Kas ir SHA-256 patiesībā

Matemātisks nospiedums, kas ietilpst sešdesmit četros simbolos un pilnībā mainās, ja tiek izmainīts pat viens oriģinālā teksta komats. Kāpēc mēs to saucam par digitālo zīmogvasku.

Vienkārša ideja aiz tehniskā nosaukuma

Iedomājieties, ka eksistē mašīna ar vienu atveri un vienu ekrānu. Pa atveri jūs ievadāt tekstu: vārdu, frāzi, veselu romānu. Ekrānā pēc mirkļa parādās secība, kas sastāv tieši no sešdesmit četriem simboliem. Profesionāls lasītājs šo secību sauc par *hash* (jaucējvērtību) jeb kriptogrāfisko kopsavilkumu; parastam lasītājam mēs to pagaidām varam saukt par teksta matemātisko nospiedumu, līdzīgi kā pirkstu nospiedums ir cilvēka identitātes zīme.

Ja ievadīsiet to pašu tekstu divreiz, mašīna abas reizes parādīs vienu un to pašu nospiedumu. Ja ievadīsiet nedaudz atšķirīgu tekstu – pārvietosiet vienu komatu, mainīsiet lielo burtu uz mazo –, mašīna parādīs pilnīgi citu nospiedumu nekā pirmo. Ne līdzīgu, bet pilnīgi atšķirīgu. Šīs divas īpašības kopā – determinisms un jutīgums – ir tā vienkāršā ideja. Viss pārējais SHA-256 sistēmā ir tikai mehānisms, kas nodrošina šo īpašību izpildi.

Svarīgi jau no paša sākuma pateikt, ko mašīna nedara. Tā nešifrē tekstu. Tā to neslēpj. Tā to nesaglabā. Mašīna apskata tekstu, aprēķina nospiedumu un tekstu aizmirst. Pēc nospieduma nav iespējams atjaunot tekstu, kas to radījis; tas ļauj tikai, ņemot vērā kandidātu tekstu, pārbaudīt, vai tas sakrīt ar oriģinālu vai nē. Tāpēc mēs sakām, ka tas ir *vienvirziena* kopsavilkums: var aiziet, bet nevar atgriezties.

Jaucējfunkcija (hash) nav tas pats, kas šifrēšana

Sajaukšana notiek bieži, tāpēc ir vērts to kliedēt: šifrēšana un „hašēšana” (jaukšana) ir dažādas darbības. Šifrēšana sastāv no teksta transformēšanas tā, lai tikai atslēgas īpašnieks varētu to atgriezt sākotnējā formā. Jaukšana sastāv no teksta nospieduma radīšanas, no kura sākotnējo tekstu nekad nevar atgūt – ne ar atslēgu, ne bez tās. Pirmā darbība pēc konstrukcijas ir atgriezeniska; otrā – neatgriezeniska.

Praktiskajām sekām ir nozīme. Kad lietotne saka „mēs glabājam jūsu paroli šifrētu”, ir kāds, kam ir atslēga tās atšifrēšanai – vismaz pašai lietotnei. Kad lietotne saka „mēs glabājam jūsu paroli hašētu”, pati lietotne nevar izlasīt sākotnējo paroli, pat ja gribētu; tā var tikai pārbaudīt, vai tas, ko jūs rakstāt, atkal rada to pašu nospiedumu. Otrais modelis, ja tas ir pareizi ieviests, ir daudz labāks paroļu glabāšanai. Vēlāk redzēsim, kāpēc „pareizi ieviests” prasa ko vairāk nekā tikai tīru SHA-256.

Četras īpašības, kas padara kriptogrāfisko kopsavilkumu noderīgu

Jaucējfunkcija, kas ir pelnījusi apzīmējumu *kriptogrāfiska*, atbilst četrām īpašībām:

1. **Determinisma princips.** Viena un tā pati ievade vienmēr rada vienu un to pašu nospiedumu.
2. **Lavīnas efekts.** Neliela izmaiņa ievadē rada pilnīgi citu nospiedumu, bez redzamas līdzības ar iepriekšējo.
3. **Noturība pret pretattēla atrašanu.** Ņemot vērā nospiedumu, nav skaitļošanas ziņā iespējams atrast tekstu, kas to radījis.
4. **Noturība pret kolīzijām.** Nav skaitļošanas ziņā iespējams atrast divus dažādus tekstus, kas rada vienu un to pašu nospiedumu.

