Was SHA-256 wirklich ist

Ein mathematischer Fingerabdruck, der in vierundsechzig Zeichen passt und sich komplett ändert, wenn auch nur ein einziges Komma im Originaltext verschoben wird. Warum wir ihn digitales Siegellack-Siegel nennen.

Um es mal so zu sagen: Stell dir eine Maschine vor, die einen beliebigen Text liest und eine Folge von 64 Zeichen ausgibt. Wenn der Text identisch ist, ist die Folge identisch. Wenn du nur ein Komma verschiebst, ist die Folge eine völlig andere. Diese Folge ist der digitale Siegellack.

Die einfache Idee hinter dem technischen Namen

Stellen Sie sich vor, es gäbe eine Maschine mit einem einzigen Schlitz und einem einzigen Bildschirm. Durch den Schlitz führen Sie einen Text ein: ein Wort, einen Satz, einen ganzen Roman. Auf dem Bildschirm erscheint Augenblicke später eine Sequenz von genau vierundsechzig Zeichen. Diese Sequenz nennen wir für den professionellen Leser *Hash* oder *kryptografische Zusammenfassung*; für den allgemeinen Leser können wir sie vorerst als einen mathematischen Fingerabdruck des Textes bezeichnen, so wie der Fingerabdruck eines Menschen für ihn charakteristisch ist.

Wenn Sie denselben Text zweimal eingeben, zeigt die Maschine beide Male denselben Fingerabdruck. Wenn Sie einen leicht veränderten Text eingeben —ein verschobenes Komma, ein Großbuchstabe, der zum Kleinbuchstaben wird— zeigt die Maschine einen Fingerabdruck, der sich völlig vom ersten unterscheidet. Nicht ähnlich: anders. Diese beiden Eigenschaften zusammen —Determinismus und Sensitivität— sind die einfache Idee. Alles andere an SHA-256 ist der Mechanismus, der dafür sorgt, dass sie gut funktionieren.

Es ist wichtig, von Anfang an zu sagen, was die Maschine nicht tut. Sie verschlüsselt den Text nicht. Sie verbirgt ihn nicht. Sie speichert ihn nicht. Die Maschine betrachtet den Text, berechnet den Fingerabdruck und vergisst den Text. Der Fingerabdruck erlaubt es nicht, den Text zu rekonstruieren, der ihn erzeugt hat; er erlaubt es nur, bei einem vorliegenden Kandidatentext zu prüfen, ob er mit dem Original übereinstimmt oder nicht. Deshalb sagen wir, dass es eine *Einweg-Zusammenfassung* ist: man geht hin, man kehrt nicht zurück.

Ein Hash ist nicht dasselbe wie Verschlüsselung

Die Verwechslung ist häufig und es ist ratsam, sie auszuräumen: Verschlüsseln und Hashen sind unterschiedliche Operationen. Verschlüsseln besteht darin, einen Text so zu transformieren, dass nur der Besitzer des Schlüssels ihn in seine ursprüngliche Form zurückführen kann. Hashen besteht darin, einen Fingerabdruck des Textes zu erzeugen, aus dem der Originaltext niemals wiederhergestellt werden kann, weder mit noch ohne Schlüssel. Ersteres ist konstruktionsbedingt umkehrbar; Letzteres ist konstruktionsbedingt unumkehrbar.

Die praktische Konsequenz ist wichtig. Wenn eine Anwendung sagt: „Wir speichern Ihr Passwort verschlüsselt“, gibt es jemanden, der den Schlüssel zum Entschlüsseln hat — in jedem Fall die Anwendung selbst. Wenn eine Anwendung sagt: „Wir speichern Ihr Passwort gehasht“, kann die Anwendung selbst das Originalpasswort nicht lesen, selbst wenn sie wollte; sie kann nur prüfen, ob das, was Sie eingeben, wieder denselben Fingerabdruck erzeugt. Das zweite Modell ist, wenn es richtig gemacht wird, für die Speicherung von Passwörtern dem ersten Modell weit vorzuziehen. Später werden wir sehen, warum „richtig gemacht“ mehr erfordert als nur reines SHA-256.

