Czym naprawdę jest SHA-256

Matematyczny odcisk, który mieści się w sześćdziesięciu czterech znakach i zmienia się całkowicie, jeśli choćby jeden przecinek w oryginalnym tekście zostanie przesunięty. Dlaczego nazywamy go cyfrową pieczęcią lakową.

Prosta idea stojąca za nazwą techniczną

Wyobraź sobie, że istnieje maszyna z jedną szczeliną i jednym ekranem. Przez szczelinę wprowadzasz tekst: słowo, zdanie, całą powieść. Na ekranie pojawia się, chwilę później, sekwencja dokładnie sześćdziesięciu czterech znaków. Tę sekwencję, dla czytelnika profesjonalnego, nazywamy *hashem* lub *skrótem kryptograficznym*; dla czytelnika ogólnego możemy ją na razie nazywać matematycznym odciskiem tekstu, tak jak odcisk palca jest odciskiem osoby.

Jeśli wprowadzisz ten sam tekst dwa razy, maszyna za każdym razem pokaże ten sam odcisk. Jeśli wprowadzisz tekst nieco inny — jeden przesunięty przecinek, wielka litera zamieniona na małą — maszyna pokaże odcisk całkowicie inny od pierwszego. Nie podobny: inny. Te dwie właściwości razem — determinizm i czułość — to prosta idea. Cała reszta SHA-256 to mechanizm, który sprawia, że są one dobrze realizowane.

Warto od początku powiedzieć, czego maszyna nie robi. Nie szyfruje tekstu. Nie ukrywa go. Nie zapisuje go. Maszyna patrzy na tekst, oblicza odcisk i zapomina o tekście. Odcisk nie pozwala na zrekonstruowanie tekstu, który go wytworzył; pozwala jedynie, przy danym tekście kandydackim, sprawdzić, czy zgadza się on z oryginałem. Dlatego mówimy, że jest to skrót *jednokierunkowy*: wychodzi, ale nie wraca.

Hash to nie to samo co szyfrowanie

Często dochodzi do nieporozumień i warto je wyjaśnić: szyfrowanie i haszowanie to różne operacje. Szyfrowanie polega na przekształceniu tekstu w taki sposób, aby tylko posiadacz klucza mógł przywrócić go do pierwotnej formy. Haszowanie polega na wytworzeniu odcisku tekstu, z którego pierwotnego tekstu nie da się nigdy odzyskać, ani z kluczem, ani bez niego. Pierwsza operacja jest z założenia odwracalna; druga, z założenia, nieodwracalna.

Konsekwencje praktyczne są ważne. Gdy aplikacja mówi: „przechowujemy Twoje hasło zaszyfrowane”, istnieje ktoś, kto ma klucz do jego odszyfrowania — w każdym razie sama aplikacja. Gdy aplikacja mówi: „przechowujemy Twoje hasło haszowane”, sama aplikacja nie może odczytać oryginalnego hasła, nawet gdyby chciała; może jedynie sprawdzić, czy to, co wpisujesz, ponownie wytwarza ten sam odcisk. Drugi model, dobrze wykonany, jest znacznie lepszy od pierwszego do przechowywania haseł. Później zobaczymy, dlaczego „dobrze wykonany” wymaga czegoś więcej niż tylko samego SHA-256.

Cztery właściwości, które czynią skrót kryptograficzny użytecznym

Funkcja skrótu, która zasługuje na miano *kryptograficznej*, spełnia cztery właściwości:

1. **Determinizm.** Te same dane wejściowe zawsze dają ten sam odcisk.
2. **Efekt lawinowy.** Mała zmiana na wejściu daje całkowicie inny odcisk, bez widocznego podobieństwa do poprzedniego.
3. **Odporność na odwracanie.** Na podstawie odcisku nie jest możliwe obliczeniowo znalezienie tekstu, który go wytworzył.
4. **Odporność na kolizje.** Nie jest możliwe obliczeniowo znalezienie dwóch różnych tekstów dających ten sam odcisk.