„Nav skaitļošanas ziņā iespējams” nenozīmē „ir matemātiski neiespējami”. Tas nozīmē, ka laika, enerģijas un naudas izmaksas, lai to sasniegtu, par daudzām kārtām pārsniedz visu saprātīgi pieejamo skaiļošanas jaudu summu. SHA-256

gadījumā šī robeža tiek mērīta tūkstošos triljonu gadu pat visoptimistiskākajām prognozēm ar specializētu aparatūru. Kas lasītājam praktiski nozīmē to pašu, ko „nevar”.

Konkrēti par SHA-256

Nosaukums pasaka visu. SHA ir akronīms no *Secure Hash Algorithm* – drošs jaucējalgoritms. Skaitlis 256 norāda nospieduma izmēru bitos: divi simti piecdesmit seši biti, t.i., trīsdesmit divi baiti, kas sešpadsmitnieku sistēmā ir tie sešdesmit četri simboli, kurus lasītājs jau atpazīst. Standartu 2001. gadā publicēja ASV NIST, organizācija, kas normē šāda veida funkcijas, kā daļu no SHA-2 saimes; pašreizējā standarta versija FIPS 180-4 ir no 2015. gada.

Tiem, kam vēl nav skaidrs, kas ir biti un baiti:

```
1 bits    →  0 vai 1        (slēdzis: ieslēgts vai izslēgts)
1 baits   →  8 biti         (256 iespējamās kombinācijas)
32 baiti  →  256 biti       (SHA-256 nospiedums)
```

Skaitlis 256 nosaukuma beigās norāda nospieduma izmēru bitos. Sešpadsmitnieku sistēmā – skaitīšanas sistēmā ar sešpadsmit simboliem desmit vietā – šie 256 biti ietilpst tieši 64 simbolos. Tie ir tie 64 simboli, kurus redzat katra „Cuaderno” apakšā.

Ir vērts brīdi padomāt par mērogiem. Divi simti piecdesmit seši biti pieļauj divi pakāpē divi simti piecdesmit seši dažādu vērtību: skaitlis ar septiņdesmit astoņiem cipariem, par vairākām kārtām lielāks nekā aprēķinātais atomu skaits novērojamajā visumā. Katrs pasaules teksts – katra grāmata, katrs e-pasts, katra ziņa – nonāk pie vienas no šīm vērtībām. Varbūtība, ka divi dažādi teksti sakritīs nejauši, praktiski neatšķiras no nulles.

Kā tas izskatās kodā

„Zig” valodā, kurā mēs rakstām „Solo2” pamatus, SHA-256 nospieduma aprēķināšana tekstam izskatās šādi:

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

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

Mēs tikko palūdzām „Zig” standarta bibliotēkai aprēķināt pēdiņās liktā teksta SHA-256. Pēc izsaukuma mainīgais *resumen* satur trīsdesmit divus baitus, kas veido nospiedumu tā neapstrādātā formā; kad tos parāda ekrānā sešpadsmitnieku sistēmā, tie ir sešdesmit četri simboli šī raksta apakšā. Ja mēs nomainītu *Cuadernos Lacre* uz *Cuadernos lacre* – par vienu lielo burtu mazāk –, nospiedums pilnībā mainītos. Tā ir tā galvenā īpašība, kas piecās koda rindiņās uztur visu pārējo. Tiem, kas vēlas redzēt, kā tas darbojas iekšpusē, raksta beigās esam iekļāvuši lasāmu algoritma versiju ar komentāriem soli pa solim.

Kāpēc mēs to saucam par zīmogvasku

Eiropas korespondencē no piecpadsmitā līdz deviņpadsmitajam gadsimtam zīmogvasks noslēdza vēstuli. Izkausēta vaska pile, virsū uzspiests zīmogs, un vēstule palika atzīmēta neatkārtojamā veidā. Tas nepasargāja saturu no apņēmīga ziņkārīgā – papīru varēja izlasīt pret gaismu, vasku varēja salauzt –, bet tas padarīja to acīmredzamu. Jebkura noslēguma izmaiņa bija redzama saņēmējam vēl pirms papīra atvēršanas. Vasks neaizkavēja kaitējumu; tas to deklarēja.