Die vier Eigenschaften, die einen kryptografischen Hash nützlich machen

Eine Hash-Funktion, die das Adjektiv *kryptografisch* verdient, erfüllt vier Eigenschaften:

1. **Determinismus.** Dieselbe Eingabe erzeugt immer denselben Fingerabdruck.
2. **Lawineneffekt.** Eine kleine Änderung an der Eingabe erzeugt einen völlig anderen Fingerabdruck, ohne erkennbare Ähnlichkeit mit dem vorherigen.
3. **Pre-image-Resistenz.** Gegeben ein Fingerabdruck, ist es rechnerisch nicht machbar, den Text zu finden, der ihn erzeugt hat.
4. **Kollisionsresistenz.** Es ist rechnerisch nicht machbar, zwei verschiedene Texte zu finden, die denselben Fingerabdruck erzeugen.

„Rechnerisch nicht machbar“ bedeutet nicht „mathematisch unmöglich“. Es bedeutet, dass die Kosten an Zeit, Energie und Geld, um dies zu erreichen, die Summe aller vernünftigerweise verfügbaren Rechenkapazitäten um Größenordnungen übersteigen. Für SHA-256 wird diese Grenze in Billionen von Jahren gemessen, selbst für die optimistischsten Ansätze mit spezialisierter Hardware. Was für den praktischen Zweck des Lesers dasselbe ist wie „es geht nicht“.

SHA-256, ganz konkret

Der Name sagt alles. SHA ist die Abkürzung für *Secure Hash Algorithm*: sicherer Hash-Algorithmus. Die Zahl 256 gibt die Größe des Fingerabdrucks in Bits an: zweihundertsechsundfünfzig Bits, also zweiunddreißig Bytes, die in hexadezimaler Darstellung die vierundsechzig Zeichen sind, die der Leser bereits kennt. Der Standard wurde vom US-amerikanischen NIST veröffentlicht, dem Gremium, das diese Art von Funktionen normiert, im Jahr 2001 als Teil der SHA-2-Familie; die aktuelle Version des Standards, FIPS 180-4, stammt aus dem Jahr 2015.

Für diejenigen, die sich noch nicht bewusst sind, was Bits und Bytes sind:

```
1 Bit    →  0 oder 1        (ein Schalter: ein oder aus)
1 Byte   →  8 Bits         (256 mögliche Kombinationen)
32 Bytes →  256 Bits       (der SHA-256-Fingerabdruck)
```

Die Zahl 256 am Ende des Namens gibt die Größe des Fingerabdrucks in Bits an. Im Hexadezimalsystem —einem Zahlensystem mit sechzehn Symbolen statt zehn— passen diese 256 Bits in genau 64 Zeichen. Das sind die 64 Zeichen, die Sie am Ende jedes Cuaderno sehen.

Die Dimensionen verdienen einen Moment Aufmerksamkeit. Zweihundertsechsundfünfzig Bits ermöglichen zwei hoch zweihundertsechsundfünfzig verschiedene Werte: eine Zahl mit achtundsiebzig Dezimalstellen, mehrere Größenordener größer als die geschätzte Anzahl der Atome im beobachtbaren Universum. Jeder Text der Welt —jedes Buch, jede E-Mail, jede Nachricht— fällt auf einen dieser Werte. Die Wahrscheinlichkeit, dass zwei verschiedene Texte zufällig übereinstimmen, ist für praktische Zwecke nicht von Null zu unterscheiden.

Wie es im Code aussieht

In Zig, der Sprache, in der wir die Teile schreiben, die Solo2 tragen, sieht die Berechnung des SHA-256-Siegels eines Textes so aus:

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

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

Wir haben gerade die Standardbibliothek von Zig gebeten, den SHA-256 des Textes in Anführungszeichen zu berechnen. Nach dem Aufruf enthält die Variable *resumen* die zweiunddreißig Bytes, die das Siegel in seiner Rohform bilden; wenn sie auf dem Bildschirm hexadezimal angezeigt werden, sind es die vierundsechzig Zeichen, die am Ende dieses Artikels erscheinen. Wenn wir *Cuadernos Lacre* in *Cuadernos lacre* ändern würden —ein Großbuchstabe weniger— würde sich das gesamte Siegel ändern. Das ist in fünf Zeilen die zentrale Eigenschaft, die den Rest trägt. Für diejenigen, die sehen wollen, wie es intern funktioniert, fügen wir am Ende des Artikels eine lesbare Version des Algorithmus mit Kommentaren Schritt für Schritt bei.

