Mi is valójában a SHA-256

Egy matematikai ujjnyomat, amely hatvannégy karakterbe belefér, és amely teljesen megváltozik, ha az eredeti szövegben akár egyetlen vessző is elmozdul. Miért nevezzük digitális pecsétviasznak.

A technikai név mögötti egyszerű ötlet

Képzeld el, hogy létezik egy gép, amelynek egyetlen nyílása és egyetlen képernyője van. A nyíláson keresztül betáplálsz egy szöveget: egy szót, egy mondatot vagy egy egész regényt. A képernyőn pillanatokkal később megjelenik egy pontosan hatvannégy karakterből álló sorozat. Ezt a sorozatot a szakmai olvasóknak *hash*-nek vagy *kriptográfiai lenyomatnak* nevezzük; az átlagolvasó számára egyelőre hívhatjuk a szöveg matematikai ujjnyomatának, ahogyan az ujjlenyomat az emberé.

Ha kétszer ugyanazt a szöveget táplálod be, a gép mindkétszer ugyanazt az ujjnyomatot mutatja. Ha egy kicsit eltérő szöveget táplálsz be — egy elmozdított vessző, egy nagybetű, amely kisbetűvé válik —, a gép az elsőtől teljesen eltérő ujjnyomatot mutat. Nem hasonlót: eltérőt. Ez a két tulajdonság együtt — a determinizmus és az érzékenység — alkotja az egyszerű ötletet. A SHA-256 minden egyéb része az a gépezet, amely biztosítja ezek helyes működését.

Érdemes már az elején tisztázni, mit nem csinál a gép. Nem titkosítja a szöveget. Nem rejti el. Nem menti el. A gép megnézi a szöveget, kiszámítja az ujjnyomatot, és elfelejti a szöveget. Az ujjnyomatból nem lehet rekonstruálni az azt létrehozó szöveget; csak azt teszi lehetővé, hogy egy adott szövegről ellenőrizzük, megegyezik-e az eredetivel. Ezért mondjuk, hogy ez egy *egyirányú* lenyomat: odafelé megy, visszafelé nem.

A hash nem ugyanaz, mint a titkosítás

Gyakori a tévedés, ezért érdemes tisztázni: a titkosítás és a hash-elés különböző műveletek. A titkosítás abból áll, hogy egy szöveget úgy alakítunk át, hogy csak a kulcs birtokosa tudja visszaállítani az eredeti formájába. A hash-elés abból áll, hogy a szövegből egy olyan ujjnyomatot hozunk létre, amelyből az eredeti szöveg soha nem nyerhető vissza, sem kulccsal, sem anélkül. Az első tervezésénél fogva visszafordítható; a második tervezésénél fogva visszafordíthatatlan.

A gyakorlati következmény fontos. Amikor egy alkalmazás azt mondja: „titkosítva tároljuk a jelszavadat”, akkor van valaki, akinél ott a kulcs a feloldáshoz — mindenképpen maga az alkalmazás. Amikor egy alkalmazás azt mondja: „hash-elve tároljuk a jelszavadat”, az alkalmazás maga sem tudja elolvasni az eredeti jelszót, még ha akarná sem; csak azt tudja ellenőrizni, hogy amit beírsz, az újra ugyanazt az ujjnyomatot hozza-e létre. A második modell, ha jól csinálják, sokkal előnyösebb a jelszavak tárolására, mint az első. Később látni fogjuk, miért igényel a „jól csinálás” többet a puszta SHA-256-nál.

A kriptográfiai hash négy hasznos tulajdonsága

Egy hash függvény, amely megérdemli a *kriptográfiai* jelzőt, négy tulajdonságnak felel meg:

1. **Determinizmus.** Ugyanaz a bemenet mindig ugyanazt az ujjnyomatot eredményezi.
2. **Lavinaeffektus.** A bemenet kismértékű változása teljesen eltérő ujjnyomatot eredményez, amelyen nem látható hasonlóság az előzővel.
3. **Visszafejtéssel szembeni ellenállás.** Egy adott ujjnyomatból kiindulva számításilag nem kivitelezhető megtalálni az azt létrehozó szöveget.
4. **Ütközésmentesség.** Számításilag nem kivitelezhető két különböző szöveget találni, amelyek ugyanazt az ujjnyomatot eredményezik.

