# Vad SHA-256 egentligen är

Ett matematiskt fingeravtryck som ryms i sextiofyra tecken och som ändras helt om ett enda kommatecken i originaltexten flyttas. Varför vi kallar det för ett digitalt lacksigill.

**För att förstå det rätt:** Tänk dig en maskin som läser en text och returnerar en sekvens på 64 tecken. Om texten är identisk blir sekvensen identisk. Om du flyttar på ett enda kommatecken blir sekvensen en helt annan. Den sekvensen är det digitala sigillacket.

## Den enkla idén bakom det tekniska namnet

Föreställ dig att det finns en maskin med en enda springa och en enda skärm. Genom springan matar du in en text: ett ord, en mening, en hel roman. På skärmen visas, ögonblicket efteråt, en sekvens på exakt sextiofyra tecken. Denna sekvens kallar vi professionella läsare för *hash* eller *kryptografisk sammanfattning*; för den allmänna läsaren kan vi tills vidare kalla det ett matematiskt fingeravtryck av texten, precis som ett fingeravtryck är det för en person.

Om du matar in samma text två gånger visar maskinen samma fingeravtryck båda gångerna. Om du matar in en något annorlunda text —ett enda flyttat kommatecken, en stor bokstav som blir liten— visar maskinen ett helt annat fingeravtryck än det första. Inte likt: annorlunda. Dessa två egenskaper tillsammans —determinism och känslighet— är den enkla idén. Allt annat med SHA-256 är maskineriet som ser till att de efterlevs väl.

Det är värt att från början säga vad maskinen inte gör. Den krypterar inte texten. Den döljer den inte. Den sparar den inte. Maskinen tittar på texten, beräknar fingeravtrycket och glömmer sedan texten. Fingeravtrycket tillåter inte att man återskapar texten som producerade det; det tillåter bara, givet en kandidattext, att kontrollera om den stämmer överens med originalet eller inte. Det är därför vi säger att det är en sammanfattning *åt ett håll*: man går dit, men man återvänder inte.

## En hash är inte detsamma som kryptering

Förvirringen är vanlig och bör redas ut: att kryptera och att hasha är olika operationer. Att kryptera består i att transformera en text så att endast innehavaren av nyckeln kan återställa den till sin ursprungliga form. Att hasha består i att producera ett fingeravtryck av texten från vilket originaltexten aldrig kan återvinnas, varken med eller utan nyckel. Den första är reversibel genom design; den andra är irreversibel genom design.

Den praktiska konsekvensen spelar roll. När en applikation säger ”vi sparar ditt lösenord krypterat”, finns det någon som har nyckeln för att dekryptera det — applikationen själv, i vilket fall som helst. När en applikation säger ”vi sparar ditt lösenord hashat”, kan applikationen själv inte läsa originallösenordet även om den ville; den kan bara kontrollera om det du skriver producerar samma fingeravtryck igen. Den andra modellen, rätt utförd, är mycket att föredra framför den första för att lagra lösenord. Senare ska vi se varför ”rätt utförd” kräver något mer än bara SHA-256.

## De fyra egenskaperna som gör en kryptografisk hash användbar

En hashfunktion som förtjänar adjektivet *kryptografisk* uppfyller fyra egenskaper:

1. **Determinism.** Samma indata ger alltid samma fingeravtryck.
2. **Lavineffekt.** En liten ändring i indata ger ett helt annat fingeravtryck, utan synlig likhet med det föregående.
3. **Motstånd mot invertering.** Givet ett fingeravtryck är det inte beräkningsmässigt genomförbart att hitta texten som producerade det.

4. **Kollisionsbeständighet.** Det är inte beräkningsmässigt genomförbart att hitta två olika texter som producerar samma fingeravtryck.

”Inte beräkningsmässigt genomförbart” betyder inte ”matematiskt omöjligt”. Det betyder att kostnaden i tid, energi och pengar för att uppnå det överstiger med många storleksordningar summan av all rimligt tillgänglig beräkningskapacitet. För SHA-256 mäts den gränsen i tusentals miljarder år även för de mest optimistiska scenarierna med specialiserad hårdvara. Vilket, för läsarens praktiska syften, är detsamma som ”det går inte”.

