Mitä SHA-256 todellisuudessa on

Matemaattinen sormenjälki, joka mahtuu kuuteenkymmeneen neljään merkkiin ja muuttuu kokonaan, jos yksikin pilkku alkuperäisessä tekstissä liikkuu. Miksi kutsumme sitä digitaaliseksi sinetiksi.

Yksinkertainen idea teknisen nimen takana

Kuvittele, että on olemassa kone, jossa on vain yksi aukko ja yksi näyttö. Aukosta syötät tekstin: sanan, lauseen tai kokonaisen romaanin. Näytölle ilmestyy hetkeä myöhemmin täsmälleen kuudenkymmenenneljän merkin pituinen jono. Tätä jonoa kutsumme ammattilaislukijalle *hashiksi* tai *kryptografiseksi tiivisteeksi*; tavalliselle lukijalle voimme kutsua sitä toistaiseksi tekstin matemaattiseksi sormenjäljeksi, aivan kuten sormenjälki on ihmiselle.

Jos syötät saman tekstin kahdesti, kone näyttää saman sormenjäljen molemmilla kerroilla. Jos syötät hieman erilaisen tekstin — yhden siirretyn pilkun tai ison kirjaimen, joka muuttuu pieneksi — kone näyttää sormenjäljen, joka on täysin erilainen kuin ensimmäinen. Ei samanlainen: erilainen. Nämä kaksi ominaisuutta yhdessä — determinismi ja herkkyys — ovat se yksinkertainen idea. Kaikki muu SHA-256:ssa on koneistoa, joka saa ne toimimaan hyvin.

On syytä sanoa alusta alkaen, mitä kone ei tee. Se ei salaa tekstiä. Se ei piilota sitä. Se ei tallenna sitä. Kone katsoo tekstiä, laskee sormenjäljen ja unohtaa tekstin. Sormenjälki ei mahdollista sen tuottaneen tekstin palauttamista; se mahdollistaa vain ehdokastekstin tarkistamisen, vastaako se alkuperäistä vai ei. Siksi sanomme, että se on *yksisuuntainen* tiiviste: se menee, mutta ei tule takaisin.

Hash ei ole sama asia kuin salaaminen

Sekaannus on yleinen ja se on syytä selvittää: salaaminen ja hashaus ovat eri operaatioita. Salaaminen tarkoittaa tekstin muuntamista niin, että vain avaimen haltija voi palauttaa sen alkuperäiseen muotoonsa. Hashaus tarkoittaa tekstistä sormenjäljen luomista, josta alkuperäistä tekstiä ei voida koskaan palauttaa, ei avaimella eikä ilman. Ensimmäinen on suunnittelultaan palautuva; toinen suunnittelultaan peruuttamaton.

Käytännön seurauksella on väliä. Kun sovellus sanoo: "Tallennamme salasanasi salattuna", on joku, jolla on avain sen avaamiseen — sovellus itse, joka tapauksessa. Kun sovellus sanoo: "Tallennamme salasanasi hashina", sovellus itse ei voi lukea alkuperäistä salasanaa, vaikka se haluaisi; se voi vain tarkistaa, tuottaako kirjoittamasi salasana saman sormenjäljen uudelleen. Jälkimmäinen malli, hyvin tehtynä, on paljon parempi salasanasäilytykseen kuin ensimmäinen. Myöhemmin näemme, miksi "hyvin tehtynä" vaatii muutakin kuin pelkän SHA-256:n.

Neljä ominaisuutta, jotka tekevät kryptografisesta hashista hyödyllisen

Hash-funktio, joka ansaitsee adjektiivin *kryptografinen*, täyttää neljä ominaisuutta:

1. **Determinismi.** Sama syöte tuottaa aina saman sormenjäljen.
2. **Lumivyöryilmiö (Avalanche effect).** Pieni muutos syötteessä tuottaa täysin erilaisen sormenjäljen, jolla ei ole näkyvää yhdennäköisyyttä edelliseen.
3. **Käänteisen kuvion vastustuskyky.** Annetun sormenjäljen perusteella ei ole laskennallisesti mahdollista löytää tekstiä, joka sen tuotti.
4. **Törmäyksen vastustuskyky.** Ei ole laskennallisesti mahdollista löytää kahta eri tekstiä, jotka tuottaisivat saman sormenjäljen.