A „számításilag nem kivitelezhető” nem azt jelenti, hogy „matematikailag lehetetlen”. Azt jelenti, hogy az eléréséhez szükséges idő, energia és pénz nagyságrendekkel meghaladja az összes ésszerűen rendelkezésre álló számítási kapacitást. A SHA-256 esetében ez a korlát több billió évben mérhető, még a legoptimistább, speciális hardvereket használó megközelítések mellett is. Ami az olvasó gyakorlati szempontjából ugyanaz, mint a „lehetetlen”.

A SHA-256-ról konkrétan

A név mindent elárul. A SHA a *Secure Hash Algorithm* rövidítése: biztonságos hash algoritmus. A 256-os szám az ujjnyomat bitben mért méretét jelzi: kétszázötvenhat bit, azaz harminckét bájt, ami hexadecimális formában az a hatvannégy karakter, amelyet az olvasó már felismer. A szabványt az amerikai NIST tette közzé 2001-ben a SHA-2 család részeként; a szabvány jelenlegi verziója, a FIPS 180-4, 2015-ös.

Azoknak, akiknek még nem világos, mik azok a bitek és bájtok:

```
1 bit    →  0 vagy 1     (egy kapcsoló: be vagy ki)
1 bájt   →  8 bit        (256 lehetséges kombináció)
32 bájt  →  256 bit      (a SHA-256 ujjnyomat)
```

A név végén lévő 256-os szám az ujjnyomat méretét jelzi bitben. Hexadecimális rendszerben — amely tíz helyett tizenhat szimbólumot használó számrendszer — ez a 256 bit pontosan 64 karakterbe fér bele. Ez az a 64 karakter, amelyet minden Cuaderno alján látsz.

A méretek megérdemelnek egy pillanatot. Kétszázötvenhat bit kettő a kétszázötvenhatodikon különböző értéket tesz lehetővé: ez egy hetvennyolc tizedesjegyből álló szám, amely több nagyságrenddel nagyobb, mint az atomok becsült száma a megfigyelhető univerzumban. A világ minden szövege — minden könyv, minden e-mail, minden üzenet — ezen értékek egyikére esik. Annak a valószínűsége, hogy két különböző szöveg véletlenül megegyezzen, a gyakorlatban megkülönböztethetetlen a nullától.

Hogyan néz ki kódban

Zig nyelven, amelyen a Solo2-t támogató részeket írjuk, egy szöveg SHA-256 pecsétjének kiszámítása így néz ki:

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

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

Épp most kértük meg a Zig szabványos könyvtárát, hogy számítsa ki az idézőjelbe tett szöveg SHA-256 értékét. A hívás után a *resumen* változó tartalmazza azt a harminckét bájtot, amely a pecsétet nyers formájában alkotja; amikor hexadecimálisan megjelenik a képernyőn, az az a hatvannégy karakter, amely a cikk alján látható. Ha megváltoztatnánk a *Cuadernos Lacre* feliratot *Cuadernos lacre*-re — egy nagybetűvel kevesebb —, a pecsét teljesen megváltozna. Öt sorban ez az a központi tulajdonság, amely a többit fenntartja. Azok számára, akik szeretnék látni, hogyan működik belülről, a cikk végén közöljük az algoritmus olvasható változatát, lépésről lépésre fűzött kommentárokkal.

Miért nevezzük pecsétviasznak

A 15. és 19. század közötti európai levelezésben pecsétviasz zárta le a levelet. Egy csepp olvadt viasz, egy rányomott pecsét, és a levél megismételhetetlen módon meg volt jelölve. Nem védte meg a tartalmat az elszánt kíváncsiskodótól — a papírt át lehetett világítani, a viaszt fel lehetett törni —, de bizonyítékként szolgált. A lezárás bármilyen módosítása látható volt a címzett számára, még mielőtt kinyitotta volna a papírt. A pecsétviasz nem akadályozta meg a kárt; hanem jelezte azt.