Warum wir es Siegellack-Siegel nennen

In der europäischen Korrespondenz des 15. bis 19. Jahrhunderts verschloss Siegellack den Brief. Ein Tropfen geschmolzenes Wachs, ein darauf gedrücktes Siegel, und der Brief war auf unwiederholbare Weise markiert. Es schützte den Inhalt nicht vor dem entschlossenen Neugierigen —das Papier konnte gegen das Licht gelesen werden, das Wachs konnte gebrochen werden—, aber es machte es offensichtlich. Jede Veränderung des Verschlusses war für den Empfänger sichtbar, noch bevor er das Papier öffnete. Der Siegellack verhinderte den Schaden nicht; er deklarierte ihn.

Der SHA-256 des Inhalts jedes Cuaderno erfüllt in seiner digitalen Version die gleiche Funktion. Wenn sich nur ein einziges Wort des Artikels zwischen dem Zeitpunkt der Veröffentlichung und dem Zeitpunkt des Lesens ändern würde, würde das hexadezimale Siegel am Ende des Textes nicht mehr mit dem SHA-256 des Textes übereinstimmen, der vor Ihnen liegt. Jeder Leser könnte dies mit fünf Zeilen Code überprüfen. Die Publikation kann ihre Geschichte nicht umschreiben, ohne dass das Siegel sie verrät. Es schützt nicht vor dem Schaden; es macht ihn verifizierbar.

Was ein Hash nicht ist

Vier Verwendungszwecke werden manchmal von SHA-256 verlangt, die ihm nicht entsprechen:

1. **Verschlüsseln.** Ein Hash fasst zusammen; er verbirgt nicht. Wenn Sie möchten, dass der Text nicht gelesen werden kann, müssen Sie ihn verschlüsseln, nicht hashen.
2. **Den Autor authentifizieren.** Ein Hash sagt nicht aus, wer den Text geschrieben hat, sondern nur, welcher Text gehasht wurde. Um eine Urheberschaft zuzuordnen, ist eine kryptografische Signatur über dem Hash erforderlich, nicht der Hash allein.
3. **Passwörter speichern.** Hier gibt es eine Falle, die man verstehen sollte. SHA-256 ist so konzipiert, dass er sehr schnell ist —was für viele Dinge gut ist, aber schlecht für diesen Zweck. Ein Angreifer mit spezialisierter Hardware kann Milliarden von Passwörtern pro Sekunde gegen einen SHA-256-Hash testen, bis er Ihres findet. Um Passwörter zu speichern, müssen absichtlich langsame Schlüsselableitungsfunktionen wie Argon2, scrypt oder bcrypt verwendet werden, kombiniert mit einem *Salt* (einem eindeutigen Zufallswert pro Benutzer, der verhindert, dass zwei Personen mit demselben Passwort denselben Hash haben).
4. **Den Hash als Identifikator des Autors lesen.** Das ist er nicht. Ein Hash identifiziert den Inhalt. Wenn zwei Personen das Wort *hola* mit SHA-256 hashen, erhalten beide dieselbe Zusammenfassung — und das ist die zentrale Eigenschaft, kein Defekt: Wären es unterschiedliche Zusammenfassungen, könnten wir die Übereinstimmung zwischen dem Veröffentlichten und dem Empfangenen nicht prüfen.

Wo SHA-256 in Ihrem Alltag vorkommt

Auch wenn Sie es nicht sehen, trägt SHA-256 einen großen Teil dessen, was Sie täglich im Internet nutzen. Die Blockchain von Bitcoin wird aufgebaut, indem der SHA-256 jedes Blocks mit dem nächsten verkettet wird; die Änderung eines vergangenen Blocks zwingt dazu, die gesamte nachfolgende Kette neu zu berechnen. Git, das System, mit dem der Code von halben Welt versioniert wird, identifiziert jeden Commit durch den SHA-256 (in neueren Versionen) oder durch seinen Vorgänger SHA-1 (in älteren Versionen) seines vollständigen Inhalts. Die HTTPS-Zertifikate, die die Identität einer Website beim Aufruf verifizieren, tragen einen zugehörigen SHA-256-Fingerabdruck. Software-Downloads werden oft von einem vom Entwickler veröffentlichten SHA-256 begleitet, damit Sie überprüfen können, ob die Datei unterwegs verändert wurde. Und, wie gesagt, am Ende jedes Cuadernos Lacre.

