Kontseptsioon · 19. mai 2026

← Eelmine·Järgmine →

Mis SHA-256 tegelikult on

Matemaatiline sõrmejälg, mis mahub kuuekümne nelja märgi sisse ja muutub täielikult, kui algteksti üksainus koma liigub. Miks me kutsume seda digitaalseks pitsatiks.

Lihtne idee tehnilise nime taga

Kujutage ette, et on olemas masin, millel on üks pilu ja üks ekraan. Pilusse sisestate teksti: sõna, lause, terve romaani. Ekraanile ilmub hetk hiljem täpselt kuuekümne nelja märgi pikkune jada. Seda jada kutsume professionaalse lugeja jaoks *hash*-iks või *krüptograafiliseks kokkuvõtteks*; üldlugeja jaoks võime seda praegu nimetada teksti matemaatiliseks sõrmejäljeks, nagu sõrmejälg on inimese jaoks.

Kui sisestate sama teksti kaks korda, näitab masin mõlemal korral sama sõrmejälge. Kui sisestate veidi erineva teksti — üksainus liigutatud koma, suurtäht, mis muutub väiketäheks — näitab masin esimesest täiesti erinevat sõrmejälge. Mitte sarnast: erinevat. Need kaks omadust koos — determinism ja tundlikkus — ongi see lihtne idee. Kõik muu SHA-256 juures on masinavärk, mis paneb need hästi toimima.

Alguses tasub öelda, mida masin ei tee. See ei krüpteeri teksti. See ei peida seda. See ei salvesta seda. Masin vaatab teksti, arvutab sõrmejälje ja unustab teksti. Sõrmejälg ei võimalda taastada teksti, mis selle tekitas; see võimaldab ainult kontrollida, kas kandidaattekst ühtib originaaliga või mitte. Seetõttu ütleme, et see on *ühesuunaline* kokkuvõte: see läheb välja, aga ei tule tagasi.

Hash ei ole sama mis krüpteerimine

Segadus on sagedane ja seda tasub selgitada: krüpteerimine ja hashimine on erinevad toimingud. Krüpteerimine tähendab teksti teisendamist nii, et ainult võtme valdaja saab selle algkujule tagasi viia. Hashimine tähendab tekstist sõrmejälje loomist, millest algteksti ei saa kunagi taastada, ei võtmega ega ilma. Esimene on disaini poolest pööratav; teine disaini poolest pöördumatu.

Praktiline tagajärg on oluline. Kui rakendus ütleb: „Salvestame teie parooli krüpteeritult“, on kellelgi olemas võti selle lahtikrüpteerimiseks — igal juhul rakendusel endal. Kui rakendus ütleb: „Salvestame teie parooli hashitult“, ei saa rakendus ise algset parooli lugeda isegi siis, kui ta seda tahaks; ta saab ainult kontrollida, kas teie sisestatud parool tekitab uuesti sama sõrmejälje. Teine mudel, kui see on hästi tehtud, on paroolide hoidmiseks esimesest palju eelistatavam. Hiljem näeme, miks „hästi tehtud“ nõuab midagi enamat kui lihtsalt puhast SHA-256.

Neli omadust, mis teevad krüptograafilise hashi kasulikuks

Hash-funktsioon, mis väärib omadussõna *krüptograafiline*, vastab neljale omadusele:

1. **Determinism.** Sama sisend annab alati sama sõrmejälje.
2. **Laviiniefekt.** Väike muutus sisendis tekitab täiesti erineva sõrmejälje, millel pole eelmise sarnasust.
3. **Pöördkujutise resistentsus.** Antud sõrmejälje põhjal ei ole arvutuslikult teostatav leida teksti, mis selle tekitas.
4. **Kollisiooniresistentsus.** Ei ole arvutuslikult teostatav leida kahte erinevat teksti, mis annaksid sama sõrmejälje.