Minden egyes Cuaderno törzsszövegének SHA-256 értéke ugyanezt a funkciót tölti be digitális formában. Ha a cikknek akár egyetlen szava is megváltozna a közzététel pillanata és az olvasás pillanata ket között, a szöveg alján lévő hexadecimális pecsét már nem egyezne meg az előtted lévő szöveg SHA-256 értékével. Bármely olvasó öt sornyi kóddal ellenőrizhetné ezt. A kiadvány nem tudja újraírni a történetét anélkül, hogy a pecsét ne árulná el. Nem véd a kár ellen; hanem ellenőrizhetővé teszi azt.

Amit a hash nem tud

Néha négy olyan feladatra kérik a SHA-256-ot, amely nem az ő asztala:

1. **Titkosítás.** A hash tömörít; nem rejt el. Ha azt szeretnéd, hogy a szöveg ne legyen olvasható, titkosítanod kell, nem hash-elned.
2. **Szerző hitelesítése.** A hash nem mondja meg, ki írta a szöveget, csak azt, hogy melyik szöveget hash-elték. A szerzőség társításához kriptográfiai aláírásra van szükség a hash felett, nem pedig magára a hash-re.
3. **Jelszavak tárolása.** Itt van egy csapda, amit érdemes megérteni. A SHA-256-ot úgy tervezték, hogy nagyon gyors legyen — ami sok mindenre jó, de erre rossz. Egy speciális hardverrel rendelkező támadó másodpercenként több milliárd jelszót próbálhat ki egy SHA-256 hash ellen, amíg meg nem találja a tiédet. A jelszavak mentéséhez szándékosan lassú kulcsszármaztatási függvényeket kell használni, mint például az Argon2, a scrypt vagy a bcrypt, kombinálva egy *sóval* (felhasználónként egyedi véletlenszerű adat, amely megakadályozza, hogy két azonos jelszóval rendelkező személynek ugyanaz legyen a hash értéke).
4. **A hash olvasása a szerző azonosítójaként.** Nem az. A hash a tartalmat azonosítja. Ha két ember a *szia* szót hash-eli SHA-256-tal, mindketten ugyanazt a lenyomatot kapják — és ez a központi tulajdonság, nem pedig hiba: ha különböző lenyomatok lennének, nem tudnánk ellenőrizni a közzétett és a kapott tartalom egyezését.

Hol bukkan fel a SHA-256 a mindennapjaidban

Még ha nem is látod, a SHA-256 támogatja nagy részét annak, amit naponta használsz az interneten. A Bitcoin blokklánca úgy épül fel, hogy minden blokk SHA-256 értékét a következőhöz láncolják; egy múltbeli blokk módosítása az egész későbbi lánc újraszámítására kényszerít. A Git, a rendszer, amellyel a világ kódjainak felét verziózzák, minden commitot a teljes tartalmának SHA-256 (újabb verziókban) vagy elődje, a SHA-1 (régebbi verziókban) értéke alapján azonosít. A webhelyek azonosítását végző HTTPS tanúsítványokhoz SHA-256 ujjnyomat társul. A szoftverletöltéseket gyakran kíséri a fejlesztő által közzétett SHA-256 érték, hogy ellenőrizhesd, nem módosult-e a fájl útközben. És mint mondtuk, minden Cuaderno Lacre alján.

A szakmai olvasó számára

Négy operatív emlékeztető azoknak, akik rendszerekről döntenek vagy auditálnak:

1. A hash nem titkosítás. Ha egy szolgáltató a két kifejezést összekeveri a műszaki dokumentációjában, érdemes megkérdezni, pontosan mire gondol.
2. Jelszavak tárolására soha nem szabad önmagában SHA-256-ot használni. A SHA-256 túl gyors ehhez a feladathoz (lásd az *Amit a hash nem tud* 3. pontját). A jelenlegi szabvány az **Argon2id**: tervezésénél fogva lassú, a szerver kapacitása szerint konfigurálható, felhasználónként eltérő véletlenszerű *sóval* kombinálva.
3. Dokumentumok — szerződések, akták, fájlok — integritásához továbbra is a SHA-256 a hivatkozási szabvány. Ezt használják az EU-ban a minősített időbélyegző szolgáltatók.
4. Hosszú távú (évtizedes) megőrzéshez érdemes a SHA-256 mellé egy SHA-3-at vagy egy SHA-512-t is kiszámítani és archiválni; a kriptográfiai óvatosság azt javasolja, hogy évszázados archívumok esetén ne hagyatkozzunk egyetlen függvényre.

