Hvad SHA-256 egentlig er

Et matematisk fingeraftryk, der fylder fireogtres tegn, og som ændrer sig fuldstændigt, hvis blot et enkelt komma i den originale tekst flyttes. Hvorfor vi kalder det et digitalt laksegl.

Den enkle idé bag det tekniske navn

Forestil dig, at der findes en maskine med én sprække og én skærm. Gennem sprækken indfører du en tekst: et ord, en sætning, en hel roman. På skærmen vises øjeblikke senere en sekvens på præcis fireogtres tegn. Denne sekvens kalder vi for den professionelle læser et *hash* eller et *kryptografisk resumé*; for den almindelige læser kan vi foreløbig kalde det et matematisk fingeraftryk af teksten, ligesom et fingeraftryk er det for et menneske.

Hvis du indfører den samme tekst to gange, viser maskinen det samme fingeraftryk begge gange. Hvis du indfører en tekst, der er en lille smule anderledes —et enkelt flyttet komma, et stort bogstav der bliver lille— viser maskinen et fingeraftryk, der er fuldstændig anderledes end det første. Ikke lignende: anderledes. Disse to egenskaber sammen —determinisme og følsomhed— er den enkle idé. Alt andet ved SHA-256 er det maskineri, der får dem til at fungere godt.

Det er værd at sige fra starten, hvad maskinen ikke gør. Den krypterer ikke teksten. Den skjuler den ikke. Den gemmer den ikke. Maskinen ser på teksten, beregner fingeraftrykket og glemmer teksten. Fingeraftrykket gør det ikke muligt at rekonstruere den tekst, der dannede det; det gør det kun muligt, givet en kandidattekst, at kontrollere om den stemmer overens med originalen eller ej. Derfor siger vi, at det er et *envejsresumé*: man går dertil, man vender ikke tilbage.

Et hash er ikke det samme som kryptering

Forvirringen er hyppig, og det er værd at rydde den af vejen: at kryptere og at hashe er forskellige operationer. Kryptering består i at transformere en tekst på en sådan måde, at kun indehaveren af nøglen kan føre den tilbage til sin oprindelige form. Hashering består i at producere et fingeraftryk af teksten, som den originale tekst aldrig kan genskabes fra, hverken med eller uden nøgle. Den første er reversibel af design; den anden er irreversibel af design.

Den praktiske konsekvens er vigtig. Når en applikation siger «vi gemmer din adgangskode krypteret», er der nogen, der har nøglen til at dekryptere den — applikationen selv, i hvert fald. Når en applikation siger «vi gemmer din adgangskode hashet», kan applikationen selv ikke læse den originale adgangskode, selvom den ville; den kan kun kontrollere, om det du skriver, producerer det samme fingeraftryk igen. Den anden model er, når den er udført korrekt, meget at foretrække frem for den første til lagring af adgangskoder. Senere skal vi se, hvorfor «udført korrekt» kræver noget mere end bare SHA-256.

De fire egenskaber, der gør et kryptografisk hash nyttigt

En hashfunktion, der fortjener tillægsordet *kryptografisk*, opfylder fire egenskaber:

1. **Determinisme.** Det samme input producerer altid det samme fingeraftryk.
2. **Lavineeffekt.** En lille ændring i inputtet producerer et fuldstændig anderledes fingeraftryk, uden synlig lighed med det foregående.
3. **Modstandsdygtighed mod inversion.** Givet et fingeraftryk er det ikke beregningsmæssigt muligt at finde den tekst, der producerede det.
4. **Modstandsdygtighed mod kollisioner.** Det er ikke beregningsmæssigt muligt at finde to forskellige tekster, der producerer det samme fingeraftryk.

«Ikke beregningsmæssigt muligt» betyder ikke «matematisk umuligt». Det betyder, at omkostningerne i tid, energi og penge for at opnå det overstiger summen af al rimeligt tilgængelig beregningskapacitet med mange størrelsesordener. For SHA-256 måles denne grænse i tusinder af milliarder af år, selv for de mest optimistiske tilgange med specialiseret hardware. Hvilket til læserens praktiske formål er det samme som «det kan man ikke».