„Ei ole arvutuslikult teostatav“ ei tähenda „on matemaatiliselt võimatu“. See tähendab, et selle saavutamise aja-, energia- ja rahaline kulu ületab suurusjärkude võrra kogu mõistlikult kättesaadava arvutusvõimsuse summa. SHA-256 puhul mõõdetakse seda piiri kvadriljonites aastates isegi kõige optimistlikumate stsenaariumide korral spetsiaalse riistvaraga. Mis on lugeja praktilistel eesmärkidel sama mis „ei saa“.

SHA-256 konkreetsemalt

Nimi ütleb kõik. SHA on lühend sõnadest *Secure Hash Algorithm* (turvaline hash-algoritm). Number 256 tähistab sõrmejälje suurust bittides: kakssada viiskümmend kuus bitti ehk kolmkümmend kaks baiti, mis kuueteistkümnendsüsteemis on need kuuskümmend neli märki, mida lugeja juba tunneb. Standardi avaldas USA NIST (asutus, mis seda tüüpi funktsioone normaliseerib) 2001. aastal osana SHA-2 perekonnast; standardi praegune versioon FIPS 180-4 on aastast 2015.

Neile, kellel pole veel silme ees, mis on bitid ja baidid:

```
1 bitt    →  0 või 1        (lüliti: sees või väljas)
1 bait    →  8 bitti        (256 võimalikku kombinatsiooni)
32 baiti  →  256 bitti      (SHA-256 sõrmejälg)
```

Number 256 nime lõpus tähistab sõrmejälje suurust bittides. Kuueteistkümnendsüsteemis — numeratsioonisüsteem, kus on kümne sümboli asemel kuusteist — mahuvad need 256 bitti täpselt 64 märki. Need on need 64 märki, mida näete iga Cuaderno allosas.

Mõõtmed väärivad hetke. Kakssada viiskümmend kuus bitti võimaldavad kaks astmel kakssada viiskümmend kuus erinevat väärtust: seitsmekümne kaheksa kümnendkohaga arv, mis on mitu suurusjärku suurem kui hinnanguline aatomite arv vaadeldavas universumis. Iga tekst maailmas — iga raamat, iga e-kiri, iga sõnum — langeb ühele neist väärtustest. Tõenäosus, et kaks erinevat teksti juhuslikult kokku langevad, on praktilistel eesmärkidel eristamatu nullist.

Kuidas see koodis välja näeb

Zig-is, keeles, milles kirjutame Solo2-d toetavad osad, näeb teksti SHA-256 pitsati arvutamine välja järgmine:

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

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

Palusime just Zig-i standardteegil arvutada jutumärkides oleva teksti SHA-256. Pärast väljakutset sisaldab muutuja *resumen* kolmekümmet kahte baiti, mis moodustavad pitsati selle toorel kujul; kui need kuvatakse ekraanil kuueteistkümnendsüsteemis, on need kuuskümmend neli märki, mis ilmuvad selle artikli allosas. Kui muudaksime *Cuadernos Lacre* nimeks *Cuadernos lacre* — üks suurtäht vähem —, muutuks kogu pitsat. See on viies reas keskne omadus, mis toetab ülejäänut. Neile, kes soovivad näha, kuidas see sisemiselt töötab, lisasime artikli lõppu algoritmi loetava versiooni koos samm-sammuliste kommentaaridega.

Miks me kutsume seda lakipitsatiks

15.–19. sajandi Euroopa kirjavahetuses suleti kiri lakiga (pitsativahaga). Tilk sulatatud vaha, sellele vajutatud pitsat ja kiri jäi märgistatuks kordumatul viisil. See ei kaitsnud sisu kindlameelse piiluja eest — paberit sai lugeda vastu valgust, lakki sai murda —, kuid see tõendas sisu. Igasugune sulgemise muutmine oli saajale nähtav juba enne paberi avamist. Lakk ei hoidnud ära kahju; see deklareeris seda.

