Hva SHA-256 egentlig er

Et matematisk fingeravtrykk på sekstifire tegn som endres fullstendig hvis bare ett komma i originalteksten flyttes. Hvorfor vi kaller det et digitalt segl.

Den enkle ideen bak det tekniske navnet

Se for deg en maskin med bare én åpning og én skjerm. Gjennom åpningen legger du inn en tekst: et ord, en setning, en hel roman. På skjermen vises det, øyeblikk senere, en sekvens på nøyaktig sekstifire tegn. Denne sekvensen kaller vi for *hash* eller *kryptografisk sammendrag*; for den vanlige leseren kan vi foreløpig kalle det et matematisk fingeravtrykk av teksten, akkurat som et fingeravtrykk er unikt for en person.

Hvis du legger inn den samme teksten to ganger, viser maskinen det samme fingeravtrykket begge gangene. Hvis du legger inn en tekst som er bare litt annerledes —et komma er flyttet, eller en stor bokstav blir liten— viser maskinen et fingeravtrykk som er fullstendig annerledes enn det første. Ikke bare likt: helt annerledes. Disse to egenskapene sammen —determinisme og følsomhet— er den enkle ideen. Alt annet ved SHA-256 er maskineriet som sørger for at disse egenskapene overholdes.

Det er greit å si med en gang hva maskinen ikke gjør. Den krypterer ikke teksten. Den skjuler den ikke. Den lagrer den ikke. Maskinen ser på teksten, beregner fingeravtrykket, og glemmer teksten. Fingeravtrykket gjør det ikke mulig å gjenskape teksten som produserte det; det gjør det bare mulig, gitt en annen tekst, å sjekke om den stemmer overens med originalen eller ikke. Derfor sier vi at det er et *enveis* sammendrag: du kan gå én vei, men ikke tilbake.

En hash er ikke det samme som kryptering

Det er ofte forvirring rundt dette, og det er greit å avklare: kryptering og hashing er to forskjellige operasjoner. Kryptering går ut på å transformere en tekst slik at bare den som har nøkkelen kan få den tilbake til original form. Hashing går ut på å produsere et fingeravtrykk av teksten der originalteksten aldri kan gjenskapes, verken med eller uten nøkkel. Den første er reversibel ved design; den andre er irreversibel ved design.

Den praktiske konsekvensen er viktig. Når en applikasjon sier «vi lagrer passordet ditt kryptert», er det noen som har nøkkelen til å dekryptere det — applikasjonen selv, i alle fall. Når en applikasjon sier «vi lagrer passordet ditt hashet», kan ikke applikasjonen selv lese originalpassordet selv om den ville; den kan bare sjekke om det du skriver inn produserer det samme fingeravtrykket igjen. Den andre modellen, når den er gjort riktig, er langt å foretrekke fremfor den første når det gjelder lagring av passord. Senere skal vi se hvorfor «gjort riktig» krever noe mer enn bare SHA-256 i seg selv.

De fire egenskapene som gjør en kryptografisk hash nyttig

En hash-funksjon som fortjener betegnelsen *kryptografisk* oppfyller fire egenskaper:

1. **Determinisme.** Samme inndata gir alltid samme fingeravtrykk.
2. **Snøballeffekt (Avalanche effect).** En liten endring i inndata gir et fullstendig annerledes fingeravtrykk, uten noen synlig likhet med det forrige.
3. **Motstand mot reversering.** Gitt et fingeravtrykk, er det ikke beregningsmessig gjennomførbart å finne teksten som produserte det.
4. **Motstand mot kollisjoner.** Det er ikke beregningsmessig gjennomførbart å finne to forskjellige tekster som produserer samme fingeravtrykk.

«Ikke beregningsmessig gjennomførbart» betyr ikke «matematisk umulig». Det betyr at kostnaden i tid, energi og penger for å oppnå det overstiger den samlede kapasiteten til all rimelig tilgjengelig datakraft med mange størrelsesordener. For SHA-256 måles denne grensen i tusenvis av billioner år, selv med de mest optimistiske anslagene med spesialisert maskinvare. Noe som for alle praktiske formål betyr at «det ikke går».

