Какво всъщност е SHA-256

Математически отпечатък, който се побира в шестдесет и четири знака и се променя изцяло, ако се премести дори една запетая в оригиналния текст. Защо го наричаме цифров восъчен печат.

Простата идея зад техническото име

Представете си, че съществува машина с един слот и един екран. През слота вкарвате текст: дума, изречение, цял роман. На екрана се появява, мигове по-късно, последователност от точно шестдесет и четири знака. Тази последователност за професионалния читател наричаме *hash* или *криптографско резюме*; за широкия читател можем да я наречем засега математически отпечатък на текста, както пръстовият отпечатък е такъв за човека.

Ако въведете един и същ текст два пъти, машината ще покаже един и същ отпечатък и двата пъти. Ако въведете малко по-различен текст —една преместена запетая, главна буква, която става малка— машината показва напълно различен отпечатък от първия. Не подобен, а различен. Тези две свойства заедно —детерминизъм и чувствителност— са простата идея. Всичко останало в SHA-256 е механизмът, който ги кара да се изпълняват добре.

Важно е да кажем от самото начало какво машината не прави. Тя не шифрова текста. Не го скрива. Не го пази. Машината поглежда текста, изчислява отпечатъка и забравя за текста. Отпечатъкът не позволява да се възстанови текстът, който го е произвел; той само позволява при даден кандидат-текст да се провери дали съвпада с оригинала или не. Ето защо казваме, че това е *еднопосочно* резюме: отива се, но не се връща.

Хеш не е същото като шифроване

Объркването е често срещано и е добре да се изясни: шифроването и хеширането са различни операции. Шифроването се състои в трансформиране на текст така, че само притежателят на ключа да може да го върне в оригиналния му вид. Хеширането се състои в създаване на отпечатък на текста, от който оригиналният текст никога не може да бъде възстановен, нито с ключ, нито без него. Първото е обратимо по дизайн; второто е необратимо по дизайн.

Практическото следствие е важно. Когато едно приложение казва „пазим паролата ви шифрована“, има някой, който има ключа за дешифрирането ѝ — самото приложение във всеки случай. Когато едно приложение казва „пазим паролата ви хеширана“, самото приложение не може да прочете оригиналната парола, дори и да иска; то може само да провери дали това, което пишете, произвежда същия отпечатък отново. Вторият модел, направен добре, е много по-предпочитан от първия за съхранение на пароли. По-нататък ще видим защо „направен добре“ изисква нещо повече от SHA-256 чисто.

Четирите свойства, които правят криптографския хеш полезен

Хеш функция, която заслужава прилагателното *криптографска*, отговаря на четири свойства:

1. **Детерминизъм.** Един и същ вход винаги произвежда един и същ отпечатък.
2. **Ефект на лавината.** Малка промяна на входа произвежда напълно различен отпечатък, без видима прилика с предишния.
3. **Устойчивост на инверсия.** При даден отпечатък не е изчислително възможно да се намери текстът, който го е произвел.

4. **Устойчивост на колизии.** Не е изчислително възможно да се намерят два различни текста, които да произвеждат един и същ отпечатък.

„Не е изчислително възможно“ не означава „математически невъзможно“. Това означава, че цената по отношение на време, енергия и пари за постигането му надвишава с порядъци сумата от целия разумен наличен изчислителен капацитет. За SHA-256 тази граница се измерва в хиляди милиарди години дори за най-оптимистичните подходи със специализиран хардуер. Което за практическите цели на читателя е същото като „не може“.

SHA-256 конкретно

Името казва всичко. SHA е съкращение от *Secure Hash Algorithm*: алгоритъм за сигурен хеш. Числото 256 показва размера на отпечатъка в битове: двеста петдесет и шест бита, тоест тридесет и два байта, които показани в шестнадесетичен вид са шестдесет и четирите знака, които читателят вече познава. Стандартът е публикуван от американския NIST, органът, който нормализира този тип функции, през 2001 г. като част от семейството SHA-2; текущата версия на стандарта, FIPS 180-4, е от 2015 г.

