Wat SHA-256 echt is

Een wiskundige vingerafdruk van vierenzestig tekens die volledig verandert als er ook maar één komma in de oorspronkelijke tekst wordt verplaatst. Waarom we het een digitaal lakzegel noemen.

Om het simpel te zeggen: Stel je een machine voor die een willekeurige tekst leest en een reeks van 64 tekens teruggeeft. Als de tekst identiek is, komt de reeks er identiek uit. Als je slechts één komma verplaatst, is de reeks compleet anders. Die reeks is de digitale zegellak.

Het eenvoudige idee achter de technische naam

Stel je een machine voor met slechts één gleuf en één scherm. Door de gleuf voer je een tekst in: een woord, een zin, een hele roman. Op het scherm verschijnt even later een reeks van precies vierenzestig tekens. Voor de professionele lezer noemen we die reeks een *hash* of *cryptografische samenvatting*; voor de algemene lezer kunnen we het voorlopig een wiskundige vingerafdruk van de tekst noemen, zoals een vingerafdruk dat is voor een persoon.

Als je twee keer dezelfde tekst invoert, toont de machine beide keren dezelfde vingerafdruk. Als je een tekst invoert die net even anders is —één verplaatste komma, een hoofdletter die een kleine letter wordt— toont de machine een vingerafdruk die totaal anders is dan de eerste. Niet een beetje anders: compleet anders. Die twee eigenschappen samen —determinisme en gevoeligheid— vormen het eenvoudige idee. Al het andere aan SHA-256 is het mechanisme dat ervoor zorgt dat dit goed gebeurt.

Het is goed om vanaf het begin te zeggen wat de machine niet doet. Het versleutelt de tekst niet. Het verbergt de tekst niet. Het slaat de tekst niet op. De machine bekijkt de tekst, berekent de vingerafdruk en vergeet de tekst. De vingerafdruk maakt het niet mogelijk om de tekst te herleiden die hem heeft geproduceerd; het maakt het alleen mogelijk om bij een kandidaat-tekst te controleren of deze overeenkomt met het origineel. Daarom zeggen we dat het een samenvatting in *één richting* is: je kunt erheen, maar niet terug.

Een hash is niet hetzelfde als versleutelen

Er is vaak verwarring over en het is goed om dat op te helderen: versleutelen en hashen zijn verschillende bewerkingen. Versleutelen bestaat uit het transformeren van een tekst op een zodanige manier dat alleen de bezitter van de sleutel deze in zijn oorspronkelijke vorm kan terugkrijgen. Hashen bestaat uit het produceren van een vingerafdruk van de tekst waarvan de oorspronkelijke tekst nooit kan worden hersteld, met of zonder sleutel. De eerste is door ontwerp omkeerbaar; de tweede is door ontwerp onomkeerbaar.

Het praktische gevolg is belangrijk. Wanneer een applicatie zegt «we bewaren je wachtwoord versleuteld», is er iemand die de sleutel heeft om het te ontsleutelen — de applicatie zelf, in elk geval. Wanneer een applicatie zegt «we bewaren je wachtwoord gehasht», kan de applicatie zelf het originele wachtwoord niet lezen, ook al zou ze dat willen; ze kan alleen controleren of wat jij typt dezelfde vingerafdruk produceert. Het tweede model is, mits goed uitgevoerd, veel beter dan het eerste voor het opslaan van wachtwoorden. Later zullen we zien waarom «goed uitgevoerd» meer vereist dan alleen SHA-256.

De vier eigenschappen die een cryptografische hash nuttig maken

Een hashfunctie die het bijvoeglijk naamwoord *cryptografisch* verdient, voldoet aan vier eigenschappen:

1. **Determinisme.** Dezelfde invoer produceert altijd dezelfde vingerafdruk.

2. **Avalanche-effect.** Een kleine wijziging in de invoer produceert een totaal andere vingerafdruk, zonder zichtbare gelijkenis met de vorige.
3. **Weerstand tegen inversie.** Gegeven een vingerafdruk is het computationeel niet haalbaar om de tekst te vinden die deze heeft geproduceerd.
4. **Weerstand tegen collisies.** Het is computationeel niet haalbaar om twee verschillende teksten te vinden die dezelfde vingerafdruk produceren.