SHA-256, helt konkret

Navnet sier alt. SHA er forkortelsen for *Secure Hash Algorithm*: en sikker hash-algoritme. Tallet 256 angir størrelsen på fingeravtrykket i bits: to hundre og femti-seks bits, det vil si trettito bytes, som i heksadesimal form er de sekstifire tegnene som leseren allerede gjenkjenner. Standarden ble publisert av amerikanske NIST, organet som standardiserer denne typen funksjoner, i 2001 som en del av SHA-2-familien; den gjeldende versjonen av standarden, FIPS 180-4, er fra 2015.

For de som ennå ikke har helt kontroll på hva bits og bytes er:

```
1 bit    →  0 eller 1      (en bryter: på eller av)
1 byte   →  8 bits         (256 mulige kombinasjoner)
32 bytes →  256 bits       (SHA-256-fingeravtrykket)
```

Tallet 256 sist i navnet angir størrelsen på fingeravtrykket i bits. I heksadesimal form —et tallsystem med seksten symboler i stedet for ti— får disse 256 bitene plass på nøyaktig 64 tegn. Dette er de 64 tegnene du ser nederst i hvert Cuaderno.

Dimensjonene fortjener et øyeblikks oppmerksomhet. To hundre og femti-seks bits gir to opphøyd i to hundre og femti-seks forskjellige verdier: et tall med syttiåtte desimaltegn, mange størrelsesordener større enn det anslåtte antallet atomer i det observerbare universet. Hver tekst i verden —hver bok, hver e-post, hver melding— havner på en av disse verdiene. Sannsynligheten for at to forskjellige tekster skal treffe samme verdi ved en tilfeldighet er for alle praktiske formål lik null.

Hvordan det ser ut i kode

I Zig, språket vi bruker for å skrive delene som bærer Solo2, ser beregningen av SHA-256-seglet til en tekst slik ut:

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

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

Vi har nettopp bedt Zigs standardbibliotek om å beregne SHA-256 av teksten i hermetegn. Etter kallet inneholder variabelen *resumen* de trettito bytene som utgjør seglet i råform; når de vises på skjermen i heksadesimal form, er det de sekstifire tegnene som vises nederst i denne artikkelen. Hvis vi endret *Cuadernos Lacre* til *Cuadernos lacre* —én stor bokstav mindre— ville hele seglet ha endret seg. Det er, på fem linjer, den sentrale egenskapen som bærer resten. For de som vil se hvordan det fungerer internt, har vi inkludert en lesbar versjon av algoritmen med kommentarer steg for steg nederst i artikkelen.

Hvorfor vi kaller det et segl

I europeisk korrespondanse fra det femtende til det nittende århundre ble brev lukket med lakk. En dråpe smeltet voks, et segl presset ned i den, og brevet ble merket på en måte som ikke kunne gjentas. Det beskyttet ikke innholdet mot en bestemt snoker —papiret kunne leses mot lyset, og lakken kunne brytes— men det gjorde det synlig. Enhver endring av lukkingen var synlig for mottakeren før brevet i det hele tatt ble åpnet. Lakken hindret ikke skaden; den erklærte den.

SHA-256 i innholdet i hvert Cuaderno fyller den samme funksjonen i digital versjon. Hvis bare ett enkelt ord i artikkelen endres mellom publiseringstidspunktet og tidspunktet du leser den, vil det heksadesimale seglet nederst i teksten ikke lenger stemme overens med SHA-256 av teksten du har foran deg. Enhver leser med fem linjer kode kan sjekke dette. En publikasjon kan ikke skrive om sin egen historie uten at seglet avslører det. Det beskytter ikke mot skade; det gjør den etterprøvbar.

Hva en hash ikke er

Det forventes av og til fire bruksområder for SHA-256 som den ikke er ment for:

1. **Kryptering.** En hash oppsummerer; den skjuler ikke. Hvis du vil at teksten ikke skal kunne leses, må du kryptere den, ikke hashe den.
2. **Autentisere forfatteren.** En hash forteller ikke hvem som skrev teksten, bare hvilken tekst som ble hashet. For å knytte forfatterskap til teksten kreves en kryptografisk signatur på toppen av hashen, ikke bare hashen alene.
3. **Lagring av passord.** Her er det en felle man bør forstå. SHA-256 er designet for å være veldig rask —noe som er bra for mye, men dårlig for akkurat dette. En angriper med spesialisert maskinvare kan prøve milliarder av passord per sekund mot en SHA-256-hash helt til de finner ditt. For å lagre passord må man bruke funksjoner for nøkkelutledning som er bevisst trege, som Argon2, scrypt eller bcrypt, kombinert med en *salt* (en unik tilfeldig verdi per bruker, som hindrer at to personer med samme passord får samme hash).
4. **Lese hashen som identifikator for forfatteren.** Det er den ikke. En hash identifiserer innholdet. Hvis to personer hasher ordet *hei* med SHA-256, får begge det samme sammendraget — og det er selve hovedegenskapen, ikke en feil: hvis de var forskjellige, kunne vi ikke kontrollert om det som ble publisert og det som ble mottatt er det samme.