Iga Cuaderno sisu SHA-256 täidab oma digitaalses versioonis sama funktsiooni. Kui artikli üksainus sõna muutuks selle avaldamise hetke ja teie lugemise hetke vahel, ei ühtiks teksti allosas olev kuueteistkümnendsüsteemis pitsat enam teie ees oleva teksti SHA-256-ga. Iga lugeja saaks seda viie koodireaga kontrollida. Väljaanne ei saa oma ajalugu ümber kirjutada ilma, et pitsat seda reedaks. See ei kaitse kahju eest; see teeb selle kontrollitavaks.

Mis hash ei ole

SHA-256-lt oodatakse mõnikord nelja kasutusviisi, mis talle ei kuulu:

1. **Krüpteerimine.** Hash teeb kokkuvõtte, mitte ei peida. Kui soovite, et tekst poleks loetav, peate selle krüpteerima, mitte hashima.

2. **Autori autentimine.** Hash ei ütle, kes teksti kirjutas, vaid ainult seda, millist teksti hashiti. Autorsuse seostamiseks on vaja hashi peale krüptograafilist allkirja, mitte ainult hashi.
3. **Paroolide salvestamine.** Siin on lõks, mida tasub mõista. SHA-256 on loodud olema väga kiire — mis on paljude asjade jaoks hea, kuid selle jaoks halb. Spetsiaalse riistvaraga ründaja saab proovida miljardeid paroole sekundis SHA-256 hashi vastu, kuni leiab teie oma. Paroolide salvestamiseks tuleb kasutada teadlikult aeglaseid võtme tuletamise funktsioone nagu Argon2, scrypt või bcrypt, kombineerituna *soolaga* (unikaalne juhuslik andmetükk kasutaja kohta, mis takistab kahel sama parooliga inimesel sama hashi omamast).
4. **Hashi lugemine autori identifikaatorina.** See ei ole nii. Hash identifitseerib sisu. Kui kaks inimest hashivad sõna *hola* (tere) SHA-256-ga, saavad mõlemad sama kokkuvõtte — ja see on keskne omadus, mitte viga: kui need oleksid erinevad kokkuvõtted, ei saaks me kontrollida avaldatu ja vastuvõetu kokkulangevust.

Kus SHA-256 ilmub teie igapäevaelus

Kuigi te seda ei näe, toetab SHA-256 suurt osa sellest, mida te internetis igapäevaselt kasutate. Bitcoini plokiahel on üles ehitatud iga ploki SHA-256 aheldamisel järgmise külge; mineviku ploki muutmine sunnib kogu järgneva ahela ümber arvutama. Git, süsteem, millega versioonitakse poolt maailma koodi, tuvastab iga commiti selle täieliku sisu SHA-256 (hiljutistes versioonides) või selle eelkäija SHA-1 (vanemates versioonides) järgi. HTTPS-sertifikaatidel, mis kontrollivad veebisaidi identiteeti, kui sinna sisenete, on kaasas SHA-256 sõrmejälg. Tarkvara allalaadimistega kaasneb sageli arendaja avaldatud SHA-256, et saaksite kontrollida, kas faili pole teel muudetud. Ja nagu me ütlesime, iga Cuaderno Lacre allosas.

Professionaalsele lugejale

Neli operatiivset meeldetuletust neile, kas otsustavad süsteemide üle või auditeerivad neid:

1. Hash ei ole krüpteerimine. Kui tarnija ajab need kaks terminit oma tehnilises dokumentatsioonis segamini, tasub küsida, mida ta täpselt mõtleb.
2. Paroolide salvestamiseks ei tohi kunagi kasutada puhast SHA-256. SHA-256 on selle ülesande jaoks liiga kiire (vt punkt 3 jaotisest *Mis hash ei ole*). Praegune standard on **Argon2id**: disainilt aeglane, konfigureeritav vastavalt serveri võimekusele, kombineeritud unikaalse juhusliku *soolaga* kasutaja kohta.
3. Dokumentide — lepingute, toimikute, failide — tervikluse tagamiseks on SHA-256 šealgi standardne etalon. Seda kasutavad EL-i kvalifitseeritud ajatempli teenuse osutajad.
4. Pikaajaliseks säilitamiseks (aastakümned) tasub lisaks SHA-256-le arvutada ja arhiveerida ka SHA-3 või SHA-512; krüptograafiline ettevaatus soovitab sajanditepikkuste arhiivide puhul mitte toetuda ainult ühele funktsioonile.