«Niet computationeel haalbaar» betekent niet «wiskundig onmogelijk». Het betekent dat de kosten in tijd, energie en geld om dit te bereiken de totale redelijkerwijs beschikbare computercapaciteit met vele ordes van grootte overtreffen. Voor SHA-256 wordt die grens gemeten in duizenden triljoenen jaren, zelfs voor de meest optimistische scenario's met gespecialiseerde hardware. Wat voor de praktische doeleinden van de lezer hetzelfde is als «het kan niet».

SHA-256, in het bijzonder

De naam zegt alles. SHA staat voor *Secure Hash Algorithm*: een veilig hash-algoritme. Het getal 256 geeft de grootte van de vingerafdruk in bits aan: tweehonderdzesenvijftig bits, oftewel tweeëndertig bytes, die in hexadecimaal weergegeven de vierenzestig tekens zijn die de lezer al herkent. De standaard werd in 2001 gepubliceerd door het Amerikaanse NIST, de instantie die dit soort functies normaliseert, als onderdeel van de SHA-2-familie; de huidige versie van de standaard, FIPS 180-4, dateert uit 2015.

Voor wie nog niet weet wat bits en bytes zijn:

```
1 bit    →  0 of 1        (een schakelaar: aan of uit)
1 byte   →  8 bits         (256 mogelijke combinaties)
32 bytes →  256 bits       (de SHA-256-vingerafdruk)
```

Het getal 256 aan het einde van de naam geeft de grootte van de vingerafdruk in bits aan. In hexadecimaal —een talstelsel met zestien symbolen in plaats van tien— passen die 256 bits in precies 64 tekens. Dat zijn de 64 tekens die je onderaan elk Cuaderno ziet.

De afmetingen verdienen een moment aandacht. Tweehonderdzesenvijftig bits maken twee tot de macht tweehonderdzesenvijftig verschillende waarden mogelijk: een getal met achtenzeventig decimalen, vele ordes van grootte groter dan het geschatte aantal atomen in het waarneembare universum. Elke tekst in de wereld —elk boek, elke e-mail, elk bericht— valt op een van die waarden. De kans dat twee verschillende teksten toevallig samenvallen, is voor praktische doeleinden niet te onderscheiden van nul.

Hoe het er in code uitziet

In Zig, de taal waarin we de onderdelen schrijven die Solo2 ondersteunen, ziet het berekenen van het SHA-256-zegel van een tekst er zo uit:

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

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

We hebben zojuist de standaardbibliotheek van Zig gevraagd om de SHA-256 van de tekst tussen aanhalingstekens te berekenen. Na de aanroep bevat de variabele *resumen* de tweeëntwintig bytes die het zegel in zijn ruwe vorm vormen; wanneer ze op het scherm in hexadecimaal worden weergegeven, zijn dit de vierenzestig tekens die onderaan dit artikel verschijnen. Als we *Cuadernos Lacre* zouden veranderen in *Cuadernos lacre* —één hoofdletter minder— zou het hele zegel veranderen. Dat is, in vijf regels, de kerneigenschap die de rest ondersteunt. Voor wie wil zien hoe het intern werkt, hebben we aan het einde van het artikel een leesbare versie van het algoritme opgenomen met stap-voor-stap commentaar.

Waarom we het een lakzegel noemen

In de Europese correspondentie van de vijftiende tot de negentiende eeuw sloot lakzegel de brief af. Een druppel gesmolten was, een zegel erop gedrukt, en de brief was op een onherhaalbare manier gemerkt. Het beschermde de inhoud niet tegen de vastberaden gluurder —het papier kon tegen het licht worden gelezen, de lak kon worden gebroken— maar

het maakte het wel zichtbaar. Elke wijziging aan de sluiting was zichtbaar voor de ontvanger voordat het papier zelfs maar was geopend. De lak voorkwam de schade niet; het maakte het kenbaar.