„Nie jest możliwe obliczeniowo” nie oznacza „jest matematycznie niemożliwe”. Oznacza to, że koszt czasu, energii i pieniędzy potrzebny do osiągnięcia tego celu przekracza o rzędy wielkości sumę wszystkich racjonalnie dostępnych mocy obliczeniowych. W przypadku SHA-256 granicę tę mierzy się w tysiącach bilionów lat, nawet przy najbardziej optymistycznych założeniach dotyczących specjalistycznego sprzętu. Co dla praktycznych celów czytelnika jest tym samym, co „nie da się”.

SHA-256, konkretnie

Nazwa mówi wszystko. SHA to skrót od *Secure Hash Algorithm*: bezpieczny algorytm skrótu. Liczba 256 wskazuje rozmiar odcisku w bitach: dwieście pięćdziesiąt sześć bitów, czyli trzydzieści dwa bajty, które wyświetlane w systemie szesnastkowym dają sześćdziesiąt cztery znaki, które czytelnik już rozpoznaje. Standard został opublikowany przez amerykański NIST, organ normalizujący tego typu funkcje, w 2001 roku jako część rodziny SHA-2; obecna wersja standardu, FIPS 180-4, pochodzi z 2015 roku.

Dla tych, którzy jeszcze nie wiedzą, czym są bity i bajty:

```
1 bit    →  0 lub 1       (przełącznik: włączony lub wyłączony)
1 bajt   →  8 bitów       (256 możliwych kombinacji)
32 bajty →  256 bitów     (odcisk SHA-256)
```

Liczba 256 na końcu nazwy określa rozmiar odcisku w bitach. W systemie szesnastkowym — systemie numeracji z szesnastoma symbolami zamiast dziesięciu — te 256 bitów mieści się w dokładnie 64 znakach. To są te same 64 znaki, które widzisz u dołu każdego Cuaderno.

Wymiary zasługują na chwilę uwagi. Dwieście pięćdziesiąt sześć bitów pozwala na dwa do potęgi dwieście pięćdziesiąt sześć różnych wartości: liczba z siedemdziesięcioma ośmioma cyframi dziesiętnymi, o kilka rzędów wielkości większa niż szacowana liczba atomów w obserwowalnym wszechświecie. Każdy tekst na świecie — każda książka, każdy e-mail, każda wiadomość — przypada na jedną z tych wartości. Prawdopodobieństwo, że dwa różne teksty zbiegną się przypadkowo, jest w praktyce nieodróżnialne od zera.

Jak to wygląda w kodzie

W Zig, języku, w którym piszemy elementy wspierające Solo2, obliczanie pieczęci SHA-256 tekstu wygląda następująco:

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

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

Właśnie poprosiliśmy bibliotekę standardową Zig o obliczenie SHA-256 tekstu w cudzysłowie. Po wywołaniu zmienna *resumen* zawiera trzydzieści dwa bajty, które tworzą pieczęć w jej surowej formie; gdy są one wyświetlane na ekranie w systemie szesnastkowym, dają sześćdziesiąt cztery znaki widoczne u dołu tego artykułu. Gdybyśmy zmienili *Cuadernos Lacre* na *Cuadernos lacre* — o jedną wielką literę mniej — pieczęć zmieniłaby się całkowicie. To właśnie w pięciu liniach zawarta jest centralna właściwość, która wspiera całą resztę. Dla tych, którzy chcą zobaczyć, jak to działa wewnętrznie, na końcu artykułu zamieszczamy czytelną wersję algorytmu z komentarzami krok po kroku.

Dlaczego nazywamy to pieczęcią lakową

W korespondencji europejskiej od XV do XIX wieku list zamykano lakiem. Kropla roztopionego laku, odciśnięta na niej pieczęć i list zostawał oznaczony w sposób niepowtarzalny. Nie chroniło to treści przed zdeterminowanym podglądaczem — papier można było przeczytać pod światło, lak można było złamać — ale czyniło to widocznym. Każda zmiana zamknięcia była widoczna dla odbiorcy jeszcze przed otwarciem papieru. Lak nie zapobiegał szkodzie; on ją deklarował.