"Ei laskennallisesti mahdollista" ei tarkoita "matemaattisesti mahdotonta". Se tarkoittaa, että sen saavuttamisen aika-, energia- ja rahakustannukset ylittävät moninkertaisesti kaiken kohtuudella saatavilla olevan laskentakapasiteetin summan. SHA-256:lle tämä raja mitataan kvadriljoonissa vuosissa jopa optimistisimmissa skenaarioissa erikoistuneella laitteistolla. Mikä on lukijan käytännön tarkoituksiin sama kuin "ei voida tehdä".

SHA-256, erityisesti

Nimi kertoo kaiken. SHA on lyhenne sanoista *Secure Hash Algorithm* (turvallinen hash-algoritmi). Numero 256 osoittaa sormenjäljen koon bitteinä: kaksisataa viisikymmentäkuusi bittiä eli kolmekymmentäkaksi tavua, jotka heksadesimaalimuodossa ovat ne kuusikymmentäneljä merkkiä, jotka lukija jo tunnistaa. Standardin julkaisi yhdysvaltalainen NIST (elin, joka normalisoi tällaisia funktioita) vuonna 2001 osana SHA-2-perhettä; standardin nykyinen versio FIPS 180-4 on vuodelta 2015.

Niille, joilla ei vielä ole mielessä, mitä bitit ja tavut ovat:

```
1 bitti   →  0 tai 1        (kytkin: päällä tai pois)
1 tavu    →  8 bittiä       (256 mahdollista yhdistelmää)
32 tavua  →  256 bittiä     (SHA-256-sormenjälki)
```

Nimen lopussa oleva numero 256 kertoo sormenjäljen koon bitteinä. Heksadesimaalijärjestelmässä — numerointijärjestelmässä, jossa on kymmenen symbolin sijasta kuusitoista — nuo 256 bittiä mahtuvat tasan 64 merkkiin. Nämä ovat ne 64 merkkiä, jotka näet kunkin Cuadernon alalaidassa.

Mitat ansaitsevat hetken huomiota. Kaksisataa viisikymmentäkuusi bittiä mahdollistavat kaksi potenssiin kaksisataaviisikymmentäkuusi erilaista arvoa: luku, jossa on seitsemänkymmentäkahdeksan desimaalinumeroa, useita kertaluokkia suurempi kuin arvioitu atomien määrä havaittavassa maailmankaikkeudessa. Jokainen maailman teksti — jokainen kirja, jokainen sähköposti, jokainen viesti — osuu yhteen näistä arvoista. Todennäköisyys sille, että kaksi eri tekstiä osuisivat sattumalta samaan, on käytännön tarkoituksiin nolla.

Miltä se näyttää koodissa

Zig-kielellä, jolla kirjoitamme Solo2:ta tukevat osat, tekstin SHA-256-sinetin laskeminen näyttää tältä:

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

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

Pyysimme juuri Zigin standardikirjastoa laskemaan lainausmerkeissä olevan tekstin SHA-256-tiivisteen. Kutsun jälkeen *resumen*-muuttuja sisältää ne kolmekymmentäkaksi tavua, jotka muodostavat sinetin sen raakamuodossa; kun ne näytetään ruudulla heksadesimaaleina, ne ovat ne kuusikymmentäneljä merkkiä, jotka näkyvät tämän artikkelin alalaidassa. Jos muuttaisimme *Cuadernos Lacre* nimeksi *Cuadernos lacre* — yksi iso kirjain vähemmän — koko sinetti muuttuisi. Tämä on viidellä rivillä se keskeinen ominaisuus, joka tukee muuta. Niille, jotka haluavat nähdä, miten se toimii sisäisesti, olemme lisänneet artikkelin loppuun algoritmin luettavan version vaiheittaisine kommentteineen.

