Modulo 3: Message Authentication Codes (MAC) e Integrità dei Dati

Indice

1 Introduzione: Integrità vs Confidenzialità

In ambito di sicurezza delle reti, è fondamentale distinguere tra due concetti che spesso vengono confusi: confidenzialità e integrità.

- **Confidenzialità:** Si occupa di nascondere il messaggio. Garantisce che solo il destinatario legittimo possa leggere il contenuto.
- **Integrità:** Si occupa dell'autenticità del messaggio. Garantisce che nessuno abbia modificato il messaggio durante la trasmissione e che solo la sorgente legittima possa averlo generato.

Un principio cardine è che la cifratura (Encryption) NON garantisce l'integrità. Un attaccante potrebbe modificare un messaggio cifrato in transito; anche se il risultato decifrato fosse privo di senso ("garbage"), il sistema ricevente potrebbe non accorgersene o, peggio, la modifica potrebbe risultare in un messaggio valido ma alterato (malleabilità).

Eccezione: Solo i cifrari AEAD (Authenticated Encryption with Associated Data) sono progettati per garantire contemporaneamente entrambe le proprietà.

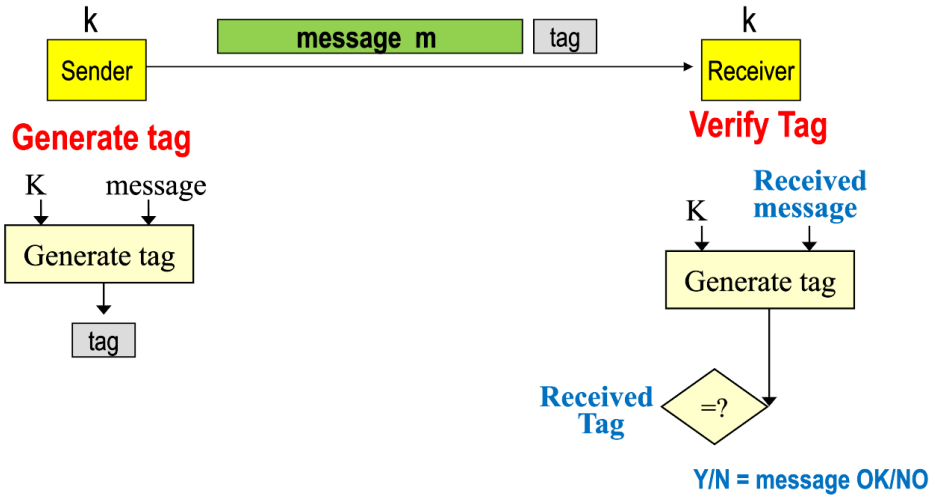

Figura 1: Schema generale della Message Autentication

Dove:

- k è il segreto conosciuto sia dal sender che dal receiver.
- `Generate-Tag` è una funzione **crittograficamente forte**: prende in input K e il msg, e come output dà un codice di autenticazione relativamente "corto" (tag, comunque più corto dell'input).
  - Perché è forte? Perché non è reversibile! (per prendere la chiave devo fare **brute-force**).
  - È anche corretto chiamare il tag la **firma del messaggio** (da non confondere con la firma digitale, che è un'altra cosa).

Esempio: MSG = 10K bits, K=128 bits → `Generate-Tag` ritorna un tag tale per cui $|TAG| \ll 10k$.

2 Requisiti per l'Autenticazione dei Messaggi (Stallings Cap. 12)

Secondo Stallings, i meccanismi di autenticazione devono proteggere contro diverse minacce. Ci sono però minacce che possono essere risolte con la Message Authentication, ed altre no.

Alcune minacce risolte sono il MITM e il M.Spoofing.

Attacchi MITM (Man in The Middle)

Un'entità non autorizzata legge il messaggio e lo modifica. Perché la MA protegge da attacchi di questo tipo? Analizziamo:

- L'attaccante può modificare tranquillamente il messaggio, però accadono le seguenti cose:
  - Lui vede sia m che TAG, ma essendo F la funzione "crittograficamente" forte, lui non potrà invertire F per ottenere K.
    - Ovviamente la dimensione della chiave deve essere "sufficientemente lunga", al giorno d'oggi almeno 128 bits, quindi per trovare la chiave devo fare brute-force, e con 128 bits abbiamo un numero di "prove" pari circa a $\frac{2^{128}}{2}$.
  - Senza conoscere K, l'attaccante non può cambiare il TAG in $TAG^\star = F(K, m^\star)$.
  - E infine, non potrà cambiare m in $m^\star$ in modo tale che $F(K, m) = F(K, m^\star)$.
- Il ricevente vede quindi $m^\star$ al posto di m, però quando poi si va a calcolare $F(K, m^\star) = TAG^\star \neq TAG$ capisce che qualcuno ha modificato l'attacco.
  - La probabilità che $TAG^\star$ sia diverso da TAG originale è $\frac{1}{2^{|TAG|}}$, per questo motivo la funzione F deve generare TAG che siano grandi.

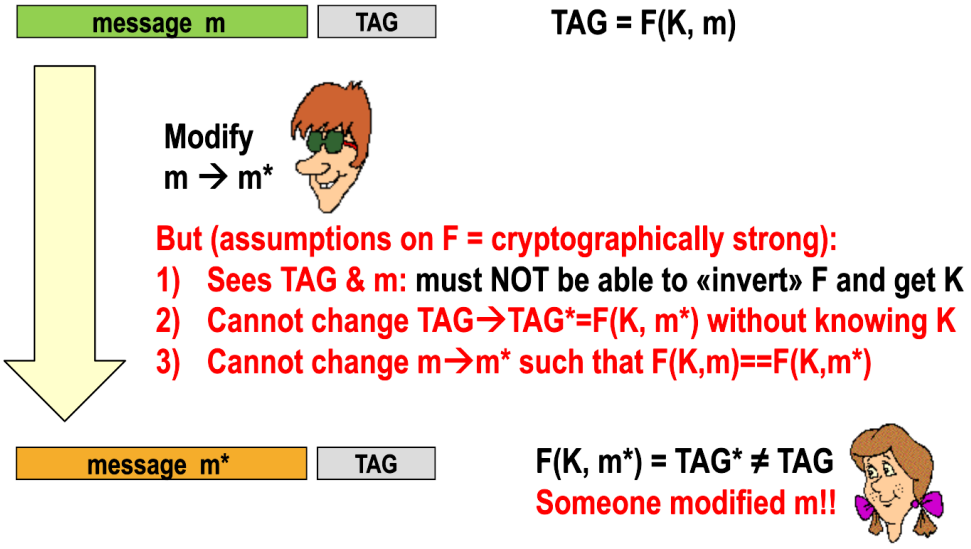

Figura 2: Schema attacco MITM con MAC

Message Spoofing

Un'entità non autorizzata finge di essere una sorgente valida. Ipotizziamo che l'attaccante crei un messaggio personalizzato X, e trovi un modo per generare una firma valida per il messaggio (ovvero il TAG) (questa proprietà, se vale, viene chiamata **Forgiability**).