Für den professionellen Leser

Vier operative Erinnerungen für diejenigen, die über Systeme entscheiden oder diese auditieren:

1. Hash ist keine Verschlüsselung. Wenn ein Anbieter in seiner technischen Dokumentation beide Begriffe verwechselt, sollte man nachfragen, was genau er damit meint.
2. Zum Speichern von Passwörtern sollte niemals reines SHA-256 verwendet werden. SHA-256 ist zu schnell für diese Aufgabe (siehe Punkt 3 von *Was ein Hash nicht ist*). Der aktuelle Standard ist **Argon2id**: konstruktionsbedingt langsam, konfigurierbar je nach Kapazität des Servers, kombiniert mit einem unterschiedlichen zufälligen *Salt* pro Benutzer.
3. Für die Integrität von Dokumenten —Verträge, Akten, Dateien— bleibt SHA-256 der Referenzstandard. Er wird von qualifizierten Zeitstempeldiensten in der EU verwendet.
4. Für die Langzeitarchivierung (Jahrzehnte) empfiehlt es sich, neben dem SHA-256 auch einen SHA-3 oder einen SHA-512 zu berechnen und zu archivieren; kryptografische Vorsicht rät davon ab, sich bei jahrhundertelangen

Archiven auf eine einzige Funktion zu verlassen.

Technisch gesehen ist diese iterierte Struktur — bei der der Zwischenzustand zwischen den Eingabeblöcken erhalten bleibt — als **Merkle-Damgård**-Konstruktion bekannt. Dies ist das Muster, auf dem SHA-1, SHA-2 (einschließlich SHA-256) und viele andere klassische Hash-Funktionen basieren. SHA-3 hingegen verzichtet auf Merkle-Damgård zugunsten einer anderen Architektur namens *Sponge* (Schwamm).

Wie SHA-256 funktioniert, Schritt für Schritt, in einfachen Worten

Stellen Sie sich vor, Sie haben die aufwendigste Domino-Bahn der Welt aufgebaut: Tausende von Steinen, Dutzende von Verzweigungen, mechanische Brücken und Rampen, die den ganzen Raum durchqueren, sorgfältig Stein für Stein platziert.

Wenn Sie den ersten Stein anstoßen, fällt die Kette in einer präzisen und wiederholbaren Sequenz. Gleicher Aufbau, gleicher Anstoß → identisches Endmuster der gefallenen Steine, immer und immer wieder.

Hier wird es interessant: Verschieben Sie vor dem Start **nur einen einzigen Stein** um einen halben Zentimeter zur Seite und stoßen Sie ihn erneut an. Eine Rampe, die hätte aktiviert werden sollen, bleibt unbewegt, eine Brücke fällt nicht, eine ganz andere Verzweigung wird ausgelöst. Das endgültige Muster der Steine auf dem Boden ist im Vergleich zum ersten völlig unkenntlich.

Mathematisch gesehen ist SHA-256 genau dieser Schaltkreis. Der Text, den Sie schreiben, ist die Ausgangsposition der Steine. Der Algorithmus ist der Anstoß, der die Kaskade auslöst. Und das Endergebnis — das, was wir *Hash* nennen — ist das Standbild des Bodens, wenn alles zum Stillstand gekommen ist. Ändern Sie ein einziges Komma im Originaltext, und das Bild wird radikal anders sein. So einfach und so drastisch.

Schritt 1. Den Text in binäre Steine übersetzen. Computer verstehen keine Buchstaben; sie übersetzen sie zuerst in Zahlen (ASCII) und die Zahlen in Binärzahlen (Einsen und Nullen). Jeder Buchstabe wird in 8 weiße oder schwarze Steine umgewandelt: Das *A* ist `01000001`, das *B* ist `01000010`, das Leerzeichen ist `00100000`. Ihr gesamter Text — ein Wort, ein Vertrag, ein Roman — wird zu einer langen Reihe von weißen und schwarzen Steinen.

