Conceito · 19 de maio de 2026

← Anterior·Seguinte →

O que é realmente o SHA-256

Uma impressão digital matemática que cabe em sessenta e quatro caracteres e que muda completamente se uma única vírgula do texto original for movida. Por que lhe chamamos selo de lacre digital.

A ideia simples por trás do nome técnico

Imagine que existe uma máquina com uma única ranhura e um único ecrã. Pela ranhura introduz um texto: uma palavra, uma frase, um romance inteiro. No ecrã aparece, instantes depois, uma sequência de exatamente sessenta e quatro caracteres. A essa sequência, para o leitor profissional, chamamos *hash* ou *resumo criptográfico*; para o leitor geral, podemos chamá-la por agora uma impressão digital matemática do texto, tal como a impressão digital o é de uma pessoa.

Se introduzir o mesmo texto duas vezes, a máquina mostra a mesma impressão digital as duas vezes. Se introduzir um texto ligeiramente diferente — uma única vírgula movida, uma maiúscula que passa a minúscula — a máquina mostra uma impressão digital completamente diferente da primeira. Não parecida: diferente. Essas duas propriedades juntas — o determinismo e a sensibilidade — são a ideia simples. Todo o resto do SHA-256 é a maquinaria que as faz cumprir bem.

Convém dizer desde o início o que a máquina não faz. Não cifra o texto. Não o oculta. Não o guarda. A máquina olha para o texto, calcula a impressão digital, e esquece o texto. A impressão digital não permite reconstruir o texto que a produziu; apenas permite, dado um texto candidato, verificar se coincide ou não com o original. Por isso dizemos que é um resumo *de uma única direção*: vai-se, não se volta.

Um hash não é o mesmo que cifrar

A confusão é frequente e convém dissipá-la: cifrar e hashear são operações diferentes. Cifrar consiste em transformar um texto de forma que apenas o detentor da chave possa devolvê-lo à sua forma original. Hashear consiste em produzir uma impressão digital do texto da qual o texto original não se pode recuperar nunca, nem com chave nem sem ela. A primeira é reversível por design; a segunda, irreversível por design.

A consequência prática importa. Quando uma aplicação diz «guardamos a tua palavra-passe cifrada», há alguém que tem a chave para a decifrar — a própria aplicação, em qualquer caso. Quando uma aplicação diz «guardamos a tua palavra-passe hasheada», a própria aplicação não pode ler a palavra-passe original mesmo que quisesse; apenas pode verificar se a que tu escreves volta a produzir a mesma impressão digital. O segundo modelo, bem feito, é muito preferível ao primeiro para armazenar palavras-passe. Mais adiante veremos por que «bem feito» exige algo mais que SHA-256 simples.

As quatro propriedades que tornam um hash criptográfico útil

Uma função hash que mereça o adjetivo *criptográfico* cumpre quatro propriedades:

1. **Determinismo.** A mesma entrada produz sempre a mesma impressão digital.
2. **Efeito avalanche.** Uma pequena mudança na entrada produz uma impressão digital completamente diferente, sem semelhança visível com a anterior.
3. **Resistência à inversão.** Dada uma impressão digital, não é viável computacionalmente encontrar o texto que a produziu.
4. **Resistência a colisões.** Não é viável computacionalmente encontrar dois textos diferentes que produzam a mesma impressão digital.

«Não é viável computacionalmente» não significa «é matematicamente impossível». Significa que o custo em tempo, energia e dinheiro de o conseguir excede em ordens de magnitude a soma de toda a capacidade de computação razoavelmente disponível. Para o SHA-256, essa cota mede-se em milhares de biliões de anos mesmo para as abordagens mais otimistas com hardware especializado. O que, para efeitos práticos do leitor, é o mesmo que «não se pode».