# SHA-256, mer specifikt

Namnet säger allt. SHA är förkortningen för *Secure Hash Algorithm*: säker hash-algoritm. Siffran 256 anger storleken på fingeravtrycket i bitar: tvåhundrafemtiosex bitar, det vill säga trettiotvå bytes, som visade i hexadecimal form är de sextiofyra tecken som läsaren redan känner igen. Standarden publicerades av amerikanska NIST, det organ som normerar denna typ av funktioner, 2001 som en del av SHA-2-familjen; den nuvarande versionen av standarden, FIPS 180-4, är från 2015.

**För den som ännu inte har koll på vad bitar och bytes är:**

```
1 bit    →  0 eller 1    (en strömbrytare: på eller av)
1 byte   →  8 bitar      (256 möjliga kombinationer)
32 bytes →  256 bitar    (SHA-256-fingeravtrycket)
```

Siffran 256 i slutet av namnet anger fingeravtryckets storlek i bitar. I hexadecimal —ett talsystem med sexton symboler istället för tio— ryms dessa 256 bitar i exakt 64 tecken. Det är de 64 tecken du ser längst ner i varje Cuaderno.

Dimensionerna förtjänar ett ögonblick. Tvåhundrafemtiosex bitar tillåter två upphöjt till tvåhundrafemtiosex olika värden: ett tal med sjuttioåtta decimala siffror, flera storleksordningar större än det uppskattade antalet atomer i det observerbara universum. Varje text i världen —varje bok, varje e-postmeddelande, varje meddelande— hamnar på ett av dessa värden. Sannolikheten för att två olika texter sammanfaller av en slump är, för praktiska ändamål, omöjlig att skilja från noll.

# Hur det ser ut i kod

I Zig, språket som vi skriver de delar som bär upp Solo2 i, ser beräkningen av SHA-256-sigillet för en text ut så här:

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

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

Vi har just bett Zigs standardbibliotek att beräkna SHA-256 för texten inom citattecken. Efter anropet innehåller variabeln *resumen* de trettiotvå bytes som utgör sigillet i dess råa form; när de visas på skärmen i hexadecimalt format är de de sextiofyra tecken som visas längst ner i denna artikel. Om vi ändrade *Cuadernos Lacre* till *Cuadernos Lacre* (med en ändring i texten) skulle sigillet ändras helt. Det är, på fem rader, den centrala egenskapen som bär upp resten. För den som vill se hur det fungerar internt inkluderar vi i slutet av artikeln en läsbar version av algoritmen med kommentarer steg för steg.

# Varför vi kallar det för lacksigill

I den europeiska korrespondensen från 1400- till 1800-talet förseglade lacket brevet. En droppe smält vax, ett sigill pressat ovanpå, och brevet blev märkt på ett unikt sätt. Det skyddade inte innehållet från den beslutsamme tjuvkikaren —papperet kunde läsas mot ljuset, lacket kunde brytas— men det bevisade det. Varje ändring av förseglingen var synlig för mottagaren redan innan papperet öppnades. Lacket förhindrade inte skadan; det deklarerade den.

SHA-256 för innehållet i varje Cuaderno fyller samma funktion i sin digitala version. Om ett enda ord i artikeln ändrades mellan det ögonblick den publicerades och det ögonblick du läser den, skulle det hexadecimala sigillet längst ner i texten inte längre stämma överens med SHA-256 för texten du har framför dig. Vilken läsare som helst med fem rader kod skulle kunna kontrollera det. Publikationen kan inte skriva om sin historia utan att sigillet avslöjar det. Det skyddar inte mot skadan; det gör den verifierbar.

## Vad en hash inte är

Fyra användningsområden begärs ibland av SHA-256 som inte tillhör den:

1. **Kryptera.** En hash sammanfattar; den döljer inte. Om du vill att texten inte ska kunna läsas behöver du kryptera den, inte hasha den.
2. **Autentisera författaren.** En hash säger inte vem som skrev texten, bara vilken text som hasglades. För att associera författarskap krävs en kryptografisk signatur ovanpå hashen, inte hashen i sig.
3. **Lagra lösenord.** Här finns en fälla som är värd att förstå. SHA-256 är utformad för att vara mycket snabb —vilket är bra för många saker, men dåligt för detta. En angripare med specialiserad hårdvara kan prova miljarder lösenord per sekund mot en SHA-256-hash tills hen hittar ditt. För att spara lösenord bör man använda medvetet långsamma nyckelderiveringsfunktioner som Argon2, scrypt eller bcrypt, kombinerat med ett *salt* (en unik slumpmässig datapunkt per användare, som förhindrar att två personer med samma lösenord får samma hash).
4. **Läsa hashen som författaridentifierare.** Det är den inte. En hash identifierar innehållet. Om två personer hashar ordet *hej* med SHA-256 får båda samma sammanfattning — och det är den centrala egenskapen, inte en brist: om det vore olika sammanfattningar skulle vi inte kunna kontrollera överensstämmelse mellan det publicerade och det mottagna.

## Var SHA-256 dyker upp i din vardag

Även om du inte ser det, bär SHA-256 upp en stor del av det du använder dagligen på internet. Bitcoins blockkedja byggs genom att länka SHA-256 för varje block till nästa; att ändra ett tidigare block tvingar fram en omberäkning av hela den efterföljande kedjan. Git, systemet som halva världens kod versionshanteras med, identifierar varje commit genom SHA-256 (i nyare versioner) eller genom dess föregångare SHA-1 (i äldre versioner) av dess fullständiga innehåll. HTTPS-certifikaten som verifierar en webbplats identitet när du går in på den bär med sig ett associerat SHA-256-fingeravtryck. Programvarunedladdningar åtföljs ofta av en SHA-256 publicerad av utvecklaren så att du kan verifiera att filen inte ändrades på vägen. Och som vi har sagt, längst ner i varje Cuadernos Lacre.

## För den professionella läsaren

Fyra operativa påminnelser för den som beslutar om eller auditerar system:

1. Hash är inte kryptering. Om en leverantör förväxlar de två termerna i sin tekniska dokumentation bör man fråga vad hen menar exakt.
2. För att lagra lösenord bör man aldrig använda enbart SHA-256. SHA-256 är för snabb för denna uppgift (se punkt 3 i *Vad en hash inte är*). Den nuvarande standarden är **Argon2id**: långsam genom design, konfigurerbar efter serverns kapacitet, kombinerat med ett unikt slumpmässigt *salt* per användare.
3. För dokumentintegritet —kontrakt, akter, filer— är SHA-256 fortfarande referensstandard. Det är den som används av kvalificerade tidsstämplingstjänster i EU.
4. För långtidsbevarande (decennier) bör man även beräkna och arkivera en SHA-3 eller en SHA-512 tillsammans med SHA-256; kryptografisk försiktighet rekommenderar att man inte förlitar sig på en enda funktion under sekelgamla arkiv.

Tekniskt sett är denna itererade struktur – där det mellanliggande tillståndet bevaras mellan ingångsblocken – känd som en **Merkle-Damgård**-konstruktion, den modell som SHA-1, SHA-2 (inklusive SHA-256) och många andra klassiska hashfunktioner bygger på. SHA-3, däremot, överger Merkle-Damgård till förmån för en annan arkitektur som kallas *svamp* (sponge).

## Hur SHA-256 fungerar, steg för steg, med enkla ord

Föreställ dig att du har byggt världens mest avancerade dominobana: tusentals brickor, dussintals förgreningar, mekaniska broar och ramper som korsar hela rummet, noggrant placerade bit för bit.

Om du ger den första brickan en knuff faller kedjan i en exakt och repeterbar sekvens. Samma uppställning, samma första knuff → identiskt slutligt mönster av fallna brickor, gång på gång.

Här är det intressanta: flytta **en enda bricka** en halv centimeter åt sidan innan du börjar och knuffa igen. En ramp som skulle ha aktiverats förblir orörlig, en bro faller inte, en annan förgrening utlöses. Det slutliga mönstret av brickor på golvet är helt oigenkännligt jämfört med det första.

SHA-256 är matematiskt sett denna bana. Texten du skriver är brickornas startposition. Algoritmen är knuffen som frigör kaskaden. Och slutresultatet – det vi kallar *hash* – är ögonblicksbilden av golvet när allt har stannat. Ändra ett enda kommatecken i originaltexten och bilden blir radikalt annorlunda. Så enkelt, och så drastiskt.