За тези, които все още нямат представа какво са битове и байтове:

```
1 бит    →  0 или 1        (ключ: включен или изключен)
1 байт   →  8 бита         (256 възможни комбинации)
32 байта →  256 бита      (отпечатъкът SHA-256)
```

Числото 256 в края на името казва размера на отпечатъка в битове. В шестнадесетична система —бройна система с шестнадесет символа вместо десет— тези 256 бита се побират точно в 64 знака. Това са 64-те знака, които виждате в долната част на всеки Cuaderno.

Размерите заслужават момент внимание. Двеста петдесет и шест бита позволяват две на степен двеста петдесет и шест различни стойности: число със седемдесет и осем десетични цифри, няколко порядъка по-голямо от прогнозния брой атоми в наблюдаваната вселена. Всеки текст по света —всяка книга, всеки имейл, всяко съобщение— попада върху една от тези стойности. Вероятността два различни текста да съвпаднат по случайност е за практически цели неразличима от нула.

Как изглежда в код

В Zig, езика, на който пишем частите, поддържащи Solo2, изчисляването на печат SHA-256 на текст изглежда така:

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

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

Току-що поискахме от стандартната библиотека на Zig да изчисли SHA-256 на текста в кавички. След извикването променливата *resumen* съдържа тридесет и двата байта, които съставляват печата в неговата сурова форма; когато се показват на екрана в шестнадесетичен вид, те са шестдесет и четирите знака, които се появяват в долната част на тази статия. Ако сменим *Cuadernos Lacre* на *Cuadernos lacre* —една главна буква по-малко— целият печат ще се промени. Това е в пет реда централното свойство, което поддържа останалото. За тези, които искат да видят как работи вътрешно, в края на статията включваме четима версия на алгоритъма с коментари стъпка по стъпка.

Защо го наричаме восъчен печат

В европейската кореспонденция от петнадесети до деветнадесети век червеният восък е запечатвал писмото. Капка разтопен восък, печат, притиснат върху него, и писмото оставало маркирано по неповторим начин. Това не предпазвало съдържанието от решителния любопитко —хартията можела да се прочете срещу светлината, восъкът можел да се счупи— но го доказвало. Всяка промяна на затварянето била видима за получателя още преди отварянето на писмото. Восъкът не предотвратявал щетата; той я обявявал.

SHA-256 на тялото на всеки Cuaderno изпълнява същата функция в своята цифрова версия. Ако дори една дума от статията се промени между момента, в който е публикувана, и момента, в който я четете, шестнадесетичният печат

в долната част на текста вече няма да съвпада с SHA-256 на текста пред вас. Всеки читател с пет реда код би могъл да го провери. Изданието не може да пренапише своята история, без печатът да го издаде. Той не предпазва от щети; той ги прави проверими.

Какво не е хеш

Четири приложения понякога се изискват от SHA-256, които не му съответстват:

1. **Шифроване.** Хешът резюмира, но не скрива. Ако искате текстът да не може да се чете, трябва да го шифровате, а не да го хеширате.
2. **Удостоверяване на автора.** Хешът не казва кой е написал текста, а само кой текст е бил хеширан. За асоцииране на авторство е необходим криптографски подпис върху хеша, а не хешът сам по себе си.
3. **Съхранение на пароли.** Тук има капан, който е добре да се разбере. SHA-256 е проектиран да бъде много бърз —което е добре за много неща, но лошо за това. Нападател със специализиран хардуер може да изпробва милиарди пароли в секунда срещу хеш SHA-256, докато намери вашата. За съхранение на пароли трябва да се използват нарочно бавни функции за извличане на ключове като Argon2, scrypt или bcrypt, комбинирани със *sol* (уникални случайни данни за всеки потребител, които предотвратяват двама души с една и съща парола да имат един и същ хеш).
4. **Четене на хеша като идентификатор на автора.** Не е такъв. Хешът идентифицира съдържанието. Ако двама души хешират думата *hola* с SHA-256, и двамата получават едно и също резюме — и това е централното свойство, а не дефект: ако бяха различни резюмета, не бихме могли да проверим съвпадението между публикуваното и полученото.

