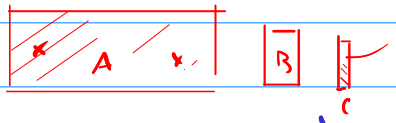
funzione hash crittografica < MDC / MAC

MDC modification detection code

- resistenza alle preimmagini
- resistenza alle II preimmagini
- resistenza alle collisioni

APPLICAZIONI → verifica di integrità
→ firma digitale → invece che firmare un messaggio firmare h(m).

→ costruzione di indici per db.



→ memorizzazione di dati di autenticazione in modo "non trasparente".

MEMORIZZAZIONE DELLE PASSWORD.

→ si vuole che un sistema possa verificare la correttezza di una password senza che essa sia esplicitamente memorizzata.

SOLUZIONE: memorizzare un hash di una password e in fase di login verificare hash(input) = dato_mem.

AUTENTICAZIONE = produrre qc. il cui hash coincide col dato memorizzato.

↓

resistenza alle preimmagini → ANCHE SE CONOSCIAMO L'HASH non abbiamo la passwd.

CALCOLARE UN HASH DEVE ESSERE FATTIBILE MA NON TROPPO VELOCE.

↓

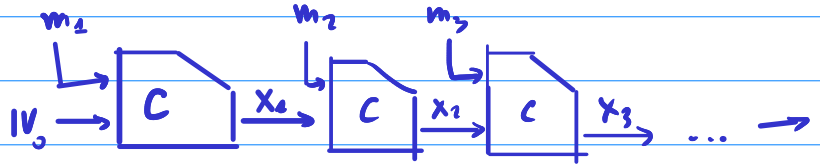
Vogliamo che un hash abbia una complessità computazionale non trascurabile.

↓

polinomiale nell'input ma non lineare.

COME COSTRUIRE UNA FUNZIONE HASH (MDC)

→ ARCHITETTURA BASE: Merkle - Damgård



Abbiamo una funzione C di compressione che prende in input un vettore "di stato" (chain vector) e un blocco di dati; ha in output un chain vector.

Dividiamo il messaggio in tanti blocchi e applichiamo questa funzione C di compressione quante volte necessario

$$H(m_3 m_2 m_1) = C(H(m_2 m_1), m_3)$$

$$H(m_2 m_1) = C(H(m_1), m_2)$$

$$H(m_1) = C(IV, m_1)$$

in questo modo si può lavorare su messaggi di lunghezza arbitraria e l'output ha sempre la stessa lunghezza.

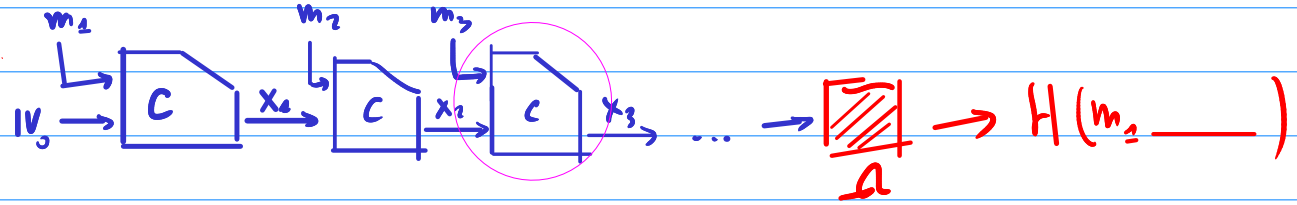
N.B Nella costruzione base è sempre possibile fare un append cioè dato

$H(m_1 m_2 \_\_ m_k)$ si può vedere che

$H(m_1 \_\_ m_k m_{k+1})$ dipende solo da $H(m_1 \_\_ m_k)$ e m_{k+1}

per cercare una collisione fra $m = m_1 \_\_ m_k$

e un $m' \neq m$ possiamo semplicemente provare ad aggiungere ad m tutti i possibili blocchi m_{k+1} e fare una ricerca esaustiva.



per evitare ciò si applica prima di restituire l'Hash una trasformazione finale non invertibile alla variabile X_k di stato (detta variabile)

↓

ad. es. un troncamento.

