Kaj dejansko je SHA-256

Matematični odtis, ki se prilega v štiriinšestdeset znakov in se popolnoma spremeni, če se v izvirnem besedilu premakne ena sama vejica. Zakaj ga imenujemo digitalni voščeni pečat.

Preprosta ideja za tehničnim imenom

Predstavljajte si stroj z eno samo režo in enim zaslonom. Skozi režo vnesete besedilo: besedo, stavek, cel roman. Na zaslonu se trenutek kasneje prikaže zaporedje natanko štiriinšestdesetih znakov. To zaporedje strokovni bralci imenujemo *zgoščevalna vrednost (hash)* ali *kriptografski povzetek*; za splošnega bralca pa ga lahko za zdaj imenujemo matematični odtis besedila, tako kot je prstni odtis človekov odtis.

Če dvakrat vnesete isto besedilo, bo stroj obakrat prikazal enak odtis. Če vnesete nekoliko spremenjeno besedilo — premaknjeno eno vejico, veliko začetnico namesto male — bo stroj prikazal popolnoma drugačen odtis od prvega. Ne podoben: drugačen. Ti dve lastnosti skupaj — determiniranost in občutljivost — sta tista preprosta ideja. Vse ostalo pri SHA-256 so mehanizmi, s katerimi to dvoje dobro deluje.

Na začetku je treba povedati, česa stroj ne počne. Besedila ne šifrira. Ne skriva ga. Ne shranjuje ga. Stroj pogleda besedilo, izračuna odtis in pozabi besedilo. Iz odtisa ni mogoče rekonstruirati besedila, ki ga je ustvarilo; omogoča zgolj preverjanje, ali se besedilo ujema z izvirnikom. Zato pravimo, da je to povzetek v *eni smeri*: gre v eno smer, v drugo ne.

Zgoščevanje ni isto kot šifriranje

Zmeda je pogosta in jo je treba razjasniti: šifriranje in zgoščevanje sta različni operaciji. Šifriranje pomeni preoblikovanje besedila tako, da ga lahko samo imetnik ključa vrne v prvotno obliko. Zgoščevanje pomeni ustvarjanje odtisa besedila, iz katerega izvirnega besedila ni mogoče nikoli več pridobiti, ne s ključem ne brez njega. Prvo je načrtno reverzibilno; drugo pa namerno ireverzibilno.

Praktična posledica je pomembna. Ko aplikacija reče "vaše geslo shranjujemo šifrirano", obstaja nekdo, ki ima ključ za njegovo dešifriranje — to je v vsakem primeru sama aplikacija. Ko aplikacija reče "vaše geslo shranjujemo zgoščeno", sama aplikacija izvirnega gesla ne more prebrati, tudi če bi želela; lahko samo preveri, ali tisto, ki ga vnesete, ustvari enak odtis. Drugi model, če je narejen pravilno, je za shranjevanje gesel veliko primernejši od prvega. Kasneje bomo videli, zakaj "narejen pravilno" zahteva več kot le sam SHA-256.

Štiri lastnosti, zaradi katerih je kriptografski odtis koristen

Zgoščevalna funkcija, ki si zasluži pridevnik *kriptografska*, izpolnjuje štiri lastnosti:

1. **Determiniranost.** Isti vnos vedno ustvari enak odtis.
2. **Učinek plazu.** Majhna sprememba na vhodu ustvari popolnoma drugačen odtis brez kakršne koli vidne podobnosti s prejšnjim.
3. **Odpornost proti inverziji.** Če imamo odtis, računsko ni izvedljivo najti besedila, ki ga je ustvarilo.
4. **Odpornost proti kolizijam.** Računsko ni izvedljivo najti dveh različnih besedil, ki ustvarita enak odtis.

"Računsko ni izvedljivo" ne pomeni "matematično nemogoče". Pomeni, da stroški časa, energije in denarja za dosego tega cilja za več velikostnih razredov presegajo vsoto vseh razumno razpoložljivih računskih zmogljivosti. Pri SHA-256 se ta meja meri v tisočih milijard let, tudi pri najbolj optimističnih pristopih z namensko strojno opremo. Kar pa je za praktične namene bralca isto kot "se ne da".