De SHA-256 van de tekst van elk Cuaderno vervult dezelfde functie in zijn digitale versie. Als er ook maar één woord in het artikel zou veranderen tussen het moment van publicatie en het moment dat jij het leest, zou het hexadecimale zegel onderaan de tekst niet meer overeenkomen met de SHA-256 van de tekst die voor je ligt. Elke lezer zou dit met vijf regels code kunnen controleren. De publicatie kan haar geschiedenis niet herschrijven zonder dat het zegel dit verraadt. Het beschermt niet tegen schade; het maakt het controleerbaar.

Wat een hash niet is

Er worden soms vier toepassingen gevraagd van SHA-256 die er niet bij horen:

1. **Versleutelen.** Een hash vat samen; het verbergt niet. Als je wilt dat de tekst niet gelezen kan worden, moet je deze versleutelen, niet hashen.
2. **De auteur authenticeren.** Een hash zegt niet wie de tekst heeft geschreven, alleen welke tekst is gehasht. Om auteurschap te koppelen, is een cryptografische handtekening bovenop de hash nodig, niet alleen de hash.
3. **Wachtwoorden opslaan.** Hier zit een valstrik die je moet begrijpen. SHA-256 is ontworpen om erg snel te zijn — wat goed is voor veel dingen, maar slecht voor dit doel. Een aanvaller met gespecialiseerde hardware kan miljarden wachtwoorden per seconde uitproberen tegen een SHA-256-hash tot hij die van jou vindt. Om wachtwoorden op te slaan, moeten opzettelijk trage sleutelafleidingsfuncties zoals Argon2, scrypt of bcrypt worden gebruikt, gecombineerd met een *salt* (unieke willekeurige gegevens per gebruiker, die voorkomen dat twee personen met hetzelfde wachtwoord dezelfde hash hebben).
4. **De hash lezen als identificatie van de auteur.** Dat is het niet. Een hash identificeert de inhoud. Als twee mensen het woord *hola* hashen met SHA-256, krijgen ze allebei dezelfde samenvatting — en dat is de kerneigenschap, geen defect: als het verschillende samenvattingen waren, zouden we de overeenkomst tussen wat gepubliceerd is en wat ontvangen is niet kunnen controleren.

Waar SHA-256 in je dagelijks leven verschijnt

Zelfs als je het niet ziet, vormt SHA-256 de basis van een groot deel van wat je dagelijks op internet gebruikt. De Bitcoin-blockchain wordt gebouwd door de SHA-256 van elk blok aan het volgende te koppelen; het wijzigen van een blok uit het verleden dwingt tot het herberekenen van de hele daaropvolgende keten. Git, het systeem waarmee de code van de halve wereld wordt bijgehouden, identificeert elke commit door de SHA-256 (in recente versies) of door zijn voorganger SHA-1 (in oudere versies) van de volledige inhoud. HTTPS-certificaten die de identiteit van een website verifiëren wanneer je deze bezoekt, hebben een bijbehorende SHA-256-vingerafdruk. Software-downloads gaan vaak vergezeld van een door de ontwikkelaar gepubliceerde SHA-256, zodat je kunt controleren of het bestand onderweg niet is gewijzigd. En, zoals we al zeiden, onderaan elk Cuaderno Lacre.

Voor de professionele lezer

Vier operationele herinneringen voor wie systemen beslist of auditeert:

1. Hash is geen versleuteling. Als een leverancier de twee termen verwisselt in zijn technische documentatie, is het verstandig om te vragen wat hij precies bedoelt.
2. Voor het opslaan van wachtwoorden mag nooit alleen SHA-256 worden gebruikt. SHA-256 is te snel voor deze taak (zie punt 3 van *Wat een hash niet is*). De huidige standaard is **Argon2id**: traag door ontwerp, configureerbaar op basis van de servercapaciteit, gecombineerd met een unieke willekeurige *salt* per gebruiker.
3. Voor documentintegriteit —contracten, dossiers, bestanden— blijft SHA-256 de referentiestandaard. Het is wat gekwalificeerde tijdstempeldiensten in de EU gebruiken.
4. Voor langetermijnconservering (decennia) is het raadzaam om ook een SHA-3 of een SHA-512 te berekenen en te archiveren naast de SHA-256; cryptografische voorzichtigheid raadt aan om niet op één enkele functie te vertrouwen bij archieven van honderd jaar oud.