Ora, per le assunzioni fatte sulla funzione F, vale che F garantisce la **non-Forgiability**, di conseguenza l'attaccante non potrà calcolare $F(K, X)$ senza conoscere K, e quindi il TAG che

arriverà sarà un TAG invalido. Il ricevente può controllare questa cosa, e potrà accorgersi che il messaggio è stato *falsificato*.

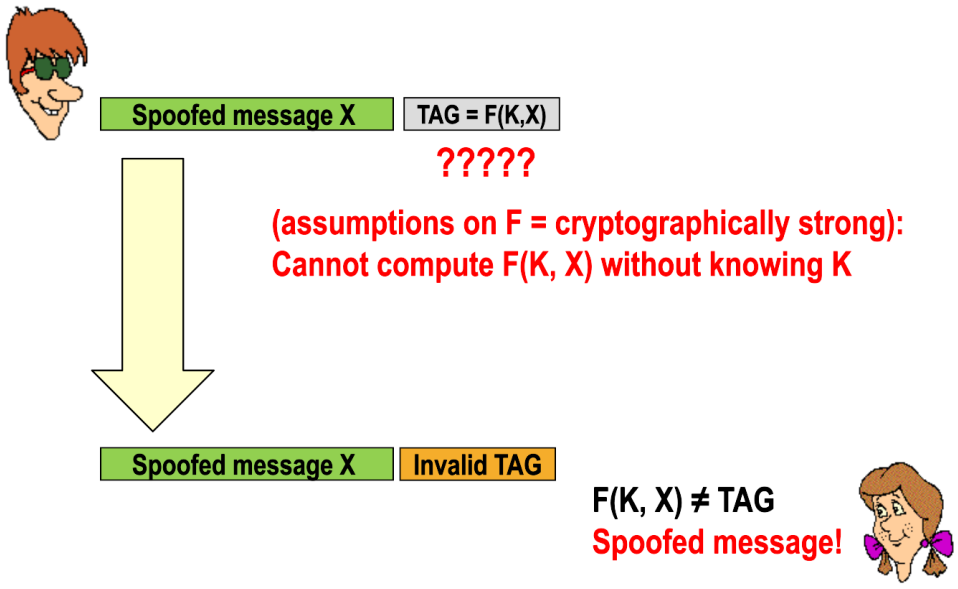

Figura 3: Schema attacco Message Spoofing con MAC

Replay Attack (Minaccia non risolta)

Un'esempio di minaccia che il MA non risolve è il così detto **Replay Attack**: i messaggi vengono ritardati o replicati. Perché non protegge? Analizziamo:

Supponiamo di voler pagare 1000$ in una transazione, e un secondo dopo voglia ripagare altri 1000$ (es. compro due telefoni). Dato che i messaggi sono *esattamente* gli stessi, e la chiave è **costante** nel tempo, allora il TAG risulterà essere lo stesso.

Ora l'attaccante può prendere il primo msg, copiarlo, e rimandarlo in un secondo momento alla, in questo caso, banca. Così facendo, l'attaccante farà in modo che la vittima paghi più volte del dovuto.

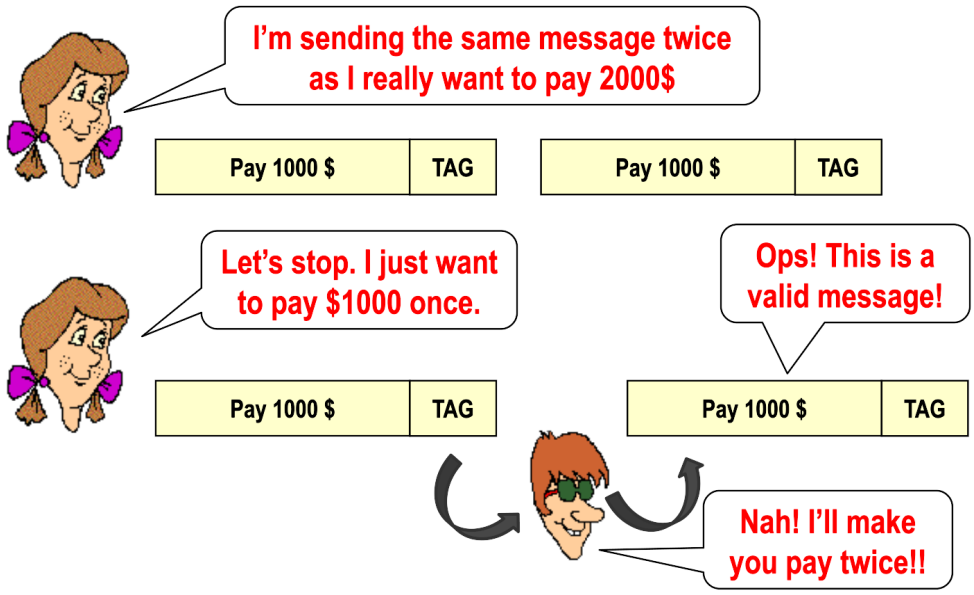

Figura 4: Schema Replay Attack con MAC

Per prevenire i replay attack, è necessario introdurre un **Nonce** (Number used ONCE), un valore che cambia per ogni messaggio e viene incluso nel calcolo del MA.

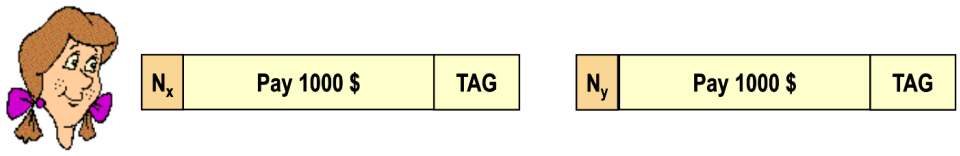

Figura 5: Aggiunta del NONCE al MAC per evitare RA

Ci sono 3 tipi di **Nonce**:

Tipo di Nonce	Descrizione	Vantaggi	Svantaggi
Sequence Number	Un contatore incrementale (N_x, N_y).	Semplice da implementare.	Gestione complessa in caso di riavvio (reboot) o perdita di sincronizzazione.
Random Number	Un valore casuale grande.	Non richiede stato o sincronizzazione temporale.	Il ricevente deve mantenere uno storico dei nonce visti per evitare duplicati (Windowing).
Timestamp	L'orario corrente.	Previene replay ritardati significativamente.	Richiede sincronizzazione degli orologi sicura e affidabile.

Il calcolo del TAG diventerà quindi: $Tag = F(K, M||Nonce)$. Osserviamo che nessuna di queste è meglio dell'altra.

3 Message Authentication Code (MAC)

Prima di parlare dei MAC, dobbiamo vedere quali differenze si porta con le Firme Digitali (DS).

La firma digitale impone che **nessuno** può modificare un messaggio firmato digitalmente (ad eccezione del creatore). Il MAC impone che **nessuno eccetto Sender e Receiver** (che hanno la chiave) può modificare un messaggio autenticato con MAC.

