Concetto · 19 maggio 2026

← Precedente·Successivo →

Cos'è veramente SHA-256

Un'impronta matematica che sta in sessantaquattro caratteri e che cambia completamente se si sposta anche solo una virgola del testo originale. Perché lo chiamiamo sigillo di ceralacca digitale.

L'idea semplice dietro il nome tecnico

Immagina che esista una macchina con una sola fessura e un solo schermo. Attraverso la fessura inserisci un testo: una parola, una frase, un intero romanzo. Sullo schermo appare, pochi istanti dopo, una sequenza di esattamente sessantaquattro caratteri. Questa sequenza, per il lettore professionista, è chiamata *hash* o *riassunto crittografico*; per il lettore comune, possiamo chiamarla per ora un'impronta matematica del testo, proprio come l'impronta digitale lo è per una persona.

Se inserisci lo stesso testo due volte, la macchina mostra la stessa impronta entrambe le volte. Se inserisci un testo leggermente diverso —una virgola spostata, una maiuscola che diventa minuscola— la macchina mostra un'impronta completamente diversa dalla prima. Non simile: diversa. Queste due proprietà insieme —il determinismo e la sensibilità— sono l'idea semplice. Tutto il resto di SHA-256 è l'ingranaggio che le fa funzionare correttamente.

Vale la pena dire fin dall'inizio cosa la macchina non fa. Non cifra il testo. Non lo nasconde. Non lo salva. La macchina guarda il testo, calcola l'impronta e dimentica il testo. L'impronta non permette di ricostruire il testo che l'ha prodotta; permette solo, dato un testo candidato, di verificare se coincide o meno con l'originale. Per questo diciamo che è un riassunto *unidirezionale*: si va, non si torna indietro.

Un hash non è la stessa cosa della cifratura

La confusione è frequente ed è opportuno chiarirla: cifrare e fare l'hash sono operazioni diverse. Cifrare consiste nel trasformare un testo in modo che solo il possessore della chiave possa riportarlo alla sua forma originale. Fare l'hash consiste nel produrre un'impronta del testo dalla quale il testo originale non può mai essere recuperato, né con la chiave né senza. La prima è reversibile per progettazione; la seconda è irreversibile per progettazione.

La conseguenza pratica è importante. Quando un'applicazione dice «salviamo la tua password cifrata», c'è qualcuno che ha la chiave per decifrarla — l'applicazione stessa, in ogni caso. Quando un'applicazione dice «salviamo la tua password hashata», l'applicazione stessa non può leggere la password originale anche se volesse; può solo verificare se quella che scrivi produce nuovamente la stessa impronta. Il secondo modello, se fatto bene, è decisamente preferibile al primo per memorizzare le password. Vedremo più avanti perché «fatto bene» richiede qualcosa di più del solo SHA-256.

Le quattro proprietà che rendono utile un hash crittografico

Una funzione hash che merita l'aggettivo *crittografico* soddisfa quattro proprietà:

1. **Determinismo.** Lo stesso input produce sempre la stessa impronta.
2. **Effetto valanga.** Un piccolo cambiamento nell'input produce un'impronta completamente diversa, senza alcuna somiglianza visibile con la precedente.
3. **Resistenza all'inversione.** Data un'impronta, non è computazionalmente fattibile trovare il testo che l'ha prodotta.
4. **Resistenza alle collisioni.** Non è computazionalmente fattibile trovare due testi diversi che producano la stessa impronta.

«Non è computazionalmente fattibile» non significa «è matematicamente impossibile». Significa che il costo in termini di tempo, energia e denaro per riuscirci supera di ordini di grandezza la somma di tutta la capacità di calcolo ragionevolmente disponibile. Per SHA-256, questo limite si misura in migliaia di miliardi di anni anche per gli approcci più ottimistici con hardware specializzato. Il che, per gli scopi pratici del lettore, equivale a «non si può fare».