SHA-256, em concreto

O nome diz tudo. SHA são as siglas de *Secure Hash Algorithm*: algoritmo de hash seguro. O número 256 indica o tamanho da impressão digital em bits: duzentos e cinquenta e seis bits, ou seja trinta e dois bytes, que mostrados em hexadecimal são os sessenta e quatro caracteres que o leitor já reconhece. O padrão foi publicado pelo NIST norte-americano, o organismo que normaliza este tipo de funções, em 2001 como parte da família SHA-2; a versão vigente do padrão, FIPS 180-4, é de 2015.

Para quem ainda não tem presente o que são bits e bytes:

```
1 bit    →  0 ou 1        (um interruptor: ligado ou desligado)
1 byte   →  8 bits        (256 combinações possíveis)
32 bytes →  256 bits      (a impressão digital SHA-256)
```

O número 256 no final do nome indica o tamanho da impressão digital em bits. Em hexadecimal — um sistema de numeração com dezasseis símbolos em vez de dez — esses 256 bits cabem em exatamente 64 caracteres. Esses são os 64 caracteres que vê no rodapé de cada Cuaderno.

As dimensões merecem um instante. Duzentos e cinquenta e seis bits permitem dois elevado a duzentos e cinquenta e seis valores diferentes: um número com setenta e oito dígitos decimais, várias ordens de magnitude maior que o número estimado de átomos no universo observável. Cada texto do mundo — cada livro, cada e-mail, cada mensagem — cai sobre um desses valores. A probabilidade de dois textos diferentes coincidirem por azar é, para efeitos práticos, indistinguível de zero.

Como se vê em código

Em Zig, linguagem em que escrevemos as peças que sustentam o Solo2, calcular o selo SHA-256 de um texto vê-se assim:

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

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

Acabámos de pedir à biblioteca padrão do Zig que calcule o SHA-256 do texto entre aspas. Depois da chamada, a variável *resumen* contém os trinta e dois bytes que compõem o selo na sua forma bruta; quando são mostrados no ecrã em hexadecimal, são os sessenta e quatro caracteres que aparecem no rodapé deste artigo. Se mudássemos *Cuadernos Lacre* para *Cuadernos lacre* — uma maiúscula a menos — o selo mudaria completamente. Essa é, em cinco linhas, a propriedade central que sustenta o resto. Para quem quiser ver como funciona internamente, no final do artigo incluímos uma versão legível do algoritmo com comentários passo a passo.

Por que lhe chamamos selo de lacre

Na correspondência europeia dos séculos quinze ao dezanove, o lacre fechava a carta. Uma gota de cera derretida, um selo pressionado em cima, e a carta ficava marcada de forma irrepetível. Não protegia o conteúdo do curioso determinado — o papel podia ser lido contra a luz, o lacre podia ser quebrado — mas evidenciava-o. Qualquer alteração do fecho era visível para o destinatário antes mesmo de abrir o papel. O lacre não impedia o dano; declarava-o.

O SHA-256 do corpo de cada Cuaderno cumpre a mesma função na sua versão digital. Se uma única palavra do artigo mudasse entre o momento em que foi publicado e o momento em que tu o lês, o selo hexadecimal no rodapé do texto já não coincidiria com o SHA-256 do texto que tens à frente. Qualquer leitor com cinco linhas de código poderia verificá-lo. A publicação não pode reescrever a sua história sem que o selo a denuncie. Não protege contra o dano; torna-o verificável.

O que um hash não é

Quatro usos são pedidos às vezes ao SHA-256 que não lhe correspondem:

1. **Cifrar.** Um hash resume; não oculta. Se queres que o texto não se possa ler, precisas de o cifrar, não de o hashear.
2. **Autenticar o autor.** Um hash não diz quem escreveu o texto, apenas qual texto foi hasheado. Para associar autoria faz falta uma assinatura criptográfica sobre o hash, não o hash simples.
3. **Armazenar palavras-passe.** Aqui há uma armadilha que convém entender. O SHA-256 está desenhado para ser muito rápido — o que é bom para muitas coisas, mas mau para esta. Um atacante com hardware especializado pode testar milhares de milhões de palavras-passe por segundo contra um hash SHA-256 até dar com a tua. Para guardar palavras-passe devem usar-se funções de derivação de chave deliberadamente lentas como Argon2, scrypt ou bcrypt, combinadas com uma *sal* (um dado aleatório único por utilizador, que evita que duas pessoas com a mesma palavra-passe tenham o mesmo hash).
4. **Ler o hash como identificador do autor.** Não o é. Um hash identifica o conteúdo. Se duas pessoas hasheiam a palavra *olá* com SHA-256, as duas obtêm o mesmo resumo — e isso é a propriedade central, não um defeito: se fossem resumos diferentes, não poderíamos verificar a coincidência entre o publicado e o recebido.