Entrambi quindi hanno lo stesso identico scopo, ovvero proteggere l'integrità dei dati/messaggi. Però, a differenza del MAC, le DS garantiscono anche il non-ripudio (ovvero l'autenticazione della sorgente).

- Abbiamo quindi che le DS sono una forma di autenticazione **più forte** dei MAC, ma richiedono nozioni di crittografia diverse, ovvero la crittografia asimmetrica.

Vediamo ora nel dettaglio i MAC. Un MAC è un blocco di dati di dimensione fissa, noto come *tag* crittografico, generato basandosi sul messaggio e su una chiave segreta.

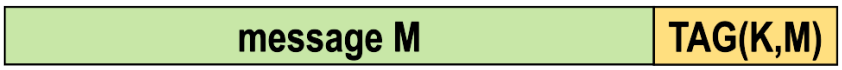

Figura 6: MAC, ovvero TAG(K,M)

3.1 Definizione Formale

Un MAC richiede tre elementi:

1. **Messaggio (M):** Di lunghezza arbitraria.
2. **Chiave Segreta (K):** Condivisa tra mittente e destinatario.
3. **Funzione MAC (F):** Una funzione crittograficamente forte che mappa K e M in un tag di lunghezza fissa.

La sicurezza dei MAC deriva dalla proprietà, descritta prima, della **non falsificabilità** (unforgeability): l'attaccante non deve essere in grado di creare/modificare un messaggio; questo implica che l'attaccante non deve essere in grado di estrarre la chiave dalla coppia $\{M, TAG(K, M)\}$.

Il processo è il seguente:

- **Generazione:** Il mittente calcola $Tag = F(K, M)$ e invia (M, Tag).
- **Verifica:** Il destinatario riceve (M', Tag'), calcola $Tag_{calc} = F(K, M')$ e lo confronta con Tag'. Se coincidono, il messaggio è autentico.

3.2 Sicurezza del MAC

- Gemini -
La sicurezza di un MAC si basa sul fatto che è computazionalmente impossibile per un attaccante (che non possiede K) calcolare un tag valido per un messaggio M^* o modificare M in M^* mantenendo valido il tag originale.

- Bianchi -
La sicurezza dei MAC deriva dalla proprietà, descritta prima, della **non falsificabilità** (unforgeability): l'attaccante non deve essere in grado di creare/modificare un messaggio; questo implica che l'attaccante non deve essere in grado di estrarre la chiave dalla coppia $\{M, TAG(K, M)\}$.

Questo modello implica che il MAC protegge dagli attacchi Man-In-The-Middle (MITM) che tentano di modificare il payload.

Però, questa benedetta funzione F come deve essere? Per rispondere a questa domanda ci vengono in aiuto le funzioni Hash, e più nel dettaglio le funzioni Hash Crittografiche. Prima di parlare quindi di HMAC, facciamo prima una piccola digressione su cosa sono le Hash Crittografiche e come funzionano.

Interludio: Hash e Hash Crittografiche

Prima di tutto, cos'è una funzione Hash? Una funzione Hash è una funzione che prende in input un messaggio X di lunghezza arbitraria (ovvero da 1 bit a qualunque dimensione), e ritorna un altro messaggio $Y = H(X)$ di lunghezza ***fissata*** (es. esattamente 256 bits se si usa SHA-256).

La stringa $Y = H(X)$ viene anche chiamata "fingerprint" (riepilogo corto) di X. La funzione $H(X)$ deve essere relativamente facile da calcolare per ogni X dato.

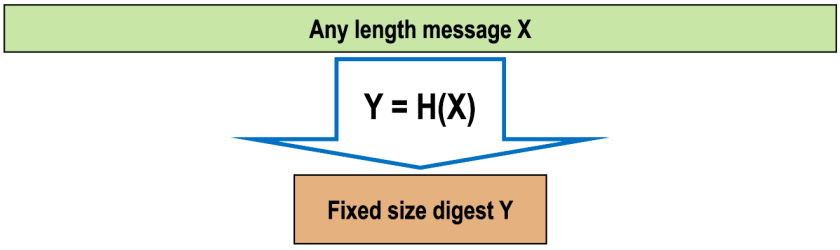

Figura 7: Schema funzionamento funzione Hash

E cosa cambia con una funzione Hash **Crittografica**? Una funzione Hash Crittografica funziona esattamente come le Hash normali, con la differenza che, se prendiamo X, generiamo $Y = H(X)$ e poi modifichiamo X in $X^\star$ (ad esempio cambiando qualche carattere nel messaggio), applicando

la *stessa* Hash Crittografica applicata ad X, quello che otterremo sarà un risultato completamente diverso:

$$Y^\star = H(X^\star), \quad Y^\star \neq Y \quad \text{(sotto ogni punto di vista)}$$

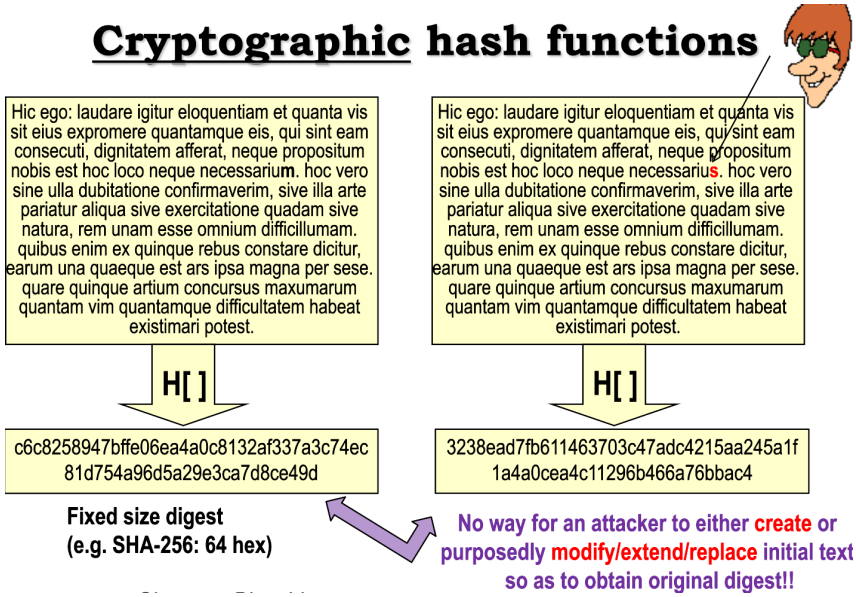

Figura 8: Schema funzione Hash Crittografica

Come possiamo vedere dall'immagine quindi, anche solo cambiare una lettera porta ad hash completamente diversi.
Le H.C. però non hanno solo questa caratteristica, ma hanno altre bellissime proprietà che ci serviranno più avanti per i MAC:

- **Preimage resistance (one-way):** Dato Y il risultato dell'hash, deve essere difficile trovare un qualunque X tale per cui $X = H(Y)$; quindi da Y non devo poter essere in grado di risalire a X.

