Concepto · 19 de mayo de 2026

← Anterior·Siguiente →

Qué es realmente SHA-256

Una huella matemática que cabe en sesenta y cuatro caracteres y que cambia entera si una sola coma del texto original se mueve. Por qué le llamamos sello de lacre digital.

La idea sencilla detrás del nombre técnico

Imagina que existe una máquina con una sola ranura y una sola pantalla. Por la ranura introduces un texto: una palabra, una frase, una novela entera. En la pantalla aparece, instantes después, una secuencia exactamente de sesenta y cuatro caracteres. Esa secuencia, al lector profesional la llamamos *hash* o *resumen criptográfico*; al lector general, podemos llamarla por ahora una huella matemática del texto, como la huella dactilar lo es de una persona.

Si introduces el mismo texto dos veces, la máquina muestra la misma huella las dos veces. Si introduces un texto ligeramente distinto —una sola coma desplazada, una mayúscula que pasa a minúscula— la máquina muestra una huella completamente distinta de la primera. No parecida: distinta. Esas dos propiedades juntas —el determinismo y la sensibilidad— son la idea sencilla. Todo lo demás del SHA-256 es la maquinaria que las hace cumplir bien.

Conviene decir desde el principio lo que la máquina no hace. No cifra el texto. No lo oculta. No lo guarda. La máquina mira el texto, calcula la huella, y se olvida del texto. La huella no permite reconstruir el texto que la produjo; solo permite, dado un texto candidato, comprobar si coincide o no con el original. Por eso decimos que es un resumen *de una sola dirección*: se va, no se vuelve.

Un hash no es lo mismo que cifrar

La confusión es frecuente y conviene despejarla: cifrar y hashear son operaciones distintas. Cifrar consiste en transformar un texto de forma que solo el poseedor de la clave pueda devolverlo a su forma original. Hashear consiste en producir una huella del texto de la que el texto original no se puede recuperar nunca, ni con clave ni sin ella. La primera es reversible por diseño; la segunda, irreversible por diseño.

La consecuencia práctica importa. Cuando una aplicación dice «guardamos tu contraseña cifrada», hay alguien que tiene la clave para descifrarla — la aplicación misma, en cualquier caso. Cuando una aplicación dice «guardamos tu contraseña hasheada», la aplicación misma no puede leer la contraseña original aunque quisiera; solo puede comprobar si la que tú escribes vuelve a producir la misma huella. El segundo modelo, hecho bien, es muy preferible al primero para almacenar contraseñas. Más adelante veremos por qué «hecho bien» exige algo más que SHA-256 a secas.

Las cuatro propiedades que hacen útil un hash criptográfico

Una función hash que merece el adjetivo *criptográfico* cumple cuatro propiedades:

1. **Determinismo.** La misma entrada produce siempre la misma huella.
2. **Efecto avalancha.** Un cambio pequeño en la entrada produce una huella completamente distinta, sin parecido visible con la anterior.
3. **Resistencia a la inversión.** Dada una huella, no es viable computacionalmente encontrar el texto que la produjo.
4. **Resistencia a colisiones.** No es viable computacionalmente encontrar dos textos distintos que produzcan la misma huella.

«No es viable computacionalmente» no significa «es matemáticamente imposible». Significa que el coste en tiempo, energía y dinero de conseguirlo excede en órdenes de magnitud la suma de toda la capacidad de cómputo razonablemente

disponible. Para SHA-256, esa cota se mide en miles de billones de años incluso para los planteamientos más optimistas con hardware especializado. Lo cual, a efectos prácticos del lector, es lo mismo que «no se puede».

SHA-256, en concreto

El nombre lo dice todo. SHA son las siglas de *Secure Hash Algorithm*: algoritmo de hash seguro. El número 256 indica el tamaño de la huella en bits: doscientos cincuenta y seis bits, es decir treinta y dos bytes, que mostrados en hexadecimal son los sesenta y cuatro caracteres que el lector reconoce ya. El estándar lo publicó el NIST estadounidense, el organismo que normaliza este tipo de funciones, en 2001 como parte de la familia SHA-2; la versión vigente del estándar, FIPS 180-4, es de 2015.