Technikailag ezt az iterált struktúrát – ahol a köztes állapot megmarad a bemeneti blokkok között – **Merkle-Damgård** konstrukciónak nevezik; ez az a minta, amelyen a SHA-1, a SHA-2 (beleértve a SHA-256-ot is) és sok más klasszikus hash-függvény alapul. A SHA-3 ezzel szemben elhagyja a Merkle-Damgårdot egy másfajta, *szivacs (sponge)* nevű architektúra javára.

Hogyan működik a SHA-256 lépésről lépésre, egyszerű szavakkal

Képzeld el, hogy felállítottad a világ legbonyolultabb dominópályáját: több ezer bábu, tucatnyi elágazás, mechanikus hidak és rámpák szelik át az egész szobát, gondosan, darabról darabra elhelyezve.

Ha megpöccinted az első bábu, a lánc pontos és megismételhető sorrendben dől el. Ugyanaz az elrendezés, ugyanaz a kezdő lökés → ugyanaz a végső minta a földön, újra és újra.

Itt jön az érdekesség: mozdíts el **egyetlen bábut** fél centiméterrel arrébb a kezdés előtt, és pöccintsd meg újra. Egy rámpa, aminek aktiválódnia kellene, mozdulatlan marad, egy híd nem dől le, egy másik elágazás lép működésbe. A földön lévő bábuk végső mintája teljesen felismerhetetlen lesz az elsőhöz képest.

A SHA-256 matematikailag pontosan ez a pálya. Az általad írt szöveg a bábuk kezdeti helyzete. Az algoritmus a lökés, amely elindítja a zuhatagot. A végső eredmény pedig – amit *hash*-nek hívunk – a földről készült állókép, miután minden megállt. Változtass meg egyetlen vesszőt az eredeti szövegben, és a kép gyökeresen más lesz. Ilyen egyszerű, és ilyen drasztikus.

1. lépés: A szöveg lefordítása bináris bábukra. A számítógépek nem értik a betűket; először számokká (ASCII), a számokat pedig bináris kódra (egyesekre és nullákra) fordítják. Minden betű 8 fehér vagy fekete bábura változik: az *A* az `01000001`, a *B* a `01000010`, a szóköz a `00100000`. Az egész szöveged – legyen az egy szó, egy szerződés vagy egy regény – fehér és fekete bábuk hosszú sorává válik.

2. lépés: Kitöltés a szabványos méretig. A pálya pontosan 512 bábuból álló *szakaszokban* dolgozza fel a sort. Ha az üzeneted nem éri el az 512 többszörösét, közvetlenül a szöveg után egy jelölőbábu kerül (`10000000` értékkel), majd nullák következnek a szakasz végéig. Minden szakasz utolsó 64 pozíciója a szöveg eredeti hosszának rögzítésére van fenntartva. Így a pálya mindig tudja, hol ért véget a valódi tartalom, és hol kezdődött a kitöltés.

3. lépés: A nyolc mesterbábu elhelyezése. A kezdés előtt **nyolc mesterbábut** helyezünk az asztalra pontos kezdőpozícióba. Ez a nyolc bábu nem titok: kezdeti értéküket egy nyilvános matematikai szabály határozza meg (az első nyolc prímszám – 2, 3, 5, 7, 11, 13, 17, 19 – négyzetgyöke, illetve az egyes gyökök tizedesjegyeinek első bitjei). Mindenki, a bolygó bármely pontján ugyanazzal a nyolc mesterbábuval kezd ugyanabban a pozícióban. Sorsuk az, hogy a zuhatag lökdösse és átalakítsa őket.

