mathsf{Gen}}, h)$ induce un MAC sicuro se Π è sicuro.

Teorema 2.3.1. *Sia $(\widetilde{\mathsf{Gen}}, H)$ il risultato della trasformazione di Merkle-Damgård applicata a $(\widetilde{\mathsf{Gen}}, h)$. Se $(\widetilde{\mathsf{Gen}}, h)$ è resistente alle collisioni e induce un MAC sicuro, allora l'NMAC è sicuro.*

Dimostrazione. Supponiamo che l'NMAC non sia sicuro, allora un avversario $\mathcal{A}$ riesce a trovare un tag valido per un messaggio m^*, del quale non sia già stato chiesto in precedenza un tag, con una probabilità non trascurabile. Ricordiamo che $\mathcal{A}$ ha accesso ad un oracolo che gli fornisce un tag per qualsiasi messaggio (eccetto m^*). Denotiamo con Q l'insieme dei messaggi per i quali $\mathcal{A}$ ha chiesto un tag all'oracolo. Abbiamo due possibilità.

- Esiste un messaggio $m \in Q$ tale che $H^s_{k_2}(m^*) = H^s_{k_2}(m)$.
  In questo caso il tag di m è uguale al tag di m^*, quindi $\mathcal{A}$ è riuscito a trovare un tag valido per il messaggio m^*, ed ha chiesto all'oracolo solo il tag per m. In particolare ne segue che $(\widetilde{\mathsf{Gen}}, H_{k_2})$ non è resistente alle collisioni, allora, per il teorema 1.4.1, $(\widetilde{\mathsf{Gen}}, h)$ non è resistente alle collisioni.

- Per ogni messaggio $m \in Q$ vale che $H^s_{k_2}(m^*) \neq H^s_{k_2}(m)$.
  Poniamo $Q' = \{H^s_{k_2}(m) | m \in Q\}$. Osserviamo che $H^s_{k_2}(m^*) \notin Q'$; allora $H^s_{k_2}(m^*)$ è un nuovo messaggio con tag valido per il MAC indotto da $(\widetilde{\mathsf{Gen}}, h)$, che risulta quindi non sicuro.

□

2.3.2 HMAC

Non sempre è possibile controllare il valore di IV, a volte è fissato e non si può modificare. L'HMAC risolve questo problema, inoltre utilizza una sola chiave segreta, dalla quale poi ne derivano due.

Definizione 2.5 (HMAC)**.** Sia $(\widetilde{\mathsf{Gen}}, h)$ una funzione hash con input di lunghezza fissata n, output $8L$ e resistente alle collisioni; sia $(\widetilde{\mathsf{Gen}}, H)$ il risultato della trasformazione di Merkle-Damgård su $(\widetilde{\mathsf{Gen}}, h)$. Siano IV, opad (abbreviazione di *outer pad*) e ipad (abbreviazione di *inner pad*) delle costanti di lunghezza $8L$ (vedremo in seguito come vengono definite).
Un HMAC è un MAC, cioè una tripla $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$, tale che:

- Gen esegue $\widetilde{\mathsf{Gen}}$ ottenendo una chiave s. Sceglie inoltre una chiave k in maniera casuale uniforme in $\{0,1\}^{8L}$. Dà in output (s, k).

- Mac prende in input le chiavi (s, k) e un messaggio $m \in \{0,1\}^*$ diviso in blocchi m_i di lunghezza $8L$. Dà in output il tag

$$t := H^s_{IV}((k \oplus \mathsf{opad}) || H^s_{IV}((k \oplus \mathsf{ipad}) || m).$$

- Vrfy prende in input (s, k), $m \in \{0,1\}^*$ e il tag t. Restituisce come output 1 se $t = Mac_{s,k}(m)$, altrimenti 0.