Para quien aún no tiene presente qué son bits y bytes:

```
1 bit    →  0 ó 1        (un interruptor: encendido o apagado)
1 byte   →  8 bits       (256 combinaciones posibles)
32 bytes →  256 bits     (la huella SHA-256)
```

El número 256 al final del nombre dice el tamaño de la huella en bits. En hexadecimal —un sistema de numeración con dieciséis símbolos en lugar de diez— esos 256 bits caben en exactamente 64 caracteres. Esos son los 64 caracteres que ves al pie de cada Cuaderno.

Las dimensiones merecen un instante. Doscientos cincuenta y seis bits permiten dos elevado a doscientos cincuenta y seis valores distintos: un número con setenta y ocho dígitos decimales, varios órdenes de magnitud mayor que el número estimado de átomos en el universo observable. Cada texto del mundo —cada libro, cada correo electrónico, cada mensaje — cae sobre uno de esos valores. La probabilidad de que dos textos distintos coincidan por azar es, a efectos prácticos, indistinguible de cero.

Cómo se ve en código

En Zig, lenguaje en el que escribimos las piezas que sostienen Solo2, calcular el sello SHA-256 de un texto se ve así:

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

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

Acabamos de pedirle a la biblioteca estándar de Zig que calcule el SHA-256 del texto entre comillas. Después de la llamada, la variable *resumen* contiene los treinta y dos bytes que componen el sello en su forma cruda; cuando se muestran en pantalla en hexadecimal, son los sesenta y cuatro caracteres que aparecen al pie de este artículo. Si cambiáramos *Cuadernos Lacre* por *Cuadernos lacre* —una mayúscula menos— el sello cambiaría entero. Esa es, en cinco líneas, la propiedad central que sostiene el resto. Para quien quiera ver cómo funciona internamente, al final del artículo incluimos una versión legible del algoritmo con comentarios paso a paso.

Por qué le llamamos sello de lacre

En la correspondencia europea de los siglos quince al diecinueve, el lacre cerraba la carta. Una gota de cera derretida, un sello presionado encima, y la carta quedaba marcada de forma irrepetible. No protegía el contenido del fisgón decidido — el papel se podía leer al trasluz, el lacre se podía romper— pero sí lo evidenciaba. Cualquier alteración del cierre era visible al destinatario antes incluso de abrir el papel. El lacre no impedía el daño; lo declaraba.

El SHA-256 del cuerpo de cada Cuaderno cumple la misma función en su versión digital. Si una sola palabra del artículo cambiara entre el momento en que se publicó y el momento en que tú lo lees, el sello hexadecimal al pie del texto ya no coincidiría con el SHA-256 del texto que tienes delante. Cualquier lector con cinco líneas de código podría comprobarlo. La publicación no puede reescribir su historia sin que el sello la delate. No protege contra el daño; lo hace verificable.

Lo que un hash no es

Cuatro usos se le piden a veces a SHA-256 que no le corresponden:

1. **Cifrar.** Un hash resume; no oculta. Si quieres que el texto no se pueda leer, necesitas cifrarlo, no hashearlo.
2. **Autenticar al autor.** Un hash no dice quién escribió el texto, solo qué texto se hasheó. Para asociar autoría hace falta una firma criptográfica encima del hash, no el hash a secas.
3. **Almacenar contraseñas.** Aquí hay una trampa que conviene entender. SHA-256 está diseñado para ser muy rápido —lo cual es bueno para muchas cosas, pero malo para esta. Un atacante con hardware especializado puede probar miles de millones de contraseñas por segundo contra un hash SHA-256 hasta dar con la tuya. Para guardar contraseñas hay que usar funciones de derivación de clave deliberadamente lentas como Argon2, scrypt o bcrypt, combinadas con una *sal* (un dato aleatorio único por usuario, que evita que dos personas con la misma contraseña tengan el mismo hash).
4. **Leer el hash como identificador del autor.** No lo es. Un hash identifica el contenido. Si dos personas hashean la palabra *hola* con SHA-256, las dos obtienen el mismo resumen — y eso es la propiedad central, no un defecto: si fueran resúmenes distintos, no podríamos comprobar coincidencia entre lo publicado y lo recibido.