SHA-256 treści każdego Cuaderno pełni tę samą funkcję w wersji cyfrowej. Gdyby choć jedno słowo w artykule zmieniło się między momentem jego publikacji a momentem, w którym go czytasz, szesnastkowa pieczęć u dołu tekstu nie zgadzałaby się już z SHA-256 tekstu, który masz przed sobą. Każdy czytelnik dysponujący pięcioma liniami kodu mógłby to sprawdzić. Publikacja nie może napisać swojej historii na nowo bez ujawnienia tego przez pieczęć. Nie chroni ona przed szkodą; czyni ją weryfikowalną.

Czym hash nie jest

Od SHA-256 oczekuje się czasem czterech zastosowań, które do niego nie należą:

1. **Szyfrowanie.** Hash streszcza; nie ukrywa. Jeśli chcesz, aby tekstu nie dało się odczytać, musisz go zaszyfrować, a nie zahaszować.
2. **Uwierzytelnianie autora.** Hash nie mówi, kto napisał tekst, a jedynie jaki tekst został zahaszowany. Aby powiązać autorstwo, potrzebny jest podpis kryptograficzny nad hashem, a nie sam hash.
3. **Przechowywanie haseł.** Tutaj kryje się pułapka, którą warto zrozumieć. SHA-256 został zaprojektowany tak, aby był bardzo szybki — co jest dobre dla wielu rzeczy, ale złe dla tej jednej. Atakujący ze specjalistycznym sprzętem może sprawdzać miliardy haseł na sekundę względem hasha SHA-256, aż znajdzie Twoje. Do zapisywania haseł należy używać celowo wolnych funkcji derywacji klucza, takich jak Argon2, scrypt lub bcrypt, w połączeniu z *solą* (unikalnymi losowymi danymi dla każdego użytkownika, co zapobiega sytuacji, w której dwie osoby z tym samym hasłem mają ten sam hash).
4. **Odczytywanie hasha jako identyfikatora autora.** Tak nie jest. Hash identyfikuje treść. Jeśli dwie osoby zahaszują słowo *cześć* za pomocą SHA-256, obie otrzymają ten sam skrót — i to jest centralna właściwość, a nie wada: gdyby były to różne skróty, nie moglibyśmy sprawdzić zgodności między tym, co opublikowano, a tym, co otrzymano.

Gdzie SHA-256 pojawia się w Twoim codzinnym życiu

Nawet jeśli tego nie widzisz, SHA-256 wspiera znaczną część tego, czego używasz na co dzień w Internecie. Łańcuch bloków Bitcoina budowany jest poprzez łączenie SHA-256 każdego bloku z następnym; zmiana przeszłego bloku wymusza ponowne obliczenie całego późniejszego łańcucha. Git, system, w którym wersjonowany jest kod połowy świata, identyfikuje każde zatwierdzenie (commit) poprzez SHA-256 (w nowszych wersjach) lub przez jego poprzednika SHA-1 (w starszych wersjach) jego pełnej treści. Certyfikaty HTTPS, które weryfikują tożsamość witryny po wejściu na nią, mają powiązany odcisk SHA-256. Pobieranym plikom oprogramowania często towarzyszy SHA-256 opublikowany przez dewelopera, abyś mógł zweryfikować, czy plik nie został zmieniony po drodze. I, jak powiedzieliśmy, u dołu każdego Cuadernos Lacre.

Dla profesjonalnego czytelnika

Cztery przypomnienia operacyjne dla tych, którzy podejmują decyzje lub audytują systemy:

1. Hash to nie szyfrowanie. Jeśli dostawca myli te dwa terminy w swojej dokumentacji technicznej, warto zapytać, co dokładnie ma na myśli.
2. Do przechowywania haseł nigdy nie należy używać samego SHA-256. SHA-256 jest zbyt szybki do tego zadania (zobacz punkt 3 w *Czym hash nie jest*). Obecnym standardem jest **Argon2id**: wolny z założenia, konfigurowalny w zależności od możliwości serwera, w połączeniu z losową *solą*, inną dla każdego użytkownika.
3. W przypadku integralności dokumentów — umów, akt, plików — SHA-256 pozostaje standardem referencyjnym. Jest to standard używany przez kwalifikowane znaczniki czasu w UE.
4. W przypadku długoterminowego przechowywania (dziesięciolecia) warto obliczyć i zarchiwizować również SHA-3 lub SHA-512 obok SHA-256; kryptograficzna roztropność zaleca, aby nie polegać na jednej funkcji w przypadku archiwów stuletnich.