Technisch gezien staat deze geïtereerde structuur — waarbij de tussenstaat behouden blijft tussen invoerblokken — bekend als een **Merkle-Damgård**-constructie, het patroon waarop SHA-1, SHA-2 (inclusief SHA-256) en vele andere klassieke

hashfuncties zijn gebaseerd. SHA-3 daarentegen stapt af van Merkle-Damgård ten gunste van een andere architectuur genaamd *sponge* (spons).

Hoe SHA-256 werkt, stap voor stap, in eenvoudige bewoordingen

Stel je voor dat je het meest uitgebreide dominocircuit ter wereld hebt gebouwd: duizenden stenen, tientallen vertakkingen, mechanische bruggen en hellingen die de hele kamer doorkruisen, zorgvuldig stuk voor stuk geplaatst.

Als je de eerste steen een tikje geeft, valt de keten in een precieze en herhaalbare reeks. Dezelfde opstelling, dezelfde eerste tik → identiek eindpatroon van gevallen stenen, keer op keer.

Hier is het interessante: verplaats **slechts één steen** een halve centimeter opzij voordat je begint en tik opnieuw. Een helling die geactiveerd had moeten worden blijft inactief, een brug valt niet, een andere vertakking wordt in gang gezet. Het eindpatroon van de stenen op de grond is compleet onherkenbaar vergeleken met het eerste.

SHA-256 is wiskundig gezien dit circuit. De tekst die je schrijft is de beginpositie van de stenen. Het algoritme is de tik die de cascade ontketent. En het eindresultaat — wat we een *hash* noemen — is de momentopname van de grond wanneer alles tot stilstand is gekomen. Verander een enkele komma van de originele tekst en de foto zal radicaal anders zijn. Zo simpel is het, en zo drastisch.

Stap 1. Vertaal de tekst naar binaire stenen. Computers begrijpen geen letters; ze vertalen ze eerst naar cijfers (ASCII) en de cijfers naar binair (enen en nullen). Elke letter wordt omgezet in 8 witte of zwarte stenen: de *A* is `01000001`, de *B* is `01000010`, de spatie is `00100000`. Je hele tekst — een woord, een contract, een roman — wordt een lange rij van witte en zwarte stenen.

Stap 2. Aanvullen tot de standaardgrootte. Het circuit verwerkt de rij in *blokken* van exact 512 stenen. Als je bericht geen veelvoud van 512 bereikt, wordt er een markeringssteen (die met de waarde `10000000`) toegevoegd net na de tekst en vervolgens nullen om het blok compleet te maken. De laatste 64 posities van elk blok zijn gereserveerd om de oorspronkelijke lengte van de tekst te noteren. Zo weet het circuit altijd waar de echte inhoud eindigde en waar de opvulling begon.

Stap 3. Plaats de acht meesterstenen. Voordat we beginnen, plaatsen we **acht meesterstenen** op tafel in een precieze beginpositie. Deze acht stenen zijn geen geheim: hun beginwaarde wordt bepaald door een publieke wiskundige regel (de vierkantswortels van de eerste acht priemgetallen — 2, 3, 5, 7, 11, 13, 17, 19 — en de eerste bits van het decimale deel van elke wortel). Iedereen, in elke uithoek van de planeet, begint met dezelfde acht meesterstenen in dezelfde positie. Hun lot is om te worden geduwd en getransformeerd door de lawine.