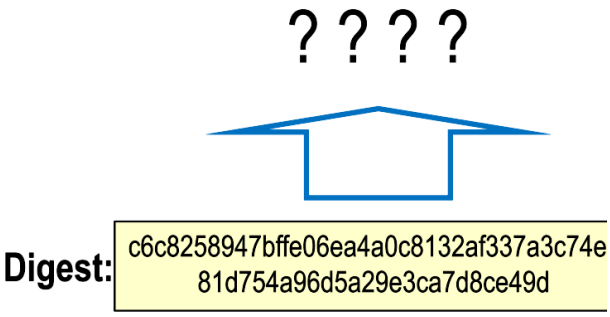

Figura 9: Preimage Resistance

- **Second Preimage Resistance (Weak collision resistance):** Dato X, deve essere difficile trovare un altro $X^\star$ tale che $H(X) = H(X^\star)$.

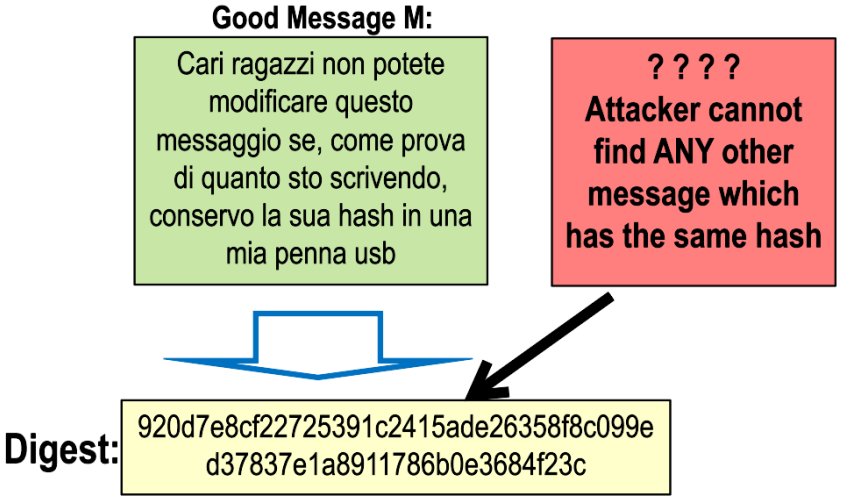

Figura 10: Second Preimage Resistance

- **Collision Resistance (Strong collision resistance):** Deve essere difficile trovare due X_1, X_2 generici tali per cui $H(X_1) = H(X_2)$.

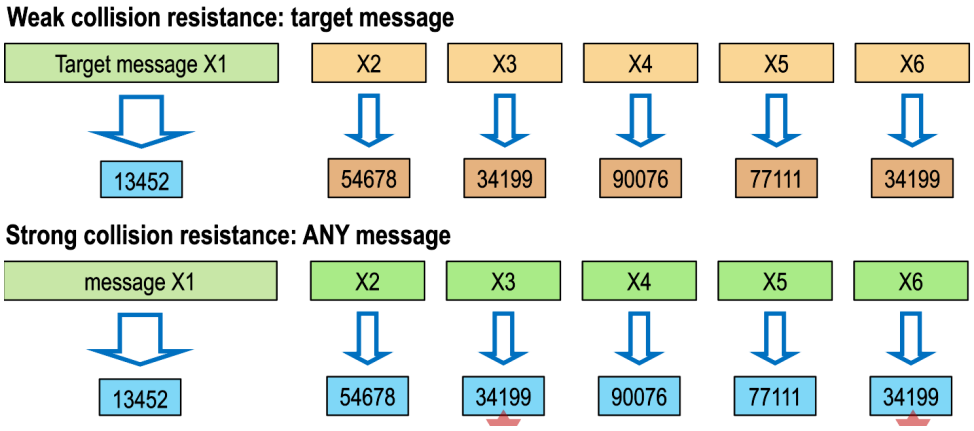

Figura 11: Collision Resistance

Abbiamo parlato quindi di questo digest e della sua dimensione, ma quanto dovrebbe essere grande per garantire sicurezza? Vediamo alcuni esempi:

- 32 bits $\rightarrow$ 4.3 bilioni di output, ma il 50% di probabilità di collisione arriva dopo $2^{16} \sim$ 60.000 messaggi (molto pochi).
- 128 bits (MD5) $\rightarrow$ il 50% di collisione arriva dopo $2^{64} = 1.8 \times 10^{19}$ (ad oggi inutile, tant'è che MD5 è stato bucato nel 2005).
- 256 bits (SHA-256) $\rightarrow$ il 50% di collisione arriva dopo $2^{128} = 3.4 \times 10^{38}$, ad oggi accettabile.

Interludio: Il Paradosso del Compleanno e la Sicurezza degli Hash

Quando parliamo di "Collision Resistance", sorge spontanea una domanda: quanto deve essere lungo un digest (n bit) per essere sicuro? L'intuizione ci suggerirebbe che per trovare una collisione in uno spazio di 2^n possibilità, servano circa 2^n tentativi (o almeno 2^{n-1} in media). Tuttavia,

esiste un fenomeno statistico contro-intuitivo noto come **Paradosso del Compleanno**, che dimezza drasticamente la sicurezza effettiva.

L'intuizione: 23 persone in una stanza

Il paradosso prende il nome dal seguente quesito:

> *Quante persone devono esserci in una stanza affinché la probabilità che almeno **due** di esse compiano gli anni lo stesso giorno sia superiore al 50%?*

Ci sono due scenari distinti:

1. **Scenario 1 (Nessun paradosso):** Qual è la probabilità che qualcuno compia gli anni lo stesso giorno di una persona specifica (es. il professore)?

   $$P = 1 - \left(\frac{364}{365}\right)^N$$

   Per avere il 50% di probabilità, servono $N = 253$ persone. Questo è intuitivo.

2. **Scenario 2 (Collisione generica):** Qual è la probabilità che **due persone qualsiasi** compiano gli anni lo stesso giorno? Qui stiamo cercando una collisione qualsiasi ($X_1 = X_2$), non una specifica. Sorprendentemente, bastano solo **23 persone** per avere una probabilità del $49.3\% \approx 50\%$.

Analisi Matematica (Birthday Paradox Math)

Applichiamo questo concetto agli Hash. Siano:

- $N = 2^n$: il numero totale di possibili output (lo spazio del digest).
- K: il numero di messaggi (o tentativi) generati.
- p: la probabilità di trovare almeno una collisione.