Il simbolo $\oplus$ indica la somma bit a bit modulo 2, oppure lo XOR (OR esclusivo).
In questa discussione è utilizzato come operazione vettoriale tra due stringhe

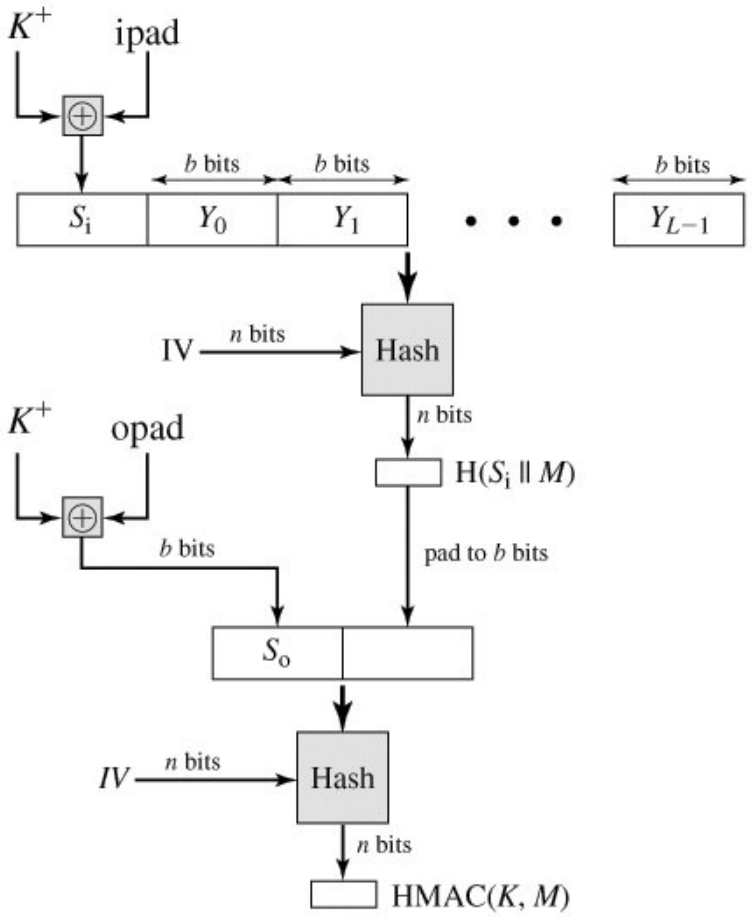

Figura 2.1: struttura HMAC

binarie di lunghezza uguale; dato che k è preso in $\{0,1\}^{8L}$ (vedremo più avanti il senso della scelta $8L$), anche ipad ed opad devono essere di lunghezza $8L$. Alla chiave k vengono eventualmente aggiunti tanti zeri quanti ne servono per farla diventare di L byte. ipad è composto dal byte 00110110 ripetuto L volte; opad è 1011100 ripetuto L volte.

Per dimostrare la sicurezza dell'HMAC si può notare, con i dovuti passaggi, che, sotto determinate condizioni, l'HMAC è un caso particolare dell'NMAC per il quale vale il teorema 2.3.1.

Infatti, prima di tutto, assumiamo che la funzione h^s, durante la trasformazione di Merkle-Damgård per ottenere H^s, venga chiamata una sola volta: il

messaggio e la sua lunghezza vengono inglobati insieme in un unico blocco x (anche se questo andrebbe contro la costruzione di trasformazione di Merkle-Damgård esposta nella sezione 4 del capitolo 1; si può cambiare l'algoritmo per rendere ciò possibile, senza nulla togliere alla sicurezza). Dunque, se x è composto da un solo blocco, $H^s_{IV}(x) = H^s(IV||x) = h^s_{IV}(x)$.
A questo punto poniamo $k_2 = h^s(IV||(k \oplus \mathsf{ipad}))$ e $k_1 = h^s(IV||(k \oplus \mathsf{opad}))$.
Risulta $H^s_{IV}((k \oplus \mathsf{opad})||H^s_{IV}((k \oplus \mathsf{ipad})||m) = H^s_{k_1}(H^s_{k_2}(m))$.
Essendo $H^s_{k_2}(m)$ di lunghezza $8L$ (un solo blocco) e considerando l'assunzione fatta sulla trasformazione di Merkle-Damgård, la formula risulta analoga a quella dell'NMAC, cioè $h^s_{k_1}(H^s_{k_2}(m))$.
Se k_1 e k_2 non fossero dipendenti fra loro allora l'HMAC sarebbe un caso particolare dell'NMAC, perciò si può concludere enunciando il seguente teorema:

Teorema 2.3.2. *Sia* $G(k) := h^s(IV||(k \oplus \mathsf{opad}))||h^s(IV||(k \oplus \mathsf{ipad})) = k_1||k_2$*, k è scelta in maniera casuale uniforme ; sia* $(\widetilde{\mathsf{Gen}}, h)$ *una funzione hash che soddisfa le condizioni del teorema 2.3.1. Se* $G(k)$ *è una funzione pseudocasuale*[2] *allora l'HMAC è sicuro.*

La scelta del valore delle costanti ipad e opad è fatta accuratamente cercando di rendere $G(k)$ pseudocasuale; per questo l'HMAC è considerato sicuro e viene utilizzato ampiamente per le firme digitali insieme a funzioni hash quali MD5 e SHA-1, delle quali parleremo nel prossimo capitolo.

[2]Una funzione pseudocasuale è una funzione deterministica che genera, nel nostro caso, chiavi, ma ha le stesse caratteristiche di una funzione casuale uniforme: dopo la generazione di k_1 non abbiamo nessuna informazione su quale potrebbe essere k_2.

Capitolo 3

L'utilizzo pratico

3.1 Le funzioni hash nella realtà

Una differenza sostanziale tra le funzioni hash usate in pratica e quelle delle nostre costruzioni è che le prime non utilizzano nessuna chiave s, né tantomeno un algoritmo Gen. Perché nei nostri ragionamenti l'abbiamo sempre utilizzata?
Per la definizione che abbiamo dato (definizione 1.6), una funzione hash senza chiave non è resistente alle collisioni. Infatti, supponiamo di conoscere (x, x') collisione per H (senza chiave), allora l'algoritmo che restituisce in output (x, x') è un algoritmo che impiega un tempo costante per trovare una collisione. Sembra banale, ma ciò non è possibile se la funzione hash cambia chiave ad ogni suo utilizzo: conoscere una collisione per H^s non implica in alcun modo conoscere un algoritmo polinomiale che trovi collisioni di $H^{s'}$.
Nonostante ciò, si utilizzano funzioni hash senza chiave; questo accade perché comunque non esistono algoritmi che impiegano un tempo ragionevole per trovare una collisione.
Le funzioni hash nella realtà devono tenere conto dell'attacco del compleanno e del progredire della tecnologia, per questo una delle caratteristiche più importanti nella costruzione di una funzione hash è la lunghezza dell'output.

3.2 MD5

Nel 1991 Ronald Rivest, noto crittografo statunitense, che tra le altre cose ha contribuito ad ideare l'RC5 (algoritmo di cifratura a blocchi) e l'RSA (algoritmo di cifratura asimmetrico), ha progettato l'MD5 (Message Digest algorithm 5). L'MD5 ha output di 128 bit, ed è rimasto lo standard per molti anni.
Analizziamo come è strutturato l'algoritmo MD5, dato un messaggio m in input; per far ciò dividiamo la struttura dell'algoritmo in 5 passaggi.

(1) Aggiunta bit di riempimento.
Si prende m in cifre binarie e si aggiunge un 1, poi tanti 0 finchè non vale (con $l(x)$ denotiamo la lunghezza di x) $l(10\dots0||m) = 448 \mod 512$. 1 viene sempre aggiunto anche se $l(m) = 448 \mod 512$. Poniamo $m' := 10\dots0||m$.

(2) Aggiunta della lunghezza di m.
Si prende $l(m)$, lo si fa diventare di 64 bit (aggiungendo 1 e zeri) e lo si aggiunge ad m'; cioè $m' = l(m)||m'$. Risulta $l(m') = 0 \mod 512$.