Dónde aparece SHA-256 en tu día a día

Aunque no lo veas, SHA-256 sostiene buena parte de lo que usas a diario en internet. La cadena de bloques de Bitcoin se construye encadenando SHA-256 de cada bloque al siguiente; alterar un bloque pasado obliga a recalcular toda la cadena posterior. Git, el sistema con el que se versiona el código de medio mundo, identifica cada confirmación por el SHA-256 (en versiones recientes) o por su predecesor SHA-1 (en versiones más antiguas) de su contenido completo. Los certificados HTTPS que verifican la identidad de un sitio web cuando entras llevan una huella SHA-256 asociada. Las descargas de software se acompañan a menudo de un SHA-256 publicado por el desarrollador para que verifiques que el archivo no se alteró por el camino. Y, como hemos dicho, al pie de cada Cuaderno Lacre.

Para el lector profesional

Cuatro recordatorios operativos para quien decide o audita sistemas:

1. Hash no es cifrado. Si un proveedor confunde los dos términos en su documentación técnica, conviene preguntar qué quiere decir exactamente.
2. Para almacenar contraseñas nunca se debe usar SHA-256 a secas. SHA-256 es demasiado rápido para esta tarea (ver punto 3 de *Lo que un hash no es*). El estándar actual es **Argon2id**: lento por diseño, configurable según la capacidad del servidor, combinado con una *sal* aleatoria distinta por usuario.
3. Para integridad de documentos —contratos, expedientes, archivos— SHA-256 sigue siendo el estándar de referencia. Es el que usan los selladores temporales cualificados en la UE.
4. Para conservación a largo plazo (decenios) conviene calcular y archivar también un SHA-3 o un SHA-512 junto al SHA-256; la prudencia criptográfica recomienda no apoyarse en una sola función durante archivos centenarios.

Técnicamente, esta estructura iterada —donde el estado intermedio se conserva entre bloques de entrada— se conoce como una construcción de **Merkle-Damgård**, el patrón en el que se basan SHA-1, SHA-2 (incluido SHA-256) y muchas otras funciones hash clásicas. SHA-3, por el contrario, abandona Merkle-Damgård en favor de una arquitectura distinta llamada *esponja*.

Cómo funciona SHA-256, paso a paso, en palabras llanas

Imagina que has montado el circuito de dominó más elaborado del mundo: miles de fichas, decenas de bifurcaciones, puentes mecánicos y rampas que cruzan toda la habitación, cuidadosamente colocadas pieza a pieza.

Si das un toque a la primera ficha, la cadena cae en una secuencia precisa y repetible. Mismo montaje, mismo toque inicial → idéntico patrón final de fichas caídas, una y otra vez.

Aquí está lo interesante: mueve **una sola ficha** medio centímetro a un lado antes de empezar y vuelve a tocar. Una rampa que debía activarse se queda inerte, un puente no cae, una bifurcación distinta se dispara. El patrón final de fichas en el suelo es completamente irreconocible comparado con el primero.

SHA-256 es matemáticamente este circuito. El texto que escribes es la posición inicial de las fichas. El algoritmo es el toque que libera la cascada. Y el resultado final —lo que llamamos *hash*— es la foto fija del suelo cuando se ha detenido todo. Cambia una sola coma del texto original y la foto será radicalmente distinta. Así de simple, y así de drástico.

Paso 1. Traducir el texto a fichas binarias. Los ordenadores no entienden de letras; las traducen primero a números (ASCII) y los números a binario (unos y ceros). Cada letra se convierte en 8 fichas blancas o negras: la *A* es `01000001`, la *B* es `01000010`, el espacio es `00100000`. Tu texto entero —una palabra, un contrato, una novela— se vuelve una larga fila de fichas blancas y negras.