Tehniliselt tuntakse seda itereeritud struktuuri — kus vaheolek säilitatakse sisendplokkide vahel — **Merkle-Damgårdi** konstruktsioonina. See on muster, millel põhinevad SHA-1, SHA-2 (sealhulgas SHA-256) ja paljud teised klassikalised räšifunktsioonid. SHA-3 seevastu loobub Merkle-Damgårdist teistsuguse arhitektuuri kasuks, mida nimetatakse *käsnaks* (sponge).

Kuidas SHA-256 töötab, samm-sammult, lihtsate sõnadega

Kujuta ette, et oled kokku pannud maailma kõige keerulisema dominoraja: tuhanded kivid, kümned hargnemised, mehaanilised sillad ja kaldteed üle terve toa, hoolikalt tükkhaaval paika pandud.

Kui sa puudutad esimest kivi, kukub ahel täpses ja korratavas järjestuses. Sama asetus, sama algne puudutus → identne lõplik mahakukkunud kivide muster, ikka ja jälle.

Siin is huvitav osa: liiguta **ainult ühte kivi** pool sentimeetrit ühele poole enne alustamist ja puuduta uuesti. Kaldtee, mis pidanuks aktiveeruma, jääb paigale, sild ei kuku, käivitub teine hargnemine. Lõplik muster põrandal on esimesega võrreldes täiesti tundmatu.

SHA-256 on matemaatiliselt seesama rada. Sinu kirjutatud tekst on kivide algpositsioon. Algoritm on puudutus, mis vallandab laviini. Ja lõpptulemus — mida me nimetame *räsiks* (hash) — on seisufoto põrandast, kui kõik on peatunud. Muuda algses tekstis kasvõi ühte koma ja foto on drastiliselt erinev. Nii lihtne ja samas nii drastiline see ongi.

Samm 1. Teksti tõlkimine binaarseteks kivideks. Arvutid ei mõista tähti; nad tõlgivad need esmalt numbriteks (ASCII) ja numbrid binaarkoodi (ühed ja nullid). Igast tähest saab 8 valget või musta kivi: *A* on `01000001`, *B* on `01000010`,

tühik on `00100000`. Kogu sinu tekst — sõna, leping, romaan — muutub pikaks valgete ja mustade kivide reaks.

Samm 2. Täitmine standardmõõduni. Rada töötleb rida täpselt 512-kiviliste **lõikudena**. Kui sinu sõnum ei ulatu 512 kordseni, lisatakse kohe pärast teksti märgistuskivi (väärtusega `10000000`) ja seejärel nullid, kuni lõik on täis. Iga lõigu viimased 64 positsiooni reserveeritakse teksti algse pikkuse märkimiseks. Nii teab rada alati, kus lõppes tegelik sisu ja kust algas täide.

Samm 3. Kaheksa juhtkivi asetamine. Enne alustamist asetame lauale **kaheksa juhtkivi** täpsesse algpositsiooni. Need kaheksa kivi pole mingi saladus: nende algväärtus on paika pandud avaliku matemaatilise reegliga (kaheksa esimese algarvu — 2, 3, 5, 7, 11, 13, 17, 19 — ruutjuured ja iga juure murdosa esimesed bitid). Kõik igas maailma nurgas alustavad samade kaheksa juhtkiviga samas asendis. Nende saatus on saada laviini poolt tõugatud ja muudetud.