Onde aparece o SHA-256 no seu dia a dia

Embora não o veja, o SHA-256 sustenta boa parte do que usa diariamente na internet. A cadeia de blocos da Bitcoin constrói-se encadeando o SHA-256 de cada bloco ao seguinte; alterar um bloco passado obriga a recalcular toda a cadeia posterior. O Git, o sistema com que se versiona o código de meio mundo, identifica cada confirmação pelo SHA-256 (em versões recentes) ou pelo seu predecessor SHA-1 (em versões mais antigas) do seu conteúdo completo. Os certificados HTTPS que verificam a identidade de um sítio web quando entra levam uma impressão digital SHA-256 associada. Os downloads de software são acompanhados frequentemente de um SHA-256 publicado pelo programador para que verifiques que o ficheiro não foi alterado pelo caminho. E, como dissemos, no rodapé de cada Cuadernos Lacre.

Para o leitor profissional

Quatro lembretes operativos para quem decide ou audita sistemas:

1. Hash não é cifragem. Se um fornecedor confunde os dois termos na sua documentação técnica, convém perguntar o que quer dizer exatamente.
2. Para armazenar palavras-passe nunca se deve usar o SHA-256 simples. O SHA-256 é demasiado rápido para esta tarefa (ver ponto 3 de *O que um hash não é*). O padrão atual é o **Argon2id**: lento por design, configurável segundo a capacidade do servidor, combinado com uma *sal* aleatória diferente por utilizador.
3. Para a integridade de documentos — contratos, expedientes, ficheiros — o SHA-256 continua a ser o padrão de referência. É o que usam os seladores temporais qualificados na UE.
4. Para conservação a longo prazo (décadas) convém calcular e arquivar também um SHA-3 ou um SHA-512 junto ao SHA-256; a prudência criptográfica recomenda não se apoiar numa única função durante arquivos centenários.

Tecnicamente, esta estrutura iterada — onde o estado intermédio é conservado entre os blocos de entrada — é conhecida como uma construção de **Merkle-Damgård**, o padrão no qual se baseiam SHA-1, SHA-2 (incluindo SHA-256) e muitas outras funções hash clássicas. SHA-3, pelo contrário, abandona Merkle-Damgård a favor de uma arquitetura diferente chamada *esponja*.

Como funciona o SHA-256, passo a passo, em palavras simples

Imagina que montaste o circuito de dominó mais elaborado do mundo: milhares de peças, dezenas de bifurcações, pontes mecânicas e rampas que cruzam todo o quarto, cuidadosamente colocadas peça por peça.

Se deres um toque na primeira peça, a cadeia cai numa sequência precisa e repetível. Mesma montagem, mesmo toque inicial → idêntico padrão final de peças caídas, uma e outra vez.

Aqui está o interessante: move **uma única peça** meio centímetro para o lado antes de começar e volta a tocar. Uma rampa que deveria ativar-se fica inerte, uma ponte não cai, uma bifurcação diferente dispara. O padrão final de peças no chão é completamente irreconhecível comparado com o primeiro.

SHA-256 é matematicamente este circuito. O texto que escreves é a posição inicial das peças. O algoritmo é o toque que liberta a cascata. E o resultado final — o que chamamos *hash* — é a foto fixa do chão quando tudo parou. Muda uma única vírgula do texto original e a foto será radicalmente diferente. Tão simples quanto isso, e tão drástico.