SHA-256, nello specifico

Il nome dice tutto. SHA sta per *Secure Hash Algorithm*: algoritmo di hash sicuro. Il numero 256 indica la dimensione dell'impronta in bit: duecentocinquantasei bit, ovvero trentadue byte, che visualizzati in esadecimale sono i sessantaquattro caratteri che il lettore già riconosce. Lo standard è stato pubblicato dal NIST statunitense, l'organismo che normalizza questo tipo di funzioni, nel 2001 come parte della famiglia SHA-2; la versione attuale dello standard, FIPS 180-4, risale al 2015.

Per chi non ha ancora presente cosa siano bit e byte:

```
1 bit    →  0 o 1        (un interruttore: acceso o spento)
1 byte   →  8 bit        (256 combinazioni possibili)
32 byte  →  256 bit      (l'impronta SHA-256)
```

Il numero 256 alla fine del nome indica la dimensione dell'impronta in bit. In esadecimale —un sistema di numerazione con sedici simboli invece di dieci— quei 256 bit occupano esattamente 64 caratteri. Questi sono i 64 caratteri che vedi in fondo a ogni Cuaderno.

Le dimensioni meritano un momento di riflessione. Duecentocinquantasei bit permettono due elevato alla duecentocinquantaseiesima valori diversi: un numero con settantotto cifre decimali, diversi ordini di grandezza superiore al numero stimato di atomi nell'universo osservabile. Ogni testo al mondo —ogni libro, ogni email, ogni messaggio— ricade su uno di questi valori. La probabilità che due testi diversi coincidano per caso è, ai fini pratici, indistinguibile dallo zero.

Come appare nel codice

In Zig, il linguaggio con cui scriviamo i componenti che sostengono Solo2, calcolare il sigillo SHA-256 di un testo appare così:

```
const std = @import("std");

const texto = "Cuadernos Lacre";
var resumen: [32]u8 = undefined;
std.crypto.hash.sha2.Sha256.hash(texto, &resumen, .{});
```

Abbiamo appena chiesto alla libreria standard di Zig di calcolare lo SHA-256 del testo tra virgolette. Dopo la chiamata, la variabile *resumen* contiene i trentadue byte che compongono il sigillo nella sua forma grezza; quando vengono visualizzati sullo schermo in esadecimale, sono i sessantaquattro caratteri che appaiono in fondo a questo articolo. Se cambiassimo *Cuadernos Lacre* in *Cuadernos lacre* —una maiuscola in meno— il sigillo cambierebbe interamente. Questa è, in cinque righe, la proprietà centrale che sostiene tutto il resto. Per chi vuole vedere come funziona internamente, alla fine dell'articolo includiamo una versione leggibile dell'algoritmo con commenti passo dopo passo.

Perché lo chiamiamo sigillo di ceralacca

Nella corrispondenza europea tra il XV e il XIX secolo, la ceralacca chiudeva la lettera. Una goccia di cera fusa, un sigillo premuto sopra, e la lettera rimaneva segnata in modo irripetibile. Non proteggeva il contenuto dal curioso deciso —la carta poteva essere letta in controluce, la ceralacca poteva essere rotta— ma lo rendeva evidente. Qualsiasi alterazione della chiusura era visibile al destinatario prima ancora di aprire la carta. La ceralacca non impediva il danno; lo dichiarava.

Lo SHA-256 del corpo di ogni Cuaderno svolge la stessa funzione nella sua versione digitale. Se una sola parola dell'articolo cambiasse tra il momento in cui è stato pubblicato e il momento in cui lo leggi, il sigillo esadecimale in fondo al testo non coinciderebbe più con lo SHA-256 del testo che hai davanti. Qualsiasi lettore con cinque righe di codice potrebbe verificarlo. La pubblicazione non può riscrivere la sua storia senza che il sigillo la tradisca. Non protegge dal danno; lo rende verificabile.