Technicznie rzecz biorąc, ta iteracyjna struktura — gdzie stan pośredni jest zachowywany między blokami wejściowymi — znana jest jako konstrukcja **Merkle-Damgård**, wzorzec, na którym opierają się SHA-1, SHA-2 (w tym SHA-256) i wiele innych klasycznych funkcji skrótu. SHA-3 z kolei rezygnuje z Merkle-Damgård na rzecz innej architektury zwanej *gąbką* (sponge).

Jak działa SHA-256, krok po kroku, prostymi słowami

Wyobraź sobie, że zbudowałeś najbardziej skomplikowany tor domina na świecie: tysiące klocków, dziesiątki rozgałęzień, mechaniczne mosty i rampy przecinające cały pokój, starannie układane element po elemencie.

Jeśli popchniesz pierwszy klocek, łańcuch upada w precyzyjnej i powtarzalnej sekwencji. Ten sam układ, to samo początkowe popchnięcie → identyczny końcowy wzór przewróconych klocków, raz za razem.

Oto co jest interesujące: przesuń **tylko jeden klocek** o pół centymetra w bok przed rozpoczęciem i popchnij ponownie. Rampa, która powinna zadziałać, pozostaje martwa, most nie opada, uruchamia się inne rozgałęzienie. Końcowy wzór klocków na podłodze jest całkowicie nierozpoznawalny w porównaniu z pierwszym.

SHA-256 to z matematycznego punktu widzenia właśnie ten tor. Tekst, który wpisujesz, to początkowe pozycje klocków. Algorytm to popchnięcie wyzwalające kaskadę. A ostateczny rezultat — to, co nazywamy *hashem* — to zdjęcie podłogi, gdy wszystko się zatrzyma. Zmień jeden jedyny przecinek w oryginalnym tekście, a zdjęcie będzie radykalnie inne. To takie proste i takie drastyczne.

Krok 1. Przetłumaczenie tekstu na binarne klocki. Komputery nie rozumieją liter; najpierw tłumaczą je na liczby (ASCII), a liczby na system binarny (jedynki i zera). Każda litera zamienia się w 8 białych lub czarnych klocków: *A* to `01000001`, *B* to `01000010`, spacja to `00100000`. Twój cały tekst — słowo, umowa, powieść — staje się długim rzędem białych i czarnych klocków.

Krok 2. Wypełnienie do standardowego rozmiaru. Tor przetwarza rzędy w *blokach* po dokładnie 512 klocków. Jeśli twoja wiadomość nie stanowi wielokrotności 512, dodaje się klocek znacznikowy (ten o wartości `10000000`) zaraz po tekście, a następnie zera, aż blok będzie pełny. Ostatnie 64 pozycje każdego bloku są zarezerwowane do zapisania oryginalnej długości tekstu. W ten sposób tor zawsze wie, gdzie skończyła się prawdziwa treść, a zaczęło wypełnienie.

Krok 3. Umieszczenie ośmiu głównych klocków. Zanim zaczniemy, kładziemy na stole **osiem głównych klocków** w precyzyjnym położeniu początkowym. Te osiem klocków to żadna tajemnica: ich początkowa wartość jest ustalona przez publiczną regułę matematyczną (pierwiastki kwadratowe z pierwszych ośmiu liczb pierwszych — 2, 3, 5, 7, 11, 13, 17, 19 — i pierwsze bity części dziesiętnej każdego pierwiastka). Wszyscy, w każdym zakątku planety, zaczynają z tymi samymi ośmioma głównymi klockami w tej samej pozycji. Ich przeznaczeniem jest zostać popchniętymi i przetransformowanymi przez lawinę.