Къде се появява SHA-256 в ежедневието ви

Дори и да не го виждате, SHA-256 поддържа голяма част от това, което използвате ежедневно в интернет. Блокчейнът на Bitcoin се изгражда чрез свързване на SHA-256 на всеки блок към следващия; промяната на минал блок налага преизчисляване на цялата последваща верига. Git, системата, с която се версиира кодът на половин свят, идентифицира всяко потвърждение (commit) чрез SHA-256 (в новите версии) или чрез неговия предшественик SHA-1 (в по-старите версии) на цялото му съдържание. Сертификатите HTTPS, които проверяват идентичността на уебсайт, когато влизате, носят асоцииран отпечатък SHA-256. Изтеглянето на софтуер често се придружава от SHA-256, публикуван от разработчика, за да проверите дали файлът не е бил променен по пътя. И както казахме, в долната част на всеки Cuadernos Lacre.

За професионалния читател

Четири оперативни напомняния за тези, които вземат решения или одитират системи:

1. Хешът не е шифроване. Ако доставчик бърка двата термина в техническата си документация, е добре да попитате какво точно има предвид.
2. За съхранение на пароли никога не трябва да се използва SHA-256 чисто. SHA-256 е твърде бърз за тази задача (вижте точка 3 от *Какво не е хеш*). Настоящият стандарт е **Argon2id**: бавен по дизайн, конфигурируем според капацитета на сървъра, комбиниран с различна случайна *sol* за всеки потребител.
3. За интегритет на документи —договори, преписки, файлове— SHA-256 продължава да бъде референтният стандарт. Той е този, който се използва от квалифицираните доставчици на удостоверителни услуги за времеви печати в ЕС.
4. За дългосрочно съхранение (десетилетия) е добре да се изчислява и архивира също SHA-3 или SHA-512 заедно с SHA-256; криптографската предпазливост препоръчва да не се разчита на една единствена функция по време на вековно архивиране.

Технически тази итеративна структура — при която междинното състояние се запазва между входните блокове — е известна като конструкция на **Merkle-Damgård**, моделът, на който се основават SHA-1, SHA-2 (включително SHA-256) и много други класически хеш функции. За разлика от тях, SHA-3 изоставя Merkle-Damgård в полза на различна архитектура, наречена *sponge* (гъба).

Как работи SHA-256, стъпка по стъпка, на разбираем език

Представете си, че сте сглобили най-сложната домино верига в света: хиляди плочки, десетки разклонения, механични мостове и рампи, пресичащи цялата стая, внимателно поставени плочка по плочка.

Ако докоснете първата плочка, веригата пада в прецизна и повтаряема последователност. Същата сглобка, същото първо докосване → идентичен краен модел от паднали плочки, отново и отново.

Тук е интересното: преместете **само една плочка** с половин сантиметър встрани преди началото и докоснете отново. Рампа, която е трябвало да се активира, остава инертна, мост не пада, задейства се съвсем различно разклонение. Крайният модел на плочките на пода е напълно неразпознаваем в сравнение с първия.

Математически SHA-256 е точно тази верига. Текстът, който пишете, е първоначалното положение на плочките. Алгоритъмът е докосването, което освобождава каскадата. А крайният резултат — това, което наричаме *hash* (хеш) — е снимката на пода, когато всичко е спряло. Променете само една запетая в оригиналния текст и снимката ще бъде радикално различна. Толкова просто и толкова драстично.

Стъпка 1. Превеждане на текста в бинарни плочки. Компютрите не разбират от букви; те ги превеждат първо в числа (ASCII), а числата в бинарни (единици и нули). Всяка буква се превръща в 8 бели или черни плочки: *A* е `01000001`, *B* е `01000010`, интервалът е `00100000`. Целият ви текст — дума, договор, роман — се превръща в дълга редица от бели и черни плочки.