Cosa un hash non è

A SHA-256 vengono a volte richiesti quattro usi che non gli competono:

1. **Cifratura.** Un hash riassume; non nasconde. Se vuoi che il testo non sia leggibile, devi cifrarlo, non farne l'hash.
2. **Autenticazione dell'autore.** Un hash non dice chi ha scritto il testo, ma solo quale testo è stato hashato. Per associare la paternità è necessaria una firma crittografica sopra l'hash, non l'hash da solo.
3. **Memorizzazione delle password.** Qui c'è una trappola che conviene capire. SHA-256 è progettato per essere molto veloce —il che è positivo per molte cose, ma negativo per questa. Un attaccante con hardware specializzato può provare miliardi di password al secondo contro un hash SHA-256 fino a trovare la tua. Per salvare le password bisogna usare funzioni di derivazione della chiave deliberatamente lente come Argon2, scrypt o bcrypt, combinate con un *sale* (un dato casuale unico per utente, che impedisce a due persone con la stessa password di avere lo stesso hash).
4. **Leggere l'hash come identificatore dell'autore.** Non lo è. Un hash identifica il contenuto. Se due persone fanno l'hash della parola *ciao* con SHA-256, entrambe ottengono lo stesso riassunto — e questa è la proprietà centrale, non un difetto: se fossero riassunti diversi, non potremmo verificare la corrispondenza tra quanto pubblicato e quanto ricevuto.

Dove appare SHA-256 nella tua vita quotidiana

Anche se non lo vedi, SHA-256 sostiene buona parte di ciò che usi quotidianamente su internet. La blockchain di Bitcoin si costruisce concatenando lo SHA-256 di ogni blocco a quello successivo; alterare un blocco passato costringe a ricalcolare l'intera catena successiva. Git, il sistema con cui viene versionato il codice di mezzo mondo, identifica ogni commit tramite lo SHA-256 (nelle versioni recenti) o il suo predecessore SHA-1 (nelle versioni più vecchie) del suo contenuto completo. I certificati HTTPS che verificano l'identità di un sito web quando vi accedi hanno un'impronta SHA-256 associata. I download di software sono spesso accompagnati da uno SHA-256 pubblicato dallo sviluppatore affinché tu possa verificare che il file non sia stato alterato durante il percorso. E, come abbiamo detto, in fondo a ogni Cuaderno Lacre.

Per il lettore professionista

Quattro promemoria operativi per chi decide o audita sistemi:

1. Hash non è cifratura. Se un fornitore confonde i due termini nella sua documentazione tecnica, è opportuno chiedere cosa intenda esattamente.
2. Per memorizzare le password non si deve mai usare SHA-256 da solo. SHA-256 è troppo veloce per questo compito (vedi punto 3 di *Cosa un hash non è*). Lo standard attuale è **Argon2id**: lento per progettazione, configurabile in base alla capacità del server, combinato con un *sale* casuale diverso per ogni utente.
3. Per l'integrità dei documenti —contratti, fascicoli, archivi— SHA-256 rimane lo standard di riferimento. È quello utilizzato dai marcatori temporali qualificati nell'UE.
4. Per la conservazione a lungo termine (decenni) è opportuno calcolare e archiviare anche un SHA-3 o un SHA-512 insieme a SHA-256; la prudenza crittografica raccomanda di non affidarsi a una singola funzione per archivi secolari.

Tecnicamente, questa struttura iterata — in cui lo stato intermedio viene conservato tra i blocchi di input — è nota come costruzione **Merkle-Damgård**, il modello su cui si basano SHA-1, SHA-2 (incluso SHA-256) e molte altre funzioni hash classiche. SHA-3, al contrario, abbandona Merkle-Damgård in favore di un'architettura diversa chiamata *spugna (sponge)*.

Come funziona lo SHA-256, passo dopo passo, in parole povere

Immagina di aver montato il circuito di domino più elaborato del mondo: migliaia di tessere (fichas), decine di biforcazioni, ponti meccanici e rampe che attraversano l'intera stanza, posizionate con cura pezzo dopo pezzo.

Se dai un tocco alla prima tessera, la catena cade in una sequenza precisa e ripetibile. Stesso montaggio, stesso tocco iniziale → identico schema finale di tessere cadute, volta dopo volta.