Miksi kutsumme sitä sinetiksi

Eurooppalaisessa kirjeenvaihdossa 1400-luvulta 1800-luvulle lakka (sinettivaha) sulki kirjeen. Pisara sulatettua vahaa, päälle painettu sinetti, ja kirje jäi merkityksi toistamattomalla tavalla. Se ei suojannut sisältöä päättäväiseltä urkkijalta — paperia voitiin lukea valoa vasten, lakka voitiin murtaa — mutta se osoitti sisällön. Mikä tahansa sulkemisen muutos oli vastaanottajan nähtävissä jo ennen paperin avaamista. Lakka ei estänyt vahinkoa; se julisti sen.

Kunkin Cuadernon rungon SHA-256 täyttää saman tehtävän digitaalisessa versiossaan. Jos artikkelin yksikin sana muuttuisi julkaisuhetken ja lukuhetkesi välillä, tekstin alalaidassa oleva heksadesimaalisinetti ei enää vastaisi edessäsi olevan tekstin SHA-256-tiivistettä. Kuka tahansa lukija voisi tarkistaa sen viidellä koodirivillä. Julkaisu ei voi kirjoittaa historiaansa uudelleen ilman, että sinetti paljastaa sen. Se ei suojaa vahingolta; se tekee siitä todennettavan.

Mitä hash ei ole

SHA-256:lta odotetaan joskus neljää käyttötapaa, jotka eivät sille kuulu:

1. **Salaaminen.** Hash tiivistää; se ei piilota. Jos haluat, ettei teksti ole luettavissa, sinun on salattava se, ei hashattava.
2. **Kirjoittajan todentaminen.** Hash ei kerro, kuka tekstin on kirjoittanut, vaan ainoastaan, mikä teksti on hashattu. Kirjoittajuuden yhdistämiseksi tarvitaan kryptografinen allekirjoitus hashin päälle, ei pelkkää hashia.
3. **Salasanojen tallentaminen.** Tässä on ansa, joka on syytä ymmärtää. SHA-256 on suunniteltu erittäin nopeaksi — mikä on hyvä moniin asioihin, mutta huono tähän. Hyökkääjä erikoistuneella laitteistolla voi kokeilla miljardeja salasanoja sekunnissa SHA-256-hashia vastaan, kunnes löytää sinun salasanasi. Salasanojen tallentamiseen on käytettävä tarkoituksella hitaita avaimenjohdannaisfunktioita (Key Derivation Functions), kuten Argon2, scrypt tai bcrypt, yhdistettynä *suolaan* (yksilöllinen satunnainen tieto käyttäjää kohden, joka estää kahta samaa salasanaa käyttävää henkilöä saamasta samaa hashia).
4. **Hashin lukeminen kirjoittajan tunnisteena.** Se ei ole sitä. Hash tunnistaa sisällön. Jos kaksi ihmistä hashaa sanan *hola* (hei) SHA-256:lla, molemmat saavat saman tiivisteen — ja se on keskeinen ominaisuus, ei vika: jos ne olisivat eri tiivisteitä, emme voisi tarkistaa julkaistun ja vastaanotetun vastaavuutta.