**Steg 1. Översätt texten till binära brickor.** Datorer förstår inte bokstäver; de översätter dem först till siffror (ASCII) och siffrorna till binärt (ettor och nollor). Varje bokstav förvandlas till 8 vita eller svarta brickor: *A* är `01000001`, *B* är `01000010`, mellanslag är `00100000`. Hela din text – ett ord, ett kontrakt, en roman – blir en lång rad vita och svarta brickor.

**Steg 2. Fyll ut till standardstorlek.** Banan bearbetar raden i *block* om exakt 512 brickor. Om ditt meddelande inte når upp till en multipel av 512, läggs en markeringsbricka (den med värdet `10000000`) till direkt efter texten och sedan nollor tills blocket är fullständigt. De sista 64 positionerna i varje block är reserverade för att anteckna textens ursprungliga längd. På så sätt vet banan alltid var det verkliga innehållet slutade och var utfyllnaden började.

**Steg 3. Placera ut de åtta huvudbrickorna.** Innan vi börjar placerar vi ut **åtta huvudbrickor** på bordet i en exakt startposition. Dessa åtta brickor är ingen hemlighet: deras startvärde bestäms av en offentlig matematisk regel (kvadratrötterna ur de åtta första primtalen – 2, 3, 5, 7, 11, 13, 17, 19 – och de första bitarna i decimaldelen av varje rot). Alla, i alla hörn av världen, börjar med samma åtta huvudbrickor i samma position. Deras öde är att knuffas och omvandlas av lavinen.

**Steg 4. Den stora lavinen: sextiofyra rundor av knuffar.** Här börjar skådespelet. Det första blocket med 512 brickor från din text får kollidera med de åtta huvudbrickorna. Men de faller inte på en gång: mekanismen utför **sextiofyra på varandra följande rundor**. I varje runda utför den tre operationer med brickorna:

- **Karusellen** (rotation). Brickorna rör sig i cirkel: de till höger flyttas till vänster. Ingen bricka går förlorad eller läggs till; de omorganiseras helt enkelt genom att åka ett helt varv i karusellen. Det är ett billigt och reversibelt sätt att omfördela information.
- **Den logiska tratten** (XOR). Brickorna passerar genom en tratt som jämför dem två och twee: om båda har samma färg kommer en vit ut; om de är olika kommer en svart ut. Det är den enklaste operationen i binär logik, mas i kombination med karusellens rotationer blir den extremt kraftfull för att blanda information utan att förlora den.
- **Överloppet** (modulär addition). Resultat adderas med en *konstant knuffbricka* hämtad från en offentlig lista med sextiofyra konstanter (kubikrötterna ur de sextiofyra första primtalen). Om additionen genererar extra brickor som inte får plats i det avsedda utrymmet för 32 brickor, kastas dessa överflödiga brickor bort. Bordet har bara plats för 32 brickor, inte en enda till.

Vid slutet av runda sextiofyra har var och en av brickorna i blocket från din text påverkat de åtta huvudbrickornas position. Knuffens energi har färdats genom hela banan.

**Steg 5. Lägg till nästa block (utan återställning).** Om din text var lång och det finns ett till block med 512 brickor att bearbeta, **återställs inte banan**. De åtta huvudbrickorna förblir som den första lavinen lämnade dem, och det andra blocket slungas mot dem för att aktivera ytterligare sextiofyra rundor. Det är som att lägga till ett nytt rum fullt med domino i slutet av det som precis har fallit: oordningen i det första rummet avgör helt hur det andra kommer att falla.

**Steg 6. Ta den slutliga bilden.** När det inte finns fler block att bearbeta stannar lavinen. Vi tittar på den slutliga positionen för de åtta huvudbrickorna. Vi översätter deras konfiguration till en kod av bokstäver och siffror i hexadecimalt system. Resultatet är en sträng på exakt sextiofyra tecken: det är ditt SHA-256-sigill.