Schritt 2. Bis zur Standardgröße auffüllen. Der Schaltkreis verarbeitet die Reihe in *Abschnitten* von genau 512 Steinen. Wenn Ihre Nachricht kein Vielfaches von 512 erreicht, wird direkt nach dem Text ein Markierungsstein (mit dem Wert `10000000`) hinzugefügt und dann mit Nullen aufgefüllt, bis der Abschnitt vollständig ist. Die letzten 64 Positionen jedes Abschnitts sind reserviert, um die ursprüngliche Länge des Textes zu notieren. So weiß der Schaltkreis immer, wo der eigentliche Inhalt endete und wo die Auffüllung begann.

Schritt 3. Die acht Master-Steine platzieren. Bevor wir beginnen, legen wir **acht Master-Steine** in einer präzisen Ausgangsposition auf den Tisch. Diese acht Steine sind kein Geheimnis: Ihr Anfangswert ist durch eine öffentliche mathematische Regel festgelegt (die Quadratwurzeln der ersten acht Primzahlen — 2, 3, 5, 7, 11, 13, 17, 19 — und die ersten Bits des Nachkommateils jeder Wurzel). Jeder an jedem Ort der Welt beginnt mit den gleichen acht Master-Steinen in der gleichen Position. Ihr Schicksal ist es, durch die Lawine angestoßen und transformiert zu werden.

Schritt 4. Die große Lawine: Vierundsechzig Runden von Anstößen. Hier beginnt das Spektakel. Der erste 512-Steine-Abschnitt Ihres Textes prallt auf die acht Master-Steine. Aber sie fallen nicht auf einmal: Der Mechanismus führt **vierundsechzig aufeinanderfolgende Runden** aus. In jeder Runde werden drei Operationen mit den Steinen durchgeführt:

- **Das Karussell** (Rotation). Die Steine bewegen sich im Kreis: Die Steine von rechts wandern nach links. Kein Stein geht verloren oder wird hinzugefügt; sie werden einfach neu angeordnet, indem sie eine komplette Runde im Karussell drehen. Dies ist ein effizienter und umkehrbarer Weg, Informationen neu zu verteilen.
- **Der logische Trichter** (XOR). Die Steine passieren einen Trichter, der sie paarweise vergleicht: Wenn beide die gleiche Farbe haben, kommt ein weißer Stein heraus; wenn sie unterschiedlich sind, kommt ein schwarzer heraus. Es ist die einfachste Operation der binären Logik, aber in Kombination mit den Rotationen des Karussells wird sie extrem mächtig, um Informationen zu mischen, ohne sie zu verlieren.
- **Der Überlauf** (modulare Addition). Das Ergebnis wird mit einem *konstanten Anstoßstein* addiert, der aus einer öffentlichen Liste von vierundsechzig Konstanten stammt (die Kubikwurzeln der ersten vierundsechzig Primzahlen). Wenn die Summe zusätzliche Steine erzeugt, die nicht in den vorgesehenen Platz von 32 Steinen passen, werden diese überschüssigen Steine verworfen. Der Tisch hat nur Platz für 32 Steine, nicht einen mehr.

Am Ende von Runde vierundsechzig hat jeder der Steine aus dem Abschnitt Ihres Textes die Position der acht Master-Steine beeinflusst. Die Energie des Anstoßes ist durch den gesamten Schaltkreis gewandert.

Schritt 5. Den nächsten Abschnitt hinzufügen (ohne Neustart). Wenn Ihr Text lang war und noch ein weiterer 512-Steine-Abschnitt zu verarbeiten ist, **wird der Schaltkreis nicht neu gestartet**. Die acht Master-Steine bleiben so, wie die erste Lawine sie verlassen hat, und der zweite Abschnitt wird gegen sie geschleudert, um weitere vierundsechzig Runden zu aktivieren. Es ist so, als würde man einen neuen Raum voller Dominosteine am Ende des Raumes hinzufügen, der gerade gefallen ist: Die Unordnung im ersten Raum bestimmt vollständig, wie der zweite fallen wird.