Passo 1. Traduzir o texto para peças binárias. Os computadores não entendem letras; traduzem-nas primeiro para números (ASCII) e os números para binário (uns e zeros). Cada letra converte-se em 8 peças brancas ou pretas: o *A* é `01000001`, o *B* é `01000010`, o espaço é `00100000`. O teu texto inteiro — uma palavra, um contrato, um romance — torna-se numa longa fila de peças brancas e pretas.

Passo 2. Preencher até ao tamanho padrão. O circuito processa a fila em *blocos* de exatamente 512 peças. Se a tua mensagem não chegar a um múltiplo de 512, adiciona-se uma peça marcadora (a do valor `10000000`) logo após o texto e depois zeros até completar o bloco. As últimas 64 posições de cada bloco são reservadas para anotar o comprimento original do texto. Assim o circuito sabe sempre onde acabou o conteúdo real e onde começou o preenchimento.

Passo 3. Colocar as oito peças mestras. Antes de começar, colocamos sobre a mesa **oito peças mestras** numa posição inicial precisa. Estas oito peças não são nenhum segredo: o seu valor inicial é fixado por uma regra matemática pública (as raízes quadradas dos oito primeiros números primos — 2, 3, 5, 7, 11, 13, 17, 19 — e os primeiros bits da parte decimal de cada raiz). Toda a gente, em qualquer canto do planeta, começa com as mesmas oito peças mestras na mesma posição. O seu destino é ser empurradas e transformadas pela avalanche.

Passo 4. A grande avalanche: sessenta e quatro rondas de empurrões. Aqui começa o espetáculo. O primeiro bloco de 512 peças do teu texto choca contra as oito peças mestras. Mas não caem de uma vez: o mecanismo executa **sessenta e quatro rondas consecutivas**. Em cada ronda faz três operações com as peças:

- **O Carrossel** (rotação). As peças movem-se em círculo: as da direita passam para a esquerda. Nenhuma peça se perde ou se adiciona; simplesmente reordenam-se dando uma volta completa ao carrossel. É uma forma barata e reversível de redistribuir a informação.
- **O Funil Lógico** (XOR). As peças passam por um funil que as compara de duas em duas: se as duas são da mesma cor, sai uma branca; se são diferentes, sai uma preta. É a operação mais simples da lógica binária, mas combinada com as rotações do carrossel torna-se poderosíssima para misturar informação sem a perder.
- **O Transbordo** (soma modular). O resultado soma-se a uma *peça de empurrão constante* trazida de uma lista pública de sessenta e quatro constantes (as raízes cúbicas dos sessenta e quatro primeiros números primos). Se a soma gera peças extras que não cabem no espaço de 32 peças previsto, essas peças sobrantes são descartadas. A mesa só tem espaço para 32 peças, nem mais uma.

No final da ronda sessenta e quatro, cada uma das peças do bloco do teu texto influenciou a posição das oito peças mestras. A energia do empurrão viajou por todo o circuito.

Passo 5. Adicionar o bloco seguinte (sem reiniciar). Se o teu texto era longo e falta outro bloco de 512 peças por processar, **o circuito não se reinicia**. As oito peças mestras ficam tal como a primeira avalanche as deixou, e o segundo bloco é lançado contra elas para ativar outras sessenta e quatro rondas. É como adicionar um quarto novo cheio de dominós no final do que acabou de cair: a desordem do primeiro condiciona inteiramente como cairá o segundo.

Passo 6. Tirar a foto final. Quando já não há mais blocos para processar, a avalanche para. Olhamos para a posição final em que ficaram as oito peças mestras. Traduzimos a sua configuração para um código de letras e números em sistema hexadecimal. O resultado é uma cadeia de exatamente sessenta e quatro caracteres: esse é o teu selo SHA-256.