Fyra egenskaper följer naturligt av hur banan är uppbyggd:

1. **Determinism.** Samma text ger alltid samma slutliga bild, på vilken dator som helst i världen. Noll slumpmässighet, noll överraskningar.
2. **Lavineffekt.** Ett tillagt kommatecken, en ändrad stor bokstav, en glömd accent: den slutliga bilden blir helt oigenkännlig. Detta är den extrema känslighet som vi beskrev i början.
3. **Enkelriktad.** Utifrån den slutliga bilden kan du inte återskapa originaltexten. Rotationerna, trattarna och överloppen förstör all riktningsinformation om *varifrån varje bit kom* och bevarar bara *vad som adderades totalt*.
4. **Kollisionsbeständighet.** Under tjugofem år av offentlig kryptoanalys har ingen lyckats hitta två olika texter vars slutliga bilder sammanfaller. Och svårigheten att göra det ligger bortom den beräkningskapacitet som någon rimligt

tänkbar civilisation kan uppnå.

Kodbilagan som följer implementerar exakt dessa sex steg i Zig. Nu kan du läsa den med vetskap om vad varje bitoperation betyder, istället för att bara blint acceptera manipulationerna.

# Teknisk ordlista

*För läsaren som vill förstå vad varje operation gör. Du kan fritt hoppa över detta: artikeln går att förstå även utan det.*

**ASCII och Unicode — hur bokstäver blir siffror.** Datorer ser inte bokstäver; de ser siffror. En standard som kallas **ASCII** (*American Standard Code for Information Interchange*, från 1963) tilldelar varje tangentbordstecken ett specifikt nummer: *A* är 65, *B* är 66, *a* är 97, *0* är 48, mellanslag är 32, kommatecken är 44. Moderna system utökar detta med **Unicode**, som tilldelar ett nummer till varje tecken i alla världens alfabet: kyrilliska, arabiska, kinesiska, japanska och till och med emojis. När du skriver ett tecken eller öppnar en textfil läser datorn bakgrundsnumret, inte formen på skärmen. SHA-256 arbetar med dessa siffror och behandlar all text som en lång sekvens av siffror. Därför kan den försegla en artikel på spanska, en dikt på japanska och en binär fil med samma algoritm.
**XOR — jämförare bit för bit.** XOR (uttalas *”ex-or”*, från engelskans *exclusive or*, ”exklusivt eller”) är en av de enklaste operationerna en dator kan utföra med två binära tal. Den jämför två bitar position för position och returnerar: **1** om exakt en av de två är 1 (en men inte båda), **0** om båda är lika (båda 0 eller båda 1). Exempel: XOR för `1010` and `1100` är `0110`. Den har en anmärkningsvärd egenskap: den är reversibel – om du utför XOR två gånger med samma nyckel kommer du tillbaka till originalet. Därför är den arbetshästen inom kryptografi: den blandar bitar utan att förlora information, men resultatet avslöjar ingenting om indata om du inte känner till en av dem.
**Hexadecimalt — räkna i bas 16.** Nästan alla siffror i vardagen använder tio siffror (0–9). Det hexadecimala systemet använder sexton: de vanliga 0–9 plus sex bokstäver som representerar följande värden: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Varför sexton? För att datorer tänker i grupper om fyra bitar, och fyra bitar kan representera exakt sexton olika värden – så ett hexadecimalt tecken motsvarar snyggt och rent fyra bitar. Ett SHA-256-avtryck mäter 256 bitar, vilket är exakt **64 hexadecimala tecken**. Om vi skrev det med vanliga decimaltal skulle det ta upp cirka 78 siffror och vara mer obekvämt. Valet är estetiskt och kompakt; bakgrundsnumret är detsamma.
**Bitrotation — den binära karusellen.** Föreställ dig en rad med sju glödlampor, vissa tända (1) och andra släckta (0): `1 0 1 1 0 0 1`. Att rotera åt höger en position innebär att man tar lampan längst till höger, flyttar den till den vänstra kanten och flyttar de andra ett steg åt höger: `1 1 0 1 1 0 0`. Ingen glödlampa går förlorad eller läggs till: de dansar helt enkelt i en cirkel. SHA-256 använder bitrotation hundratals gånger i varje beräkning; det är ett billigt och förlustfritt sätt att omfördela informationen i tillståndet.
**Konstanter *”nothing-up-my-sleeve”* — varför de kommer från primtal.** De åtta huvudbrickorna och de sextiofyra rundkonstanterna i SHA-256 valdes inte slumpmässigt. De kommer från kvadrat- och kubikrötterna ur de första primtalen. Varför? För att dess designer ville ha konstanter *”utan något i ärmen”* (*”nothing-up-my-sleeve”*): värden vars ursprung vem som helst kan verifiera. Om någon sa till dig: *”lita on mig: använd det här slumpmässiga 32-bitarsnumret”*, skulle du med rätta misstänka en dold svaghet eller en bakdörr. Men vem som helst med en miniräknare kan kontrollera att de första 32 bitarna i kvadratroten ur 2 är `0x6a09e667`. Värdena är matematiska, offentliga och reproducerbara: inget dolt trick kan smyga sig in i receptet.