Konkretno, SHA-256

Že samo ime pove vse. SHA je kratica za *Secure Hash Algorithm*: varni zgoščevalni algoritem. Število 256 označuje velikost odtisa v bitih: dvesto šestinpetdeset bitov, torej dvaintrideset bajtov, kar v heksadecimalnem zapisu predstavlja tistih štiriinšestdeset znakov, ki jih bralec že prepozna. Standard je leta 2001 izdal ameriški NIST, organ, ki standardizira takšne funkcije, kot del družine SHA-2; trenutna veljavna različica standarda, FIPS 180-4, je iz leta 2015.

Za tiste, ki še ne vedo, kaj so biti in bajti:

```
1 bit    →  0 ali 1      (stikalo: vklopljeno ali izklopljeno)
1 bajt   →  8 bitov      (256 možnih kombinacij)
32 bajtov→  256 bitov    (odtis SHA-256)
```

Številka 256 na koncu imena pove velikost odtisa v bitih. V heksadecimalnem sistemu — številskem sistemu s šestnajstimi simboli namesto desetih — se teh 256 bitov prilega natanko v 64 znakov. To je tistih 64 znakov, ki jih vidite na dnu vsake izdaje Cuadernos Lacre.

Mere si zaslužijo nekaj pozornosti. Dvesto šestinpetdeset bitov omogoča dva na dvesto šestinpetdeset različnih vrednosti: število z oseminsedemdesetimi decimalkami, kar je za več velikostnih razredov več kot je ocenjeno število atomov v vidnem vesolju. Vsako besedilo na svetu — vsaka knjiga, vsako elektronsko sporočilo, vsako sporočilo — pade na eno od teh vrednosti. Verjetnost, da se bosta dve različni besedili po naključju ujemali, je za praktične namene nerazpoznavna od ničle.

Kako to izgleda v kodi

V jeziku Zig, v katerem pišemo dele, ki podpirajo Solo2, je izračun pečata SHA-256 besedila videti takole:

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

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

Pravkar smo standardno knjižnico jezika Zig prosili, naj izračuna SHA-256 besedila v narekovajih. Po klicu spremenljivka *resumen* vsebuje dvaintrideset bajtov, ki sestavljajo pečat v njegovi surovi obliki; ko so prikazani na zaslonu v heksadecimalnem zapisu, je to tistih štiriinšestdeset znakov, ki so navedeni na koncu tega članka. Če bi *Cuadernos Lacre* spremenili v *Cuadernos lacre* — z eno veliko začetnico manj — bi se spremenil celoten pečat. To je v petih vrsticah tista osrednja lastnost, ki podpira vse ostalo. Za tiste, ki želijo videti, kako deluje od znotraj, smo na konec članka vključili berljivo različico algoritma s komentarji po korakih.

Zakaj ga imenujemo voščeni pečat

V evropski korespondenci od petnajstega do devetnajstega stoletja je pečatni vosek zapiral pismo. Kaplja stopljenega voska, pritisnjen pečat in pismo je bilo neponovljivo označeno. Vsebine ni zaščitil pred odločnim vohunom — papir je bilo mogoče brati proti svetlobi, vosek zlomiti — je pa poseg naredil očiten. Vsaka sprememba pečata je bila prejemniku vidna še pred odprtjem papirja. Voščeni pečat ni preprečil poškodbe; razglasil jo je.

SHA-256 jedra vsake izdaje Cuadernos opravlja v digitalni različici povsem enako vlogo. Če bi se spremenila samo ena beseda članka od trenutka, ko je bil objavljen, do trenutka, ko ga preberete, se heksadecimalni pečat na dnu besedila ne bi več ujemal s SHA-256 besedila, ki ga imate pred seboj. To bi lahko preveril vsak bralec s petimi vrsticami kode. Publikacija ne more prepisati svoje zgodovine, ne da bi jo pečat izdal. Ne varuje pred poškodbami; poškodbe naredi preverljive.

Kaj zgoščevanje ni

Od SHA-256 se včasih zahtevajo štiri uporabe, ki mu ne pripadajo:

1. **Šifriranje.** Zgoščevanje povzame, ne skrije. Če želite, da je besedilo neberljivo, ga morate šifrirati, ne zgostiti.