Stap 4. De grote lawine: vierenzestig rondes van duwtjes. Hier begint de show. Het eerste blok van 512 stenen van je tekst botst tegen de acht meesterstenen. Maar ze vallen niet in één keer: het mechanisme voert **vierenzestig opeenvolgende rondes** uit. In elke ronde voert het drie bewerkingen met de stenen uit:

- **De Draaimolen** (rotatie). De stenen bewegen in een cirkel: die aan de rechterkant gaan naar links. Er gaat geen steen verloren en er wordt er geen toegevoegd; ze worden simpelweg herschikt door een volledige ronde op de draaimolen te maken. Het is een goedkope en omkeerbare manier om informatie te herverdelen.
- **De Logische Trechter** (XOR). De stenen gaan door een trechter die ze twee aan twee vergelijkt: als beide dezelfde kleur hebben, komt er een witte uit; als ze verschillend zijn, komt er een zwarte uit. Het is de eenvoudigste bewerking in de binaire logica, maar gecombineerd met de rotaties van de draaimolen wordt het enorm krachtig om informatie te mengen zonder deze te verliezen.
- **De Overloop** (modulair optellen). Het resultaat wordt opgeteld met een *constante duwsteen* gehaald uit een publieke lijst van vierenzestig constanten (de derdemachtswortels van de eerste vierenzestig priemgetallen). Als de som extra stenen genereert die niet passen in de voorziene ruimte van 32 stenen, worden deze overtollige stenen weggegooid. De tafel heeft slechts ruimte voor 32 stenen, niet één meer.

Aan het einde van ronde vierenzestig heeft elk van de stenen uit het blok van je tekst de positie van de acht meesterstenen beïnvloed. De energie van de duw is door het hele circuit gereisd.

Stap 5. Voeg het volgende blok toe (zonder te herstarten). Als je tekst lang was en er is nog een blok van 512 stenen te verwerken, **dan wordt het circuit niet gereset**. De acht meesterstenen blijven precies zoals de eerste lawine ze achterliet, en het tweede blok wordt ertegenaan gelanceerd om nog eens vierenzestig rondes te activeren. Het is als het toevoegen van

een nieuwe kamer vol dominostenen aan het einde van de kamer die net is gevallen: de wanorde van de eerste bepaalt volledig hoe de tweede zal vallen.

Stap 6. Neem de eindfoto. Wanneer er geen blokken meer te verwerken zijn, stopt de lawine. We kijken naar de eindpositie waarin de acht meesterstenen zijn gebleven. We vertalen hun configuratie naar een code van letters en cijfers in het hexadecimale systeem. Het resultaat is een reeks van exact vierenzestig tekens: dat is je SHA-256-zegel.

Vier eigenschappen vloeien vanzelf voort uit hoe het circuit is opgebouwd:

1. **Determinisme.** Dezelfde tekst levert altijd dezelfde eindfoto op, in elke computer ter wereld. Nul willekeur, nul verrassingen.
2. **Lawine-effect.** Een toegevoegde komma, een gewijzigde hoofdletter, een vergeten accent: de foto wordt compleet onherkenbaar. Dit is de extreme gevoeligheid die we al in het begin hebben beschreven.
3. **Eénrichtingsverkeer.** Gegeven de eindfoto, kun je de originele tekst niet reconstrueren. De rotaties, trechters en overlopen vernietigen alle directionele informatie over *waar elke bit vandaan kwam* en behouden alleen *wat in totaal is opgeteld*.
4. **Botsingsbestendigheid.** In vijfentwintig jaar van publieke cryptanalyse is het niemand gelukt om twee verschillende teksten te vinden waarvan de eindfoto's overeenkomen. En de moeilijkheid om dit te doen ligt buiten het computationele bereik van elke redelijkerwijs voorstelbare beschaving.

De codebijlage die volgt implementeert exact deze zes stappen in Zig. Nu kun je deze lezen met de wetenschap wat elke bitbewerking betekent, in plaats van de manipulaties blindelings te accepteren.

Technisch glossarium

Voor de lezer die wil begrijpen wat elke bewerking doet. Voel je vrij om dit over te slaan: het artikel is ook zonder dit te begrijpen.