SHA-256, helt specifikt

Navnet siger det hele. SHA står for *Secure Hash Algorithm*: sikker hash-algoritme. Tallet 256 angiver størrelsen af fingeraftrykket i bits: to hundrede og seksoghalvtreds bits, det vil sige toogtredive bytes, som vist i hexadecimal er de fireogtres tegn, som læseren allerede kender. Standarden blev udgivet af det amerikanske NIST, det organ der normaliserer denne type funktioner, i 2001 som en del af SHA-2-familien; den nuværende version af standarden, FIPS 180-4, er fra 2015.

Til dem, der endnu ikke har styr på, hvad bits og bytes er:

```
1 bit    →  0 eller 1      (en kontakt: tændt eller slukket)
1 byte   →  8 bits         (256 mulige kombinationer)
32 bytes →  256 bits       (SHA-256 fingeraftrykket)
```

Tallet 256 i slutningen af navnet fortæller størrelsen af fingeraftrykket i bits. I hexadecimal —et talsystem med seksten symboler i stedet for ti— fylder disse 256 bits præcis 64 tegn. Det er de 64 tegn, du ser i bunden af hver Cuaderno.

Dimensionerne fortjener et øjeblik. To hundrede og seksoghalvtreds bits tillader to i to hundrede og seksoghalvtredsende forskellige værdier: et tal med otteoghalvfjerds decimalcifre, flere størrelsesordener større end det anslåede antal atomer i det observerbare univers. Hver tekst i verden —hver bog, hver e-mail, hver besked— lander på en af disse værdier. Sandsynligheden for, at to forskellige tekster falder sammen ved et tilfælde, er til praktiske formål umulig at skelne fra nul.

Hvordan det ser ud i kode

I Zig, det sprog vi skriver de dele i, som Solo2 hviler på, ser beregningen af SHA-256-seglet for en tekst således ud:

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

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

Vi har lige bedt Zigs standardbibliotek om at beregne SHA-256 for teksten i anførselstegn. Efter kaldet indeholder variablen *resumen* de toogtredive bytes, som udgør seglet i dets rå form; når de vises på skærmen i hexadecimal, er det de fireogtres tegn, der vises i bunden af denne artikel. Hvis vi ændrede *Cuadernos Lacre* til *Cuadernos lacre* —ét stort bogstav mindre— ville hele seglet ændre sig. Det er, på fem linjer, den centrale egenskab, som resten hviler på. For dem, der vil se, hvordan det fungerer internt, inkluderer vi en læselig version af algoritmen med kommentarer trin for trin i slutningen af artiklen.

Hvorfor vi kalder det et laksegl

I europæisk korrespondance fra det femtende til det nittende århundrede lukkede lak brevet. En dråbe smeltet voks, et segl presset ned i det, og brevet var markeret på en måde, der ikke kunne gentages. Det beskyttede ikke indholdet mod den besluttede nysgerrige —papiret kunne læses mod lyset, lakken kunne brydes— men det dokumenterede det. Enhver ændring af lukningen var synlig for modtageren, endog før papiret blev åbnet. Lakken forhindrede ikke skaden; den erklærede den.

SHA-256 af indholdet i hver Cuaderno udfører den samme funktion i sin digitale version. Hvis blot et enkelt ord i artiklen ændrede sig mellem det øjeblik, den blev udgivet, og det øjeblik, du læser den, ville det hexadecimale segl i bunden af teksten ikke længere stemme overens med SHA-256 af den tekst, du har foran dig. Enhver læser med fem linjers kode kunne kontrollere det. Udgiveren kan ikke omskrive sin historie, uden at seglet afslører det. Det beskytter ikke mod skade; det gør den verificerbar.

Hvad et hash ikke er

Fire anvendelser kræves undertiden af SHA-256, som det ikke er beregnet til:

1. **Kryptering.** Et hash opsummerer; det skjuler ikke. Hvis du ønsker, at teksten ikke skal kunne læses, skal du kryptere den, ikke hashe den.
2. **Autentificering af forfatteren.** Et hash fortæller ikke, hvem der skrev teksten, kun hvilken tekst der blev hashet. For at knytte forfatterskab til kræves en kryptografisk signatur oven på hashet, ikke hashet i sig selv.
3. **Lagring af adgangskoder.** Her er en fælde, som det er værd at forstå. SHA-256 er designet til at være meget hurtig —hvilket er godt til mange ting, men dårligt til dette. En angriber med specialiseret hardware kan afprøve milliarder af adgangskoder i sekundet mod et SHA-256 hash, indtil vedkommende finder din. Til at gemme adgangskoder skal man bruge bevidst langsomme nøgleafledningsfunktioner som Argon2, scrypt eller bcrypt, kombineret med et *salt* (en unik tilfældig data for hver bruger, der forhindrer to personer med samme adgangskode i at få det samme hash).
4. **Læsning af hashet som identifikation af forfatteren.** Det er det ikke. Et hash identificerer indholdet. Hvis to personer hasher ordet *hola* med SHA-256, får de begge det samme resumé — og det er den centrale egenskab, ikke en fejl: hvis de var forskellige resuméer, kunne vi ikke kontrollere overensstemmelse mellem det udgivne og det modtagne.

Hvor SHA-256 optræder i din hverdag

Selvom du ikke ser det, understøtter SHA-256 en stor del af det, du bruger dagligt på internettet. Bitcoins blockchain er bygget ved at lænke SHA-256 fra hver blok til den næste; at ændre en tidligere blok tvinger til at genberegne hele den efterfølgende kæde. Git, det system som halvdelen af verden bruger til versionsstyring af kode, identificerer hver bekræftelse (commit) ved SHA-256 (i nyere versioner) eller ved dens forgænger SHA-1 (i ældre versioner) af dens fulde indhold. De HTTPS-certifikater, der verificerer en hjemmesides identitet, når du går ind på den, har et tilknyttet SHA-256 fingeraftryk. Software-downloads ledsages ofte af et SHA-256, som udvikleren har udgivet, så du kan kontrollere, at filen ikke blev ændret undervejs. Og som vi har sagt, i bunden af hver Cuadernos Lacre.

Til den professionelle læser

Fire operationelle påmindelser til dem, der træffer beslutninger eller auditerer systemer:

1. Hash er ikke kryptering. Hvis en leverandør forveksler de to udtryk i sin tekniske dokumentation, er det værd at spørge, hvad vedkommende præcis mener.
2. Til lagring af adgangskoder bør man aldrig bruge SHA-256 alene. SHA-256 er for hurtig til denne opgave (se punkt 3 i *Hvad et hash ikke er*). Den nuværende standard er **Argon2id**: langsom af design, konfigurerbar efter serverens kapacitet, kombineret med et forskelligt tilfældigt *salt* for hver bruger.
3. For dokumentintegritet —kontrakter, sager, filer— er SHA-256 fortsat referencestandarden. Det er det, der bruges af kvalificerede tidsstempeltjenester i EU.
4. For langtidskonservering (årtier) er det værd også at beregne og arkivere et SHA-3 eller et SHA-512 sammen med SHA-256; kryptografisk forsigtighed anbefaler ikke at forlade sig på en enkelt funktion under hundredårige arkiveringer.

Teknisk set er denne itererede struktur — hvor den mellemliggende tilstand bevares mellem indgangsblokke — kendt som en **Merkle-Damgård**-konstruktion, det mønster som SHA-1, SHA-2 (inklusive SHA-256) og many andre klassiske hash-funktioner er baseret på. SHA-3 forlader derimod Merkle-Damgård til fordel for en anden arkitektur kaldet *sponge* (svamp).

Hvordan SHA-256 fungerer, trin for trin, i jævne ord

Forestil dig, at du har bygget den mest avancerede dominobane i verden: tusindvis af brikker, snesevis af forgreninger, mekaniske broer og ramper, der krydser hele rummet, omhyggeligt placeret brik for brik.

Hvis du giver den første brik et puf, falder kæden i en præcis og gentagelig sekvens. Samme opstilling, samme første puf → identisk slutresultat af faldne brikker, gang på gang.

Her er det interessante: Flyt **en enkelt brik** en halv centimeter til siden, før du starter, og puf igen. En rampe, der skulle have været aktiveret, forbliver inaktiv, en bro falder ikke, en helt anden forgrening udløses. Det endelige mønster af brikker på gulvet er fuldstændig uigenkendeligt sammenlignet med det første.

SHA-256 er matematisk set denne bane. Teksten, du skriver, er brikkernes startposition. Algoritmen er puffet, der frigiver kaskaden. And det endelige resultat — det, vi kalder et *hash* — er det fastfrosne billede af gulvet, når alt er stoppet. Skift et enkelt komma i den oprindelige tekst, og billedet vil være radikalt anderledes. Så enkelt og så drastisk.