La probabilità che **non** ci siano collisioni $(1 - p)$ si calcola moltiplicando le probabilità che ogni nuovo messaggio abbia un hash diverso da tutti i precedenti:

$$P(\text{no collision}) = 1 \cdot \left(1 - \frac{1}{N}\right) \cdot \left(1 - \frac{2}{N}\right) \cdots \left(1 - \frac{K-1}{N}\right)$$

Questa produttoria può essere scritta come:

$$1 - p = \prod_{i=1}^{K-1} \left(1 - \frac{i}{N}\right)$$

Ricordando l'approssimazione di Taylor per x piccolo, dove $e^{-x} \approx 1 - x$, possiamo riscrivere l'equazione come:

$$1 - p \approx \prod_{i=1}^{K-1} e^{-\frac{i}{N}} = e^{-\sum_{i=1}^{K-1} \frac{i}{N}}$$

Sapendo che la somma dei primi interi è $\sum_{i=1}^{K-1} i = \frac{K(K-1)}{2} \approx \frac{K^2}{2}$, otteniamo:

$$1 - p \approx e^{-\frac{K^2}{2N}}$$

Implicazioni sulla Sicurezza: La regola della radice quadrata

Ora vogliamo sapere quanti messaggi K dobbiamo generare per avere una probabilità di collisione p (ad esempio il 50%). Risolviamo l'equazione rispetto a K:

$$\ln(1-p) \approx -\frac{K^2}{2N} \implies K^2 \approx -2N\ln(1-p) \implies K \approx \sqrt{2N\ln\left(\frac{1}{1-p}\right)}$$

Se poniamo $p = 0.5$ (50% di probabilità di collisione), abbiamo $\ln(2) \approx 0.693$:

$$K \approx \sqrt{2 \cdot 0.693 \cdot N} \approx 1.177\sqrt{N}$$

Poiché nello scenario crittografico $N = 2^n$ (dove n è il numero di bit del digest):

$$K \approx 1.177\sqrt{2^n} = 1.177 \cdot (2^n)^{1/2} \approx \mathbf{2^{n/2}}$$

Conclusione Fondamentale

Il livello di sicurezza di un digest a n bit contro gli attacchi di collisione **NON è 2^n, ma $2^{n/2}$**.

Esempio pratico:

- **SHA-256** ($n = 256$ bit): Non servono 2^{256} operazioni per trovare una collisione, ma ne bastano 2^{128}.
- **MD5** ($n = 128$ bit): La sicurezza è 2^{64}, un numero oggi raggiungibile computazionalmente (infatti MD5 è rotto).

4 MAC basati su Funzioni Hash (HMAC)

Vediamo quindi, dopo aver brevemente elencato le proprietà delle Hash Crittografiche, come usarle all'interno dei MAC.

Primo ingrediente: una buona funzione Hash.

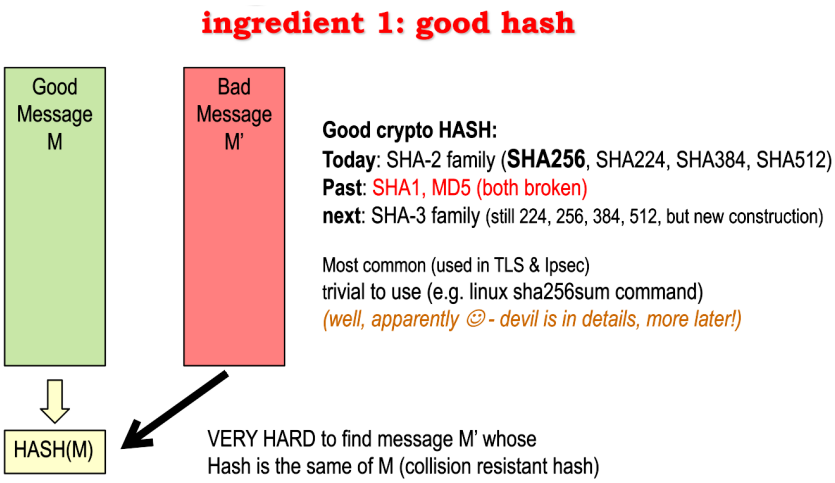

Figura 12: Primo Ingrediente

Secondo ingrediente: includere il secret nell'hash.

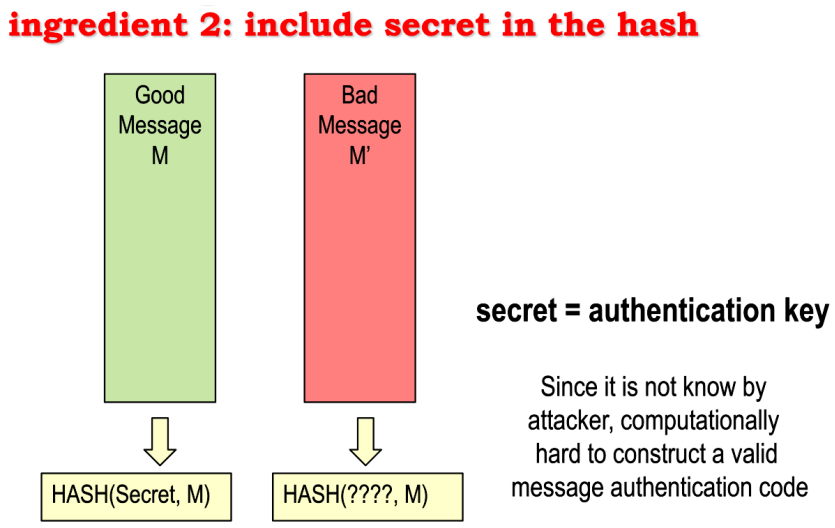

Figura 13: Secondo Ingrediente

Una funzione hash crittografica $H(M)$ garantisce l'impronta digitale, ma non l'autenticazione, perché chiunque può calcolare l'hash di un messaggio modificato. È necessario includere la chiave segreta K.

Infatti la domanda che ci poniamo è: dove mettere il secret? come suffisso o come prefisso del messaggio?

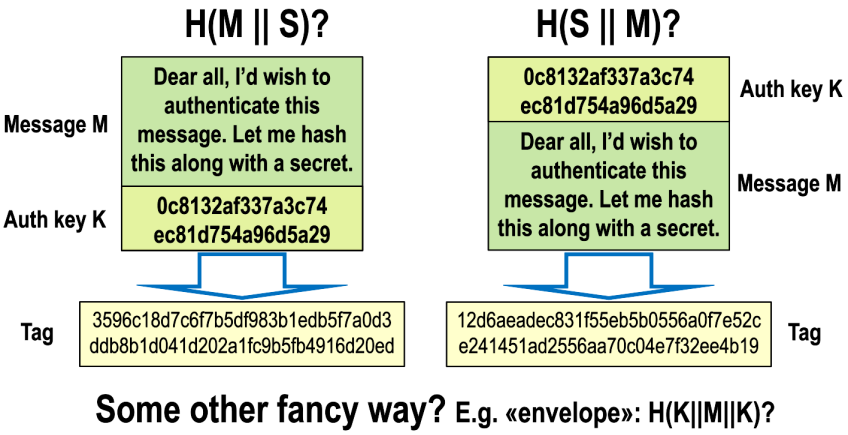

Figura 14: Differenze con Suffisso o Prefisso

All'apparenza potrebbe essere una domanda stupida, ma all'atto pratico assolutamente no.
Noi potremmo non fregarci di questa questione SOLO se la funzione Hash scelta fosse un **Random Oracle Perfetto**, ovvero una struttura che ha la seguente proprietà:

- Per ogni X distinto, $H(X)$ è un **vero** valore randomico (dove per vero valore randomico intendiamo un valore che a prescindere da tutto sia generato veramente in maniera casuale), ma con la proprietà che allo stesso X deve sempre corrispondere lo stesso $H(X)$.