Hvor SHA-256 dukker opp i hverdagen din

Selv om du ikke ser det, støtter SHA-256 mye av det du bruker daglig på internett. Bitcoins blokkjede er bygget ved å lenke SHA-256 av hver blokk til den neste; å endre en tidligere blokk tvinger frem en ny beregning av hele den etterfølgende kjeden. Git, systemet som brukes til versjonskontroll av kode over hele verden, identifiserer hver bekreftelse (commit) med SHA-256 (i nyere versjoner) eller med forgjengeren SHA-1 (i eldre versjoner) av hele innholdet. HTTPS-sertifikater som bekrefter identiteten til et nettsted når du besøker det, har et tilknyttet SHA-256-fingeravtrykk. Programvarenedlastinger følges ofte av en SHA-256 publisert av utvikleren, slik at du kan bekrefte at filen ikke ble endret underveis. Og, som vi har sagt, nederst i hvert Cuaderno Lacre.

For den profesjonelle leseren

Fire påminnelser til de som tar beslutninger eller auditerer systemer:

1. Hash er ikke kryptering. Hvis en leverandør forveksler de to begrepene i sin tekniske dokumentasjon, er det lurt å spørre nøyaktig hva de mener.
2. For lagring av passord bør man aldri bruke SHA-256 alene. SHA-256 er for rask til denne oppgaven (se punkt 3 under *Hva en hash ikke er*). Gjeldende standard er **Argon2id**: treg ved design, konfigurerbar etter serverens kapasitet, kombinert med en unik og tilfeldig *salt* per bruker.
3. For dokumentintegritet —kontrakter, saksmapper, filer— er SHA-256 fortsatt referansestandarden. Det er dette som brukes av kvalifiserte tidsstemplingstjenester i EU.
4. For langtidslagring (tiår) er det lurt å beregne og arkivere en SHA-3 eller SHA-512 sammen med SHA-256; kryptografisk forsiktighet tilsier at man ikke bør stole på bare én funksjon for arkiver som skal vare i hundre år.

Teknisk sett er denne itererte strukturen — der den mellomliggende tilstanden bevares mellom inndatablokkene — kjent som en **Merkle-Damgård**-konstruksjon, mønsteret som SHA-1, SHA-2 (inkludert SHA-256) og mange andre klassiske hashfunksjoner er basert på. SHA-3 forlater derimot Merkle-Damgård til fordel for en annen arkitektur kalt *svamp* (sponge).

Hvordan SHA-256 fungerer, trinn for trinn, med enkle ord

Forestill deg at du har bygget verdens mest forseggjorte dominobane: tusenvis av brikker, dusinvis av forgreninger, mekaniske broer og ramper som krysser hele rommet, møysommelig plassert brikke for brikke.

Hvis du gir den første brikken et dytt, faller rekken i en nøyaktig og repeterbar sekvens. Samme oppsett, samme første dytt → identisk sluttmønster av falne brikker, gang på gang.

Her er det interessante: flytt **bare én brikke** en halv centimeter til siden før du begynner og dytt igjen. En rampe som burde ha blitt aktivert forblir død, en bro faller ikke, en annen forgrening utløses. Sluttmønsteret av brikker på gulvet er helt ugjenkjennelig sammenlignet med det første.

SHA-256 er matematisk sett denne banen. Teksten du skriver er brikkenes startposisjon. Algoritmen er dyttet som utløser kaskaden. Og sluttresultatet — det vi kaller *hash* — er det frosne bildet av gulvet når alt har stoppet. Endre ett eneste komma i originalteksten og bildet blir radikalt annerledes. Så enkelt, og så drastisk.