Trin 1. Oversæt teksten til binære brikker. Computere forstår ikke bogstaver; de oversætter dem først til tal (ASCII) og tallene til binære tal (et-taller og nuller). Hvert bogstav bliver til 8 hvide eller sorte brikker: *A* er `01000001`, *B* er `01000010`, mellemrum er `00100000`. Hele din tekst — et ord, en kontrakt, en roman — bliver til en lang række af hvide og sorte brikker.

Trin 2. Udfyld til standardstørrelsen. Banen behandler rækken i *stykker* på præcis 512 brikker. Hvis din besked ikke når et multiplum af 512, tilføjes en markørbrik (værdien `10000000`) lige efter teksten og derefter nuller, indtil stykket er komplet. De sidste 64 pladser i hvert stykke er reserveret til at notere tekstens oprindelige længde. På den måde ved banen altid, hvor det reelle indhold sluttede, og hvor udfyldningen begyndte.

Trin 3. Placer de otte masterbrikker. Før vi starter, placerer vi **otte masterbrikker** på bordet i en præcis startposition. Disse otte brikker er ingen hemmelighed: Deres startværdi er fastsat af en offentlig matematisk regel (kvadratrødderne af de første otte primtal — 2, 3, 5, 7, 11, 13, 17, 19 — og de første bits af decimaldelen af hver rod). Alle, i ethvert hjørne af planeten, starter med de samme otte masterbrikker i samme position. Deres skæbne er at blive puffet til og transformeret af lavinen.

Trin 4. Den store lavine: fireogtres runder med puf. Her begynder forestillingen. Det første stykke på 512 brikker af din tekst bringes til at kollidere med de otte masterbrikker. Men de falder ikke på én gang: Mekanismen udfører **fireogtres på hinanden følgende runder**. I hver runde udfører den tre operationer med brikkerne:

- **Karrusellen** (rotation). Brikkerne bevæger sig i en cirkel: Dem til højre flytter til venstre. Ingen brik går tabt eller bliver tilføjet; de bliver blot omorganiseret ved at tage en hel tur i karrusellen. Det er en billig og reversibel måde at omfordele information på.
- **Den logiske tragt** (XOR). Brikkerne passerer gennem en tragt, der sammenligner dem to og to: Hvis de to har samme farve, kommer der en hvid ud; hvis de er forskellige, kommer der en sort ud. Det er den enkleste operation i binær logik, men kombineret med karrusellens rotationer bliver den ekstremt kraftfuld til at blande information uden at miste den.
- **Overløbet** (modulær addition). Resultatet lægges sammen med en *konstant pufbrik* hentet fra en offentlig liste med fireogtres konstanter (kubikrødderne af de første fireogtres primtal). Hvis additionen genererer ekstra brikker, der ikke kan være på de 32 pladser, der er afsat, bliver de overskydende brikker kasseret. Bordet har kun plads til 32 brikker, hverken mere eller mindre.

Ved slutningen af runde fireogtres har hver af brikkerne fra stykket i din tekst påvirket masterbrikkernes position. Energien fra puffet har rejst gennem hele banen.

Trin 5. Tilføj det næste stykke (uden genstart). Hvis din tekst var lang, og der er endnu et stykke på 512 brikker, der skal behandles, **genstarter banen ikke**. De otte masterbrikker forbliver, som den første lavine efterlod dem, og det andet stykke sendes mod dem for at aktivere yderligere fireogtres runder. Det er ligesom at tilføje et nyt rum fyldt med dominoer for enden af det, der lige er faldet: Uordenen i det første rum betinger fuldstændig, hvordan det andet vil falde.

Trin 6. Tag det endelige billede. Når der ikke er flere stykker at behandle, stopper lavinen. Vi ser på den endelige position, som de otte masterbrikker er endt i. Vi oversætter deres konfiguration til en kode af bogstaver og tal i det hexadecimale system. Resultatet er en streng på præcis fireogtres tegn: Det er dit SHA-256-segl.