Ecco la cosa interessante: sposta **una sola tessera** di mezzo centimetro da un lato prima di iniziare e tocca di nuovo. Una rampa che doveva attivarsi rimane inerte, un ponte non cade, scatta una biforcazione diversa. Lo schema finale delle tessere a terra è completamente irriconoscibile rispetto al primo.

Lo SHA-256 è matematicamente questo circuito. Il testo che scrivi è la posizione iniziale delle tessere. L'algoritmo è il tocco che libera la cascata. E il risultato finale — quello che chiamiamo *hash* — è la foto fissa del pavimento quando tutto si è fermato. Cambia una sola virgola del testo originale e la foto sarà radicalmente diversa. Così semplice, e così drastico.

Passo 1. Tradurre il testo in tessere binarie. I computer non capiscono le lettere; le traducono prima in numeri (ASCII) e i numeri in binario (uno e zero). Ogni lettera si trasforma in 8 tessere bianche o nere: la *A* è `01000001`, la *B* è `01000010`, lo spazio è `00100000`. L'intero tuo testo — una parola, un contratto, un romanzo — diventa una lunga fila di tessere bianche e nere.

Passo 2. Riempire fino alla dimensione standard. Il circuito elabora la fila in *tratti* di esattamente 512 tessere. Se il tuo messaggio non raggiunge un multiplo di 512, viene aggiunta una tessera marcatore (quella col valore `10000000`) subito dopo il testo e poi degli zeri fino a completare il tratto. Le ultime 64 posizioni di ogni tratto sono riservate per annotare la lunghezza originale del testo. Così il circuito sa sempre dove finisce il contenuto reale e dove inizia il riempimento.

Passo 3. Collocare le otto tessere maestre. Prima di iniziare, sistemiamo sul tavolo **otto tessere maestre** in una posizione iniziale precisa. Queste otto tessere non sono un segreto: il loro valore iniziale è fissato da una regola matematica pubblica (le radici quadrate dei primi otto numeri primi — 2, 3, 5, 7, 11, 13, 17, 19 — e i primi bit della parte decimale di ogni radice). Chiunque, in ogni angolo del pianeta, inizia con le stesse otto tessere maestre nella stessa posizione. Il loro destino è essere spinte e trasformate dalla cascata.

Passo 4. La grande cascata: sessantaquattro round di spinte. Qui inizia lo spettacolo. Il primo tratto di 512 tessere del tuo testo viene fatto urtare contro le otto tessere maestre. Ma non cadono di colpo: il meccanismo esegue **sessantaquattro round consecutivi**. In ogni round compie tre operazioni con le tessere:

- **La Giostra (Tiovivo)** (rotazione). Le tessere si muovono in cerchio: quelle a destra passano a sinistra. Nessuna tessera viene persa né aggiunta; semplicemente si riordinano facendo un giro completo di giostra. È un modo economico e reversibile per ridistribuire l'informazione.
- **L'Imbuto Logico (Embudo Lógico)** (XOR). Le tessere passano attraverso un imbuto che le confronta a due a due: se sono dello stesso colore, ne esce una bianca; si sono diverse, ne esce una nera. È l'operazione più semplice della logica binaria, ma combinata con le rotazioni della giostra diventa potentissima per mescolare le informazioni senza perderle.
- **Il Trabocco (Desborde)** (somma modulare). Il risultato viene sommato a una *tessera di spinta costante* tratta da una lista pubblica di sessantaquattro costanti (le radici cubiche dei primi sessantatré numeri primi). Se la somma genera tessere extra che non entrano nello spazio di 32 tessere previsto, quelle tessere in eccedenza vengono scartate. Il tavolo ha spazio solo per 32 tessere, non una di più.

Alla fine del round sessantaquattro, ognuna delle tessere del tratto del tuo testo ha influenzato la posizione delle otto tessere maestre. L'energia della spinta ha viaggiato attraverso l'intero circuito.