Trinn 1. Oversett teksten til binære brikker. Datamaskiner forstår ikke bokstaver; de oversetter dem først til tall (ASCII) og tallene til binært (enere og nuller). Hver bokstav blir gjort om til 8 hvite eller svarte brikker: *A* er `01000001`, *B* er `01000010`, mellomrom er `00100000`. Hele teksten din — et ord, en kontrakt, en roman — blir en lang rekke med hvite og svarte brikker.

Trinn 2. Fyll ut til standardstørrelse. Banen behandler rekken i *blokker* på nøyaktig 512 brikker. Hvis meldingen din ikke når et multiplum av 512, legges det til en markørbrikke (den med verdien `10000000`) rett etter teksten og deretter nuller for å fylle ut blokken. De siste 64 posisjonene i hver blokk er reservert for å notere tekstens opprinnelige lengde. Dermed vet banen alltid hvor det faktiske innholdet endte og hvor fyllet begynte.

Trinn 3. Plasser de åtte mesterbrikkene. Før vi begynner, plasserer vi **åtte mesterbrikker** på bordet i en nøyaktig startposisjon. Disse åtte brikkene er ingen hemmelighet: startverdien er fastsatt av en offentlig matematisk regel (kvadratrøttene av de åtte første primtallene — 2, 3, 5, 7, 11, 13, 17, 19 — og de første bitene av desimaldelen til hver rot). Alle, uansett hvor de er i verden, starter med de samme åtte mesterbrikkene i samme posisjon. Deres skjebne er å bli dyttet og transformert av skredet.

Trinn 4. Det store skredet: sekstifire runder med dytt. Her begynner showet. Den første blokken på 512 brikker av teksten din kolliderer med de åtte mesterbrikkene. Men de faller ikke med en gang: mekanismen utfører **sekstifire påfølgende runder**. I hver runde gjøres tre operasjoner med brikkene:

- **Karusellen** (rotasjon). Brikkene beveger seg i sirkel: de til høyre flyttes til venstre. Ingen brikker mistes eller legges til; de omorganiseres ganske enkelt ved å ta en hel runde på karusellen. Det er en billig og reversibel måte å omfordele informasjonen på.
- **Den Logiske Trakten** (XOR). Brikkene går gjennom en trakt som sammenligner dem to og to: hvis begge har samme farge, kommer det ut en hvit; hvis de er forskjellige, kommer det ut en svart. Det er den enkleste operasjonen i binær logikk, men kombinert med karusellens rotasjoner blir den utrolig kraftig for å blande informasjon uten å miste den.
- **Overløpet** (modulær addisjon). Resultatet legges sammen med en *konstant dyttbrikke* hentet fra en offentlig liste på sekstifire konstanter (kuberøttene av de sekstifire første primtallene). Hvis summen genererer ekstra brikker som ikke passer på plassen til de 32 planlagte brikkene, kastes disse overflødige brikkene. Bordet har bare plass til 32 brikker, ikke en eneste mer.

På slutten av runde sekstifire har hver av brikkene fra tekstblokken din påvirket posisjonen til de åtte mesterbrikkene. Energien fra dyttet har reist gjennom hele banen.

Trinn 5. Legg til neste blokk (uten å starte på nytt). Hvis teksten din var lang og det gjenstår en ny blokk på 512 brikker å behandle, **nullstilles ikke banen**. De åtte mesterbrikkene blir liggende akkurat slik det første skredet etterlot dem, og den andre blokken kastes mot dem for å aktivere ytterligere sekstifire runder. Det er som å legge til et nytt rom fullt av dominobrikker i enden av det som nettopp falt: uorden i det første bestemmer fullt ut hvordan det andre vil falle.

Trinn 6. Ta det endelige bildet. Når det ikke gjenstår flere blokker å behandle, stopper skredet. Vi ser på den endelige posisjonen de åtte mesterbrikkene har havnet i. Vi oversetter konfigurasjonen deres til en kode av bokstaver og tall i det heksadesimale systemet. Resultatet er en streng på nøyaktig sekstifire tegn: det er ditt SHA-256-segl.