Samm 4. Suur laviin: kuuskümmend neli tõukeringi. Siit algab etendus. Sinu teksti esimene 512-kiviline lõik põrkub kokku kaheksa juhtkiviga. Kuid nad ei kuku korraga: mehhanism sooritab **kuuskümmend neli järjestikust ringi**. Igas ringis tehakse kividega kolm operatsiooni:

- **Karussell** (rotatsioon). Kivid liiguvad ringis: parempoolsed lähevad vasakule. Ühtegi kivi ei kao ega lisandu; nad lihtsalt paigutuvad ümber, tehes karussellil täistiiru. See on odav ja pööratav viis teabe ümberjaotamiseks.
- **Loogiline lehter** (XOR). Kivid läbivad lehtri, mis võrdleb neid paarikaupa: kui mõlemad on sama värvi, väljub valge kivi; kui nad on erinevad, väljub must. See on binaarloogika lihtsaim operatsioon, kuid kombineerituna karusselli rotatsioonidega muutub see ülimalt võimsaks teabe segamiseks ilma seda kaotamata.
- **Ületäitumine** (modulaarne liitmine). Tulemus liidetakse *konstantse tõukekiviga*, mis on võetud kuuekümne nelja konstandi avalikust loendist (esimese kuuekümne nelja algarvu kuupjuured). Kui liitmine tekitab lisakive, mis ei mahu ettenähtud 32 kivi suurusele alale, siis need ülejäävad kivid visatakse ära. Laual on ruumi vaid 32 kivile, mitte ühelegi enam.

Kuuekümne neljanda ringi lõpuks on iga kivi sinu teksti lõigust mõjutanud kaheksa juhtkivi asendit. Tõuke energia on läbinud kogu raja.

Samm 5. Järgmise lõigu lisamine (ilma lähtestamata). Kui sinu tekst oli pikk ja töötlemiseks on veel üks 512-kiviline lõik, siis **rada ei lähtestata**. Kaheksa juhtkivi jäävad täpselt selliseks, nagu esimene laviin nad jättis, ja teine lõik paisatakse nende vastu, et aktiveeriva veel kuuskümmend neli ringi. See on nagu uue dominode täis toa lisamine äsja mahakukkunud toa lõppu: esimese segadus tingib täielikult selle, kuidas teine kukub.

Samm 6. Lõpliku foto tegemine. Kui enam lõike proovida pole, laviin peatub. Vaatame lõppasendit, kuhu kaheksa juhtkivi on jäänud. Tõlgime nende seisu kuueteistkümnendsüsteemi tähtede ja numbrite koodiks. Tulemuseks on täpselt kuuekümne nelja märgi pikkune jada: see on sinu SHA-256 pitser.

Raja ülesehitusest tulenevad iseenesest neli omadust:

1. **Determinism.** Sama tekst tekitab alati sama lõppfoto igas maailma arvutis. Null juhuslikkust, null üllatust.
2. **Laviini efekt.** Lisatud koma, muudetud suurtäht, unustatud täppitäht: foto osutub täiesti tundmatuks. See ongi see äärmuslik tundlikkus, mida kirjeldasime alguses.
3. **Ühesuunalisus.** Arvestades lõppfotot, ei saa sa taastada algset teksti. Rotatsioonid, lehtrid ja ületäitumised hävitavad kogu suunalise teabe selle kohta, *kust iga bit tuli*, ja säilitavad vaid selle, *mis kokku liideti*.
4. **Kollisioonikindlus.** Viiekümne aasta pikkuse avaliku krüptoanalüüsi jooksul pole kellelgi õnnestunud leida kahte erinevat teksti, mille lõppfotod ühtiksid. Ja selle tegemise keerukus ületab igasuguse mõistlikult ettekujutatava tsivilisatsiooni arvutusvõimsuse.

Järgnev koodilisa rakendab täpselt need kuus sammu Zigis. Nüüd saad seda lugeda, teades, mida iga bititehe tähendab, selle asemel et manipuleerimisi pimesi aktsepteerida.

Tehniline sõnastik

Lugejale, kes soovib mõista, mida iga operatsioon teeb. Võid selle vabalt vahele jätta: artiklist saab aru ka ilma selleta.