Inoltre si può aggiungere in testa al messaggio un blocco m_0 di Header che ne rappresenta la lunghezza in bit.

→ costruzione → $m = m_1 \_\_ m_k$

$$h(m) = \Omega(H(m_0 \| m_1 \_\_ m_k))$$

IV = vettore fissato.

ove $H(x_0) = C(IV, x_0)$

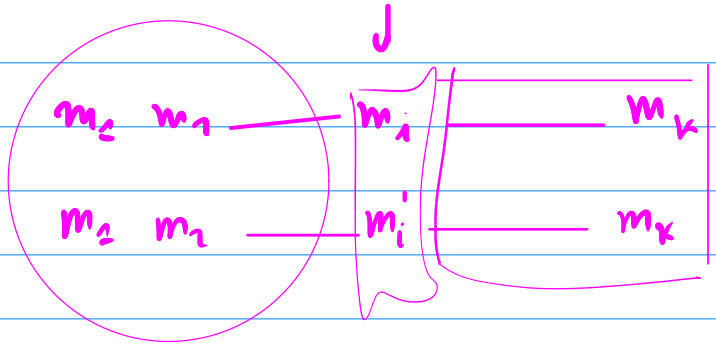
$H(x_0 x_1 \dots x_t) = C(H(x_0 \dots x_{t-1}), x_t)$

ed $\Omega(x_k)$ = trasformata

Elemento fondamentale è la funzione C.

C deve essere resistente alle collisioni

Se C è resistente alle collisioni $\Rightarrow$ l'Hash funziona bene.

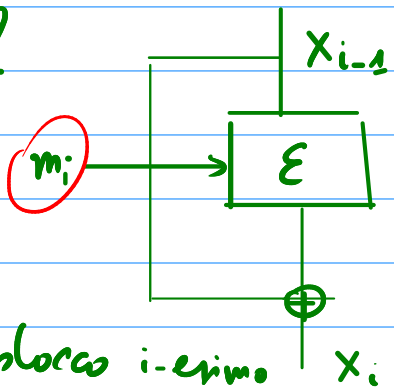


con $m_i \neq m_i'$ e

$C(x_{i-1}, m_i) = C(x_{i-1}, m_i')$.

come costruire C?

Davis Mayer.



E crittosistema simmetrico.

chiave = blocco i-esimo

$C(X_{i-1}, m_i) := E(m_i, X_{i-1}) \oplus X_{i-1} = X_i$

se valgono le proprietà di diffusione e confusione questa costruzione fornisce una funzione resistente alle collisioni.

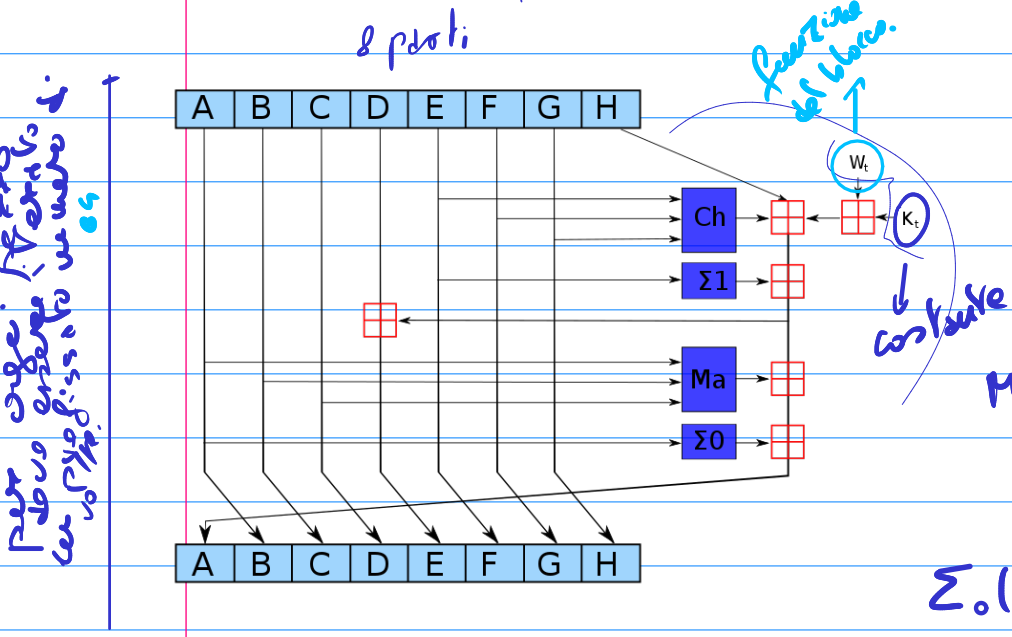
problema: la chain variable in questo caso ha la stessa lunghezza di un blocco per un cifrario simmetrico.