Fire egenskaper følger naturlig av hvordan banen er satt opp:

1. **Determinisme.** Samme tekst gir alltid samme sluttbilde, på en hvilken som helst datamaskin i verden. Null tilfeldighet, null overraskelser.
2. **Skredeffekt (Avalanche).** Et ekstra komma, en endret stor bokstav, en glemt aksent: bildet blir fullstendig ugjenkjennelig. Dette er den ekstreme følsomheten vi allerede har beskrevet i begynnelsen.
3. **Enveis (One-way).** Gitt sluttbildet kan du ikke rekonstruere originalteksten. Rotasjonene, traktene og overløpene ødelegger all retningsinformasjon om *hvor hver bit kom fra* og bevarer bare *hva som ble lagt sammen totalt*.
4. **Kollisjonsmotstand.** Gjennom tjuefem år med offentlig kryptoanalyse har ingen klart å finne to forskjellige tekster der sluttbildene er like. Og vanskeligheten med å gjøre det er utenfor den datakraften noen sivilisasjon rimeligvis kan forestille seg.

Kodetillegget som følger implementerer nøyaktig disse seks trinnene i Zig. Nå kan du lese det og vite hva hver bit-operasjon betyr, i stedet for å akseptere manipulasjonene i blinde.

Teknisk ordliste

For leseren som vil forstå hva hver operasjon gjør. Hopp over det hvis du vil: artikkelen er fortsatt forståelig uten det.

ASCII og Unicode — hvordan bokstaver blir tall. Datamaskiner ser ikke bokstaver; de ser tall. En standard kalt **ASCII** (*American Standard Code for Information Interchange*, fra 1963) tilordner hvert tastaturtegn et spesifikt nummer: *A* er 65, *B* er 66, *a* er 97, *0* er 48, mellomrom er 32, komma er 44. Moderne systemer utvider dette med **Unicode**, som tilordner et tall til hvert tegn i alle verdens alfabeter: kyrillisk, arabisk, kinesisk, japansk og til og med emojier. Når du skriver et tegn eller åpner en tekstfil, leser datamaskinen tallet i bakgrunnen, ikke formen på skjermen. SHA-256 jobber med disse tallene, og behandler all tekst som en lang sekvens av siffer. Derfor kan den forsegle en artikkel på spansk, et dikt på japansk og en binærfil med den samme algoritmen.
XOR — bit-for-bit komparatoren. XOR (uttales *«exor»*, fra engelsk *exclusive or*, «eksklusiv eller») er en av de enkleste operasjonene en datamaskin kan gjøre med to binære tall. Den sammenligner to bits posisjon for posisjon og returnerer: **1** hvis nøyaktig en av de to er 1 (en, men ikke begge), **0** hvis de to er like (begge 0 eller begge 1). Eksempel: XOR av `1010` og `1100` er `0110`. Den har en bemerkelsesverdig egenskap: den er reversibel — hvis du gjør XOR to ganger med den samme nøkkelen, kommer du tilbake til originalen. Derfor er den kryptografiens arbeidshest: den blander bits uten å miste informasjon, men resultatet avslører ingenting om inngangene hvis du ikke kjenner en av dem.
Heksadesimalt — telle med base 16. Nesten alle hverdagstall bruker ti sifre (0-9). Heksadesimalt bruker seksten: de vanlige 0-9 pluss seks bokstaver som representerer følgende verdier: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Hvorfor seksten? Fordi datamaskiner tenker i grupper på fire bits, og fire bits kan representere nøyaktig seksten forskjellige verdier — dermed tilsvarer et heksadesimalt tegn pent fire bits. En SHA-256-hash (segl) måler 256 bits, som er nøyaktig **64 heksadesimale tegn**. Hvis vi skrev det i vanlig desimal, ville det ta opp omtrent 78 sifre og vært mer uhåndterlig. Valget er estetisk og kompakt; det underliggende tallet er det samme.
Bitrotasjon — den binære karusellen. Forestill deg en rekke med syv lyspærer, noen på (1) og andre av (0): `1 0 1 1 0 0 1`. Å rotere en posisjon til høyre består i å ta lyspæren lengst til høyre, flytte den helt til venstre og forskyve de andre en plass til høyre: `1 1 0 1 1 0 0`. Ingen lyspære går tapt eller legges til: de danser ganske enkelt i en sirkel. SHA-256 bruker bitrotasjon hundrevis av ganger i hver beregning; det er en billig og tapsfri måte å omfordele informasjon i tilstanden.
***«Nothing-up-my-sleeve»*-konstanter — hvorfor de kommer fra primtall.** De åtte mesterbrikkene og de sekstifire runde-konstantene i SHA-256 ble ikke valgt tilfeldig. De kommer fra kvadratrøttene og kuberøttene av de første primtallene. Hvorfor? Fordi designerne ville ha konstanter *«uten noe i ermet»*: verdier hvis opprinnelse hvem som helst kan verifisere. Hvis noen fortalte deg *«stol på meg: bruk dette tilfeldige 32-bits tallet»*, ville du med rimelighet mistenkt en skjult svakhet eller en bakdør. Men alle med en kalkulator kan sjekke at de første 32 bitene av kvadratroten til 2 er `0x6a09e667`. Verdiene er matematiske, offentlige og reproduserbare: ingen skjulte triks kan snike seg inn i oppskriften.