Paso 2. Rellenar hasta el tamaño estándar. El circuito procesa la fila en *tramos* de exactamente 512 fichas. Si tu mensaje no llega a un múltiplo de 512, se añade una ficha marcadora (la del valor `10000000`) justo después del texto y luego ceros hasta completar el tramo. Las últimas 64 posiciones de cada tramo se reservan para anotar la longitud original del texto. Así el circuito siempre sabe dónde acabó el contenido real y dónde empezó el relleno.

Paso 3. Colocar las ocho fichas maestras. Antes de empezar, situamos sobre la mesa **ocho fichas maestras** en una posición inicial precisa. Estas ocho fichas no son ningún secreto: su valor inicial está fijado por una regla matemática pública (las raíces cuadradas de los ocho primeros números primos —2, 3, 5, 7, 11, 13, 17, 19— y los primeros bits de la parte decimal de cada raíz). Todo el mundo, en cualquier rincón del planeta, empieza con las mismas ocho fichas maestras en la misma posición. Su destino es ser empujadas y transformadas por la avalancha.

Paso 4. La gran avalancha: sesenta y cuatro rondas de empujones. Aquí empieza el espectáculo. El primer tramo de 512 fichas de tu texto se hace chocar contra las ocho fichas maestras. Pero no caen de golpe: el mecanismo ejecuta **sesenta y cuatro rondas consecutivas**. En cada ronda hace tres operaciones con las fichas:

- **El Tiovivo** (rotación). Las fichas se mueven en círculo: las de la derecha pasan a la izquierda. Ninguna ficha se pierde ni se añade; simplemente se reordenan dando una vuelta completa al tiovivo. Es una manera barata y reversible de redistribuir la información.
- **El Embudo Lógico** (XOR). Las fichas pasan por un embudo que las compara de dos en dos: si las dos son del mismo color, sale una blanca; si son distintas, sale una negra. Es la operación más sencilla de la lógica binaria, pero combinada con las rotaciones del tiovivo se vuelve poderosísima para mezclar información sin perderla.
- **El Desborde** (suma modular). El resultado se suma con una *ficha de empuje constante* traída de una lista pública de sesenta y cuatro constantes (las raíces cúbicas de los sesenta y cuatro primeros números primos). Si la suma genera fichas extras que no caben en el espacio de 32 fichas previsto, esas fichas sobrantes se descartan. La mesa solo tiene espacio para 32 fichas, ni una más.

Al final de la ronda sesenta y cuatro, cada una de las fichas del tramo de tu texto ha influido en la posición de las ocho fichas maestras. La energía del empujón ha viajado por todo el circuito.

Paso 5. Añadir el siguiente tramo (sin reiniciar). Si tu texto era largo y queda otro tramo de 512 fichas por procesar, **el circuito no se reinicia**. Las ocho fichas maestras se quedan tal y como las dejó la primera avalancha, y el segundo tramo se lanza contra ellas para activar otras sesenta y cuatro rondas. Es como añadir una habitación nueva llena de dominós al final de la que acaba de caer: el desorden de la primera condiciona enteramente cómo caerá la segunda.

Paso 6. Hacer la foto final. Cuando ya no quedan más tramos por procesar, la avalancha se detiene. Miramos la posición final en la que han quedado las ocho fichas maestras. Traducimos su configuración a un código de letras y números en sistema hexadecimal. El resultado es una cadena de exactamente sesenta y cuatro caracteres: ese es tu sello SHA-256.