Schritt 6. Das Zielfoto machen. Wenn keine weiteren Abschnitte mehr zu verarbeiten sind, kommt die Lawine zum Stillstand. Wir betrachten die endgültige Position, in der die acht Master-Steine stehen geblieben sind. Wir übersetzen ihre Konfiguration in einen Code aus Buchstaben und Zahlen im Hexadezimalsystem. Das Ergebnis ist eine Zeichenfolge von genau vierundsechzig Zeichen: Das ist Ihr SHA-256-Siegel.

Vier Eigenschaften ergeben sich von selbst aus der Art und Weise, wie der Schaltkreis aufgebaut ist:

1. **Determinismus.** Derselbe Text erzeugt auf jedem Computer der Welt immer dasselbe Zielfoto. Null Zufall, null Überraschungen.
2. **Lawinen-Effekt.** Ein hinzugefügtes Komma, ein geänderter Großbuchstabe, ein vergessener Akzent: Das Foto ist völlig unkenntlich. Dies ist die extreme Empfindlichkeit, die wir bereits zu Beginn beschrieben haben.
3. **Einweg-Funktion.** Aus dem fertigen Foto lässt sich der Originaltext nicht rekonstruieren. Die Rotationen, Trichter und Überläufe zerstören alle Richtungsinformationen darüber, *woher jedes Bit kam*, und bewahren nur das auf, *was insgesamt addiert wurde*.
4. **Kollisionsresistenz.** In 25 Jahren öffentlicher Kryptoanalyse ist es niemandem gelungen, zwei verschiedene Texte zu finden, deren Zielfotos übereinstimmen. Und die Schwierigkeit, dies zu tun, liegt außerhalb der Rechenkapazität jeder vernünftigerweise vorstellbaren Zivilisation.

Der folgende Code-Anhang implementiert genau diese sechs Schritte in Zig. Jetzt können Sie ihn lesen und wissen, what jede Bit-Operation bedeutet, anstatt die Manipulationen blind zu akzeptieren.

Technisches Glossar

Für den Leser, der verstehen möchte, was jede Operation bewirkt. Überspringen Sie es ruhig: Der Artikel ist auch ohne dieses Glossar verständlich.

ASCII und Unicode — wie Buchstaben zu Zahlen werden. Computer sehen keine Buchstaben; sie sehen Zahlen. Ein Standard namens **ASCII** (*American Standard Code for Information Interchange*, von 1963) weist jedem Tastaturzeichen eine bestimmte Nummer zu: Das *A* ist 65, das *B* ist 66, das *a* ist 97, die *0* ist 48, das Leerzeichen ist 32, das Komma ist 44. Moderne Systeme erweitern dies durch **Unicode**, das jedem Zeichen jedes Alphabets der Welt eine Nummer zuweist: Kyrillisch, Arabisch, Chinesisch, Japanisch und sogar Emojis. Wenn Sie ein Zeichen tippen oder eine Textdatei öffnen, liest der Computer die dahinterstehende Zahl, nicht die Form auf dem Bildschirm. SHA-256 arbeitet mit diesen Zahlen und behandelt jeden Text als eine lange Folge von Ziffern. Deshalb kann es einen Artikel auf Spanisch, ein Gedicht auf Japanisch und eine Binärdatei mit demselben Algorithmus versiegeln.
XOR — der bitweise Vergleicher. XOR (ausgesprochen *'ex-or'*, vom Englischen *exclusive or*, 'eksklusives Oder') ist eine der einfachsten Operationen, die ein Computer mit zwei Binärzahlen durchführen kann. Er vergleicht zwei Bits an der gleichen Position und gibt Folgendes zurück: **1**, wenn genau eines der beiden 1 ist (eines, aber nicht beide), **0**, wenn beide gleich sind (beide 0 oder beide 1). Beispiel: XOR von `1010` und `1100` ist `0110`. Es hat eine bemerkenswerte Eigenschaft: Es ist umkehrbar — wenn Sie zweimal XOR mit demselben Schlüssel durchführen, kehren Sie zum Original zurück. Deshalb ist es das Arbeitstier der Kryptografie: Es mischt Bits ohne Informationsverlust, aber das Ergebnis verrät nichts über die Eingaben, wenn man eine davon nicht kennt.
Hexadezimal — Zählen zur Basis 16. Fast alle Zahlen im Alltag verwenden zehn Ziffern (0-9). Das Hexadezimalsystem verwendet sechzehn: die üblichen 0-9 plus sechs Buchstaben, die die folgenden Werte darstellen: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Warum sechzehn? Parce que Computer in Gruppen von vier Bits denken und vier Bits genau sechzehn verschiedene Werte darstellen können — so entspricht ein Hexadezimalzeichen sauber vier Bits. Ein SHA-256-Abdruck ist 256 Bits lang, was genau **64 Hexadezimalzeichen** entspricht. Würden wir ihn in gewöhnlichen Dezimalzahlen schreiben, würde er etwa 78 Stellen einnehmen und wäre unhandlicher. Die Wahl ist ästhetisch und kompakt; die dahinterstehende Zahl ist dieselbe.
Bit-Rotation — das binäre Karussell. Stellen Sie sich eine Reihe von sieben Glühbirnen vor, von denen einige leuchten (1) und andere aus sind (0): `1 0 1 1 0 0 1`. Eine Rotation um eine Position nach rechts besteht darin, die Glühbirne ganz