Missä SHA-256 esiintyy arjessasi

Vaikka et näe sitä, SHA-256 tukee suurta osaa siitä, mitä käytät päivittäin internetissä. Bitcoinin lohkoketju rakentuu ketjuttamalla kunkin lohkon SHA-256 seuraavaan; menneen lohkon muuttaminen pakottaa laskemaan koko seuraavan ketjun uudelleen. Git, järjestelmä, jolla puolet maailman koodista versioidaan, tunnistaa jokaisen commitin sen täyden sisällön SHA-256-tiivisteellä (uusissa versioissa) tai sen edeltäjällä SHA-1:llä (vanhemmissa versioissa). HTTPS-varmenteilla, jotka todentavat verkkosivuston identiteetin sinne mennessäsi, on mukanaan SHA-256-sormenjälki. Ohjelmistolatausten mukana on usein kehittäjän julkaisema SHA-256, jotta voit tarkistaa, ettei tiedostoa ole muutettu matkan varrella. Ja kuten sanoimme, kunkin Cuaderno Lacren alalaidassa.

Ammattilaislukijalle

Neljä operatiivista muistutusta niille, jotka päättävät järjestelmistä tai auditoivat niitä:

1. Hash ei ole salausta. Jos toimittaja sekoittaa nämä kaksi termiä teknisessä dokumentaatiossaan, on syytä kysyä, mitä hän tarkalleen ottaen tarkoittaa.
2. Salasanojen tallentamiseen ei saa koskaan käyttää pelkkää SHA-256:ta. SHA-256 on liian nopea tähän tehtävään (katso kohta 3 kohdassa *Mitä hash ei ole*). Nykyinen standardi on **Argon2id**: suunnittelultaan hidas, konfiguroitavissa palvelimen kapasiteetin mukaan, yhdistettynä yksilölliseen satunnaiseen *suolaan* käyttäjää kohden.
3. Asiakirjojen — sopimusten, asiakirja-aineistojen, tiedostojen — eheydelle SHA-256 on edelleen viitestandardi. Sitä käyttävät EU:n pätevöidyt aikaleimapalvelun tarjoajat.
4. Pitkäaikaiseen säilytykseen (vuosikymmenet) kannattaa laskea ja arkistoida myös SHA-3 tai SHA-512 SHA-256:n rinnalle; kryptografinen varovaisuus suosittelee olemaan turvautumatta vain yhteen funktioon vuosisataisessa arkistoinnissa.

Teknisesti tätä iteroitua rakennetta — jossa välitila säilytetään syötelohkojen välillä — kutsutaan **Merkle-Damgård** -rakenteeksi. Se on malli, johon SHA-1, SHA-2 (mukaan lukien SHA-256) ja monet muut perinteiset tiiviste- eli hash-funktiot perustuvat. SHA-3 sen sijaan hylkää Merkle-Damgårdin ja käyttää erilaista arkkitehtuuria nimeltään *sieni* (sponge).

Miten SHA-256 toimii, vaihe vaiheelta, selkokielellä

Kuvittele, että olet rakentanut maailman monimutkaisimman dominoradan: tuhansia palikoita, kymmeniä haaroja, mekaanisia siltoja ja ramppeja, jotka risteilevät huoneen halki, huolellisesti aseteltuina palikka palikalta.

Jos napautat ensimmäistä palikkaa, ketju kaatuu täsmällisessä ja toistettavassa järjestyksessä. Sama asennus, sama aloitusnapautus → identtinen loppukuvio kaatuneista palikoista, kerta toisensa jälkeen.

Tässä on mielenkiintoinen seikka: siirrä **yhtä ainoaa palikkaa** puoli senttimetriä sivulle ennen aloitusta ja napauta uudelleen. Ramppi, jonka piti aktivoitua, jää liikkumattomaksi, silta ei putoa, eri haara laukeaa. Lopullinen palikkakuvio

lattialla on täysin tunnistamaton verrattuna ensimmäiseen.

SHA-256 on matemaattisesti tämä rata. Kirjoittamasi teksti on palikoiden alkuasento. Algoritmi on napautus, joka vapauttaa ketjureaktion. Ja lopputulos — jota kutsumme nimellä *hash* — on lattiasta otettu pysäytyskuva, kun kaikki on pysähtynyt. Muuta yksi ainoa pilkku alkuperäisessä tekstissä, ja kuva on radikaalisti erilainen. Niin yksinkertaista ja niin dramaattista.