Fire egenskaber følger naturligt af, hvordan banen er bygget:

1. **Determinisme.** Den samme tekst producerer altid det samme endelige billede på enhver computer i verden. Nul tilfældighed, nul overraskelser.
2. **Lavineeffekt.** Et tilføjet komma, et ændret stort bogstav, en glemt accent: Billedet bliver fuldstændig uigenkendeligt. Dette is den ekstreme følsomhed, som vi allerede beskrev i begyndelsen.
3. **Kun én vej.** Givet det endelige billede kan du ikke rekonstruere den oprindelige tekst. Rotationerne, tragtene og overløbene ødelægger al retningsbestemt information om, *hvor hver bit kom fra*, og bevarer kun, *hvad der i alt blev*

lagt sammen.
4. **Resistens mod kollisioner.** I femogtyve år med offentlig kryptoanalyse er det ikke lykkedes nogen at finde to forskellige tekster, hvis endelige billeder stemmer overens. Og sværhedsgraden ved at gøre det ligger uden for den beregningsmæssige rækkevidde for enhver civilisation, man med rimelighed kan forestille sig.

Det efterfølgende kodebilag implementerer netop disse seks trin i Zig. Nu kan du læse det med viden om, hvad hver bit-operation betyder, i stedet for blot at acceptere manipulationerne blindt.

Teknisk ordliste

Til læseren, der ønsker at forstå, hvad hver operation gør. Spring det frit over: Artiklen kan stadig forstås uden det.

ASCII og Unicode — hvordan bogstaver bliver til tal. Computere ser ikke bogstaver; de ser tal. En standard kaldet **ASCII** (*American Standard Code for Information Interchange*, fra 1963) tildeler hvert tegn på tastaturet et specifikt tal: *A* is 65, *B* is 66, *a* is 97, *0* is 48, mellemrum er 32, komma er 44. Moderne systemer udvider dette med **Unicode**, som tildeler et tal til hvert tegn i alle alfabeter i verden: kyrillisk, arabisk, kinesisk, japansk og endda emojis. Når du skriver et tegn eller åbner en tekstfil, læser computeren det bagvedliggende tal, ikke formen på skærmen. SHA-256 arbejder på disse tal og behandler enhver tekst som en lang række af cifre. Derfor kan det forsegle en artikel på spansk, et digt på japansk og en binær fil med den samme algoritme.
XOR — bit-sammenligneren. XOR (udtales *'eks-or'*, fra engelsk *exclusive or*, 'eksklusivt eller') er en af de enkleste operationer, en computer kan udføre med to binære tal. Den sammenligner to bits position for position og returnerer: **1** hvis præcis én af de to er 1 (den ene, men ikke begge), **0** hvis de to er ens (begge 0 eller begge 1). Eksempel: XOR af `1010` og `1100` er `0110`. Den har en bemærkelsesværdig egenskab: Den er reversibel — hvis du laver XOR to gange med den samme nøgle, vender du tilbage til udgangspunktet. Derfor er den kryptografiens arbejdshest: Den blander bits uden at miste information, men resultatet afslører intet om inputtene, hvis du ikke kender et af dem.
Hexadecimal — at tælle i talsystemet med grundtal 16. Næsten alle tal i hverdagen bruger ti cifre (0-9). Det hexadecimale system bruger seksten: de sædvanlige 0-9 plus seks bogstaver, der repræsenterer de følgende værdier: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Hvorfor seksten? Fordi computere tænker i grupper af fire bits, og fire bits kan repræsentere præcis seksten forskellige værdier — således svarer ét hexadecimalt tegn rent til fire bits. Et SHA-256-aftryk måler 256 bits, hvilket er præcis **64 hexadecimale tegn**. Hvis vi skrev det med almindelige decimaltal, ville det fylde omkring 78 cifre og være mere besværligt. Valget is æstetisk og kompakt; det bagvedliggende tal er det samme.
Bit-rotation — den binære karrusel. Forestil dig en række med syv pærer, hvor nogle er tændt (1) og andre er slukket (0): `1 0 1 1 0 0 1`. At rotere én position til højre består i at tage pæren helt til højre, flytte den til den yderste venstre fløj og rykke de andre én plads til højre: `1 1 0 1 1 0 0`. Ingen pære går tabt eller bliver tilføjet: De danser blot i en cirkel. SHA-256 bruger bit-rotation hundredvis af gange i hver beregning; det er en billig og tabsfri måde at omfordele informationen i tilstanden på.
Konstanter af typen *'nothing-up-my-sleeve'* — hvorfor de stammer fra primtal. De otte masterbrikker og de fireogtres runde-konstanter i SHA-256 blev ikke valgt tilfældigt. De stammer fra kvadratrødderne og kubikrødderne af de første primtal. Hvorfor? Fordi deres designere ønskede konstanter *'uden noget i ærmet'*: værdier, hvis oprindelse alle kan verificere. Hvis nogen sagde til dig *'stol på mig: brug dette tilfældige 32-bit tal'*, ville du med rimelighed mistænke en skjult svaghed eller en bagdør. Men alle med en lommeregner kan kontrollere, at de første 32 bits af kvadratroden af 2 er `0x6a09e667`. Værdierne er matematiske, offentlige og reproducerbare: Ingen skjulte tricks kan snige sig ind i opskriften.