Стъпка 2. Допълване до стандартния размер. Веригата обработва редицата на *сегменти* от точно 512 плочки. Ако съобщението ви не достига кратно на 512, непосредствено след текста се добавя маркираща плочка (със стойност `10000000`), а след това нули до завършване на сегмента. Последните 64 позиции на всеки сегмент са запазени за записване на оригиналната дължина на текста. Така веригата винаги знае къде свършва реалното съдържание и къде започва пълнежът.

Стъпка 3. Поставяне на осемте главни плочки. Преди да започнем, поставяме на масата **осем главни плочки** в прецизна начална позиция. Тези осем плочки не са тайна: началната им стойност е фиксирана от общоизвестно математическо правило (квадратните корени на първите осем прости числа — 2, 3, 5, 7, 11, 13, 17, 19 — и първите битове на десетичната част на всеки корен). Всеки по всяко кътче на планетата започва с едни и същи осем главни плочки в една и съща позиция. Съдбата им е да бъдат избутани и трансформирани от лавината.

Стъпка 4. Голямата лавина: шестдесет и четири кръга от избутвания. Тук започва зрелището. Първият сегмент от 512 плочки от вашия текст се сблъсква с осемте главни плочки. Но те не падат наведнъж: механизмът изпълнява **шестдесет и четири последователни кръга**. Във всеки кръг се извършват три операции с плочките:

- **Въртележката** (ротация). Плочките се движат в кръг: тези отдясно отиват вляво. Нито една плочка не се губи и не се добавя; те просто се пренареждат, правейки пълно завъртане на въртележката. Това е евтин и обратим начин за преразпределяне на информацията.
- **Логическата фуния** (XOR). Плочките преминават през фуния, която ги сравнява две по две: ако и двете са от един и същи цвят, излиза бяла; ако са различни, излиза черна. Това е най-простата операция в бинарната логика, но комбинирана с ротациите на въртележката, тя става изключително мощна за смесване на информация без загубата ѝ.
- **Преливането** (модулно събиране). Резултатът се събира с *плочка с константен тласък*, взета от публичен списък с шестдесет и четири константи (кубичните корени на първите шестдесет и четири прости числа). Ако събирането генерира излишни плочки, които не се побират в предвиденото пространство от 32 плочки, тези излишни плочки се отхвърлят. Масата има място само за 32 плочки, нито една повече.

В края на кръг шестдесет и четири всяка от плочките от сегмента на вашия текст е повлияла на позицията на осемте главни плочки. Енергията на тласъка е преминала през цялата верига.

Стъпка 5. Добавяне на следващия сегмент (без рестартиране). Ако текстът ви е бил дълъг и остава друг сегмент от 512 плочки за обработка, **веригата не се рестартира**. Осемте главни плочки остават така, както ги е оставила първата лавина, и вторият сегмент се изстрелва срещу тях, за да задейства други шестдесет и четири кръга. Това е като да добавите нова стая, пълна с домино, в края на тази, която току-що е паднала: безпорядъкът в първата обуславя изцяло как ще падне втората.

Стъпка 6. Направа на финалната снимка. Когато вече не останат сегменти за обработка, лавината спира. Гледаме финалната позиция, в която са останали осемте главни плочки. Превеждаме конфигурацията им в код от

букви и цифри в шестнадесетична система. Резултатът е низ от точно шестдесет и четири знака: това е вашият SHA-256 печат.