Vaihe 1. Tekstin kääntäminen binääripalikoiksi. Tietokoneet eivät ymmärrä kirjaimia; ne kääntävät ne ensin numeroiksi (ASCII) ja numerot binäärimuotoon (ykkösiksi ja nolliksi). Jokainen kirjain muuttuu 8 valkoiseksi tai mustaksi palikaksi: *A* on `01000001`, *B* on `01000010`, välilyönti on `00100000`. Koko tekstistäsi — sanasta, sopimuksesta tai romaanista — tulee pitkä jono valkoisia ja mustia palikoita.

Vaihe 2. Täyttäminen vakiokokoon. Rata käsittelee jonon täsmälleen 512 palikan *lohkoissa*. Jos viestisi ei yllä 512:n monikertaan, lisätään merkkipalikka (arvoltaan `10000000`) heti tekstin jälkeen ja sitten nollia lohkon loppuun asti. Kunkin lohkon viimeiset 64 paikkaa varataan tekstin alkuperäisen pituuden merkitsemiseen. Näin rata tietää aina, mihin todellinen sisältö päättyi ja mistä täyte alkoi.

Vaihe 3. Kahdeksan mestarinappulan asettaminen. Ennen aloitusta asetamme pöydälle **kahdeksan mestarinappulaa** täsmälliseen alkuasentoon. Nämä kahdeksan nappulaa eivät ole salaisuus: niiden alkuarvot on määritetty julkisella matemaattisella säännöllä (kahdeksan ensimmäisen alkuluvun — 2, 3, 5, 7, 11, 13, 17, 19 — neliöjuurten ensimmäiset bitit desimaaliosasta). Kaikki kaikkialla maailmassa aloittavat samalla kahdeksalla mestarinappulalla samassa asennossa. Niiden kohtalona on tulla vyöryn tönimiksi ja muuntamiksi.

Vaihe 4. Suuri vyöry: kuusikymmentäneljä tönäisykierrosta. Tässä esitys alkaa. Tekstisi ensimmäinen 512 palikan lohko törmää kahdeksaan mestarinappulaan. Ne eivät kuitenkaan kaadu kerralla: mekanismi suorittaa **kuusikymmentäneljä peräkkäistä kierrosta**. Jokaisella kierroksella palikoille tehdään kolme operaatiota:

- **Karuselli** (kierto). Palikat liikkuvat ympyrässä: oikealla olevat siirtyvät vasemmalle. Yhtään palikkaa ei häviä eikä lisätä; ne vain järjestetään uudelleen karusellin täydellä kierroksella. Se on halpa ja palautuva tapa jakaa tietoa uudelleen.
- **Looginen suppilo** (XOR). Palikat kulkevat suppilon läpi, joka vertaa niitä pareittain: jos molemmat ovat samanvärisiä, ulos tulee valkoinen; jos ne ovat erivärisiä, ulos tulee musta. Se on binäärilogiikan yksinkertaisin operaatio, mutta yhdistettynä karusellin kiertoihin se muuttuu erittäin tehokkaaksi tiedon sekoittajaksi ilman tiedon menetystä.
- **Ylivuoto** (modulaarinen summaus). Tulos lasketaan yhteen *vakiotönäisypalikan* kanssa, joka on otettu julkisesta kuudenkymmenenneljän vakion listasta (kuudenkymmenenneljän ensimmäisen alkuluvun kuutiojuuret). Jos summa tuottaa ylimääräisiä palikoita, jotka eivät mahdu varattuun 32 palikan tilaan, nämä ylimääräiset palikat hylätään. Pöydällä on tilaa vain 32 palikalle, ei yhdellekään enemmälle.

Kuudenkymmenenneljännen kierroksen lopussa jokainen tekstisi lohkon palikka on vaikuttanut kahdeksan mestarinappulan asentoon. Tönäisyn energia on kulkenut koko radan läpi.