Krok 4. Wielka lawina: sześćdziesiąt cztery rundy popchnięć. Tu zaczyna się widowisko. Pierwszy blok 512 klocków twojego tekstu zderza się z ośmioma głównymi klockami. Ale nie przewracają się one od razu: mechanizm wykonuje **sześćdziesiąt cztery kolejne rundy**. W każdej rundzie z klockami wykonywane są trzy operacje:

- **Karuzela** (rotacja). Klocki poruszają się w kółko: te po prawej przechodzą na lewo. Żaden klocek nie ginie ani nie jest dodawany; po prostu zmieniają swoje położenie, kręcąc się wokół karuzeli. To tani i odwracalny sposób na redystrybucję informacji.
- **Logiczny Lejek** (XOR). Klocki przechodzą przez lejek, który porównuje je parami: jeśli oba są tego samego koloru, wypada biały; jeśli się różnią, wypada czarny. To najprostsza operacja w logice binarnej, ale w połączeniu z obrotami karuzeli staje się potężnym narzędziem do mieszania informacji bez jej utraty.
- **Przepełnienie** (dodawanie modularne). Wynik jest sumowany z *stałym klockiem popychającym* wziętym z publicznej listy sześćdziesięciu czterech stałych (pierwiastków sześciennych z pierwszych sześćdziesięciu czterech liczb pierwszych). Jeśli suma generuje dodatkowe klocki, które nie mieszczą się w przewidzianej przestrzeni 32 klocków, te nadmiarowe klocki są odrzucane. Stół ma miejsce tylko na 32 klocki, ani jednego więcej.

Pod koniec sześćdziesiątej czwartej rundy każdy z klocków z bloku twojego tekstu wpłynął na pozycję ośmiu głównych klocków. Energia z popchnięcia przebyła cały tor.

Krok 5. Dodanie kolejnego bloku (bez restartu). Jeśli twój tekst był długi i pozostał jeszcze jeden blok 512 klocków do przetworzenia, **tor nie jest resetowany**. Osiem głównych klocków zostaje tak, jak pozostawiła je pierwsza lawina, a drugi blok zostaje wystrzelony w ich stronę, by uruchomić kolejne sześćdziesiąt cztery rundy. To tak, jakby dodać nowy pokój pełen domina na końcu tego, który właśnie upadł: nieład z pierwszego całkowicie determinuje sposób upadku drugiego.

Krok 6. Zrobienie końcowego zdjęcia. Gdy nie ma już więcej bloków do przetworzenia, lawina się zatrzymuje. Patrzymy na ostateczną pozycję, w jakiej znalazło się osiem głównych klocków. Tłumaczymy ich konfigurację na kod składający się z liter i cyfr w systemie szesnastkowym. Rezultatem jest ciąg dokładnie sześćdziesięciu czterech znaków: to twoja pieczęć SHA-256.

Cztery właściwości wynikają bezpośrednio z tego, jak tor jest zbudowany:

1. **Determinizm.** Ten sam tekst zawsze daje to samo zdjęcie końcowe, na każdym komputerze na świecie. Zero przypadkowości, zero niespodzianek.
2. **Efekt lawinowy.** Dodany przecinek, zmieniona wielka litera, zapomniany akcent: zdjęcie staje się całkowicie nierozpoznawalne. To jest ta ekstremalna wrażliwość, którą opisaliśmy już na początku.

3. **Jednokierunkowość.** Mając zdjęcie końcowe, nie możesz zrekonstruować oryginalnego tekstu. Rotacje, lejki i przepełnienia niszczą wszelkie kierunkowe informacje o tym, *skąd pochodził dany bit* i zachowują jedynie to, *co łącznie dodano*.
4. **Odporność na kolizje.** Przez dwadzieścia pięć lat publicznej kryptoanalizy nikt nie zdołał znaleźć dwóch różnych tekstów, których zdjęcia końcowe by się zgadzały. A trudność w dokonaniu tego wykracza poza możliwości obliczeniowe jakiejkolwiek cywilizacji, którą można sobie racjonalnie wyobrazić.

Poniższy dodatek z kodem realizuje dokładnie te sześć kroków w języku Zig. Teraz możesz go przeczytać, wiedząc, co oznacza każda operacja na bitach, zamiast przyjmować manipulacje w ciemno.

Słowniczek techniczny

Dla czytelnika, który chce zrozumieć, co robi każda operacja. Możesz to swobodnie pominąć: artykuł jest zrozumiały również bez tego.