Passo 5. Aggiungere il tratto successivo (senza riavviare). Se il tuo testo era lungo e rimane un altro tratto di 512 tessere da elaborare, **il circuito non si riavvia**. Le otto tessere maestre rimangono come le ha lasciate la prima cascata, e il secondo tratto viene lanciato contro di esse per attivare altri sessantaquattro round. È come aggiungere una nuova stanza piena di domino alla fine di quella appena caduta: il disordine della prima condiziona interamente come cadrà la seconda.

Passo 6. Fare la foto finale. Quando non rimangono più tratti da elaborare, la cascata si ferma. Guardiamo la posizione finale in cui sono rimaste le otto tessere maestre. Traduciamo la loro configurazione in un codice di lettere e numeri in sistema esadecimale. Il risultato è una stringa di esattamente sessantaquattro caratteri: quello è il tuo sigillo SHA-256.

Quattro proprietà derivano da come è montato il circuito:

1. **Determinismo.** Lo stesso testo produce sempre la stessa foto finale, su qualsiasi computer del mondo. Zero casualità, zero sorprese.
2. **Effetto valanga.** Una virgola aggiunta, una maiuscola cambiata, un accento dimenticato: la foto risulta completamente irriconoscibile. Questa è la sensibilità estrema che abbiamo già descritto all'inizio.
3. **Sola andata.** Data la foto finale, non puoi ricostruire il testo originale. Le rotazioni, gli imbuti e i trabocchi distruggono ogni informazione direzionale su *da dove proveniva ogni bit* e conservano solo *cosa è stato sommato in*

totale.

4. **Resistenza alle collisioni.** In venticinque anni di crittoanalisi pubblica, nessuno è riuscito a trovare due testi diversi le cui foto finali coincidano. E la difficoltà di farlo è al di fuori della portata computazionale di qualsiasi civiltà ragionevolmente immaginabile.

L'appendice di codice che segue implementa esattamente questi sei passi in Zig. Ora puoi leggerlo sapendo cosa significa ogni operazione sui bit, invece di accettare le manipolazioni alla cieca.

Glossario tecnico

Per il lettore che vuole capire cosa fa ogni operazione. Saltalo pure: l'articolo resta comprensibile anche senza.