All'effettivo, nessuna funzione hash può essere un random oracle. Tuttavia, le costruzioni "ingenue" sono insicure a causa della struttura iterativa (**Merkle-Damgård**) delle funzioni hash comuni (MD5, SHA-1, SHA-2).

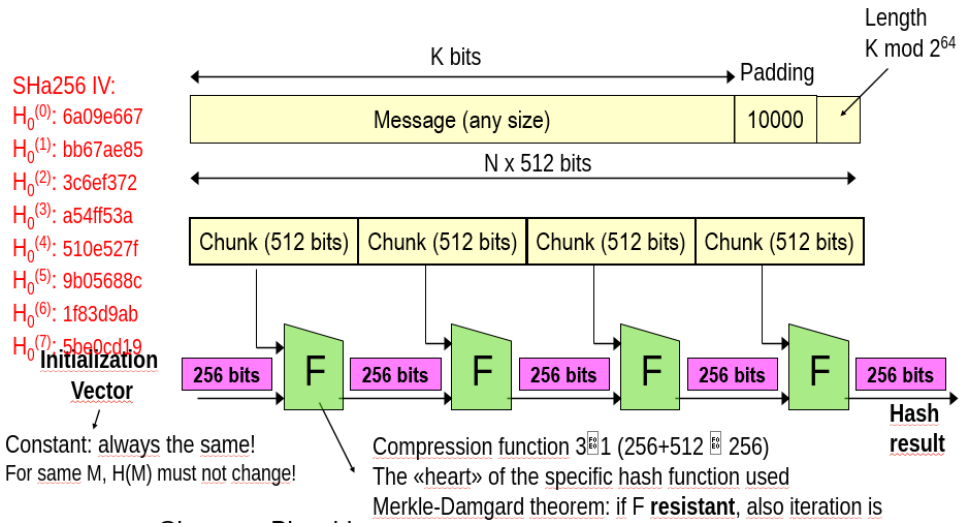

Figura 15: Costruzione Iterativa: Merkle-Damgard

Come vediamo da questo schema, al messaggio originale di K bits viene aggiunto un padding, ovvero 10000 (il numero di zeri varia, perchè serve per riempire l'ultimo chunk) e la lunghezza del messaggio, che sarà $K \mod 2^{64}$ A questo punto il messaggio viene diviso in N chunks, ognugno composto da 512 bits Viene poi aggiunto un vettore chiamato Initialization Vector (IV), composto da 8 valori, ovvero quelli che vediamo in foto, che sono ***COSTANTI*** (IMPORTANTE) A questo

punto la funzione hash lavora nel modo seguente:

- Prende il primo chunk (512 bits), e l'IV (256 bits), passa dentro la funzione F (che viene chiamata **Compression Block**) che ritorna un valore a 256 bits
- Procede iterativamente in questo modo fino a che non finisce i chunk
- Alla fine, il risultato sarà proprio il digest della funzione hash

Parliamo un secondo del blocco F

- è una funzione che come vediamo prende in input due valori, uno da 512 bits e uno da 256 bits, e ritorna come output un valore da 256 bits
- c'è un teorema molto importante di Merkle-Damgard che dice : fintanto che F è **resistente (sicura)**, allora anche l'iterazione è sicura
- alla prima esecuzione, il blocco F prende in input il primo chunk e l'IV

Vediamo ora come risolvere il problema del secret

4.1 Secret Suffix

Vediamo ora una delle vulnerabilità critiche della costruzione **Secret Suffix** per i MAC, ovvero quando si tenta di autenticare un messaggio calcolando:

$$MAC = H(Messaggio \mid\mid Chiave)$$

Dove "||" indica la concatenazione e la chiave segreta (Secret) viene messa **alla fine**.

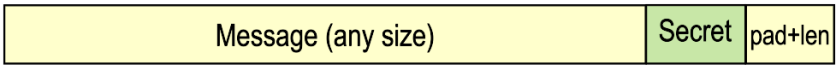

Figura 16: Secret come suffisso del messaggio

1. Attacco Brute-Force Ottimizzato ("State Precomputation")

Immaginiamo che un attaccante abbia intercettato un messaggio M molto lungo e il suo MAC, e voglia scoprire la Chiave Segreta (K) facendo un attacco di forza bruta (provando tutte le possibili chiavi).

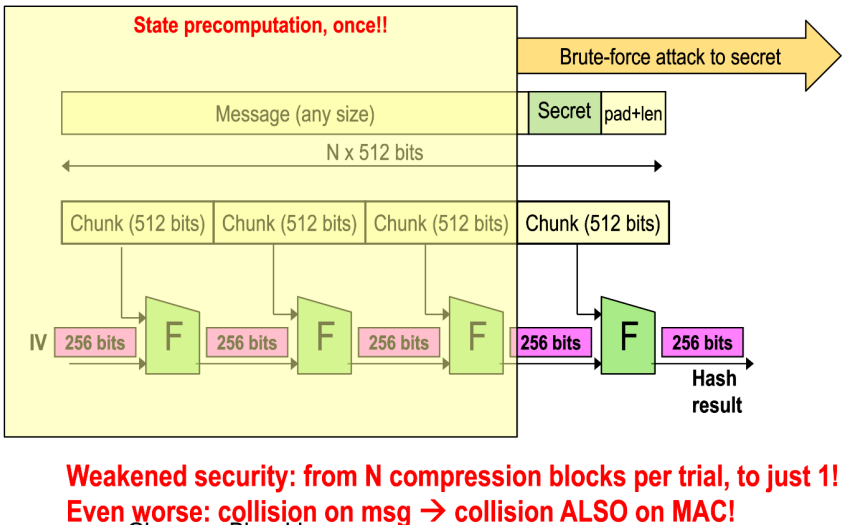

Figura 17: Schema Brute-Force per Secret Suffix

- **Scenario Normale (senza ottimizzazione):** Se la chiave fosse all'inizio o mescolata, per ogni tentativo di chiave ($K_1, K_2, K_3...$), l'attaccante dovrebbe ricalcolare l'hash dell'intero messaggio M (che potrebbe essere lungo, e quindi richiedere molto tempo). Se il messaggio è composto da N blocchi, il costo per ogni tentativo sarebbe N operazioni di compressione.

$$\text{Costo Totale} = \text{Numero Tentativi} \times N$$

- **Scenario Secret Suffix (con ottimizzazione):** Poiché la chiave è alla fine, l'attaccante osserva che la parte del messaggio M è nota e fissa.
  1. **Precomputazione:** L'attaccante calcola lo stato interno della funzione hash dopo aver processato tutto il messaggio M. Questo calcolo viene fatto **una volta sola**. Chiamiamo questo stato intermedio H_{stato}.
  2. **Attacco:** Ora, per provare una chiave candidata K', l'attaccante deve solo prendere lo stato precomputato H_{stato} e processare **solo l'ultimo blocco** che contiene la chiave.

Conseguenza: La sicurezza è drasticamente indebolita. Non importa se il messaggio è lungo 100 GB; l'attaccante "salta" tutta la computazione del messaggio e attacca solo l'ultimo blocco. Il costo per tentativo scende da N operazioni a **1 sola operazione**. Questo rende l'attacco brute-force molto più veloce.