Katra „Cuaderno” teksta SHA-256 pilda to pašu funkciju digitālajā versijā. Ja rakstā mainītos kaut viens vārds no tā publicēšanas brīža līdz brīdim, kad jūs to lasāt, sešpadsmitnieku nospiedums teksta apakšā vairs nesakristu ar jūsu priekšā esošā teksta SHA-256 vērtību. Jebkurš lasītājs ar piecām koda rindiņām varētu to pārbaudīt. Izdevējs nevar pārrakstīt savu vēsturi, lai nospiedums viņu nenodotu. Tas neaizsargā pret kaitējumu; tas padara to pārbaudāmu.

Kas kopsavilkums (hash) nav

Reizēm no SHA-256 tiek gaidīti četri lietošanas veidi, kas tam nepienākas:

1. **Šifrēšana.** Jaucējfunkcija apkopo; tā neslēpj. Ja vēlaties, lai tekstu nevarētu izlasīt, tas ir jāšifrē, nevis jāhašē.

2. **Autora autentificēšana.** Jaucējfunkcija nepasaka, kas uzrakstīja tekstu, tikai to, kurš teksts tika hašēts. Lai sasaistītu autorību, ir nepieciešams kriptogrāfisks paraksts virs kopsavilkuma, nevis tikai pats kopsavilkums.
3. **Paroļu glabāšana.** Šeit ir slazds, kuru ir vērts saprast. SHA-256 ir izstrādāts, lai būtu ļoti ātrs – kas ir labi daudzām lietām, bet slikti šai. Uzbrucējs ar specializētu aparatūru var pārbaudīt miljardiem paroļu sekundē pret SHA-256 nospiedumu, līdz atrod jūsējo. Paroļu glabāšanai ir jāizmanto apzināti lēnas atslēgu atvasināšanas funkcijas, piemēram, „Argon2”, „scrypt” vai „bcrypt”, apvienojumā ar *sāli* (salt – unikāli nejauši dati katram lietotājam, kas neļauj diviem cilvēkiem ar vienādu paroli iegūt vienādu nospiedumu).
4. **Nospieduma izmantošana kā autora identifikators.** Tas tāds nav. Nospiedums identificē saturu. Ja divi cilvēki hašē vārdu *hola* ar SHA-256, abi iegūst vienu un to pašu rezultātu – un tā ir galvenā īpašība, nevis trūkums: ja rezultāti būtu dažādi, mēs nevarētu pārbaudīt sakritību starp publicēto un saņemto.

Kur SHA-256 parādās jūsu ikdienā

Lai gan jūs to neredzat, SHA-256 uztur lielu daļu no tā, ko ikdienā izmantojat internetā. „Bitcoin” blokķēde tiek veidota, savienojot katra bloka SHA-256 ar nākamo; mainot kādu pagātnes bloku, ir jāpārrēķina visa turpmākā ķēde. „Git”, sistēma, ar kuru tiek uzturētas koda versijas visā pasaulē, identificē katru izmaiņu (commit) pēc tās pilnā satura SHA-256 (jaunākajās versijās) vai pēc tā priekšteča SHA-1 (vecākajās versijās). HTTPS sertifikāti, kas apstiprina vietnes identitāti, ietver saistītu SHA-256 nospiedumu. Programmatūras lejupielādēm bieži pievieno izstrādātāja publicētu SHA-256, lai jūs varētu pārbaudīt, vai fails nav mainīts ceļā. Un, kā jau teicām, katra „Cuaderno Lacre” apakšā.

Profesionālam lasītājam

Četri operatīvi atgādinājumi tam, kurš pieņem lēmumus vai auditē sistēmas:

1. Jaucējfunkcija nav šifrēšana. Ja piegādātājs savā tehniskajā dokumentācijā jauc abus terminus, ir vērts pajautāt, ko tieši viņš ar to domā.
2. Paroļu glabāšanai nekad nedrīkst izmantot tīru SHA-256. SHA-256 šim uzdevumam ir pārāk ātrs (skat. 3. punktu sadaļā *Kas kopsavilkums nav*). Pašreizējais standarts ir **Argon2id**: pēc konstrukcijas lēns, konfigurējams atbilstoši servera jaudai, apvienots ar atšķirīgu nejaušu *sāli* katram lietotājam.
3. Dokumentu – līgumu, lietu, failu – integritātei SHA-256 joprojām ir atsauces standarts. Tieši to izmanto kvalificēti laika zīmogu pakalpojumu sniedzēji ES.
4. Ilgtermiņa saglabāšanai (gadu desmitiem) ir vērts aprēķināt un arhivēt arī SHA-3 vai SHA-512 kopā ar SHA-256; kriptogrāfiskā piesardzība iesaka nepaļauties tikai uz vienu funkciju gadsimtu arhīviem.