ASCII ja Unicode — kuidas tähtedest saavad numbrid. Arvutid ei näe tähti; nad näevad numbreid. Standard nimega **ASCII** (*American Standard Code for Information Interchange*, aastast 1963) määrab igale klaviatuuri märgile konkreetse numbri: *A* on 65, *B* on 66, *a* on 97, *0* on 48, tühik on 32, koma on 44. Kaasaegsed süsteemid laiendavad seda **Unicode**'iga,

mis määrab numbri igale märgile igas maailma tähestikus: kirillitsas, araabia, hiina, jaapani ja isegi emotikonidele. Kui trükid märgi või avad tekstifaili, loeb arvuti selle taga olevat numbrit, mitte ekraanil olevat kuju. SHA-256 töötab nende numbritega, käsitledes mis tahes teksti pika numbrite jadana. Seetõttu saab ta sama algoritmiga pitserdada hispaaniakeelse artikli, jaapanikeelse luuletuse ja binaarfaili.

XOR — bitiviisiline võrdleja. XOR (häälдatakse *«ex-or»*, ingliskeelsest terminist *exclusive or*, välistav või) on üks lihtsamaid tehteid, mida arvuti saab kahe binaararvuga teha. See võrdleb kahte bitti positsiooni kaupa ja tagastab: **1**, kui täpselt üks neist kahest on 1 (üks, aga mitte mõlemad), **0**, kui mõlemad on samad (mõlemad 0 või mõlemad 1). Näide: `1010` ja `1100` XOR on `0110`. Sellel on märkimisväärne omadus: see on pööratav — kui teed sama võtmega kaks korda XOR-i, jõuad tagasi algse variandini. Seetõttu on see krüptograafia tööloom: see segab mitte ilma teavet kaotamata, mas el resultado no revela nada sobre las entradas si no conoces una de ellas.

Kuueteistkümnendsüsteem — loendamine alusel 16. Peaaegu kõik igapäevased numbrid kasutavad kümmet numbrit (0-9). Kuueteistkümnendsüsteem ehk heksadetsimaalsüsteem kasutab kuueteistkümmet: tavalised 0-9 pluss kuus tähte, mis tähistavad järgmisi väärtusi: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Miks kuueteistkümmend? Sest arvutid mõtlevad nelja biti kaupa ja neli bitti saavad esindada täpselt kuueteistkümmet erinevat väärtust — seega vastab üks heksamärk puhtalt neljale bitile. SHA-256 sõrmejälg on 256 bitti pikk, mis on täpselt **64 heksamärki**. Kui kirjutaksime selle tavalises kümnendsüsteemis, võtaks see umbes 78 numbrit ja oleks ebamugavam. Valik on esteetiline ja kompaktne; selle taga olev number on sama.

Bititõste ehk rotatsioon — binaarne karussell. Kujuta ette rida seitsmest lambipirnist, millest mõned põlevad (1) ja teised on kustus (0): `1 0 1 1 0 0 1`. Rotatsioon paremale ühe positsiooni võrra tähendab parempoolseima pirni võtmist, selle viimist vasakusse otsa ja teiste nihutamist ühe koha võrra paremale: `1 1 0 1 1 0 0`. Ühtegi pirni ei kao ega lisandu: nad lihtsalt tantsivad ringis. SHA-256 kasutab igas arvutuses bititõstet sadu kordi; see on odav ja kadudeta viis teabe ümberjaotamiseks oleku sees.

***«Nothing-up-my-sleeve»* konstandid — miks nad pärinevad algarvudest.** SHA-256 kaheksat juhtkivi ja kuutkümmet nelja ringikonstanti ei valitud juhuslikult. Need pärinevad esimeste algarvude ruut- ja kuupjuurtest. Miks? Sest nende loojad tahtsid **«ilma peidetud trikkideta»** (nothing-up-my-sleeve) konstante: väärtusi, mille päritolu saab igaüks kontrollida. Kui keegi ütleb sulle *«usalda mind: kasuta seda 32-bitist suvalist numbrit»*, kahtlustaksid sa põhjendatult peidetud nõrkust või tagaust. Kuid igaüks, kellel on kalkulaator, saab kontrollida, et ruutjuurest 2 esimesed 32 bitti on `0x6a09e667`. Väärtused on matemaatilised, avalikud ja korratavad: ühtegi peidetud lõksu ei saa retsepti sisse hiilida.