ASCII e Unicode — come le lettere diventano numeri. I computer non vedono lettere; vedono numeri. Uno standard chiamato **ASCII** (*American Standard Code for Information Interchange*, del 1963) assegna a ogni carattere della tastiera un numero specifico: la *A* è 65, la *B* è 66, la *a* è 97, lo *0* è 48, lo spazio è 32, la virgola è 44. I sistemi moderni lo estendono con **Unicode**, che assegna un numero a ogni carattere di ogni alfabeto del mondo: cirillico, arabo, cinese, giapponese e persino le emoji. Quando scrivi un carattere o apri un file di testo, il computer legge il numero di fondo, non la forma a video. Lo SHA-256 lavora su questi numeri, trattando qualsiasi testo come una lunga sequenza di cifre. Per questo può sigillare un articolo in spagnolo, un poema in giapponese e un file binario con lo stesso algoritmo.
XOR — il comparador bit a bit. XOR (pronunciato *«exor»*, dall'inglese *exclusive or*, «o esclusivo») è una delle operazioni più semplici che un computer possa fare con due numeri binari. Confronta due bit posizione per posizione e restituisce: **1** se esattamente uno dei due è 1 (uno ma non entrambi), **0** se entrambi sono uguali (entrambi 0 o entrambi 1). Esempio: lo XOR di `1010` e `1100` è `0110`. Ha una proprietà notevole: è reversibile — se fai lo XOR due volte con la stessa chiave, torni all'originale. Per questo è il cavallo di battaglia della crittografia: mescola i bit senza perdere informazioni, ma il risultato non rivela nulla sugli input se non ne conosci uno.
Esadecimale — contare in base 16. Quasi tutti i numeri della vita quotidiana usano dieci cifre (0-9). L'esadecimale ne usa sedici: le abituali 0-9 più sei lettere che rappresentano i seguenti valori: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Perché sedici? Perché i computer ragionano in gruppi di quattro bit, e quattro bit possono rappresentare esattamente sedici valori diversi — così, un carattere esadecimale corrisponde perfettamente a quattro bit. Un'impronta (huella) SHA-256 misura 256 bit, che sono esattamente **64 caratteri esadecimali**. Se la scrivessimo in decimale corrente, occuperebbe circa 78 cifre e risulterebbe più scomoda. La scelta è estetica e compatta; il numero di fondo è lo stesso.
Rotazione dei bit — la giostra binaria (tiovivo binario). Immagina una fila di sette lampadine, alcune accese (1) e altre spente (0): `1 0 1 1 0 0 1`. Ruotare a destra di una posizione consiste nel prendere la lampadina all'estrema destra, portarla all'estrema sinistra e spostare le altre di un posto a destra: `1 1 0 1 1 0 0`. Nessuna lampadina va persa né viene aggiunta: semplicemente danzano in cerchio. Lo SHA-256 utilizza la rotazione dei bit centinaia di volte in ogni calcolo; è un modo economico e senza perdite di ridistribuire l'informazione all'interno dello stato.
Costanti *«nothing-up-my-sleeve»* — perché derivano dai numeri primi. Le otto tessere maestre e le sessantaquattro costanti di round dello SHA-256 non sono state scelte a caso. Derivano dalle radici quadrate e cubiche dei primi numeri primi. Perché? Perché i loro progettisti volevano costanti *«senza nulla sotto la manica»* (*«nothing-up-my-sleeve»*): valori la cui origine chiunque possa verificare. Se qualcuno ti dicesse *«fidati di me: usa questo numero casuale a 32 bit»*, sospetteresti ragionevolmente di una debolezza occulta o di una backdoor. Ma chiunque abbia una calcolatrice può verificare che i primi 32 bit della radice quadrata di 2 sono `0x6a09e667`. I valori sono matematici, pubblici e riproducibili: nessun trucco nascosto può infilarsi nella ricetta.