Tehniski šī iterētā struktūra — kur starpstāvoklis tiek saglabāts starp ievades blokiem — ir pazīstama kā **Merkle-Damgård** konstrukcija. Tas ir modelis, uz kura balstās SHA-1, SHA-2 (ieskaitot SHA-256) un daudzas citas klasiskās jaucējfunkcijas. Turpretī SHA-3 atsakās no Merkle-Damgård par labu citādai arhitektūrai, ko sauc par *sūkli* (angl. sponge).

Kā darbojas SHA-256: soli pa solim vienkāršos vārdos

Iedomājieties, ka esat uzbūvējis pasaulē sarežģītāko domino ķēdi: tūkstošiem kauliņu, desmitiem atzarojumu, mehāniskie tilti un rampas visā istabā, rūpīgi izvietoti pa vienai detaļai.

Ja pieskaraties pirmajam kauliņam, ķēde gāžas precīzā un atkārtojamā secībā. Tas pats izvietojums, tas pats sākotnējais pieskāriens → identisks galīgais nokritušo kauliņu raksts, atkal un atkal.

Šeit sākas interesantākais: pirms sākšanas pabīdiet **tikai vienu kauliņu** par puscentimetru uz sānu un atkal pieskarieties. Rampa, kurai vajadzēja nostrādāt, paliek nekustīga, tilts negāžas, nostrādā cits atzarojums. Galīgais kauliņu raksts uz grīdas ir pilnīgi neatpazīstams salīdzinājumā ar pirmo.

Matemātiski SHA-256 ir tieši šī ķēde. Jūsu rakstītais teksts ir kauliņu sākotnējā pozīcija. Algoritms ir pieskāriens, kas palaiž kaskādi. Un galīgais rezultāts — ko mēs saucam par *hašu* (angl. hash) — ir grīdas fotoattēls, kad viss ir apstājies. Nomainiet kaut vienu komatu oriģinālajā tekstā, un fotoattēls būs radikāli citāds. Tik vienkārši un tik drastiski.

1. solis. Teksta tulkošana bināros kauliņos. Datori nesaprot burtus; tie vispirms tos pārvērš skaitļos (ASCII), bet skaitļus — binārajā sistēmā (vieniniekos un nullēs). Katrs burts pārvēršas par 8 baltiem vai melniem kauliņiem: *A* ir 65 (ASCII), *B*

ir 66. Datora atmiņā *A* kļūst par `01000001`, *B* — `01000010`, atstarpe — `00100000`. Viss jūsu teksts — vārds, līgums, romāns — kļūst par garu baltu un melnu kauliņu rindu.

2. solis. Papildināšana līdz standarta izmēram. Ķēde apstrādā rindu precīzi pa 512 kauliņu *blokiem*. Ja jūsu ziņojums nesasniedz 512 daudzkārtni, tūlīt pēc teksta tiek pievienots marķiera kauliņš (kura vērtība ir `10000000`), un pēc tam nulles, līdz tiek aizpildīts bloks. Katra bloka pēdējās 64 pozīcijas ir rezervētas oriģinālā teksta garuma atzīmēšanai. Tādējādi ķēde vienmēr zina, kur beidzās īstais saturs un kur sākās papildinājums.

3. solis. Astoņu meistarkauliņu novietošana. Pirms sākšanas uz galda precīzās sākotnējās pozīcijās novietojam **astoņus meistarkauliņus**. Šie astoņi kauliņi nav nekāds noslēpums: to sākotnējā vērtība ir noteikta pēc publiska matemātiska noteikuma (pirmo astoņu pirmskaitļu — 2, 3, 5, 7, 11, 13, 17, 19 — kvadrātsaknes un katras saknes daļveida daļas pirmie biti). Ikviens visā pasaulē sāk ar tiem pašiem astoņiem meistarkauliņiem tajās pašās pozīcijās. To liktenis ir tikt grūstītiem un transformētiem nogruvuma rezultātā.