2. Vulnerabilità alle Collisioni ("Collision on MSG -> Collision on MAC")

La slide menziona: *"Even worse: collision on msg → collision ALSO on MAC!"*. Questa è una debolezza strutturale ancora più grave.

Se l'attaccante riesce a trovare due messaggi diversi, M_1 e M_2, che producono lo stesso hash (una collisione sull'hash, senza chiave):

$$H(M_1) = H(M_2)$$

Allora, a causa della natura iterativa della funzione hash, lo stato interno della funzione dopo aver processato M_1 sarà identico a quello dopo aver processato M_2. Se aggiungiamo la stessa chiave segreta K alla fine di entrambi:

$$H(M_1 \mid\mid K) = H(M_2 \mid\mid K)$$

L'Attacco Pratico:

1. L'attaccante trova offline due messaggi M_1 (es. "Trasferisci 10€") e M_2 (es. "Trasferisci 1000€") che hanno lo stesso hash (collisione). Non serve conoscere la chiave per farlo.
2. L'attaccante chiede alla vittima di autenticare/firmare M_1. La vittima produce il MAC valido per M_1.
3. L'attaccante sostituisce M_1 con M_2 e allega lo stesso MAC.
4. Poiché $H(M_1||K) = H(M_2||K)$, il MAC risulta valido anche per M_2.

La costruzione **Secret Suffix** fallisce perché non isola la chiave dalle debolezze della funzione hash sottostante. Permette di velocizzare gli attacchi di forza bruta (precomputando lo stato del messaggio) e trasforma le collisioni dell'hash (che dovrebbero essere difficili ma gestibili) in falsificazioni immediate del MAC.

Queste vulnerabilità sono il motivo per cui è stato inventato **HMAC**, che usa una struttura annidata ($H(K \oplus opad \mid\mid H(K \oplus ipad \mid\mid M))$) proprio per prevenire sia l'attacco di estensione (tipico del Secret Prefix) sia le debolezze mostrate qui del Secret Suffix.

4.2 Secret Prefix

Vediamo invece cosa succede se mettiamo il secret come prefisso del messaggio, ovvero:

$$MAC = H(Chiave \mid\mid Messaggio)$$

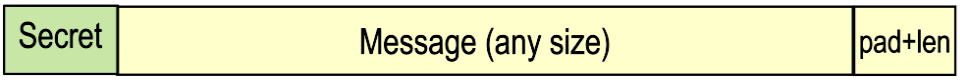

Figura 18: Secret come prefisso del messaggio

A prima vista sembra sicuro perché la chiave influenza tutto il calcolo fin dall'inizio. Tuttavia, questa costruzione è vulnerabile a causa della struttura iterativa di funzioni hash come MD5, SHA-1 e SHA-2 (la costruzione Merkle-Damgård).

Vediamo invece un attacco possibile, chiamato **Length Extension Attack**.

L'Attacco di Estensione della Lunghezza (Length Extension Attack)

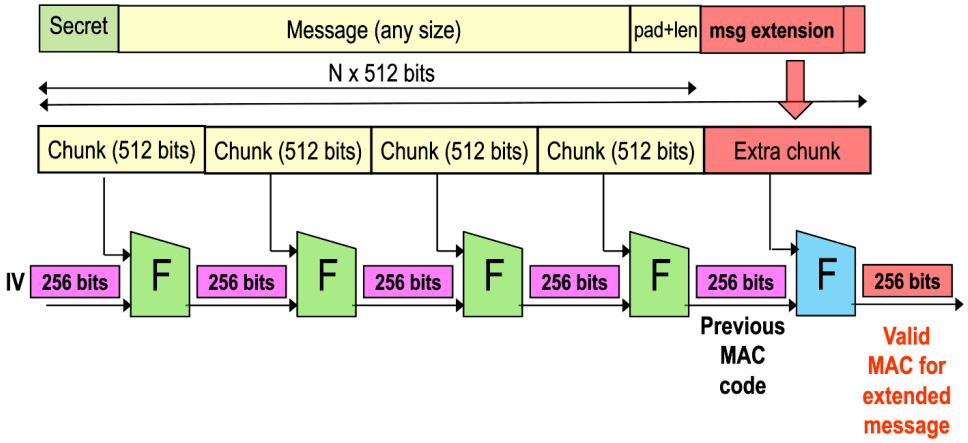

Figura 19: Lenght Extension Attack

Le funzioni hash operano a blocchi. Lo stato interno dell'hash dopo aver processato $K||M$ è l'output finale dell'hash originale. Un attaccante può prendere l'hash legittimo e usarlo come stato iniziale per processare un blocco aggiuntivo (estensione), calcolando un nuovo hash valido per il messaggio $M||Padding||Estensione$ senza conoscere K.

Vediamolo come "gioco a 2". Immaginiamo che:

- **Alice** invii a Bob un messaggio M: "Trasferisci 100€".
- Il sistema calcola il MAC: $Hash(Chiave \mid\mid M)$.
- **Eva** (l'attaccante) intercetta il messaggio M e il suo MAC. Eva **non conosce** la chiave segreta.

L'obiettivo di Eva è creare un nuovo messaggio valido senza conoscere la chiave. Vuole aggiungere qualcosa al messaggio originale, ad esempio "Trasferisci 100€ *...e 1000€ a Eva*".

Il processo:

1. **Partenza dal MAC Esistente:** Eva prende il `Previous MAC code` intercettato. Matematicamente, questo valore rappresenta lo stato della funzione hash *dopo* aver processato $(Chiave \mid\mid M \mid\mid Padding)$.
2. **Estensione:** Eva usa questo MAC esistente come **Stato Iniziale** (o IV) per la funzione hash, invece di usare l'IV standard.
3. **Aggiunta dell'Extra Chunk:** Eva fa processare alla funzione hash un nuovo blocco (l'`Extra chunk` o `msg extension`) partendo da quello stato interrotto.
4. **Risultato:** La funzione hash produce un nuovo output (`Valid MAC for extended message`).

Eva ha calcolato correttamente:

$$Nuovo\ MAC = H(Chiave \mid\mid M \mid\mid Padding \mid\mid Estensione)$$

È riuscita a farlo **senza mai conoscere la "Chiave"**. Ha semplicemente ripreso il calcolo da dove il sistema legittimo si era fermato.
L'immagine dimostra che nella costruzione **Secret Prefix**, conoscere l'hash di un messaggio equivale a conoscere lo stato interno della funzione hash. Questo permette a chiunque di aggiungere dati alla fine del messaggio (estenderlo) e calcolare il nuovo MAC valido, violando completamente l'integrità del sistema

4.3 HMAC: La Soluzione Standard

Per risolvere questi problemi, nel 1996 è stato introdotto **HMAC** (RFC 2104), standardizzato anche dal NIST. HMAC utilizza una struttura a due passaggi ("nested") che impedisce gli attacchi di estensione e isola l'uso della chiave.