2. **Preverjanje avtorja.** Zgoščevanje ne pove, kdo je besedilo napisal, le katero besedilo je bilo zgoščeno. Za določitev avtorstva je potreben kriptografski podpis nad odtisom, ne pa sam odtis.
3. **Shranjevanje gesel.** Tu se skriva past, ki jo je dobro razumeti. SHA-256 je zasnovan tako, da je zelo hiter — kar je dobro za mnoge stvari, za to pa slabo. Napadalec z namensko strojno opremo lahko preveri več milijard gesel na sekundo proti zgoščeni vrednosti SHA-256, dokler ne najde vašega. Za shranjevanje gesel je treba uporabiti namerno počasne funkcije izpeljave ključa, kot so Argon2, scrypt ali bcrypt, v kombinaciji s *soljo* (edinstven naključen podatek za vsakega uporabnika, ki preprečuje, da bi dve osebi z istim geslom imeli enak odtis).
4. **Branje odtisa kot identifikatorja avtorja.** To ne drži. Zgoščevanje identificira vsebino. Če dve osebi z SHA-256 zgostita besedo *živijo*, obe dobita enak povzetek — in to je ključna lastnost, ne napaka: če bi bila povzetka različna, ne bi mogli preveriti ujemanja med objavljenim in prejetim.

Kje se SHA-256 pojavlja v vašem vsakdanjem življenju

Čeprav ga ne vidite, SHA-256 podpira velik del tistega, kar vsakodnevno uporabljate na internetu. Veriga blokov omrežja Bitcoin se gradi z veriženjem odtisa SHA-256 vsakega bloka z naslednjim; sprememba prejšnjega bloka zahteva ponovni izračun celotne verige. Git, sistem za nadzor različic kode, ki ga uporablja pol sveta, identificira vsako objavo po SHA-256 (v novejših različicah) ali njenem predhodniku SHA-1 (v starejših različicah) njene celotne vsebine. Certifikati HTTPS, ki preverjajo identiteto spletnega mesta ob vstopu, imajo povezan odtis SHA-256. Prenose programske opreme pogosto spremlja SHA-256, ki ga objavi razvijalec, da lahko preverite, ali datoteka ni bila na poti spremenjena. In kot smo rekli, na dnu vsake izdaje Cuadernos Lacre.

Za strokovnega bralca

Štirje operativni opomniki za tiste, ki odločajo o sistemih ali jih preverjajo:

1. Zgoščevanje ni šifriranje. Če ponudnik v svoji tehnični dokumentaciji zameša oba pojma, je dobro vprašati, kaj natančno misli.
2. Za shranjevanje gesel se nikoli ne sme uporabljati sam SHA-256. SHA-256 je za to nalogo prehiter (glejte 3. točko v *Kaj zgoščevanje ni*). Trenutni standard je **Argon2id**: po zasnovi počasen, prilagodljiv zmogljivosti strežnika in združen z različno naključno *soljo* za vsakega uporabnika.
3. Za celovitost dokumentov — pogodb, zapisov, datotek — ostaja SHA-256 referenčni standard. Tega uporabljajo kvalificirani elektronski časovni žigi v EU.
4. Za dolgoročno shranjevanje (desetletja) je priporočljivo poleg SHA-256 izračunati in arhivirati tudi SHA-3 ali SHA-512; kriptografska previdnost priporoča, da se pri arhiviranju za celo stoletje ne zanašamo na eno samo funkcijo.

Tehnično je ta iterirana struktura – kjer se vmesno stanje ohranja med vhodnimi bloki – znana kot konstrukcija **Merkle-Damgård**, model, na katerem temeljijo SHA-1, SHA-2 (vključno s SHA-256) in številne druge klasične hash funkcije. SHA-3 nasprotno opušča konstrukcijo Merkle-Damgård v korist drugačne arhitekture, imenovane *goba* (sponge).

Kako deluje SHA-256, korak za korakom, s preprostimi besedami

Predstavljajte si, da ste sestavili najbolj dodelano domino vezje na svetu: na tisoče žetonov, na desetine razcepov, mehanskih mostov in klančin čez celo sobo, ki so skrbno postavljeni kos za kosom.

Če se dotaknete prvega žetona, veriga pade v natančnem in ponovljivem zaporedju. Enaka postavitev, enak začetni dotik → enak končni vzorec padlih žetonov, znova in znova.

Tukaj postane zanimivo: premaknite **en sam žeton** za pol centimetra v stran, preden začnete, in se ga ponovno dotaknite. Klančina, ki bi se morala aktivirati, ostane nepremična, most ne pade, sproži se drugačen razcep. Končni vzorec žetonov na tleh je popolnoma neprepoznaven v primerjavi s prvim.

SHA-256 je matematično to vezje. Besedilo, ki ga napišete, je začetni položaj žetonov. Algoritem je dotik, ki sprosti kaskado. In končni rezultat – tisto, ko imenujemo *hash* – je fotografija tal, ko se je vse ustavilo. Spremenite eno samo vejico v izvornem besedilu in fotografija bo popolnoma drugačna. Tako preprosto in tako drastično.