Appendiks: SHA-256 i læselig kode

Dette appendiks er til den læser, der ønsker at se algoritmen indefra. Det er en didaktisk implementering i Zig, der følger FIPS 180-4 specifikationen. Det er ikke den version, Solo2 bruger —den rigtige findes i `std.crypto.hash.sha2.Sha256` i Zigs standardbibliotek, optimeret og auditeret—. Men algoritmen er den samme: det, du ser her, er trin for trin, hvad der sker, når det kald på fem tegn udfører sit arbejde.

```
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
}
```

Enhver omskrivning til et andet sprog, der følger den samme struktur —startkonstanter, udvidelse af skemaet, fireogtres runder, akkumulering— producerer det samme resultat. Algoritmen har ingen hemmeligheder: dens værdi ligger i, at de ovennævnte egenskaber stadig holder efter to årtiers offentlig kryptoanalyse af tusindvis af øjne.

Hvis du vender tilbage til bunden af denne artikel, vil du se et hexdecimalt segl på fireogtres tegn. Det er SHA-256 af den tekst, du lige har læst, på dette sprog. Hvis vi oversatte artiklen, ville seglet være et andet; hvis et enkelt ord i den danske version ændrede sig, ville det danske segl ændre sig. Seglet beskytter ikke indholdet —til det findes der andre værktøjer— men det identificerer det entydigt. Og det er, uanset hvor beskedent det lyder, nok til at intet led i den redaktionelle kæde

kan ændre det sagte, uden at det bemærkes. Resten —at kryptere, at signere, at identificere— bygges oven på denne enkle idé.

Kilder og yderligere læsning

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, august 2015. Officiel specifikation for SHA-2-familien, herunder SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, maj 2011. Normativ version for implementører.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Kapitel 5 og 6 dækker hashfunktioner og deres legitime og illegitime anvendelser.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Praktisk eksempel på brugen af SHA-256 til at lænke blokke i en struktur, der er uforanderlig af konstruktion.
- Forordning (EU) 910/2014 (eIDAS) — ramme for kvalificerede tidsstempler. SHA-256 er referencefunktionen for kvalificerede elektroniske signaturer og segl udstedt i EU.
- Referenceimplementering i Zig: `std.crypto.hash.sha2.Sha256` i sprogets officielle repository (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). Det er den optimerede og auditerede version, som Solo2 faktisk bruger. Nyttig til at sammenligne med den didaktiske implementering i appendikset.

← ForrigeCUADERNOS_LIST_SCHREMS_TITLENæste →CUADERNOS_LIST_KILLSWITCH_TITLE

Seneste læsning

- CUADERNOS_LIST_PREGUNTAS_TITLE
- CUADERNOS_LIST_SELFHOST_TITLE
- CUADERNOS_LIST_IDENTIDAD_TITLE

Tag denne artikel med dig, hvor du har brug for den.

↓ Markdown ↓ Almindelig tekst ↓ PDF

Filen downloades til din enhed. Derfra kan du gemme den, importere den til Solo2 eller dele den, hvor du vil. Cuadernos beslutter ikke destinationen for dig.

Laksegl · SHA-256 14aa67524983f9b28cff4e6475b106832b7c9e9f486660a2730cebfc7e797a91

Cuadernos Lacre · En udgivelse fra Menzuri Gestión S.L. ·
skrevet af R.Eugenio · redigeret af holdet bag Solo2.