(3) Inizializzazione delle variabili.
Si introducono quattro variabili A, B, C, D di lunghezza 32 bit definite come segue (in cifre esadecimali):

```
A: 01 23 45 67
B: 89 ab cd ef
C: fe dc ba 98
D: 76 54 32 10
```

Si definiscono quattro funzioni:
$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z))$
$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$
$H(X,Y,Z) = Z \oplus Y \oplus Z$
$I(X,Y,Z) = Y \oplus (X \vee \neg Z)$.
Si definisce un vettore $T = (t_0, \dots, t_{63})$, dove $t_i := \lfloor |\sin i + 1| \cdot 2^{32} \rfloor$

(4) Elaborazione dell'hash.
Si divide il messaggio m' in N blocchi di 512 bit; $N = \frac{l(m')}{512}$. Per ogni blocco esegue il seguente algoritmo:
si salvano le variabili A, B, C, D in a, b, c, d;
si divide il blocco in ulteriori 16 parti da 32 bit ciascuna che chiamiamo $w(j)$, $j \in \{0, \ldots, 15\}$.
Per ogni blocco si eseguono quattro passaggi:

(i) La scrittura $[ABCD\ k\ s\ i]$ denota l'operazione

$$A = B + {}^{1}(A + F(B, C, D) + w(k) + t_i <<< {}^{2}s).$$

Vengono eseguite le seguenti 16 operazioni:
[ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3] [BCDA 3 22 4] [ABCD 4 7 5] [DABC 5 12 6] [CDAB 6 17 7] [BCDA 7 22 8] [ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12] [ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]

(ii) La scrittura $[ABCD\ k\ s\ i]$ denota ora l'operazione

$$A = B + (A + G(B, C, D) + w(k) + t_i <<< s).$$

Vengono eseguite le seguenti 16 operazioni:
[ABCD 1 5 17] [DABC 6 9 18] [CDAB 11 14 19] [BCDA 0 20 20] [ABCD 5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA 4 20 24] [ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28] [ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]

(iii)]La scrittura $[ABCD\ k\ s\ i]$ denota ora l'operazione

$$A = B + (A + H(B, C, D) + w(k) + t_i <<< s).$$

[1] quì il simbolo + denota la somma modulo 2^{32}, si noti che tutte le variabili sono composte da 32 bit.

[2] $X <<< s$ denota l'operazione di spostamento dei bit di s posizioni verso sinistra. Se $X = x_0 x_1 \ldots x_{31}$, allora $X <<< s = x_s x_{s+1} \ldots x_0 \ldots x_{s-1}$.

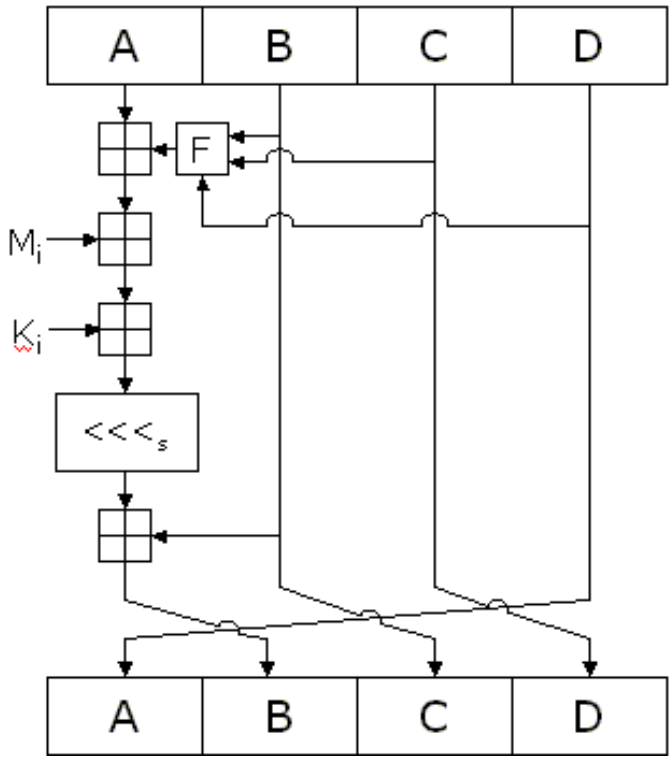

Figura 3.1: operazione al passo (i)