Quatro propriedades surgem naturalmente de como o circuito está montado:

1. **Determinismo.** O mesmo texto produz sempre a mesma foto final, em qualquer computador do mundo. Zero aleatoriedade, zero surpresas.
2. **Efeito avalanche.** Uma vírgula adicionada, uma maiúscula mudada, um acento esquecido: a foto fica completamente irreconhecível. Esta é a sensibilidade extrema que já descrevemos no início.
3. **Numa única direção.** Dada a foto final, não podes reconstruir o texto original. As rotações, os funis e os transbordos destroem toda a informação direcional sobre *de onde vinha cada bit* e conservam apenas *o que se somou no total*.
4. **Resistência a colisões.** Em vinte e cinco anos de criptoanálise pública, ninguém conseguiu encontrar dois textos diferentes cujas fotos finais coincidam. E a dificuldade de o fazer está fora do alcance computacional de qualquer civilização razoavelmente imaginável.

O apêndice de código que se segue implementa exatamente estes seis passos em Zig. Agora podes lê-lo sabendo o que significa cada operação de bits, em vez de aceitar as manipulações às cegas.

Glossário técnico

Para o leitor que queira entender o que faz cada operação. Pode saltá-lo livremente: o artigo continua a entender-se sem ele.

ASCII e Unicode — como as letras se tornam números. Os computadores não veem letras; veem números. Um padrão chamado **ASCII** (*American Standard Code for Information Interchange*, de 1963) atribui a cada carácter do teclado um número específico: o *A* é 65, o *B* é 66, o *a* é 97, o *0* é 48, o espaço é 32, a vírgula é 44. Os sistemas modernos estendem-no com o **Unicode**, que atribui um número a cada carácter de cada alfabeto do mundo: cirílico, árabe, chinês, japonês, e até emojis. Quando escreves um carácter ou abres um ficheiro de texto, o computador lê o número de fundo, não a forma no ecrã. SHA-256 trabalha sobre estes números, tratando qualquer texto como uma longa sequência de algarismos. Por isso pode selar um artigo em espanhol, um poema em japonês e um ficheiro binário com o mesmo algoritmo.
XOR — o comparador bit a bit. XOR (pronunciado *«exor»*, do inglês *exclusive or*, «ou exclusivo») é uma das operações mais simples que um computador pode fazer com dois números binários. Compara dois bits posição por posição e devolve: **1** se exatamente um dos dois for 1 (um mas não os dois), **0** se os dois forem iguais (ambos 0 ou ambos 1). Exemplo: XOR de `1010` e `1100` é `0110`. Tem uma propriedade notável: é reversível — se fizeres XOR duas vezes com a mesma chave, voltas ao original —. Por isso é o cavalo de batalha da criptografia: mistura bits sem perder informação, mas o resultado não revela nada sobre as entradas se não conheceres uma delas.
Hexadecimal — contar na base 16. Quase todos os números do dia a dia usam dez dígitos (0-9). O hexadecimal usa dezasseis: os habituais 0-9 mais seis letras que representam os seguintes valores: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Porquê dezasseis? Porque os computadores pensam em grupos de quatro bits, e quatro bits podem representar exatamente dezasseis valores diferentes — assim, um carácter hexadecimal corresponde perfeitamente a quatro bits —. Uma huella (hash) SHA-256 mede 256 bits, o que são exatamente **64 caracteres hexadecimais**. Se a escrevêssemos em decimal comum, ocuparia uns 78 dígitos e seria mais incómoda. A escolha é estética e compacta; o número de fundo é o mesmo.
Rotação de bits — o carrossel binário. Imagina uma fila de sete lâmpadas, umas acesas (1) e outras apagadas (0): `1 0 1 1 0 0 1`. Rodar à direita uma posição consiste em pegar na lâmpada mais à direita, levá-la para o extremo esquerdo e deslocar as outras uma posição para a direita: `1 1 0 1 1 0 0`. Nenhuma lâmpada se perde ou se adiciona: simplesmente dançam em círculo. SHA-256 utiliza a rotação de bits centenas de vezes em cada cálculo; é uma forma barata e sem perdas de redistribuir a informação dentro do estado.
Constantes *«nothing-up-my-sleeve»* — porque provêm de números primos. As oito peças mestras e as sessenta e quatro constantes de ronda do SHA-256 não foram escolhidas ao acaso. Provêm das raízes quadradas e cúbicas dos primeiros números primos. Porquê? Porque os seus desenhadores queriam constantes *«sem nada na manga»*: valores cuja origem qualquer pessoa possa verificar. Se alguém te dissesse *«confia em mim: usa este número aleatório de 32 bits»*, suspeitarias razoavelmente de uma fraqueza oculta ou de uma porta traseira (backdoor). Mas qualquer um com uma calculadora pode comprovar que os primeiros 32 bits da raiz quadrada de 2 são `0x6a09e667`. Os valores são matemáticos, públicos e reprodutíveis: nenhuma armadilha oculta pode entrar na receita.