ASCII en Unicode — hoe letters getallen worden. Computers zien geen letters; ze zien getallen. Een standaard genaamd **ASCII** (*American Standard Code for Information Interchange*, uit 1963) wijst aan elk toetsenbordteken een specifiek nummer toe: de *A* is 65, de *B* is 66, de *a* is 97, de *0* is 48, de spatie is 32, de komma is 44. Moderne systemen breiden dit uit met **Unicode**, dat een nummer toewijst aan elk teken van elk alfabet ter wereld: cyrillisch, Arabisch, Chinees, Japans en zelfs emoji's. Wanneer je een teken typt of een tekstbestand opent, leest de computer op de achtergrond het getal, niet de vorm op het scherm. SHA-256 werkt met deze getallen en behandelt elke tekst als een lange reeks cijfers. Daarom kan het een artikel in het Spaans, een gedicht in het Japans en een binair bestand met hetzelfde algoritme verzegelen.
XOR — de bit-voor-bit vergelijker. XOR (uitgesproken als *«exor»*, van het Engelse *exclusive or*, «exclusieve of») is een van de eenvoudigste bewerkingen die een computer kan uitvoeren met twee binaire getallen. Het vergelijkt twee bits positie per positie en retourneert: **1** als exact één van de twee 1 is (een, maar niet beide), **0** als ze beide gelijk zijn (beide 0 of beide 1). Voorbeeld: XOR van `1010` en `1100` is `0110`. Het heeft een opmerkelijke eigenschap: het is omkeerbaar — als je twee keer XOR doet met dezelfde sleutel, kom je terug bij het origineel. Daarom is het het werkpaard van de cryptografie: het mengt bits zonder informatie te verliezen, maar het resultaat verraadt niets over de invoer als je een van beide niet kent.
Hexadecimaal — tellen in basis 16. Bijna alle alledaagse getallen gebruiken tien cijfers (0-9). Hexadecimaal gebruikt er zestien: de gebruikelijke 0-9 plus zes letters die de volgende waarden vertegenwoordigen: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Waarom zestien? Omdat computers in groepen van vier bits denken, en vier bits precies zestien verschillende waarden kunnen vertegenwoordigen — een hexadecimaal teken komt dus keurig overeen met vier bits. Een SHA-256 hash is 256 bits lang, wat neerkomt op exact **64 hexadecimale tekens**. Als we het in gewoon decimaal zouden schrijven, zou het ongeveer 78 cijfers in beslag nemen en onhandiger zijn. De keuze is esthetisch en compact; het onderliggende getal is hetzelfde.
Bitrotatie — de binaire draaimolen. Stel je een rij van zeven gloeilampen voor, sommige aan (1) en andere uit (0): `1 0 1 1 0 0 1`. Een positie naar rechts roteren bestaat uit het nemen van de meest rechtse lamp, deze naar uiterst links te brengen en de rest één plaats naar rechts op te schuiven: `1 1 0 1 1 0 0`. Er gaat geen lamp verloren en er komt er geen bij: ze dansen simpelweg in een cirkel. SHA-256 gebruikt bitrotatie honderden keren in elke berekening; het is een goedkope en verliesvrije manier om informatie binnen de staat te herverdelen.
***«Nothing-up-my-sleeve»* constanten — waarom ze afkomstig zijn van priemgetallen.** De acht meesterstenen en de vierenzestig ronde-constanten van SHA-256 zijn niet willekeurig gekozen. Ze zijn afgeleid van de vierkantswortels en derdemachtswortels van de eerste priemgetallen. Waarom? Omdat de ontwerpers constanten wilden *«zonder iets in de mouw»*: waarden waarvan iedereen de oorsprong kan verifiëren. Als iemand je zou vertellen *«vertrouw me: gebruik dit willekeurige 32-bits getal»*, zou je redelijkerwijs een verborgen zwakte of een achterdeur (backdoor) vermoeden. Maar

iedereen met een rekenmachine kan controleren dat de eerste 32 bits van de vierkantswortel van 2 `0x6a09e667` zijn. De waarden zijn wiskundig, openbaar en reproduceerbaar: er kan geen verborgen truc in het recept sluipen.