1. korak: Prevajanje besedila v binarne žetone. Računalniki ne razumejo črk; najprej jih prevedejo v številke (ASCII), številke pa v binarne zapise (enice in ničle). Vsaka črka se spremeni v 8 belih ali črnih žetonov: *A* je `01000001`, *B* je

`01000010`, presledek je `00100000`. Celotno vaše besedilo – beseda, pogodba, roman – postane dolga vrsta belih in črnih žetonov.

2. korak: Dopolnjevanje do standardne velikosti. Vezje procesira vrsto v *blokih* z natanko 512 žetoni. Če vaše sporočilo ne doseže večkratnika 512, se takoj za besedilom doda označevalni žeton (tisti z vrednostjo `10000000`), nato pa ničle do dopolnitve bloka. Zadnjih 64 položajev vsakega bloka je rezerviranih za zapis prvotne dolžine besedila. Tako vezje vedno ve, kje se je končala dejanska vsebina in kje se je začelo polnilo.

3. korak: Postavitev osmih glavnih žetonov. Preden začnemo, na mizo postavimo **osem glavnih žetonov** v natančen začetni položaj. Teh osem žetonov ni nobena skrivnost: njihova začetna vrednost je določena z javnim matematičnim pravilom (kvadratni koreni prvih osmih prastevil – 2, 3, 5, 7, 11, 13, 17, 19 – in prvi biti decimalnega dela vsakega korena). Vsakdo, v katerem koli kotičku planeta, začne z istimi osmimi glavnimi žetoni na istem položaju. Njihova usoda je, da jih plaz potisne in preoblikuje.

4. korak: Veliki plaz: štiriinšestdeset krogov potiskanja. Tukaj se začne spektakel. Prvi blok 512 žetonov vašega besedila trči v osem glavnih žetonov. Toda ti ne padejo hkrati: mehanizem izvede **štiriinšestdeset zaporednih krogov**. V vsakem krogu z žetoni izvede tri operacije:

- **Vrtiljak** (rotacija). Žetoni se premikajo v krogu: tisti na desni preidejo na levo. Noben žeton se ne izgubi ali doda; preprosto se prerazporedijo tako, da naredijo cel krog na vrtiljaku. To je poceni in povratni način za prerazporejanje informacij.
- **Logični lijak** (XOR). Žetoni gredo skozi lijak, ki jih primerja po dva in dva: če sta oba iste barve, izstopi bel; če sta različna, izstopi črn. To je najpreprostejša operacija binarne logike, toda v kombinaciji s rotacijami vrtiljaka postane izjemno močna za mešanje informacij brez njihove izgube.
- **Preliv** (modularno seštevanje). Rezultat se sešteje s *žetonom konstantnega potiska*, vzetim iz javnega seznama štiriinšestdesetih konstant (kubični koreni prvih štiriinšestdesetih prastevil). Če seštevanje ustvari dodatne žetone, ki ne sodijo v predviden prostor za 32 žetonov, se ti odvečni žetoni zavržejo. Na mizi je prostor samo za 32 žetonov, niti enega več.

Na koncu štiriinšestdesetega kroga je vsak od žetonov v bloku vašega besedila vplival na položaj osmih glavnih žetonov. Energija potiska je potovala po celotnem vezju.

5. korak: Dodajanje naslednjega bloka (brez ponastavitve). Če je bilo vaše besedilo dolgo in ostane še en blok 512 žetonov za obdelavo, **se vezje ne ponastavi**. Osem glavnih žetonov ostane takšnih, kot jih je pustil prvi plaz, drugi blok pa se sproži proti njim, da aktivira naslednjih štiriinšestdeset krogov. To je tako, kot bi dodali novo sobo, polno domin, na konec tiste, ki je pravkar padla: nered prve sobe v celoti določa, kako bo padla druga.

6. korak: Izdelava končne fotografije. Ko ni več blokov za obdelavo, se plaz ustavi. Pogledamo končni položaj osmih glavnih žetonov. Njihovo konfiguracijo prevedemo v kodo črk in številk v šestnajstiškem sistemu. Rezultat este niz natanko štiriinšestdesetih znakov: to je vaš žig SHA-256.