ASCII i Unicode — jak litery stają się liczbami. Komputery nie widzą liter; widzą liczby. Standard o nazwie **ASCII** (*American Standard Code for Information Interchange*, z 1963 roku) przypisuje każdemu znakowi na klawiaturze konkretną liczbę: *A* to 65, *B* to 66, *a* to 97, *0* to 48, spacja to 32, przecinek to 44. Nowoczesne systemy rozszerzają to za pomocą **Unicode**, który przypisuje liczbę każdemu znakowi z każdego alfabetu na świecie: cyrylicy, arabskiego, chińskiego, japońskiego, a nawet emotikonom. Gdy wpisujesz znak lub otwierasz plik tekstowy, komputer czyta w tle liczbę, a nie kształt na ekranie. SHA-256 pracuje na tych liczbach, traktując każdy tekst jako długi ciąg cyfr. Dlatego tym samym algorytmem można opieczętować artykuł po hiszpańsku, wiersz po japońsku i plik binarny.
XOR — komparator bit po bicie. XOR (wymawiane *«exor»*, od angielskiego *exclusive or*, «alternatywa wykluczająca») to jedna z najprostszych operacji, jakie komputer może wykonać na dwóch liczbach binarnych. Porównuje ona dwa bity pozycja po pozycji i zwraca: **1**, jeśli dokładnie jeden z nich to 1 (jeden, ale nie oba), **0**, jeśli oba są takie same (oba 0 lub oba 1). Przykład: XOR dla `1010` i `1100` to `0110`. Ma ona niezwykłą właściwość: jest odwracalna — jeśli wykonasz XOR dwa razy z tym samym kluczem, wrócisz do oryginału. Dlatego jest to koń pociągowy kryptografii: miesza bity bez utraty informacji, ale wynik nie ujawnia niczego o wejściach, jeśli nie znasz żadnego z nich.
System szesnastkowy — liczenie przy podstawie 16. Prawie wszystkie codzienne liczby używają dziesięciu cyfr (0-9). System szesnastkowy (heksadecymalny) używa szesnastu: zwykłych 0-9 plus sześciu liter reprezentujących następujące wartości: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Dlaczego szesnaście? Ponieważ komputery myślą w grupach po cztery bity, a cztery bity mogą reprezentować dokładnie szesnaście różnych wartości — więc jeden znak szesnastkowy odpowiada równo cztery bitom. Skrót (hash) SHA-256 ma długość 256 bitów, co daje dokładnie **64 znaki szesnastkowe**. Gdybyśmy zapisali to w zwykłym systemie dziesiętnym, zajęłoby to około 78 cyfr i byłoby znacznie mniej poręczne. Wybór jest estetyczny i zwięzły; ukryta wartość jest ta sama.
Rotacja bitowa — binarna karuzela. Wyobraź sobie rząd siedmiu żarówek, z których jedne się świecą (1), a inne nie (0): `1 0 1 1 0 0 1`. Obrót o jedną pozycję w prawo polega na wzięciu najbardziej wysuniętej w prawo żarówki, przeniesieniu jej na skrajnie lewy koniec i przesunięciu pozostałych o jedno miejsce w prawo: `1 1 0 1 1 0 0`. Żadna żarówka nie zostaje stracona ani dodana: po prostu tańczą w kółko. SHA-256 wykorzystuje obroty bitowe setki razy w każdym obliczeniu; to tani i bezstratny sposób na zmianę rozkładu informacji wewnątrz stanu.
Stałe *«nothing-up-my-sleeve»* — dlaczego pochodzą z liczb pierwszych. Osiem głównych klocków i sześćdziesiąt cztery stałe rundy SHA-256 nie zostały wybrane losowo. Pochodzą one z pierwiastków kwadratowych i sześciennych pierwszych liczb pierwszych. Dlaczego? Ponieważ ich projektanci chcieli stałych *«bez asów w rękawie»*: wartości, których pochodzenie każdy może zweryfikować. Gdyby ktoś powiedział ci: *«Zaufaj mi: użyj tej losowej liczby 32-bitowej»*, słusznie podejrzewałbyś ukrytą słabość lub tylne drzwi (backdoor). Ale każdy z kalkulatorem może sprawdzić, że pierwsze 32 bity pierwiastka kwadratowego z 2 to `0x6a09e667`. Wartości te są matematyczne, publiczne i powtarzalne: do przepisu nie wkradnie się żadna ukryta pułapka.