Appendice: SHA-256 in codice leggibile

Questa appendice è per il lettore che vuole vedere l'algoritmo dall'interno. È un'implementazione didattica in Zig che segue la specifica FIPS 180-4. Non è la versione utilizzata da Solo2 —quella reale si trova in `std.crypto.hash.sha2.Sha256` della libreria standard di Zig, ottimizzata e auditata. Ma l'algoritmo è lo stesso: quello che vedi qui è, passo dopo passo, ciò che accade quando quella chiamata di cinque caratteri esegue il suo lavoro.

```
const std = @import("std");

// SHA-256 — implementación didáctica.
// Sigue la especificación FIPS 180-4. Prioriza la claridad sobre la
// velocidad y la robustez frente a entradas hostiles. Para producción,
// usa std.crypto.hash.sha2.Sha256, que está optimizada y auditada.

// H0: las ocho palabras del estado inicial. Primeros 32 bits de la parte
```

```
// fraccionaria de las raíces cuadradas de los primeros ocho primos
// (2, 3, 5, 7, 11, 13, 17, 19).
const H0 = [_]u32{
    0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
    0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19,
};

// K: 64 constantes de ronda. Primeros 32 bits de la parte fraccionaria
// de las raíces cúbicas de los primeros 64 primos.
const K = [_]u32{
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2,
};

// Rotación circular a la derecha de un u32.
inline fn rotr(x: u32, n: u5) u32 {
    return std.math.rotr(u32, x, n);
}

// Lee 4 bytes consecutivos como un u32 big-endian.
inline fn readU32(b: []const u8) u32 {
    return @as(u32, b[0]) << 24 | @as(u32, b[1]) << 16 | @as(u32, b[2]) << 8 | @as(u32, b[3]);
}

// Escribe un u32 como 4 bytes consecutivos big-endian.
inline fn writeU32(b: []u8, v: u32) void {
    b[0] = @truncate(v >> 24);
    b[1] = @truncate(v >> 16);
    b[2] = @truncate(v >> 8);
    b[3] = @truncate(v);
}

// Compresión de un bloque de 64 bytes sobre el estado del hash. Sigue §6.2.2 de FIPS 180-4.
fn compress(state: *[8]u32, block: [16]u32) void {

    // 1. Expansión del schedule: 16 palabras → 64. Las nuevas se obtienen
    //    combinando cuatro anteriores con dos funciones de mezcla (s0 y s1)
    //    que usan rotación, XOR y desplazamiento. El "+%" es suma con
    //    truncado u32 (overflow-wrap), tal como exige el estándar.
    var w: [64]u32 = undefined;
    for (0..16) |i| w[i] = block[i];
    for (16..64) |i| {
        const s0 = rotr(w[i-15], 7) ^ rotr(w[i-15], 18) ^ (w[i-15] >> 3);
        const s1 = rotr(w[i-2], 17) ^ rotr(w[i-2], 19) ^ (w[i-2] >> 10);
        w[i] = w[i-16] +% s0 +% w[i-7] +% s1;
    }

    // 2. Variables de trabajo: copia del estado actual.
    var a = state[0]; var b = state[1]; var c = state[2]; var d = state[3];
    var e = state[4]; var f = state[5]; var g = state[6]; var h = state[7];

    // 3. 64 rondas de mezcla no lineal.
    //    S1, S0  : combinaciones rotacionales de 'e' y 'a'.
    //    ch      : "choose" — multiplexor bit a bit, elige entre f y g según e.
    //    maj     : "majority" — bit mayoritario entre a, b, c.
    //    t1 + t2 : se inyecta al top de la cascada cada ronda.
    for (0..64) |i| {
        const S1 = rotr(e, 6) ^ rotr(e, 11) ^ rotr(e, 25);
        const ch = (e & f) ^ (~e & g);
        const t1 = h +% S1 +% ch +% K[i] +% w[i];
        const S0 = rotr(a, 2) ^ rotr(a, 13) ^ rotr(a, 22);
        const maj = (a & b) ^ (a & c) ^ (b & c);
        const t2 = S0 +% maj;
        h = g; g = f; f = e; e = d +% t1;
        d = c; c = b; b = a; a = t1 +% t2;
```

```
    }

    // 4. Acumular las variables de trabajo en el estado.
    state[0] +%= a; state[1] +%= b; state[2] +%= c; state[3] +%= d;
    state[4] +%= e; state[5] +%= f; state[6] +%= g; state[7] +%= h;
}

// Hash completo: procesa el mensaje en bloques, padea el último, escribe el resumen.
pub fn sha256(msg: []const u8, out: *[32]u8) void {
    var state = H0;
    var block: [64]u8 = undefined;
    var block_w: [16]u32 = undefined;

    // Procesar bloques completos del mensaje original.
    var i: usize = 0;
    while (i + 64 <= msg.len) : (i += 64) {
        @memcpy(block[0..64], msg[i..i+64]);
        for (0..16) |j| block_w[j] = readU32(block[j*4..j*4+4]);
        compress(&state, block_w);
    }

    // Padding del último bloque: byte 0x80, después ceros, después la
    // longitud original (en bits) como u64 big-endian en los 8 últimos bytes.
    const remaining = msg.len - i;
    @memcpy(block[0..remaining], msg[i..]);
    block[remaining] = 0x80;
    const bit_len: u64 = @as(u64, msg.len) * 8;

    if (remaining + 1 + 8 <= 64) {
        // El padding cabe en el mismo bloque.
        for (remaining + 1..56) |k| block[k] = 0;
        var k: usize = 0;
        while (k < 8) : (k += 1) block[56 + k] = @truncate(bit_len >> @as(u6, @intCast((7 - k) * 8)));
        for (0..16) |j| block_w[j] = readU32(block[j*4..j*4+4]);
        compress(&state, block_w);
    } else {
        // El padding requiere un bloque adicional.
        for (remaining + 1..64) |k| block[k] = 0;
        for (0..16) |j| block_w[j] = readU32(block[j*4..j*4+4]);
        compress(&state, block_w);
        for (0..56) |k| block[k] = 0;
        var k: usize = 0;
        while (k < 8) : (k += 1) block[56 + k] = @truncate(bit_len >> @as(u6, @intCast((7 - k) * 8)));
        for (0..16) |j| block_w[j] = readU32(block[j*4..j*4+4]);
        compress(&state, block_w);
    }

    // Escribir el estado final como 32 bytes big-endian.
    for (0..8) |j| writeU32(out[j*4..j*4+4], state[j]);
}

// Ejemplo de uso.
pub fn main() void {
    var resumen: [32]u8 = undefined;
    sha256("Cuadernos Lacre", &resumen);
    for (resumen) |byte| std.debug.print("{x:0>2}", .{byte});
    std.debug.print("\n", .{});
    // Imprime: ae6bdea6bbf5476889e0651a31f3dc1612fc61497477e21a95cabae2a6886c3e
}
```