Apêndice: SHA-256 em código legível

Este apêndice é para o leitor que queira ver o algoritmo por dentro. É uma implementação didática em Zig que segue a especificação FIPS 180-4. Não é a versão que o Solo2 usa — a real está em `std.crypto.hash.sha2.Sha256` da biblioteca padrão do Zig, optimizada e auditada —. Mas o algoritmo é o mesmo: o que vês aqui é, passo a passo, o que acontece quando aquela chamada de cinco caracteres executa o seu trabalho.

```
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
}
```

Qualquer reescrita noutra linguagem que siga a mesma estrutura — constantes iniciais, expansão do schedule, sessenta e quatro rondas, acumulação — produz o mesmo resultado. O algoritmo não tem segredos: o seu valor reside em que as propriedades enumeradas mais acima continuam a sustentar-se depois de duas décadas de criptoanálise pública sobre milhares de olhos.

Se voltares ao rodapé deste artigo, verás um selo hexadecimal de sessenta e quatro caracteres. É o SHA-256 do texto que acabaste de ler, neste idioma. Se traduzíssemos o artigo, o selo seria outro; se mudasse uma palavra da versão espanhola, o selo espanhol mudaria. O selo não protege o conteúdo — para isso existem outras ferramentas — mas identifica-o univocamente. E isso, por modesto que soe, basta para que nenhum passo da cadeia editorial possa alterar o que foi dito sem que se note. O resto — cifrar, assinar, identificar — constrói-se sobre esta ideia simples.

Fontes e leitura adicional

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, agosto de 2015. Especificação oficial da família SHA-2, incluindo SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, maio de 2011. Versão normativa para implementadores.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Capítulos 5 e 6 cobrem funções hash e os seus usos legítimos e ilegítimos.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Exemplo prático do uso de SHA-256 para encadear blocos numa estrutura imutável por construção.
- Regulamento (UE) 910/2014 (eIDAS) — quadro dos seladores temporais qualificados. O SHA-256 é a função de referência para as assinaturas e selos eletrónicos qualificados emitidos na UE.
- Implementação de referência em Zig: `std.crypto.hash.sha2.Sha256` no repositório oficial da linguagem (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). É a versão optimizada e auditada que de facto o Solo2 usa. Útil para contrastar com a implementação didática do apêndice.

← AnteriorCUADERNOS_LIST_SCHREMS_TITLESeguinte →CUADERNOS_LIST_KILLSWITCH_TITLE

Leituras recentes

- CUADERNOS_LIST_PREGUNTAS_TITLE
- CUADERNOS_LIST_SELFHOST_TITLE
- CUADERNOS_LIST_IDENTIDAD_TITLE

Leve este artigo para onde precisar.

↓ Markdown ↓ Texto simples ↓ PDF

O arquivo é descarregado no seu dispositivo. A partir daí, pode guardá-lo, importá-lo no Solo2 ou partilhá-lo onde quiser. Cuadernos não decide o destino por si.

Selo de lacre · SHA-256 5ec909ce34e4dc2c985abc02a64f61cd40dac13b8ada7dc93e6cb157558c23c0

Cuadernos Lacre · Uma publicação da Menzuri Gestión S.L. ·
escrita por R.Eugenio · editada pela equipa do Solo2.