Dodatek: SHA-256 w czytelnym kodzie

Ten dodatek jest przeznaczony dla czytelnika, który chce zobaczyć algorytm od środka. Jest to dydaktyczna implementacja w języku Zig, zgodna ze specyfikacją FIPS 180-4. Nie jest to wersja używana przez Solo2 — prawdziwa znajduje się w `std.crypto.hash.sha2.Sha256` w bibliotece standardowej Zig, zoptymalizowana i audytowana. Ale algorytm jest ten sam: to, co tu widzisz, to krok po kroku to, co dzieje się, gdy to pięcioznakowe wywołanie wykonuje swoją pracę.

```
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
}
```

Każde przepisanie na inny język z zachowaniem tej samej struktury — stałe początkowe, rozszerzenie harmonogramu, sześćdziesiąt cztery rundy, akumulacja — daje ten sam wynik. Algorytm nie ma tajemnic: jego wartość polega na tym, że wymienione wyżej właściwości pozostają nienaruszone po dwóch dekadach publicznej kryptoanalizy przeprowadzonej przez tysiące oczu.

Jeśli wrócisz do dołu tego artykułu, zobaczysz szesnastkową pieczęć o długości sześćdziesięciu czterech znaków. Jest to SHA-256 tekstu, który właśnie przeczytałeś, w tym języku. Gdybyśmy przetłumaczyli artykuł, pieczęć byłaby inna; gdyby zmieniło się choć jedno słowo w wersji hiszpańskiej, pieczęć hiszpańska uległaby zmianie. Pieczęć nie chroni treści — do tego służą inne narzędzia — lecz identyfikuje ją w sposób jednoznaczny. I to, jakkolwiek skromnie by to nie brzmiało, wystarczy, aby żaden etap łańcucha redakcyjnego nie mógł zmienić tego, co powiedziano, bez bycia zauważonym. Cała reszta — szyfrowanie, podpisywanie, identyfikacja — opiera się na tej prostej idei.

Źródła i dodatkowa lektura

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, sierpień 2015. Oficjalna specyfikacja rodziny SHA-2, w tym SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, maj 2011. Wersja normatywna dla implementatorów.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Rozdziały 5 i 6 omawiają funkcje skrótu oraz ich legalne i nielegalne zastosowania.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Praktyczny przykład użycia SHA-256 do łączenia bloków w strukturę niezmienną z konstrukcji.
- Rozporządzenie (UE) 910/2014 (eIDAS) — ramy dla kwalifikowanych znaczników czasu. SHA-256 jest referencyjną funkcją dla kwalifikowanych podpisów i pieczęci elektronicznych wydawanych w UE.
- Implementacja referencyjna w Zig: `std.crypto.hash.sha2.Sha256` w oficjalnym repozytorium języka (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). Jest to zoptymalizowana i audytowana wersja, której faktycznie używa Solo2. Przydatna do porównania z implementacją dydaktyczną z dodatku.

← PoprzedniCUADERNOS_LIST_SCHREMS_TITLENastępny →CUADERNOS_LIST_KILLSWITCH_TITLE

Ostatnie lektury

- CUADERNOS_LIST_PREGUNTAS_TITLE
- CUADERNOS_LIST_SELFHOST_TITLE
- CUADERNOS_LIST_IDENTIDAD_TITLE

Zabierz ten artykuł tam, gdzie go potrzebujesz.

↓ Markdown ↓ Zwykły tekst ↓ PDF

Plik zostanie pobrany na Twoje urządzenie. Stamtąd możesz go zapisać, zaimportować do Solo2 lub udostępnić w dowolnym miejscu. Cuadernos nie decyduje o miejscu docelowym za Ciebie.

Pieczęć lakowa · SHA-256 8fb104cd4669ff2a9a342b207ca2fc30e1fd44fc0e2c7424e8874d5b09cfaa14

Cuadernos Lacre · Publikacja Menzuri Gestión S.L. ·
napisana przez R.Eugenio · redagowana przez zespół Solo2.