Vaihe 5. Seuraavan lohkon lisääminen (ilman nollausta). Jos tekstisi oli pitkä ja jäljellä on toinen 512 palikan lohko käsiteltäväksi, **rataa ei nollata**. Kahdeksan mestarinappulaa jäävät siihen asentoon, johon ensimmäinen vyöry ne jätti, ja toinen lohko laukaistaan niitä vasten käynnistämään toiset kuusikymmentäneljä kierrosta. Se on kuin lisäisi uuden huoneen täynnä dominoita juuri kaatuneen huoneen perään: ensimmäisen huoneen epäjärjestys määrää täysin, miten toinen huone kaatuu.

Vaihe 6. Lopullisen kuvan ottaminen. Kun käsiteltäviä lohkoja ei ole enää jäljellä, vyöry pysähtyy. Katsomme kahdeksan mestarinappulan lopullista asentoa. Muunnamme niiden kokoonpanon kirjain- ja numerokoodiksi heksadesimaalijärjestelmässä. Tuloksena on täsmälleen kuudenkymmenenneljän merkin pituinen jono: se on SHA-256 -sinettisi.

Radan rakenteesta johtuen neljä ominaisuutta toteutuvat itsestään:

1. **Determinismi.** Sama teksti tuottaa aina saman lopullisen kuvan millä tahansa tietokoneella maailmassa. Nolla satunnaisuutta, nolla yllätystä.
2. **Vyöryilmiö.** Lisätty pilkku, muutettu suuraakkonen, unohdettu tilde: loppukuva on täysin tunnistamaton. Tämä on se äärimmäinen herkkyys, jota kuvasimme alussa.

3. **Yksisuuntaisuus.** Lopullisesta kuvasta ei voi päätellä alkuperäistä tekstiä. Kierrot, suppilot ja ylivuodot tuhoavat kaiken suunnatun tiedon siitä, *mistä kukin bitti tuli*, ja säilyttävät vain tiedon siitä, *mikä oli loppusumma*.
4. **Törmäyskestävyys.** Viidenkolmatta vuoden julkisen kryptoviestinnän analyysin aikana kukaan ei ole onnistunut löytämään kahta eri tekstiä, joiden loppukuvat täsmäisivät. Tämän tekemisen vaikeus on minkä tahansa kohtuudella kuviteltavissa olevan sivilisaation laskentatehon ulottumattomissa.

Seuraava koodiliite toteuttaa täsmälleen nämä kuusi vaihetta Zig-kielellä. Nyt voit lukea sen tietäen, mitä kukin bittioperaatio tarkoittaa, sen sijaan että hyväksyisit käsittelyt sokeasti.

Glosario técnico

Lukijalle, joka haluaa ymmärtää kunkin operaation merkityksen. Voit ohittaa tämän vapaasti: artikkeli on ymmärrettävissä ilman sitäkin.