Cuatro propiedades caen por sí solas de cómo está montado el circuito:

1. **Determinismo.** El mismo texto produce siempre la misma foto final, en cualquier ordenador del mundo. Cero aleatoriedad, cero sorpresas.
2. **Efecto avalancha.** Una coma añadida, una mayúscula cambiada, una tilde olvidada: la foto resulta completamente irreconocible. Esta es la sensibilidad extrema que ya hemos descrito al principio.
3. **Una sola dirección.** Dada la foto final, no puedes reconstruir el texto original. Las rotaciones, los embudos y los desbordes destruyen toda la información direccional sobre *de dónde venía cada bit* y conservan solo *qué se sumó en total*.
4. **Resistencia a colisiones.** En veinticinco años de criptoanálisis público, nadie ha conseguido encontrar dos textos distintos cuyas fotos finales coincidan. Y la dificultad de hacerlo está fuera del alcance computacional de cualquier civilización razonablemente imaginable.

El apéndice de código que sigue implementa exactamente estos seis pasos en Zig. Ahora puedes leerlo sabiendo qué significa cada operación de bits, en vez de aceptar las manipulaciones a ciegas.

Glosario técnico

Para el lector que quiera entender qué hace cada operación. Sáltatelo libremente: el artículo se sigue entendiendo sin él.

ASCII y Unicode — cómo las letras se vuelven números. Los ordenadores no ven letras; ven números. Un estándar llamado **ASCII** (*American Standard Code for Information Interchange*, de 1963) asigna a cada carácter de teclado un número específico: la *A* es 65, la *B* es 66, la *a* es 97, el *0* es 48, el espacio es 32, la coma es 44. Los sistemas modernos lo extienden con **Unicode**, que asigna un número a cada carácter de cada alfabeto del mundo: cirílico, árabe, chino, japonés, e incluso emojis. Cuando escribes un carácter o abres un fichero de texto, el ordenador lee el número de fondo, no la forma en pantalla. SHA-256 trabaja sobre estos números, tratando cualquier texto como una secuencia larga de cifras. Por eso puede sellar un artículo en español, un poema en japonés y un archivo binario con el mismo algoritmo.
XOR — el comparador bit a bit. XOR (pronunciado *«exor»*, del inglés *exclusive or*, «o exclusivo») es una de las operaciones más sencillas que un ordenador puede hacer con dos números binarios. Compara dos bits posición por posición y devuelve: **1** si exactamente uno de los dos es 1 (uno pero no los dos), **0** si los dos son iguales (ambos 0 o ambos 1). Ejemplo: XOR de `1010` y `1100` es `0110`. Tiene una propiedad notable: es reversible —si haces XOR dos veces con la misma clave, vuelves al original—. Por eso es el caballo de batalla de la criptografía: mezcla bits sin perder información, pero el resultado no revela nada sobre las entradas si no conoces una de ellas.
Hexadecimal — contar en base 16. Casi todos los números del día a día usan diez dígitos (0-9). El hexadecimal usa dieciséis: los habituales 0-9 más seis letras que representan los valores siguientes: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. ¿Por qué dieciséis? Porque los ordenadores piensan en grupos de cuatro bits, y cuatro bits pueden representar exactamente dieciséis valores distintos —así, un carácter hexadecimal corresponde limpiamente a cuatro bits—. Una huella SHA-256 mide 256 bits, lo que son exactamente **64 caracteres hexadecimales**. Si la escribiéramos en decimal corriente, ocuparía unos 78 dígitos y resultaría más incómoda. La elección es estética y compacta; el número de fondo es el mismo.
Rotación de bits — el tiovivo binario. Imagina una fila de siete bombillas, unas encendidas (1) y otras apagadas (0): `1 0 1 1 0 0 1`. Rotar a la derecha una posición consiste en tomar la bombilla de la derecha del todo, llevarla al extremo izquierdo y desplazar las demás un sitio a la derecha: `1 1 0 1 1 0 0`. Ninguna bombilla se pierde ni se añade: simplemente bailan en círculo. SHA-256 utiliza la rotación de bits cientos de veces en cada cálculo; es una manera barata y sin pérdidas de redistribuir la información dentro del estado.
Constantes *«nothing-up-my-sleeve»* — por qué provienen de números primos. Las ocho fichas maestras y las sesenta y cuatro constantes de ronda de SHA-256 no se eligieron al azar. Provienen de las raíces cuadradas y cúbicas de los primeros números primos. ¿Por qué? Porque sus diseñadores querían constantes *«sin nada bajo la manga»*: valores cuyo origen cualquiera pueda verificar. Si alguien te dijera *«fíate de mí: usa este número aleatorio de 32 bits»*, sospecharías razonablemente de una debilidad oculta o de una puerta trasera. Pero cualquiera con una calculadora puede comprobar que los primeros 32 bits de la raíz cuadrada de 2 son `0x6a09e667`. Los valores son matemáticos, públicos y reproducibles: ninguna trampa oculta puede colarse en la receta.