# Bilaga: SHA-256 i läsbar kod

Denna bilaga är för läsaren som vill se algoritmen inifrån. Det är en didaktisk implementering i Zig som följer specifikationen FIPS 180-4. Det är inte den version som Solo2 använder —den riktiga finns i `std.crypto.hash.sha2.Sha256` i Zigs standardbibliotek, optimerad och auditerad—. Men algoritmen är densamma: det du ser här är, steg för steg, vad som händer när det där anropet på fem tecken utför sitt arbete.

```
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
}
```

Varje omskrivning i ett annat språk som följer samma struktur —initiala konstanter, expansion av schemat, sextiofyra ronder, ackumulering— ger samma resultat. Algoritmen har inga hemligheter: dess värde ligger i att egenskaperna som räknades upp ovan fortfarande håller efter två decennier av offentlig kryptanalys av tusentals ögon.

---

*Om du går tillbaka till slutet av denna artikel ser du ett hexadecimalt sigill på sextiofyra tecken. Det är SHA-256 för texten du just har läst, på detta språk. Om vi översatte artikeln skulle sigillet vara ett annat; om ett ord i den svenska versionen ändrades skulle det svenska sigillet ändras. Sigillet skyddar inte innehållet —det finns andra verktyg för det— utan det identifierar det unikt. Och det, så blygsamt det än låter, räcker för att inget steg i den redaktionella kedjan ska kunna ändra det som sagts utan att det märks. Resten —kryptering, signering, identifiering— byggs ovanpå denna enkla idé.*

**Redaktionell not:** när dessa Cuadernos nämner företag eller produkter är det inte för att anklaga. De som bygger dem gör ett jobb som miljontals människor använder och uppskattar. Det vi pekar på är strukturellt — modellen, inte varumärket. Varumärken visas som exempel eftersom det är dessa läsaren känner igen.

## Källor och vidare läsning


- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, augusti 2015. Officiell specifikation för SHA-2-familjen, inklusive SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, maj 2011. Normativ version för implementatörer.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Kapitlen 5 och 6 täcker hashfunktioner och deras legitima och illegitima användningsområden.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Praktiskt exempel på användning av SHA-256 för att länka block i en struktur som är oföränderlig genom konstruktion.
- Förordning (EU) 910/2014 (eIDAS) — ramverk för kvalificerade tidsstämplingstjänster. SHA-256 är referensfunktionen för kvalificerade elektroniska signaturer och sigill utfärdade i EU.
- Referensimplementering i Zig: `std.crypto.hash.sha2.Sha256` i språkets officiella repository (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). Det är den optimerade och auditerade versionen som Solo2 faktiskt använder. Användbar för att jämföra med bilagans didaktiska implementering.



← FöregåendeSchrems II, fem år senareNästa →Kill switch och institutionell fångst


## Senaste läsning


- Analys · 18 maj 2026 Verklig vs skenbar integritet: Frågorna man bör ställa sig
- Analys · 18 maj 2026 Self-hosting som yrkespraxis
- Koncept · 18 maj 2026 De 24 orden: vad en kryptografisk identitet är


Ta med dig den här artikeln dit du behöver den.

↓ Markdown ↓ Klartext ↓ PDF

Filen laddas ner till din enhet. Därifrån kan du spara den, importera den till Solo2 eller dela den var du vill. Cuadernos bestämmer inte destinationen åt dig.

Lacksigill · SHA-256 40b161bcd993d0619584067a0ba25fe12a35ae0c472ef0ac657eed263f58b4eb


Cuadernos Lacre · En utgåva från Menzuri Gestión S.L. ·
skriven av R.Eugenio · redigerad av teamet bakom Solo2.