ASCII ja Unicode — miten kirjaimista tulee numeroita. Tietokoneet eivät näe kirjaimia, vaan numeroita. Standardi nimeltään **ASCII** (*American Standard Code for Information Interchange*, vuodelta 1963) antaa kullekin näppäimistön merkille tietyn numeron: *A* on 65, *B* on 66, *a* on 97, *0* on 48, välilyönti on 32 ja pilkku on 44. Nykyjärjestelmät laajentavat tätä **Unicode**-standardilla, joka antaa numeron maailman jokaisen aakkoston jokaiselle merkille: kyrillisille, arabialaisille, kiinalaisille, japanilaisille ja jopa emojeille. Kun kirjoitat merkin tai avaat tekstitiedoston, tietokone lukee taustalla olevan numeron, ei näytöllä näkyvää muotoa. SHA-256 työskentelee näillä numeroilla ja käsittelee mitä tahansa tekstiä pitkänä lukujonona. Siksi se voi sinetöidä espanjankielisen artikkelin, japaninkielisen runon ja binääritiedoston samalla algoritmilla.
XOR — bitti kerrallaan toimiva vertailija. XOR (lausutaan *"eksor"*, englannin sanoista *exclusive or*, poissulkeva tai) on yksi yksinkertaisimmista operaatioista, joita tietokone voi tehdä kahdella binääriluvulla. Se vertaa kahta bittiä paikka kerrallaan ja palauttaa: **1** jos tasan toinen niistä on 1 (toinen mutta eivät molemmat), ja **0** jos molemmat ovat samoja (molemmat 0 tai molemmat 1). Esimerkki: XOR luvuille `1010` ja `1100` on `0110`. Sillä on huomattava ominaisuus: se on palautuva — jos teet XOR-operaation kahdesti samalla avaimella, palaat alkuperäiseen. Siksi se on kryptografian työjuhta: se sekoittaa bitit tietoa menettämättä, mutta lopputulos ei paljasta mitään syötteistä, ellet tiedä toista niistä.
Heksadesimaali — laskeminen 16-kantaisessa järjestelmässä. Melkein kaikki jokapäiväiset numerot käyttävät kymmentä numeroa (0–9). Heksadesimaalijärjestelmä käyttää kuuttatoista: tavalliset 0–9 sekä kuusi kirjainta, jotka edustavat seuraavia arvoja: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Miksi kuusitoista? Koska tietokoneet ajattelevat neljän bitin ryhmissä, ja neljä bittiä voi edustaa täsmälleen kuuttatoista eri arvoa — näin yksi heksadesimaalimerkki vastaa siististi neljää bittiä. Yksi SHA-256 -tunniste on 256 bittiä pitkä, mikä on täsmälleen **64 heksadesimaalimerkkiä**. Jos kirjoittaisimme sen tavallisina kymmenjärjestelmän numeroina, se veisi noin 78 numeroa ja olisi epämukavampi. Valinta on esteettinen ja kompakti; taustalla oleva luku on sama.
Bittien kierto — binäärinen karuselli. Kuvittele rivi, jossa on seitsemän lamppua, joista osa palaa (1) ja osa on sammuksissa (0): `1 0 1 1 0 0 1`. Kierto oikealle yhdellä paikalla tarkoittaa, että otetaan oikeanpuoleisin lamppu, viedään se vasempaan reunaan ja siirretään muita yksi paikka oikealle: `1 1 0 1 1 0 0`. Yhtään lamppua ei häviä eikä lisätä: ne vain tanssivat ympyrää. SHA-256 käyttää bittien kiertoa satoja kertoja jokaisessa laskutoimituksessa; se on halpa ja häviötön tapa jakaa tietoa uudelleen tilan sisällä.
Vakiot *"nothing-up-my-sleeve"* — miksi ne tulevat alkuluvuista. SHA-256:n kahdeksaa mestarinappulaa ja kuuttakymmentäneljää kierrosvakiota ei valittu sattumanvaraisesti. Ne on johdettu ensimmäisten alkulukujen neliö- ja kuutiojuurista. Miksi? Koska niiden suunnittelijat halusivat vakiot, joissa ei ole *"mitään hihassa"* (engl. *nothing-up-my-sleeve*): arvoja, joiden alkuperän kuka tahansa voi tarkistaa. Jos joku sanoisi *"luota minuun: käytä tätä satunnaista 32-bittistä lukua"*, epäilisit syystä piilotettua heikkoutta tai takaporttia. Mutta kuka tahansa laskimen omistaja voi tarkistaa, että luvun 2 neliöjuuren ensimmäiset 32 bittiä ovat `0x6a09e667`. Arvot ovat matemaattisia, julkisia ja toistettavia: mikään piilotettu ansa ei voi päästä mukaan reseptiin.