4. solis. Lielais nogruvums: sešdesmit četras grūstīšanās kārtas. Šeit sākas izrāde. Pirmais 512 jūsu teksta kauliņu bloks saduras ar astoņiem meistarkauliņiem. Taču tie negāžas uzreiz: mehānisms veic **sešdesmit četras secīgas kārtas**. Katrā kārtā ar kauliņiem tiek veiktas trīs operācijas:

- **Karuselis** (rotācija). Kauliņi kustas pa apli: labās puses kauliņi pāriet uz kreiso. Neviens kauliņš netiek pazaudēts vai pievienots; tie tiek vienkārši pārgrupēti, apmetot pilnu loku karuselī. Tas ir lēts un atgriezenisks veids, kā pārdalīt informāciju.
- **Loģiskā piltuve** (XOR). Kauliņi iet cauri piltuvei, kas tos salīdzina pa pāriem: ja abi ir vienā krāsā, iznāk balts; ja dažādās — melns. Tā ir vienkāršākā binārās loģikas operācija, taču kombinācijā ar karuseļa rotācijām tā kļūst ārkārtīgi spēcīga informācijas sajaukšanai, to nezaudējot.
- **Pārpilde** (modulārā saskaitīšana). Rezultāts tiek saskaitīts ar *pastāvīgā grūdiena kauliņu*, kas paņemts no publiska sešdesmit četru konstantu saraksta (pirmo sešdesmit četru pirmskaitļu kubsaknes). Ja saskaitīšana rada papildu kauliņus, kas neietilpst paredzētajā 32 kauliņu telpā, šie liekie kauliņi tiek atmesti. Uz galda ir vieta tikai 32 kauliņiem, ne par vienu vairāk.

Sešdesmit ceturtās kārtas beigās katrs jūsu teksta bloka kauliņš ir ietekmējis astoņu meistarkauliņu pozīciju. Grūdiena enerģija ir apceļojusi visu ķēdi.

5. solis. Nākamā bloka pievienošana (neatsākot no jauna). Ja jūsu teksts bija garš un palicis vēl kāds 512 kauliņu bloks, **ķēde netiek restartēta**. Astoņi meistarkauliņi paliek tādi, kādus tos atstāja pirmais nogruvums, un otrais bloks tiek palaists pret tiem, aktivizējot vēl sešdesmit četras kārtas. Tas ir tāpat kā pievienot jaunu istabu, pilnu ar domino, tās galā, kura tikko nogruva: pirmās istabas nekārtība pilnībā nosaka to, kā bruks otrā.

6. solis. Galīgā fotoattēla uzņemšana. Kad vairs nav bloku apstrādei, nogruvums apstājas. Apskatām galīgo pozīciju, kurā palikuši astoņi meistarkauliņi. Pārvēršam to konfigurāciju burtu un skaitļu kodā sešpadsmitnieku sistēmā. Rezultāts ir precīzi sešdesmit četru rakstzīmju virkne: tas ir jūsu SHA-256 zīmogs.

Četras īpašības izriet pašas no sevis no tā, kā uzbūvēta ķēde:

1. **Determinizms.** Tas pats tekstas vienmēr rada to pašu galīgo fotoattēlu jebkurā pasaules datorā. Nulle nejaušības, nulle pārsteigumu.
2. **Nogruvuma efekts.** Pievienots komats, nomainīts lielais burts, aizmirsta garumzīme — fotoattēls kļūst pilnīgi neatpazīstams. Tā ir tā īpašā jutība, ko aprakstījām sākumā.
3. **Viens virziens.** No galīgā fotoattēla nevar atjaunot oriģinālo tekstu. Rotācijas, piltuves un pārpildes iznīcina visu virziena informāciju par to, *no kurienes nāca katrs bits*, un saglabā tikai to, *kas tika saskaitīts kopā*.
4. **Izturība pret kolīzijām.** Divdesmit piecos publiskās kriptoanalīzes gados nevienam nav izdevies atrast divus dažādus tekstus, kuru galīgie fotoattēli sakristu. Un sarežģītība to izdarīt pārsniedz jebkuras saprātīgi iedomājamas civilizācijas skaitļošanas jaudu.

Turpmākajā koda pielikumā šie seši soļi ir precīzi īstenoti Zig valodā. Tagad jūs varat to lasīt, zinot, ko nozīmē katra bitu operācija, nevis akli pieņemot manipulācijas.