Štiri lastnosti izhajajo same po sebi iz načina, kako je vezje postavljeno:

1. **Determinizm.** Isto besedilo vedno ustvari isto končno fotografijo na katerem koli računalniku na svetu. Nič naključja, nič presenečenj.
2. **Učinek plaza.** Dodana vejica, spremenjena velika črka, pozabljen naglas: končna fotografija postane popolnoma neprepoznavna. To je tista izjemna občutljivost, ki smo jo opisali že na začetku.
3. **Enosmernost.** Iz končne fotografije ne morete rekonstruirati prvotnega besedila. Rotacije, lijaki in prelivi uničijo vse usmerjene informacije o tem, *od kod je prišel vsak bit*, in ohranijo le, *kaj se je skupaj seštelo*.
4. **Odpornost na kolizije.** V petindvajsetih letih javne kriptoanalize nikomur ni uspelo najti dveh različnih besedil, katerih končni fotografiji bi se ujemali. In težavnost te naloge je onkraj računske zmogljivosti katere koli razumno predstavljive civilizacije.

Dodatek s kodo, ki sledi, implementira natanko teh šest korakov v jeziku Zig. Zdaj ga lahko berete z vedenjem, kaj pomeni vsaka bitna operacija, namesto da slepo sprejemate te manipulacije.

Tehnični glosar

Za bralca, ki želi razumeti, kaj počne posamezna operacija. To lahko prosto preskočite: članek bo razumljiv tudi brez tega.

ASCII in Unicode — kako črke postanejo številke. Računalniki ne vidijo črk; vidijo številke. Standard, imenovan **ASCII** (*American Standard Code for Information Interchange*, iz leta 1963), vsakemu znaku na tipkovnici dodeli določeno številko: *A* je 65, *B* je 66, *a* je 97, *0* je 48, presledek je 32, vejica je 44. Sodobni sistemi to razširijo z **Unicode**, ki dodeli številko vsakemu znaku vsake abecede na svetu: cirilice, arabščine, kitajščine, japonščine in celo emojijem. Ko napišete znak ali odprete besedilno datoteko, računalnik prebere številko v ozadju, ne oblike na zaslonu. SHA-256 deluje na teh številkah in vsako besedilo obravnava kot dolgo zaporedje številk. Zato lahko z istim algoritmom zapečati članek v španščini, pesem v japonščini in binarno datoteko.
XOR — bitni primerjalnik. XOR (izgovori se *»ex-or«*, iz angleščine *exclusive or*, »izključni ali«) je ena najpreprostejših operacij, ki jih računalnik lahko izvede z dvema binarnima številoma. Primerja dva bita položaj za položajem in vrne: **1**, če je natanko eden od njiju 1 (eden, vendar ne oba), **0**, če sta oba enaka (oba 0 ali oba 1). Primer: XOR za `1010` in `1100` je `0110`. Ima pomembno lastnost: je povraten – če dvakrat izvedete XOR z istim ključem, se vrnete na prvotno vrednost. Zato je delovni konj kriptografije: meša bite brez izgube informacij, vendar rezultat ne razkrije ničesar o vhodih, če ne poznate enega od njih.
Šestnajstiški sistem — štetje v bazi 16. Skoraj vse številke v vsakdanjem življenju uporabljajo deset števk (0–9). Šestnajstiški sistem jih uporablja šestnajst: običajne 0–9 in šest črk, ki predstavljajo naslednje vrednosti: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Zakaj šestnajst? Ker računalniki razmišljajo v skupinah po štiri bite, štirje biti pa lahko predstavljajo natanko šestnajst različnih vrednosti – tako en šestnajstiški znak čisto ustreza štirim bitom. Odtis SHA-256 meri 256 bitov, kar je natanko **64 šestnajstiških znakov**. Če bi ga zapisali v običajnem desetiškem sistemu, bi zavzel približno 78 števk in bi bil bolj neroden. Izbira este estetska in kompaktna; številka v ozadju je ista.
Rotacija bitov — binarni vrtiljak. Predstavljajte si vrsto sedmih žarnic, nekatere svetijo (1), druge pa ne (0): `1 0 1 1 0 0 1`. Rotacija v desno za eno mesto pomeni, da vzamete žarnico na skrajni desni, jo prestavite na skrajno levo, ostale pa premaknete za eno mesto v desno: `1 1 0 1 1 0 0`. Nobena žarnica se ne izgubi ali doda: preprosto plešejo v krogu. SHA-256 pri vsakem izračunu stokrat uporabi rotacijo bitov; to je poceni način prerazporejanja informacij znotraj stanja brez izgub.
Konstante *«nothing-up-my-sleeve»* — zakaj izhajajo iz prastevil. Osem glavnih žetonov in štiriinšestdeset konstant krogov SHA-256 ni bilo izbranih naključno. Izhajajo iz kvadratnih in kubičnih korenov prvih prastevil. Zakaj? Ker so njihovi načrtovalci želeli konstante *«brez asov v rokavu»* (*«nothing-up-my-sleeve»*): vrednosti, katerih izvor lahko preveri vsakdo. Če bi vam kdo rekel: *«zaupaj mi: uporabi to naključno 32-bitno številko»*, bi upravičeno posumili na skrito šibkost ali stranska vrata. Toda vsakdo s kalkulatorjem lahko preveri, ali so prvi 32 biti kvadratnega korena števila 2 enaki `0x6a09e667`. Vrednosti so matematične, javne in ponovljive: v recept se ne more prikrasti nobena skrita past.