Liite: SHA-256 luettavassa koodissa

Tämä liite on lukijalle, joka haluaa nähdä algoritmin sisältäpäin. Se on didaktinen Zig-toteutus, joka noudattaa FIPS 180-4 -määrittelyä. Se ei ole Solo2:n käyttämä versio — varsinainen versio on Zigin standardikirjastossa `std.crypto.hash.sha2.Sha256`, joka on optimoitu ja auditoitu. Mutta algoritmi on sama: se, mitä tässä näet, on vaihe vaiheelta se, mitä tapahtuu, kun tuo viiden merkin kutsu suorittaa tehtävänsä.

```
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
}
```

Mikä tahansa toisella kielellä tehty uudelleenkirjoitus, joka noudattaa samaa rakennetta — alkuperäiset vakiot, aikataulun laajennus (schedule expansion), kuusikymmentäneljä kierrosta, kertyminen — tuottaa saman tuloksen. Algoritmi ei ole salainen: sen arvo perustuu siihen, että edellä luetellut ominaisuudet pitävät edelleen paikkansa kahden vuosikymmenen julkisen kryptanalyysin jälkeen tuhansien silmien alla.

Jos palaat tämän artikkelin alalaitaan, näet kuudenkymmenenneljän merkin heksadesimaalisinetin. Se on juuri lukemasi tekstin SHA-256-tiiviste tällä kielellä. Jos kääntäisimme artikkelin, sinetti olisi toinen; jos suomenkielisessä versiossa muuttuisi yksi sana, suomalainen sinetti muuttuisi. Sinetti ei suojaa sisältöä — sitä varten on muita työkaluja — vaan tunnistaa sen yksiselitteisesti. Ja se, vaikka se kuulostaakin vaatimattomalta, riittää siihen, ettei mikään toimitusketjun vaihe voi muuttaa sanottua ilman, että se huomataan. Kaikki muu — salaaminen, allekirjoittaminen, tunnistaminen — rakentuu tämän yksinkertaisen idean päälle.

Lähteet ja lisälukemista

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, elokuu 2015. SHA-2-perheen virallinen määrittely, mukaan lukien SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, toukokuu 2011. Normatiivinen versio toteuttajille.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Luvut 5 ja 6 käsittelevät hash-funktioita sekä niiden oikeutettua ja oikeudetonta käyttöä.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Käytännön esimerkki SHA-256:n käytöstä lohkojen ketjuttamiseen rakenteessa, joka on muuttumaton jo rakenteeltaan.
- Asetus (EU) 910/2014 (eIDAS) — pätevöityjen aikaleimojen kehys. SHA-256 on viitefunktio EU:ssa myönnettäville pätevöidyille sähköisille allekirjoituksille ja sineteille.
- Viitetoteutus Zigillä: `std.crypto.hash.sha2.Sha256` kielen virallisessa arkistossa (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). Se on optimoitu ja auditoitu versio, jota Solo2 itse asiassa käyttää. Hyödyllinen vertailukohta liitteen didaktiselle toteutukselle.

← EdellinenCUADERNOS_LIST_SCHREMS_TITLESeuraava →CUADERNOS_LIST_KILLSWITCH_TITLE

Viimeaikaiset lukemiset

- CUADERNOS_LIST_PREGUNTAS_TITLE
- CUADERNOS_LIST_SELFHOST_TITLE
- CUADERNOS_LIST_IDENTIDAD_TITLE

Ota tämä artikkeli mukaasi minne tarvitset.

↓ Markdown ↓ Pelkkä teksti ↓ PDF

Tiedosto ladataan laitteellesi. Voit tallentaa sen, tuoda sen Solo2-sovellukseen tai jakaa sen haluamallasi tavalla. Cuadernos ei päätä tiedoston kohtaloa puolestasi.

Sinetti · SHA-256 f64feb69b162cd19ac2f350426d6410088f2175bcaad5e64e90c7a72a2da0891

Cuadernos Lacre · Menzuri Gestión S.L. -julkaisu ·
kirjoittanut R.Eugenio · toimittanut Solo2-tiimi.