4. lépés: A nagy zuhatag: hatvannégy lökés-forduló. Itt kezdődik a látványosság. A szöveged első 512 bábuból álló szakaszát nekiütköztetjük a nyolc mesterbábunak. De nem egyszerre dőlnek el: a mechanizmus **hatvannégy egymást követő fordulót** hajt végre. Minden fordulóban három műveletet végez a bábukkal:

- **A körhinta (Tiovivo)** (rotáció). A bábuk körben mozognak: a jobb oldaliak átkerülnek a bal oldalra. Egyetlen bábu sem vész el vagy adódik hozzá; egyszerűen átrendeződnek, miközben tesznek egy teljes kört a körhintán. Ez az információ újraelosztásának olcsó és megfordítható módja.
- **A logikai tölcsér (Embudo Lógico)** (XOR). A bábuk egy tölcséren haladnak át, amely párosával hasonlítja össze őket: ha mindkettő azonos színű, egy fehér jön ki; ha különböznek, egy fekete. Ez a bináris logika legegyszerűbb művelete, de a körhinta forgatásaival kombinálva rendkívül erőssé válik az információk veszteségmentes keveréséhez.
- **A túlcsordulás (Desborde)** (moduláris összeadás). Az eredményt összeadjuk egy *állandó lökőbábuval*, amelyet egy hatvannégy állandóból álló nyilvános listából veszünk (az első hatvannégy prímszám köbgyöke). Ha az összeadás során olyan extra bábuk keletkeznek, amelyek nem férnek el a tervezett 32 bábunyi helyen, ezek a felesleges bábuk elvesznek. Az asztalon csak 32 bábu számára van hely, egyetlen eggyel sem többnek.

A hatvannegyedik forduló végén a szöveged szakaszának minden egyes bábuja befolyásolta a nyolc mesterbábu helyzetét. A lökés energiája bejárta az egész pályát.

5. lépés: A következő szakasz hozzáadása (újraindítás nélkül). Ha a szöveged hosszú volt, és maradt még egy 512 bábuból álló szakasz feldolgozásra, **a pálya nem indul újra**. A nyolc mesterbábu úgy marad, ahogy az első zuhatag hagyta őket, és a második szakaszt ellenük indítjuk el újabb hatvannégy fordulóra. Olyan ez, mintha egy új, dominókkal teli szobát adnánk az éppen kidőlt szoba végéhez: az első rendetlensége teljes mértékben meghatározza, hogyan dől el a második.

6. lépés: Az állókép elkészítése. Amikor nincs több feldolgozandó szakasz, a zuhatag megáll. Megnézzük a nyolc mesterbábu végső helyzetét. Konfigurációjukat betűkből és számokból álló kódra fordítjuk a hexadecimális rendszerben. Az eredmény pontosan hatvannégy karakterből álló sorozat: ez a te SHA-256 pecséted.

A pálya felépítéséből négy tulajdonság magától értetődik:

1. **Determinizmus.** Ugyanaz a szöveg mindig ugyanazt a végső képet eredményezi a világ bármely számítógépén. Nulla véletlenszerűség, nulla meglepetés.
2. **Lavina-effekt.** Egy hozzáadott vessző, egy megváltoztatott nagybetű, egy elfelejtett ékezet: a kép teljesen felismerhetetlenné válik. Ez az a szélsőséges érzékenység, amelyet már az elején leírtunk.

3. **Egyirányúság.** A végső állókép alapján nem lehet rekonstruálni az eredeti szöveget. A rotációk, a tölcsérek és a túlcsordulások elpusztítanak minden irányadó információt arról, hogy *honnan jött az egyes bit*, és csak azt őrzik meg, hogy *mit adtak hozzá összesen*.
4. **Ütközésállóság.** A nyilvános kriptoanalízis huszonöt éve alatt senkinek sem sikerült két olyan különböző szöveget találnia, amelynek végső képe megegyezne. Ennek nehézsége pedig meghaladja bármely ésszerűen elképzelhető civilizáció számítási kapacitását.

A következő kódfüggelék pontosan ezt a hat lépést valósítja meg Zig nyelven. Most már úgy olvashatod, hogy tudod, mit jelent az egyes bitműveletek, ahelyett, hogy vakon elfogadnád a manipulációkat.

Technikai szószedet

Az olvasónak, aki meg akarja érteni, mit csinálnak az egyes műveletek. Nyugodtan ugord át: a cikk e nélkül is érthető marad.