Dodatek: SHA-256 v berljivi kodi

Ta dodatek je namenjen bralcu, ki si želi algoritem ogledati od znotraj. Gre za didaktično implementacijo v jeziku Zig, ki sledi specifikaciji FIPS 180-4. To ni različica, ki jo uporablja Solo2 — tista prava se nahaja v `std.crypto.hash.sha2.Sha256` standardne knjižnice Zig, optimizirana in revidirana. Vendar pa je algoritem enak: to, kar vidite tukaj, korak za korakom prikazuje, kaj se zgodi, ko ta petznakovni klic opravi svoje delo.

```
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
}
```

Kakršen koli prepis v drug jezik, ki sledi enaki strukturi — začetne konstante, širitev urnika, štiriinšestdeset krogov, zbiranje — bo dal enak rezultat. Algoritem nima skrivnosti: njegova vrednost je v tem, da zgoraj naštete lastnosti veljajo še po dveh desetletjih javne kriptografske analize na tisoče oči.

Če se vrnete na dno tega članka, boste videli heksadecimalni pečat s štiriinšestdesetimi znaki. To je SHA-256 besedila, ki ste ga pravkar prebrali, v tem jeziku. Če bi članek prevedli, bi bil pečat drugačen; če bi se spremenila ena sama beseda v španski različici, bi se španski pečat spremenil. Pečat ne varuje vsebine — za to obstajajo druga orodja — temveč jo nedvoumno identificira. In to, čeprav zveni skromno, zadošča, da noben korak v založniški verigi ne more neopaženo spremeniti povedanega. Vse ostalo — šifriranje, podpisovanje, identifikacija — se gradi na tej preprosti zamisli.

Viri in nadaljnje branje

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, avgust 2015. Uradna specifikacija družine SHA-2, vključno z SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, maj 2011. Normativna različica za implementatorje.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Poglavji 5 in 6 obravnavata zgoščevalne funkcije ter njihovo legitimno in nelegitimno uporabo.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Praktičen primer uporabe SHA-256 za veriženje blokov v strukturah, ki so po zasnovi nespremenljive.

- Uredba (EU) 910/2014 (eIDAS) — okvir za kvalificirane elektronske časovne žige. SHA-256 je referenčna funkcija za kvalificirane elektronske podpise in pečate, izdane v EU.
- Referenčna implementacija v jeziku Zig: `std.crypto.hash.sha2.Sha256` v uradnem repozitoriju jezika (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). To je optimizirana in revidirana različica, ki jo Solo2 dejansko uporablja. Koristna je za primerjavo z didaktično implementacijo v dodatku.

← PrejšnjiCUADERNOS_LIST_SCHREMS_TITLENaslednji →CUADERNOS_LIST_KILLSWITCH_TITLE

Zadnja branja

- CUADERNOS_LIST_PREGUNTAS_TITLE
- CUADERNOS_LIST_SELFHOST_TITLE
- CUADERNOS_LIST_IDENTIDAD_TITLE

Vzemite ta članek s seboj, kamor koli ga potrebujete.

↓ Markdown ↓ Navadno besedilo ↓ PDF

Datoteka bo prenesena v vašo napravo. Od tam jo lahko shranite, uvozite v Solo2 ali delite, kjer koli želite. Cuadernos ne odloča o cilju namesto vas.

Voščeni pečat · SHA-256 9d4faaea48830466c58ee4ec9bc6e3fe7b8759e9c7523c0ba9e74ebb34efa347

Cuadernos Lacre · Publikacija podjetja Menzuri Gestión S.L. ·
napisal R.Eugenio · uredila ekipa Solo2.