Tehnisko terminu vārdnīca

Lasītājam, kurš vēlas saprast, ko dara katra operācija. Varat droši izlaist: raksts ir saprotams arī bez tā.

ASCII un Unicode — kā burti kļūst par skaitļiem. Datori neredz burtus; tie redz skaitļus. Standarts, ko sauc par **ASCII** (angl. American Standard Code for Information Interchange, 1963. gads), katrai tastatūras rakstzīmei piešķir konkrētu skaitli: *A* ir 65, *B* ir 66, *a* ir 97, *0* ir 48, atstarpe ir 32, komats ir 44. Mūsdienu sistēmas to paplašina ar **Unicode**, kas piešķir skaitli katram burtam katrā pasaules alfabētā: kirilicā, arābu, ķīniešu, japāņu un pat emociju zīmēm. Kad rakstāt rakstzīmi vai atverat teksta failu, dators nolasa paslēpto skaitli, nevis formu ekrānā. SHA-256 darbojas ar šiem skaitļiem, apstrādājot jebkuru tekstu kā garu ciparu secību. Tāpēc tas var noplombēt rakstu spāņu valodā, dzejoli japāņu valodā un bināro failu ar to pašu algoritmu.
XOR — bitu salīdzināšanas rīks. XOR (izrunā kā „eksor", no angl. exclusive or — „izslēdzošais vai") ir viena no vienkāršākajām operācijām, ko dators var veikt ar diviem bināriem skaitļiem. Tas salīdzina divus bitus pozīciju pa pozīcijai un atgriež: **1**, ja tieši viens no abiem ir 1 (viens, bet ne abi), **0**, ja abi ir vienādi (abi 0 vai abi 1). Piemērs: `1010` un `1100` XOR ir `0110`. Tam piemīt īpaša īpašība: tas ir atgriezenisks — ja veicat XOR divreiz ar to pašu atslēgu, atgriežaties pie oriģināla. Tāpēc tas ir kriptogrāfijas darba zirgs: sajauc bitus, nezaudējot informāciju, taču rezultāts neko neatklāj par ievadēm, ja nezināt vienu no tām.
Heksadecimālā sistēma — skaitīšana ar bāzi 16. Gandrīz visi ikdienas skaitļi izmanto desmit ciparus (0-9). Heksadecimālā sistēma izmanto sešpadsmit: parastos 0-9 un sešus burtus, kas apzīmē šādas vērtības: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Kāpēc sešpadsmit? Tāpēc, ka datori domā četru bitu grupās, un četri biti var reprezentēt precīzi sešpadsmit dažādas vērtības — tādējādi heksadecimāls simbols tīri atbilst četriem bitiem. SHA-256 pirksta nospiedums ir 256 bitu garš, kas ir precīzi **64 heksadecimālas rakstzīmes**. Ja mēs to rakstītu parastajā decimālajā sistēmā, tas aizņemtu apmēram 78 ciparus un būtu neērtāks. Šī izvēle ir estētiska un kompakta; paslēptais skaitlis ir tas pats.
Bitų rotācija — binārais karuselis. Iedomājieties septiņu spuldzīšu rindu, dažas deg (1) un dažas ne (0): `1 0 1 1 0 0 1`. Rotācija pa labi par vienu pozīciju nozīmē paņemt spuldzīti no paša labā malas, aiznest to uz kreiso galu un pabīdīt pārējās par vienu vietu pa labi: `1 1 0 1 1 0 0`. Neviena spuldzīte netiek pazaudēta vai pievienota: tās vienkārši dejo aplī. SHA-256 izmanto bitu rotāciju simtiem reižu katrā aprēķinā; tas ir lēts un bezzaudējumu veids, kā pārdalīt informāciju stāvokļa iekšienē.
Konstantes „bez viltus" (angl. nothing-up-my-sleeve) — kāpēc tās nāk no pirmskaitļiem. Astoņi meistarkauliņi un sešdesmit četras SHA-256 kārtas konstantes netika izvēlētas nejauši. Tās nāk no pirmo pirmskaitļu kvadrātsaknēm un kubsaknēm. Kāpēc? Jo to izstrādātāji vēlējās konstantes *„bez nekā paslēpta piedurknē"*: vērtības, kuru izcelsmi ikviens var pārbaudīt. Ja kāds jums teiktu *„uzticies man: izmanto šo 32 bitu nejaušo skaitli"*, jūs pamatoti aizdomātos par slēptu vājību vai sētas durvīm. Bet ikviens, kam ir kalkulators, var pārbaudīt, vai kvadrātsaknes no 2 pirmie 32 biti ir `0x6a09e667`. Vērtības ir matemātiskas, publiskas un reproducējamas: receptē nevar iezagties nekādas slēptas lamatas.