Qualsiasi riscrittura in un altro linguaggio che segua la stessa struttura —costantes iniziali, espansione dello schedule, sessantaquattro round, accumulo— produce lo stesso risultato. L'algoritmo non ha segreti: il suo valore risiede nel fatto che le proprietà elencate sopra continuano a reggere dopo due decenni di crittoanalisi pubblica condotta da migliaia di esperti.

Se torni in fondo a questo articolo, vedrai un sigillo esadecimale di sessantaquattro caratteri. È lo SHA-256 del testo che hai appena letto, in questa lingua. Se traducessimo l'articolo, il sigillo sarebbe diverso; se cambiasse una parola della versione italiana, il sigillo italiano cambierebbe. Il sigillo non protegge il contenuto —per quello ci sono altri strumenti— ma lo identifica univocamente. E questo, per quanto modesto possa sembrare, basta affinché nessun passaggio della

catena editoriale possa alterare quanto detto senza che si noti. Tutto il resto —cifrare, firmare, identificare— si costruisce sopra questa idea semplice.

Fonti e letture aggiuntive

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, agosto 2015. Specifica ufficiale della famiglia SHA-2, incluso SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, maggio 2011. Versione normativa per gli implementatori.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). I capitoli 5 e 6 trattano le funzioni hash e i loro usi legittimi e illegittimi.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Esempio pratico dell'uso di SHA-256 per concatenare i blocchi in una struttura immutabile per costruzione.
- Regolamento (UE) 910/2014 (eIDAS) — quadro dei marcatori temporali qualificati. SHA-256 è la funzione di riferimento per le firme e i sigilli elettronici qualificati emessi nell'UE.
- Implementazione di riferimento in Zig: `std.crypto.hash.sha2.Sha256` nel repository ufficiale del linguaggio (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). È la versione ottimizzata e auditata effettivamente utilizzata da Solo2. Utile per il confronto con l'implementazione didattica dell'appendice.

← PrecedenteCUADERNOS_LIST_SCHREMS_TITLESuccessivo →CUADERNOS_LIST_KILLSWITCH_TITLE

Letture recenti

- CUADERNOS_LIST_PREGUNTAS_TITLE
- CUADERNOS_LIST_SELFHOST_TITLE
- CUADERNOS_LIST_IDENTIDAD_TITLE

Porta questo articolo dove ne hai bisogno.

↓ Markdown ↓ Testo semplice ↓ PDF

Il file viene scaricato sul tuo dispositivo. Da lì puoi salvarlo, importarlo in Solo2 o condividerlo come preferisci. Cuadernos non decide la destinazione per te.

Sigillo di cera · SHA-256 1b489087a469a95dcdc56e2d91930a8d8e5d43f5d1237ea06119a438e5da6bc2

Cuadernos Lacre · Una pubblicazione di Menzuri Gestión S.L. ·
scritta da R.Eugenio · a cura del team di Solo2.