ASCII és Unicode – hogyan lesznek a betűkből számok. A számítógépek nem betűket látnak; számokat látnak. Az **ASCII** (*American Standard Code for Information Interchange*, 1963) nevű szabvány minden billentyűzetkarakterhez egy konkrét számot rendel: az *A* a 65, a *B* a 66, az *a* a 97, a *0* a 48, a szóköz a 32, a vessző a 44. A modern rendszerek ezt kiterjesztik az **Unicode**-dal, amely a világ minden ábécéjének minden karakteréhez rendel egy számot: cirill, arab, kínai, japán, sőt még az emojikhoz is. Amikor leírsz egy karaktert vagy megnyitsz egy szöveges fájlt, a számítógép a háttérben lévő számot olvassa ki, nem a képernyőn látható alakot. A SHA-256 ezekkel a számokkal dolgozik, bármilyen szöveget számjegyek hosszú sorozataként kezel. Ezért tud ugyanazzal az algoritmussal lepecsételni egy spanyol nyelvű cikket, egy japán verset és egy bináris fájlt is.
XOR – a bitenkénti összehasonlító. A XOR (kiejtése: *„exor”*, az angol *exclusive or*, azaz „kizáró vagy” kifejezésből) az egyik legegyszerűbb művelet, amelyet egy számítógép két bináris számmal végezhet. Pozíciónként hasonlít össze két bitet, és az eredmény: **1**, ha a kettő közül pontosan az egyik 1-es (az egyik, de nem mindkettő), **0**, ha a kettő megegyezik (mindkettő 0 vagy mindkettő 1). Példa: `1010` és `1100` XOR-ja `0110`. Van egy figyelemre méltó tulajdonsága: megfordítható – ha ugyanazzal a kulccsal kétszer végzel XOR-műveletet, visszakapod az eredetit. Ezért ez a kriptográfia igáslova (caballo de batalla): információvesztés nélkül keveri össze a biteket, de az eredmény semmit nem árul el a bemenetekről, ha nem ismered az egyiket.
Hexadecimális – 16-os számrendszerben való számolás. A mindennapi számok szinte kivétel nélkül tíz számjegyet használnak (0–9). A hexadecimális tizenhatot használ: a megszokott 0–9 számjegyeket, plusz hat betűt, amelyek a következő értékeket képviselik: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Miért tizenhat? Mert a számítógépek négybites csoportokban gondolkodnak, és négy bit pontosan tizenhat különböző értéket képviselhet – így egy hexadecimális karakter tisztán négy bitnek felel meg. Egy SHA-256 ujjnyomat 256 bites, ami pontosan **64 hexadecimális karakternek** felel meg. Ha hagyományos tizedesrendszerben írnánk le, körülbelül 78 számjegyet foglalna el, és kényelmetlenebb lenne. A választás esztétikai és tömör; a háttérben lévő szám ugyanaz.
Bitforgatás – a bináris körhinta (tiovivo binario). Képzelj el egy hét izzóból álló sort, amelyek közül néhány világít (1), néhány pedig nem (0): `1 0 1 1 0 0 1`. Egy pozícióval jobbra forgatás abból áll, hogy vesszük a jobb szélső izzót, átvisszük a bal szélre, a többit pedig egy hellyel jobbra toljuk: `1 1 0 1 1 0 0`. Egyetlen izzó sem vész el vagy adódik hozzá: egyszerűen körbe-körbe táncolnak. A SHA-256 bitforgatást több százszor használ minden egyes számításnál; ez az információk állapotbeli újraelosztásának olcsó és veszteségmentes módja.
„*Nothing-up-my-sleeve*” állandók – miért prímszámokból származnak. A SHA-256 nyolc mesterbábuját és hatvannégy fordulóállandóját nem véletlenszerűen választották ki. Az első prímszámok négyzet- és köbgyökeiből származnak. Miért? Mert a tervezők *„semmi nincs a ruhaujjamban”* (*„nothing-up-my-sleeve”*) állandókat akartak: olyan értékeket, amelyek eredetét bárki ellenőrizheti. Ha valaki azt mondaná neked: *„bízz bennem: használd ezt a véletlenszerű 32 bites számot”*, joggal gyanakodnál rejtett gyengeségre vagy hátsó kapura. De bárki, akinek van számológépe, ellenőrizheti, hogy a 2 négyzetgyökének első 32 bitje a `0x6a09e667`. Az értékek matematikaiak, nyilvánosak és reprodukálhatóak: semmilyen rejtett trükk nem kerülhet a receptbe.