Appendiks: SHA-256 i lesbar kode

Dette appendikset er for leseren som vil se algoritmen fra innsiden. Det er en pedagogisk implementering i Zig som følger FIPS 180-4-spesifikasjonen. Det er ikke versjonen som Solo2 bruker —den ekte finnes i `std.crypto.hash.sha2.Sha256` i Zigs standardbibliotek, optimalisert og auditert—. Men algoritmen er den samme: det du ser her er, steg for steg, det som skjer når det kallet på fem tegn utfører jobben sin.

```
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 omskriving til et annet språk som følger den samme strukturen —startkonstanter, utvidelse av skjemaet (schedule), sekstifire runder, akkumulering— gir det samme resultatet. Algoritmen har ingen hemmeligheter: verdien ligger i at egenskapene nevnt ovenfor fortsatt holder stand etter to tiår med offentlig kryptanalyse av tusenvis av øyne.

Hvis du går tilbake til slutten av denne artikkelen, vil du se et heksadesimalt segl på sekstifire tegn. Det er SHA-256 av teksten du nettopp har lest, på dette språket. Hvis vi oversatte artikkelen, ville seglet vært et annet; hvis ett ord i den norske versjonen ble endret, ville det norske seglet endret seg. Seglet beskytter ikke innholdet —det finnes andre verktøy for det— men det identifiserer det unikt. Og det, uansett hvor beskjedent det høres ut, er nok til at ingen ledd i den redaksjonelle kjeden kan endre det som er sagt uten at det merkes. Resten —kryptering, signering, identifisering— er bygget på toppen av denne enkle ideen.

Kilder og videre lesing

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, august 2015. Offisiell spesifikasjon for SHA-2-familien, inkludert SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, mai 2011. Normativ versjon for implementører.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Kapittel 5 og 6 dekker hash-funksjoner og deres legitime og illegitime bruksområder.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Praktisk eksempel på bruk av SHA-256 for å lenke sammen blokker i en struktur som er uforanderlig ved konstruksjon.
- Forordning (EU) 910/2014 (eIDAS) — rammeverk for kvalifiserte tidsstemplingstjenester. SHA-256 er referansefunksjonen for kvalifiserte elektroniske signaturer og segl utstedt i EU.
- Referanseimplementering i Zig: `std.crypto.hash.sha2.Sha256` i språkets offisielle repositorium (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). Dette er den optimaliserte og auditerte versjonen som faktisk brukes av Solo2. Nyttig for å sammenligne med den pedagogiske implementeringen i appendikset.

← ForrigeCUADERNOS_LIST_SCHREMS_TITLENeste →CUADERNOS_LIST_KILLSWITCH_TITLE

Siste lesninger

- CUADERNOS_LIST_PREGUNTAS_TITLE
- CUADERNOS_LIST_SELFHOST_TITLE
- CUADERNOS_LIST_IDENTIDAD_TITLE

Ta med deg denne artikkelen dit du trenger den.

↓ Markdown ↓ Klartekst ↓ PDF

Filen lastes ned til enheten din. Derfra kan du lagre den, importere den til Solo2 eller dele den hvor du vil. Cuadernos bestemmer ikke destinasjonen for deg.

Lakksegl · SHA-256 2e0a06c2f5630a26cfeee937fc7ef2e118ea1dfcbdc564739f3b1b3ae7af75f5

Cuadernos Lacre · En utgivelse fra Menzuri Gestión S.L. ·
skrevet av R.Eugenio · redigert av teamet bak Solo2.