		lunghezza hash
DES →	64 bit.]	→ 32
AES →	128 bit.]	→ 64

modi

CI SONO ALTRE VARIANTI.

1) costruire delle funzioni C "ad hoc" che non vengono da cifrari



sha2.

$Ch(E,F,G) = (E \wedge F) \oplus (\neg E \wedge G)$

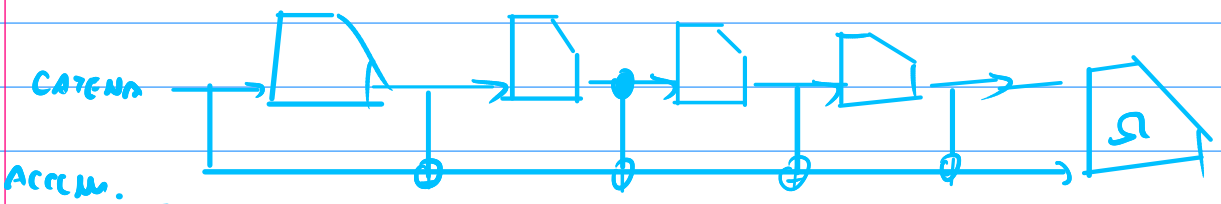
$Ma(A,B,C) = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)$

$\Sigma_0(A) = (A \gg 2) \oplus (A \gg 13) \oplus (A \gg 22)$

$\Sigma_1(E) = (E \gg 6) \oplus (E \gg 11) \oplus (E \gg 25)$

⊞ = somma modulo 2^{32}

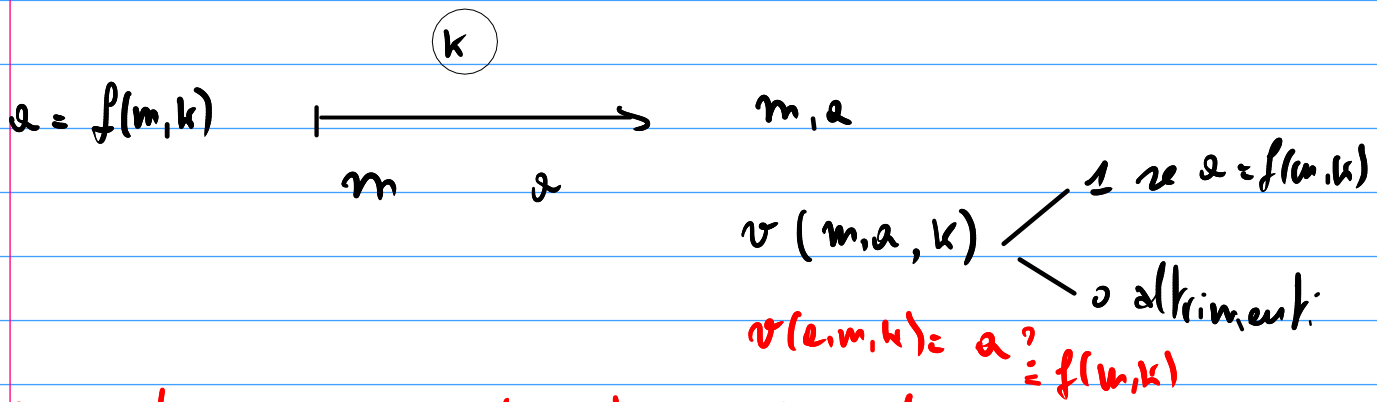
ARCHITETTURE DIVERSE DA M/D hash.