Четири свойства произтичат сами по себе си от начина, по който е сглобена веригата:

1. **Детерминизъм.** Един и същи текст винаги произвежда една и съща финална снимка на всеки компютър в света. Нулева произволност, нула изненади.
2. **Ефект на лавината.** Добавена запетая, променена главна буква, забравено ударение: снимката се оказва напълно неразпознаваема. Това е екстремната чувствителност, която вече описахме в началото.
3. **Еднопосочност.** Като имате финалната снимка, не можете да възстановите оригиналния текст. Ротациите, фуниите и преливанията унищожават цялата насочена информация за това *откъде е дошъл всеки бит* и запазват само *какво е събрано общо*.
4. **Устойчивост на колизии.** За двадесет и пет години публичен криптоанализ никой не е успял да намери два различни текста, чиито финални снимки съвпадат. А трудността да се направи това е извън изчислителния обхват на всяка разумно представима цивилизация.

Приложеният кодов апендикс изпълнява точно тези шест стъпки на Zig. Сега можете да го прочетете, знаейки какво означава всяка битова операция, вместо да приемате манипулациите на сляпо.

Технически речник

За читателя, който иска да разбере какво прави всяка операция. Прескочете го свободно: статията се разбира и без него.

ASCII и Unicode — как буквите стават числа. Компютрите не виждат букви; те виждат числа. Стандарт, наречен **ASCII** (*American Standard Code for Information Interchange*, от 1963 г.), присвоява на всеки символ от клавиатурата специфично число: *A* е 65, *B* е 66, *a* е 97, *0* е 48, интервалът е 32, запетаята е 44. Модерните системи го разширяват с **Unicode**, който присвоява число на всеки символ от всяка азбука в света: кирилица, арабска, китайска, японска и дори емотикони. Когато пишете символ или отваряте текстов файл, компютърът чете скритото число, а не формата на екрана. SHA-256 работи върху тези числа, третирайки всеки текст като дълга последователност от цифри. Ето защо той може да запечата статия на испански, стихотворение на японски и бинарен файл със същия алгоритъм.
XOR — сравнение бит по бит. XOR (произнася се *„екс-ор“*, от английското *exclusive or*, „изключващо или“) е една от най-простите операции, които компютърът може да извърши с две бинарни числа. Сравнява два бита позиция по позиция и връща: **1**, ако точно един от двата е 1 (единия, но не и двата), **0**, ако двата са еднакви (и двата 0 или и двата 1). Пример: XOR на `1010` и `1100` е `0110`. Има забележително свойство: той е обратим — ако направите XOR два пъти с един и същи ключ, се връщате към оригинала. Ето защо той е основният инструмент на криптографията: смесва битове без загуба на информация, но резултатът не разкрива нищо за входовете, ако не познавате единия от тях.
Шестнадесетична система — броене в основа 16. Почти всички числа в ежедневието използват десет цифри (0-9). Шестнадесетичната система използва шестнадесет: обичайните 0-9 плюс шест букви, които представляват следните стойности: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Защо шестнадесет? Защото компютрите мислят в групи от четири бита, а четири бита могат да представляват точно шестнадесет различни стойности — така един шестнадесетичен символ съответства чисто на четири бита. Един SHA-256 отпечатък е 256 бита, което са точно **64 шестнадесетични знака**. Ако го напишем в обикновена десетична система, той би заемал около 78 цифри и би бил по-неудобен. Изборът е естетичен и компактен; скритото число е същото.
Ротация на битове — бинарната въртележка. Представете си редица от седем лампички, едни светещи (1), а други угаснали (0): `1 0 1 1 0 0 1`. Ротирането надясно с една позиция се състои в това да вземете най-дясната лампичка, да я преместите в левия край и да изместите останалите с едно място надясно: `1 1 0 1 1 0 0`. Нито една лампичка не се губи и не се добавя: те просто танцуват в кръг. SHA-256 използва ротация на битове стотици пъти във всяко изчисление; това е евтин начин без загуби за преразпределяне на информацията в състоянието.
Константи *„nothing-up-my-sleeve“* — защо произлизат от прости числа. Осемте главни плочки и шестдесет и четирите кръгови константи на SHA-256 не са избрани случайно. Те произлизат от квадратните и кубичните корени на първите прости числа. Защо? Защото дизайнерите му са искали константи *„без нищо скрито в ръкава“*: стойности, чийто произход всеки може да провери. Ако някой ви каже *„довери ми се: използвай това произволно 32-битово число“*, основателно бихте се усъмнили в скрита слабост или задна врата. Но всеки с калкулатор може да провери, че първите 32 бита от квадратния корен на 2 са `0x6a09e667`. Стойностите са математически, публични и възпроизводими: никаква скрита клопка не може да се промъкне в рецептата.