HMAC tratta la funzione hash H come una "scatola nera". La formula di HMAC è:

$$HMAC(K, M) = H((K^+ \oplus opad) \mid\mid H((K^+ \oplus ipad) \mid\mid M))$$

Dove K^+ è la chiave K riempita con zeri (padding) fino alla dimensione del blocco della funzione hash.

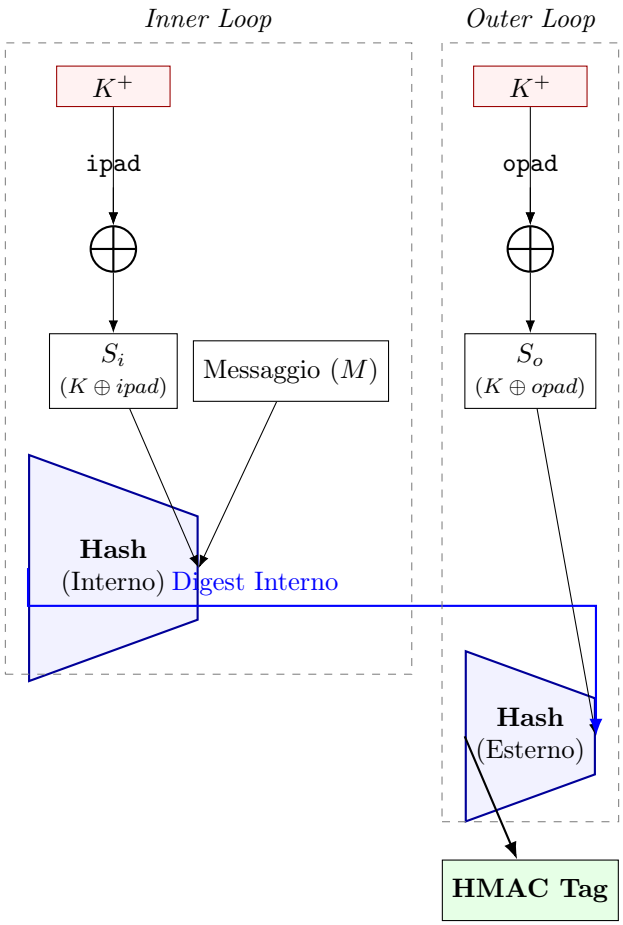

Figura 20: Struttura HMAC

Le costanti sono:

- **ipad (inner pad):** $0x36$ ripetuto (in bit sarebbe 00110110).
- **opad (outer pad):** $0x5C$ ripetuto (in bit sarebbe 01011100).

Funzionamento Logico:

1. Si deriva una chiave interna $K_{in} = K^+ \oplus ipad$.
2. Si calcola l'hash interno: $Hash_{in} = H(K_{in}||M)$.
3. Si deriva una chiave esterna $K_{out} = K^+ \oplus opad$.
4. Si calcola l'hash finale: $Tag = H(K_{out}||Hash_{in})$.

Spieghiamo la struttura di HMAC:

- All'inizio abbiamo i chunk del messaggio, a cui attacchiamo come prefisso la InnerKey (ovvero il risultato dell'operazione $K^+ \oplus ipad$).
- Il valore della InnerKey viene messo come input del primo CompressionBlock F, insieme agli IV, formando così una sorta di IV "cifrati" per il messaggio.
- Poi parte il processo iterativo classico della struttura Merkle-Damgard, fino a che non finiscono i chunk del messaggio.

  - Ora, se la costruzione si fosse fermata qua, saremmo ritornati al problema dell'attacco ad espansione, dato che l'ultimo valore generato, ovvero l'InnerHash è espandibile. Per questo si va avanti aggiungendo roba.

- Si prende poi il risultato di tutto il processo iterativo, e all'output risultate (InnerHash) viene aggiunto un padding, e una OuterKey (che come la corrispettiva precedente, è il risultato dell'operazione $K^+ \oplus opad$).
- L'OuterKey viene messa in input al CompressionBlock, insieme agli IV originali.
- Come ultimo passaggio, la concatenazione di InnerHash + Padding viene passata all'ultimo compression block insieme al risultato dell'operazione precedente, ottenendo così l'HMAC per il messaggio.

La cosa bella è che la complessità di tutta questa mega operazione viene incrementata di un fattore 2 rispetto alla costruzione di Merkle-Damgard, ovvero passiamo da $N \to N + 2$.

Questa struttura "blocca" il risultato dell'hash interno all'interno di un secondo hash, rendendo impossibile l'attacco di estensione della lunghezza. La sicurezza di HMAC è dimostrabile ed è legata alla pseudocasualità della funzione hash sottostante, non necessariamente alla sua resistenza alle collisioni.

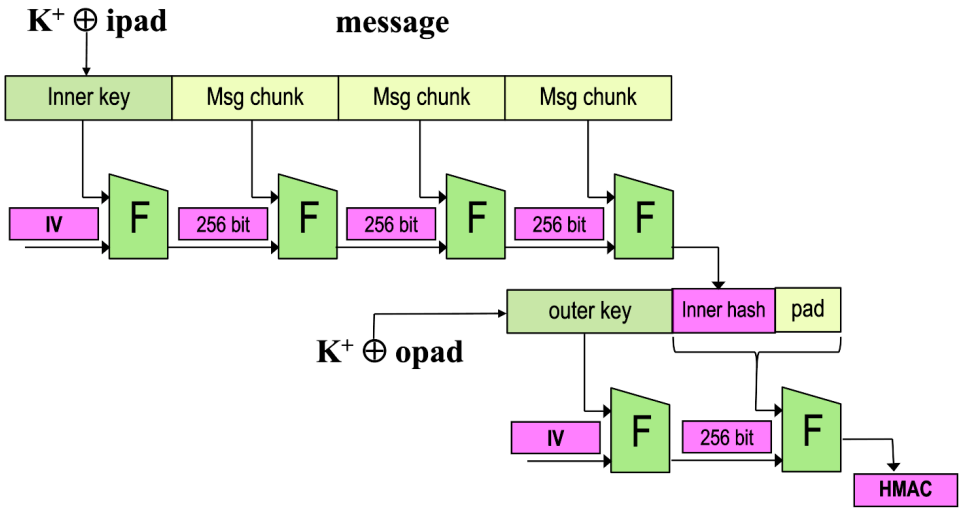

Figura 21: Struttura HMAC Bianchi

5 Conclusioni e Best Practices

1. **Non inventare crittografia:** Mai usare funzioni Hash "così come sono" per l'autenticazione o tentare di inserire la chiave in modi creativi (es. $H(M||K)$).
2. **Usare Standard:** Utilizzare **HMAC-SHA256** o **AES-CMAC** per l'integrità pura. Utilizzare **AES-GCM** se serve anche confidenzialità.
3. **Gestione Replay:** Implementare sempre Timestamp o Numeri di Sequenza nei protocolli di autenticazione per prevenire il riutilizzo dei messaggi validi.
4. **Verifica a Tempo Costante:** La verifica del tag deve avvenire in tempo costante per evitare attacchi laterali (timing attacks) che potrebbero rivelare il contenuto del tag byte per byte.