Pielikums: SHA-256 saprotamā kodā

Šis pielikums ir domāts lasītājam, kurš vēlas redzēt algoritmu no iekšpuses. Tas ir didaktisks īstenojums „Zig" valodā, kas atbilst FIPS 180-4 specifikācijai. Tā nav tā versija, ko izmanto „Solo2" – īstā ir `std.crypto.hash.sha2.Sha256` standarta „Zig" bibliotēkā, kas ir optimizēta un auditēta. Bet algoritms ir tas pats: tas, ko redzat šeit, ir soli pa solim tas, kas notiek, kad tas piecu rakstzīmju izsaukums veic savu darbu.

```
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
}
```

Jebkurš pārrakstījums citā valodā, kas ievēro to pašu struktūru – sākotnējās konstantes, grafika paplašināšanu, sešdesmit četras kārtas, akumulāciju –, dod to pašu rezultātu. Algoritmam nav noslēpumu: tā vērtība slēpjas tajā, ka iepriekš uzskaitītās īpašības turpina saglabāties pēc divu desmitgažu publiskas kriptoanalīzes tūkstošiem acu.

Ja atgriezīsieties šī raksta apakšā, redzēsiet sešdesmit četru simbolu sešpadsmitnieku nospiedumu. Tas ir SHA-256 tekstam, ko tikko izlasījāt šajā valodā. Ja mēs iztulkotu rakstu, nospiedums būtu cits; ja mainītos kaut vārds spāņu versijā, spāņu nospiedums mainītos. Nospiedums neaizsargā saturu – tam ir citi rīki –, bet tas to vienozīmīgi identificē. Un ar to, lai cik pieticīgi tas neizklausītos, pietiek, lai neviens redakcijas ķēdes posms nevarētu mainīt teikto, to nepamanot. Viss pārējais – šifrēšana, parakstīšana, identificēšana – tiek būvēts uz šīs vienkāršās idejas.

Avoti un papildu literatūra

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, 2015. gada augusts. Oficiālā SHA-2 saimes specifikācija, ieskaitot SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, 2011. gada maijs. Normatīvā versija programmētājiem.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). 5. un 6. nodaļa aptver jaucējfunkcijas un to likumīgu un nelikumīgu izmantošanu.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Praktisks piemērs SHA-256 izmantošanai bloku sasaistīšanai nemainīgā struktūrā.

- Regula (ES) 910/2014 (eIDAS) — kvalificētu laika zīmogu ietvars. SHA-256 ir atsauces funkcija kvalificētiem elektroniskajiem parakstiem un zīmogiem, kas izsniegti ES.
- Atsauces īstenojums „Zig” valodā: `std.crypto.hash.sha2.Sha256` oficiālajā valodas krātuvē (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). Tā ir optimizētā un auditētā versija, ko faktiski izmanto „Solo2”. Noderīga salīdzināšanai ar pielikuma didaktisko īstenojumu.

← IepriekšējaisCUADERNOS_LIST_SCHREMS_TITLENākamais →CUADERNOS_LIST_KILLSWITCH_TITLE

Jaunākie lasījumi

- CUADERNOS_LIST_PREGUNTAS_TITLE
- CUADERNOS_LIST_SELFHOST_TITLE
- CUADERNOS_LIST_IDENTIDAD_TITLE

Paņemiet šo rakstu līdzi tur, kur jums nepieciešams.

↓ Markdown ↓ Parasts teksts ↓ PDF

Fails tiks lejupielādēts jūsu ierīcē. No turienes varat to saglabāt, importēt Solo2 vai kopīgot jebkur. Cuadernos nepieņem lēmumu par galamērķi jūsu vietā.

Vaska zīmogs · SHA-256 a2b31c484a471065c75b590f8b6ff0f778da2d94cc0a7e9f76361968393e0711

Cuadernos Lacre · Menzuri Gestión S.L. publikācija ·
autors R.Eugenio · rediģējusi Solo2 komanda.