Vengono eseguite le seguenti 16 operazioni:
[ABCD 5 4 33] [DABC 8 11 34] [CDAB 11 16 35] [BCDA 14 23 36] [ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40] [ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44] [ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]

(iv)]La scrittura $[ABCD\ k\ s\ i]$ denota ora l'operazione

$$A = B + (A + I(B, C, D) + w(k) + t_i <<< s).$$

Vengono eseguite le seguenti 16 operazioni:
[ABCD 0 6 49] [DABC 7 10 50] [CDAB 14 15 51] [BCDA 5 21 52] [ABCD 12 6 53] [DABC 3 10 54] [CDAB 10 15 55] [BCDA 1 21 56] [ABCD 8 6 57] [DABC 15 10 58] [CDAB 6 15 59] [BCDA 13

21 60] [ABCD 4 6 61] [DABC 11 10 62] [CDAB 2 15 63] [BCDA 9 21 64]

Infine, si pone
$A = A + a$; $B = B + b$; $C = C + c$; $D = D + d$.
Ricordiamo che questo procedimento viene eseguito 16 volte per ognuno degli N blocchi di cui è composto m'.

(5) Output.
L'output è la stringa $A||B||C||D$ che è lunga 128 bit.

3.3 Collisioni dell'MD5

Già nel 1993 era stata trovata una collisione; negli anni successivi si cercò di valutare la facilità con cui si possono trovare collisioni dell'MD5, finché, nel 2004, due crittografi cinesi, Xiaoyun Wang e Hongbo Yu, riuscirono a trovare una collisione in un'ora tra due stringhe di 128 byte, scritte in caratteri esadecimali :

```
d131dd02c5e6eec4693d9a0698aff95c 2fcab58712467eab4004583eb8fb7f89
55ad340609f4b30283e488832571415a 085125e8f7cdc99fd91dbdf280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e2b487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080a80d1e c69821bcb6a8839396f9652b6ff72a70
```

e

```
d131dd02c5e6eec4693d9a0698aff95c 2fcab50712467eab4004583eb8fb7f89
55ad340609f4b30283e4888325f1415a 085125e8f7cdc99fd91dbd7280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e23487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080280d1e c69821bcb6a8839396f965ab6ff72a70
```

Il cui comune valore di hash MD5 è:

```
79054025255fb1a26e4bc422aef54eb4.
```

Negli anni successivi (grazie anche alla sempre crescente velocità dei calcolatori) sono stati messi a punto algoritmi capaci di trovare collisioni in pochi minuti, addirittura algoritmi capaci di trovare collisioni di senso compiuto. In rete si possono trovare ad esempio due file .eps di finte lettere inviate da Giulio Cesare contenenti due messaggi diversissimi ma con stesso hash MD5:

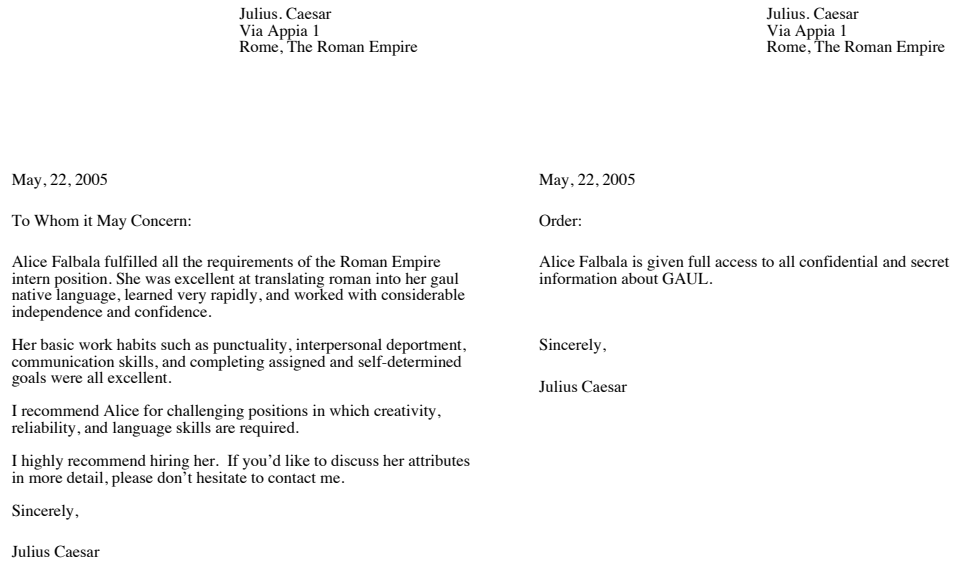

Figura 3.2: letter.eps e order.eps

Ecco cosa risulta valutando l'hash MD5 dei due file:

```
MD5(letter.eps)= a25f7f0b29ee0b3968c860738533a4b9
MD5(order.eps)= a25f7f0b29ee0b3968c860738533a4b9
```

E' stato sviluppato inoltre un algoritmo che permette di rendere due programmi diversi con lo stesso valore hash. Ecco un esempio:

- ```
  .\hello.exe
  Hello, world!
  ```
  ```

```
(press enter to quit)
```

- ```
  .\erase.exe
  This program is evil!!!
  Erasing hard drive...1Gb...2Gb... just kidding!
  Nothing was erased.


  (press enter to quit)
  ```

Ecco cosa risulta valutando l'hash MD5 dei due file:

```
MD5(hello)= da5c61e1edc0f18337e46418e48c1290
MD5(erase)= da5c61e1edc0f18337e46418e48c1290
```

Nonostante ciò l'MD5 trova tutt'oggi applicazioni utili (vedi esempio 1.2) e campi in cui è ancora sicuro il suo utilizzo: ad esempio per il controllo d'integrità di un hard disk; è difficile per un avversario far sparire o corrompere parte dei dati dell'hard disk di una vittima in quanto si trova costretto a lavorare con input di diversi gigabyte, non 128 byte, e quindi la valutazione della funzione hash impiega molto più tempo.

3.4 SHA

Un altro algoritmo molto utilizzato è l'SHA-1 (Secure Hash Algorithm) che ha output di 160 bit; un attacco del compleanno riuscirebbe a trovare una collisione in 2^{80} operazioni, contro le 2^{64} dell'MD5. Nel 2005 è stato teorizzato un attacco secondo il quale è possibile trovare collisioni con 2^{69} valutazioni, in seguito sono diventate 2^{63} e, nel 2009, addirittura 2^{52}; in contrasto a tutto ciò rimane il fatto che non sono ancora state trovate collisioni esplicite.
Già nel 2001 è stato pubblicato l'algoritmo SHA-2, che è in realtà una famiglia di due funzioni hash, che utilizzano una output di 256 e l'altra 512 bit, e tra il 2007 e il 2008 si è svolto un concorso per scegliere l'algoritmo che diventerà l'SHA-3 (verrà data la notizia della scelta nel 2012).
```

Bibliografia

[1] Bellare, Mihir, *New Proofs for NMAC and HMAC: Security without Collision-Resistance*,2006, http://eprint.iacr.org/2006/043.pdf.

[2] Buchmann, Johannes, *Introduction to Cryptography*, 2004, Springer.

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

[4] Hash'em all!, http://www.hashemall.com.

[5] Katz, Jonatan & Lindell, Yehuda, *Introduction to Modern Cryptography*, 2007, Chapman & Hall/CRC Press.

[6] Mattiucci, Marco, *Hash MD5 - collisioni e probabilità*, 2007, http://www.marcomattiucci.it/md5.php.

[7] MD5 Collision Demo, 2009, http://www.mscs.dal.ca/∼selinger/md5collision/.

[8] Mattiucci, Marco, *Hash MD5 - collisioni e probabilità*, 2007, http://www.marcomattiucci.it/md5.php.

[9] Menezes, Alfred, van Oorschot, Paul & Vanstone, Scott, *Handbook of Applied Cryptography*, 1997, CRC Press.

[10] Rivest, Ronald, *The MD5 Message-Digest Algorithm*, 1992, http://www.ietf.org/rfc/rfc1321.txt.