Bijlage: SHA-256 in leesbare code

Deze bijlage is voor de lezer die het algoritme van binnenuit wil bekijken. Het is een didactische implementatie in Zig die de FIPS 180-4-specificatie volgt. Het is niet de versie die Solo2 gebruikt —de echte bevindt zich in `std.crypto.hash.sha2.Sha256` van de standaardbibliotheek van Zig, geoptimaliseerd en geauditeerd—. Maar het algoritme is hetzelfde: wat je hier ziet is, stap voor stap, wat er gebeurt als die aanroep van vijf tekens zijn werk doet.

```
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
}
```

Elke herschrijving in een andere taal die dezelfde structuur volgt —beginconstanten, uitbreiding van het schema (schedule), vierenzestig rondes, accumulatie— levert hetzelfde resultaat op. Het algoritme heeft geen geheimen: de waarde ervan ligt in het feit dat de hierboven genoemde eigenschappen na twee decennia van publieke cryptanalyse door duizenden ogen nog steeds standhouden.

Als je teruggaat naar de voet van dit artikel, zie je een hexadecimaal zegel van vierenzestig tekens. Het is de SHA-256 van de tekst die je zojuist hebt gelezen, in deze taal. Als we het artikel zouden vertalen, zou het zegel anders zijn; als er een woord van de Nederlandse versie zou veranderen, zou het Nederlandse zegel veranderen. Het zegel beschermt de inhoud niet —daar zijn andere hulpmiddelen voor— maar het identificeert deze op unieke wijze. En dat, hoe bescheiden het ook klinkt, is voldoende zodat geen enkele stap in de redactionele keten wat gezegd is kan wijzigen zonder dat het opvalt. De rest —versleutelen, ondertekenen, identificeren— is gebouwd op dit eenvoudige idee.

Noot van de redactie: wanneer deze Cuadernos bedrijven of producten noemen, is dat niet om te beschuldigen. Degenen die ze bouwen, leveren werk dat miljoenen mensen gebruiken en waarderen. Wat we aanstippen is structureel — het model, niet het merk. Merken verschijnen als voorbeeld omdat dit de merken zijn die de lezer herkent.

Bronnen und verdere lectuur

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, augustus 2015. Officiële specificatie van de SHA-2-familie, inclusief SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, mei 2011. Normatieve versie voor implementeerders.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Hoofdstukken 5 en 6 behandelen hashfuncties en hun legitieme en onwettige gebruik.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Praktisch voorbeeld van het gebruik van SHA-256 om blokken te koppelen in een onveranderlijke structuur door constructie.
- Verordening (EU) 910/2014 (eIDAS) — kader voor gekwalificeerde tijdstempeldiensten. SHA-256 is de referentiefunctie voor gekwalificeerde elektronische handtekeningen en zegels die in de EU worden uitgegeven.
- Referentie-implementatie in Zig: `std.crypto.hash.sha2.Sha256` in de officiële repository van de taal (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). Het is de geoptimaliseerde en geauditeerde versie die Solo2 daadwerkelijk gebruikt. Nuttig om te contrasteren met de didactische implementatie van de bijlage.

← VorigeSchrems II, vijf jaar laterVolgende →Kill switch en institutionele inbeslagname

Recente artikelen

- Analyse · 18 mei 2026 Echte vs. schijnbare privacy: de vragen die men zich moet stellen
- Analyse · 18 mei 2026 Self-hosting als professionele praktijk
- Concept · 18 mei 2026 De 24 woorden: wat een cryptografische identiteit is

Neem dit artikel mee naar waar u het nodig heeft.

↓ Markdown ↓ Platte tekst ↓ PDF

Het bestand wordt gedownload naar uw apparaat. Van daaruit kunt u het opslaan, importeren in Solo2 of delen waar u maar wilt. Cuadernos beslist niet voor u over de bestemming.

Lakzegel · SHA-256 1bf922423ec94808389195d5998df970e2fdd2a2be81c4740cd3199b5a23ee24

Cuadernos Lacre · Een uitgave van Menzuri Gestión S.L. ·
geschreven door R.Eugenio · geredigeerd door het team van Solo2.

Deze website gebruikt geen cookies en laadt geen bronnen van derden. Er wordt gebruikgemaakt van een anonieme bezoeksteller (Umami, op onze Europese server) en het minimale JavaScript dat nodig is voor de twee knoppen in de header: licht of donker thema, en taalkeuze. Geen trackers, geen profilering, geen delen van gegevens. Als u ons wilt volgen: RSS.