Приложение: SHA-256 в четим код

Това приложение е за читателя, който иска да види алгоритъма отвътре. Това е дидактическа реализация на Zig, която следва спецификацията FIPS 180-4. Това не е версията, която Solo2 използва —истинската е в `std.crypto.hash.sha2.Sha256` от стандартната библиотека на Zig, оптимизирана и одитирана—. Но алгоритъмът е същият: това, което виждате тук, е стъпка по стъпка какво се случва, когато това извикване от пет знака изпълнява своята работа.

```
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
}
```

Всяко пренаписване на друг език, което следва същата структура —начални константи, разширяване на графика, шестдесет и четири кръга, натрупване— произвежда същия резултат. Алгоритъмът няма тайни: неговата стойност се състои в това, че изброените по-горе свойства продължават да се поддържат след две десетилетия публичен криптоанализ от хиляди очи.

Ако се върнете в долната част на тази статия, ще видите шестнадесетичен печат от шестдесет и четири знака. Това е SHA-256 на текста, който току-що прочетохте, на този език. Ако преведем статията, печатът ще бъде друг; ако се промени една дума от българската версия, българският печат ще се промени. Печатът не предпазва съдържанието —за това има други инструменти— а го идентифицира еднозначно. И това, колкото и скромно да звучи, е достатъчно, за да не може нито една стъпка от редакционната верига да промени казаното, без това да се забележи. Останалите неща —шифроване, подписване, идентифициране— се изграждат върху тази проста идея.

Източници и допълнително четиво

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, август 2015 г. Официална спецификация на семейството SHA-2, включително SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, май 2011 г. Нормативна версия за разработчици.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Глави 5 и 6 обхващат хеш функциите и техните легитимни и нелегитимни употреби.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Практически пример за използване на SHA-256 за свързване на блокове в имутабилна по конструкция структура.
- Регламент (ЕС) 910/2014 (eIDAS) — рамка на квалифицираните доставчици на времеви печати. SHA-256 е референтната функция за квалифицираните електронни подписи и печати, издавани в ЕС.
- Референтна реализация в Zig: `std.crypto.hash.sha2.Sha256` в официалното хранилище на езика (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). Това е оптимизираната и одитирана версия, която всъщност Solo2 използва. Полезно за сравнение с дидактическата реализация в приложението.

← ПредишнаCUADERNOS_LIST_SCHREMS_TITLEСледваща →CUADERNOS_LIST_KILLSWITCH_TITLE

Скорошни четива

- CUADERNOS_LIST_PREGUNTAS_TITLE
- CUADERNOS_LIST_SELFHOST_TITLE
- CUADERNOS_LIST_IDENTIDAD_TITLE

Вземете тази статия със себе си навсякъде, където ви е необходима.

↓ Markdown ↓ Обикновен текст ↓ PDF

Файлът ще бъде изтеглен на вашето устройство. Оттам можете да го запазите, импортирате в Solo2 или споделите където пожелаете. Cuadernos не решава дестинацията вместо вас.

Восъчен печат · SHA-256 835d1ad771b6a1b316a7a1c1f99c336fe860013c2c5b3f13588c1436ff0e5fb3

Cuadernos Lacre · Публикация на Menzuri Gestión S.L. ·
написана от R.Eugenio · редактирана от екипа на Solo2.

предпочитанието ви за светла/тъмна тема. Без тракери, без профилиране, без споделяне на данни. Ако искате да ни последвате: RSS.