Apéndice: SHA-256 en código legible

Este apéndice es para el lector que quiera ver el algoritmo por dentro. Es una implementación didáctica en Zig que sigue la especificación FIPS 180-4. No es la versión que usa Solo2 —la real está en `std.crypto.hash.sha2.Sha256` de la biblioteca estándar de Zig, optimizada y auditada—. Pero el algoritmo es el mismo: lo que ves aquí es, paso a paso, lo que ocurre cuando aquella llamada de cinco caracteres ejecuta su trabajo.

```
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
}
```

Cualquier reescritura en otro lenguaje que siga la misma estructura —constantes iniciales, expansión del schedule, sesenta y cuatro rondas, acumulación— produce el mismo resultado. El algoritmo no tiene secretos: su valor reside en que las propiedades enumeradas más arriba siguen sosteniéndose después de dos décadas de criptanálisis público sobre miles de ojos.

Si vuelves al pie de este artículo, verás un sello hexadecimal de sesenta y cuatro caracteres. Es el SHA-256 del texto que acabas de leer, en este idioma. Si tradujéramos el artículo, el sello sería otro; si cambiara una palabra de la versión española, el sello español cambiaría. El sello no protege el contenido —para eso están otras herramientas— sino que lo identifica unívocamente. Y eso, por modesto que suene, basta para que ningún paso de la cadena editorial pueda alterar lo dicho sin que se note. Lo demás —cifrar, firmar, identificar— se construye encima de esta idea sencilla.

Fuentes y lectura adicional

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, agosto de 2015. Especificación oficial de la familia SHA-2, incluyendo SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, mayo de 2011. Versión normativa para implementadores.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Capítulos 5 y 6 cubren funciones hash y sus usos legítimos e ilegítimos.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Ejemplo práctico del uso de SHA-256 para encadenar bloques en una estructura inmutable por construcción.
- Reglamento (UE) 910/2014 (eIDAS) — marco de los selladores temporales cualificados. SHA-256 es la función de referencia para las firmas y sellos electrónicos cualificados emitidos en la UE.
- Implementación de referencia en Zig: `std.crypto.hash.sha2.Sha256` en el repositorio oficial del lenguaje (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). Es la versión optimizada y auditada que de hecho usa Solo2. Útil para contrastar con la implementación didáctica del apéndice.

← AnteriorSchrems II, cinco años después Siguiente →Kill switch y la captura institucional

Lecturas recientes

- Análisis · 18 de mayo de 2026 Privacidad real vs aparente: las preguntas que conviene hacerse
- Análisis · 18 de mayo de 2026 Self-hosting como práctica profesional
- Concepto · 18 de mayo de 2026 Las 24 palabras: qué es una identidad criptográfica

Llévate este artículo donde lo necesites.

↓ Markdown ↓ Texto plano ↓ PDF

El archivo se descarga a tu dispositivo. Desde ahí puedes guardarlo, importarlo a Solo2, o compartirlo donde quieras. Cuadernos no decide el destino por ti.

Sello de lacre · SHA-256 4cfae316bd48c6f49ed922f451c5240bebc9d81dd6a446da4e4f87008cf2b7db

Cuadernos Lacre · Una publicación de Menzuri Gestión S.L. ·
escrita por R.Eugenio · editada por el equipo de Solo2.