MAC = Message authentication code.

→ controparte privata della firma digitale.

ALICE BOB.



Si vuole avere 2 algoritmi. Tali che

1) Generazione del tag $m, k \rightarrow a = MAC(m,k)$

2) VERIFICA DEL TAG che restituisce 1 se il tag è stato generato con la chiave del messaggio.

Vogliamo che $\forall m_1 \text{ —— } m_m$ di messaggi e di tag $a_i = MAC(m_i,k)$, sia "impossibile" vista:

$(m_1, a_1) \text{ —— } (m_n, a_n)$ valida

generare una coppia (m_{n+1}, a'_{n+1})

valida senza conoscere k con $m_{n+1} \notin \{m_1 \text{ —— } m_n\}$.

$$Pr\left((m_{n+1}, a'_{n+1}) \text{ valido} \mid (m_1, a_1) \text{ —— } (m_n, a_n) \right) = Pr\left(m_{n+1}, a'_{n+1} \right)$$

valido senza aver visto nulla

N.B un MAC non deve essere omomorfico

NON DEVE VALERE

N2 | $MAC(m_2 m_1, a_1 a_2) = MAC(m_2, a_2)\, MAC(m_1, a_1)$

Esempio di MAC che non funziona

COSTRUZIONE BASE M/D.

$$MAC(m, k) = H(k \| m)$$

in questo allora $MAC(m \| x, k) =$

$$= H(k \| m \| x) = C(H(k \| m), x).$$

Posso accettare $m \| x$ dato il MAC di m

senza conoscere k.

come ottenere un MAC da un MDC?

costruzione HMAC

$$HMAC(K, m) = H(K' \oplus opad \| H((K' \oplus ipad) \| m))$$

con $K' = H(K)$ se lunghezza $K >$ lunghezza blocco

altrimenti $K' = K$

ed ipad/opad sono costanti

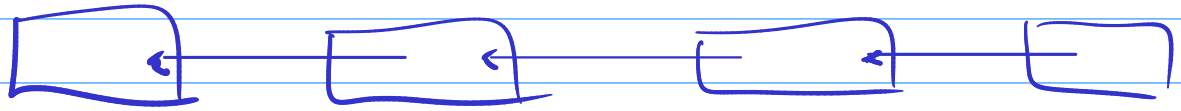
$$H\left(H(k \oplus opad) \,\|\, H((k \oplus ipad) \,\|\, m)\right)$$

per un attacco di tipo append dovremmo poter aggiungere il blocco x alla funzione di hash interna ma l'output finale dipende dall'hash di questo hash concatenato con la chiave. #

Applicazione delle funzioni hash

→ blockchain.

blockchain → struttura dati lineare in cui ogni blocco è indicizzato in funzione del contenuto dei blocchi precedenti.



ogni blocco contiene un puntatore al contenuto del blocco precedente (linked list).

2) Obiettivo degli algoritmi di aggiornamento e riconciliazione dati.

E' possibile 1) AGGIUNGERE BLOCCHI

2) AVERE UN CONSENSO SU QUALI SIANO I BLOCCHI.

Database distribuito immutabile con un algoritmo di riconciliazione (=> tutti gli utenti vedono gli stessi dati).

Teorema CAP: in un db distribuito è impossibile avere queste 3 proprietà contemporaneamente

1) CONSISTENZA → ogni utente riceve i dati più aggiornati o errore.

2) DISPONIBILITÀ → ogni utente riceve una risposta ad una sua query

3) Resistenza alle PARTIZIONI: il sistema può operare anche se alcuni messaggi sono persi o cancellati.