Lisa: SHA-256 loetavas koodis

See lisa on mõeldud lugejale, kes soovib näha algoritmi seestpoolt. See on didaktiline Zig-i implementatsioon, mis järgib FIPS 180-4 spetsifikatsiooni. See ei ole Solo2 kasutatav versioon — pärisversioon asub Zig-i standardteegis `std.crypto.hash.sha2.Sha256`, mis on optimeeritud ja auditeeritud. Kuid algoritm on sama: see, mida siin näete, on samm-sammult see, mis juhtub, kui see viiemärgiline väljakutse oma tööd teeb.

```
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
}
```

Igasugune ümberkirjutamine mõnes muus keeles, mis järgib sama struktuuri — algkonstandid, graafiku laiendamine, kuuskümmend neli vooru, akumulatsioon — annab sama tulemuse. Algoritmil pole saladusi: selle väärtus seisneb selles, et eespool loetletud omadused peavad paika ka pärast kahte aastakümmet kestnud avalikku krüptoanalüüsi tuhandete silmade all.

Kui lähete tagasi selle artikli allossa, näete kuuekümne nelja märgi pikkust kuueteistkümnendsüsteemis pitsatit. See on äsja loetud teksti SHA-256 selles keeles. Kui tõlgiksime artikli, oleks pitsat teistsugune; kui eestikeelses versioonis muutuks üks sõna, muutuks eesti pitsat. Pitsat ei kaitse sisu — selleks on teised tööriistad —, vaid identifitseerib selle üheselt. Ja sellest, kui tagasihoidlikult see ka ei kõlaks, piisab, et toimetuse ahela ükski samm ei saaks öeldut muuta ilma, et see välja tuleks. Kõik muu — krüpteerimine, allkirjastamine, identifitseerimine — on ehitatud selle lihtsa idee peale.

Allikad ja täiendav lugemine

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, august 2015. SHA-2 perekonna, sealhulgas SHA-256, ametlik spetsifikatsioon.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, mai 2011. Normatiivne versioon rakendajatele.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Peatükid 5 ja 6 käsitlevad hash-funktsioone ning nende õiguspäraseid ja väärkasutusi.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Praktiline näide SHA-256 kasutamisest plokkide aheldamiseks muutumatus struktuuris.
- Määrus (EL) 910/2014 (eIDAS) — kvalifitseeritud ajatemplite raamistik. SHA-256 on EL-is väljastatavate kvalifitseeritud e-allkirjade ja -pitsatite etalonfunktsioon.

- Zig-i viiteimplementatsioon: `std.crypto.hash.sha2.Sha256` keele ametlikus hoidlas (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). See on optimeeritud ja auditeeritud versioon, mida Solo2 tegelikult kasutab. Kasulik lisa didaktilise implementatsiooniga võrdlemiseks.

← EelmineCUADERNOS_LIST_SCHREMS_TITLEJärgmine →CUADERNOS_LIST_KILLSWITCH_TITLE

Viimased lugemised

- CUADERNOS_LIST_PREGUNTAS_TITLE
- CUADERNOS_LIST_SELFHOST_TITLE
- CUADERNOS_LIST_IDENTIDAD_TITLE

Võtke see artikkel endaga kaasa kuhu iganes vaja.

↓ Markdown ↓ Lihttekst ↓ PDF

Fail laaditakse alla teie seadmesse. Sealt saate selle salvestada, importida Solo2-sse või jagada kus iganes soovite. Cuadernos ei otsusta sihtkohta teie eest.

Lakipitsat · SHA-256 34a5437ba1b29c3182dfd1c3a3bf1d9d15accc7e7ca76a836c4e9fc55d52c00c

Cuadernos Lacre · Ettevõtte Menzuri Gestión S.L. väljaanne ·
kirjutanud R.Eugenio · toimetanud Solo2 meeskond.