Függelék: SHA-256 olvasható kódban

Ez a függelék azoknak az olvasóknak szól, akik belülről szeretnék látni az algoritmust. Ez egy didaktikai megvalósítás Zig nyelven, amely a FIPS 180-4 specifikációt követi. Ez nem az a verzió, amelyet a Solo2 használ — az igazi a Zig szabványos könyvtárának `std.crypto.hash.sha2.Sha256` részében található, optimalizálva és auditálva. De az algoritmus ugyanaz: amit itt látsz, az lépésről lépésre az, ami történik, amikor az az öt karakteres hívás végrehajtja a munkáját.

```
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
}
```

Bármilyen más nyelven történő újraírás, amely ugyanazt a struktúrát követi — kezdeti konstansok, ütemezés kiterjesztése, hatvannégy kör, akkumuláció — produceolja ugyanazt az eredményt. Az algoritmusnak nincsenek titkai: értéke abban

rejlik, hogy a fent felsorolt tulajdonságok két évtizednyi, több ezer szem előtt zajló nyilvános kriptoanalízis után is fennállnak.

Ha visszatérsz a cikk aljára, egy hatvannégy karakteres hexadecimális pecsétet fogsz látni. Ez az általad éppen elolvasott szöveg SHA-256 értéke ezen a nyelven. Ha lefordítanánk a cikket, a pecsét más lenne; ha a magyar verzió egyetlen szava megváltozna, a magyar pecsét megváltozna. A pecsét nem védi a tartalmat — erre más eszközök szolgálnak —, hanem egyedileg azonosítja azt. És ez, bármilyen szerényen is hangzik, elég ahhoz, hogy a szerkesztői lánc egyetlen lépése se tudja észrevétlenül megváltoztatni az elmondottakat. Minden más — titkosítás, aláírás, azonosítás — erre az egyszerű ötletre épül.

Források és további olvasnivalók

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, 2015. augusztus. A SHA-2 család hivatalos specifikációja, beleértve a SHA-256-ot.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, 2011. május. Normatív verzió a megvalósítók számára.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Az 5. és 6. fejezet a hash függvényekkel és azok jogos és jogosulatlan felhasználásával foglalkozik.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Gyakorlati példa a SHA-256 használatára a blokkok láncolásához egy konstrukciójánál fogva megváltoztathatatlan struktúrában.
- 910/2014/EU rendelet (eIDAS) — a minősített időbélyegző-szolgáltatók keretrendszere. Az EU-ban kibocsátott minősített elektronikus aláírások és pecsétek hivatkozási függvénye a SHA-256.
- Referencia-megvalósítás Zig nyelven: `std.crypto.hash.sha2.Sha256` a nyelv hivatalos tárolójában (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). Ez az optimalizált és auditált verzió, amelyet a Solo2 valójában használ. Hasznos a függelékben található didaktikai megvalósítással való összevetéshez.

← ElőzőCUADERNOS_LIST_SCHREMS_TITLEKövetkező →CUADERNOS_LIST_KILLSWITCH_TITLE

Legutóbbi olvasmányok

- CUADERNOS_LIST_PREGUNTAS_TITLE
- CUADERNOS_LIST_SELFHOST_TITLE
- CUADERNOS_LIST_IDENTIDAD_TITLE

Vigye magával ezt a cikket, ahová csak szüksége van rá.

↓ Markdown ↓ Egyszerű szöveg ↓ PDF

A fájl letöltődik az Ön eszközére. Onnan elmentheti, importálhatja a Solo2-be, vagy megoszthatja bárhol. A Cuadernos nem dönt Ön helyett a fájl sorsáról.

Viaszpecsét · SHA-256 8aee4499a2e172d34243bb71ef5984679149fd36675b065f5ebb9fd6d18cbc56

Cuadernos Lacre · A Menzuri Gestión S.L. kiadványa ·
írta R.Eugenio · szerkesztette a Solo2 csapata.