rechts zu nehmen, sie an das linke Ende zu setzen und die anderen um eine Position nach rechts zu verschieben: `1 1 0 1 1 0 0`. Keine Glühbirne geht verloren oder wird hinzugefügt: Sie tanzen einfach im Kreis. SHA-256 verwendet die Bit-Rotation hunderte Male bei jeder Berechnung; es ist ein effizienter und verlustfreier Weg, die Informationen innerhalb des Zustands neu zu verteilen.

***'Nothing-up-my-sleeve'*-Konstanten — warum sie von Primzahlen abstammen.** Die acht Master-Steine und die vierundsechzig Runden-Konstanten von SHA-256 wurden nicht zufällig gewählt. Sie stammen von den Quadrat- und Kubikwurzeln der ersten Primzahlen ab. Warum? Weil die Entwickler Konstanten wollten, bei denen *'nichts im Ärmel versteckt ist'*: Werte, deren Ursprung jeder überprüfen kann. Wenn Ihnen jemand sagen würde: *'Vertrau mir: Benutze diese zufällige 32-Bit-Zahl'*, würden Sie berechtigterweise eine versteckte Schwachstelle oder eine Hintertür vermuten. Aber jeder mit einem Taschenrechner kann überprüfen, dass die ersten 32 Bits der Quadratwurzel von 2 `0x6a09e667` sind. Die Werte sind mathematisch, öffentlich und reproduzierbar: Kein versteckter Trick kann sich in das Rezept einschleichen.

Anhang: SHA-256 in lesbarem Code

Dieser Anhang ist für den Leser gedacht, der den Algorithmus von innen sehen möchte. Es handelt sich um eine didaktische Implementierung in Zig, die der Spezifikation FIPS 180-4 folgt. Es ist nicht die Version, die Solo2 verwendet —die echte befindet sich in `std.crypto.hash.sha2.Sha256` in der Standardbibliothek von Zig, optimiert und auditiert—. Aber der Algorithmus ist derselbe: Was Sie hier sehen, ist Schritt für Schritt das, was passiert, wenn jener fünfzeichenlange Aufruf seine Arbeit verrichtet.

```
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
}
```

Jede Umschreibung in einer anderen Sprache, die derselben Struktur folgt —Anfangskonstanten, Erweiterung des Schedules, vierundsechzig Runden, Akkumulation—, erzeugt das gleiche Ergebnis. Der Algorithmus hat keine Geheimnisse: Sein Wert liegt darin, dass die oben genannten Eigenschaften auch nach zwei Jahrzehnten öffentlicher Kryptoanalyse durch Tausende von Augen weiterhin Bestand haben.

Wenn Sie zum Ende dieses Artikels zurückkehren, sehen Sie ein hexadezimales Siegel mit vierundsechzig Zeichen. Es ist der SHA-256 des Textes, den Sie gerade gelesen haben, in dieser Sprache. Wenn wir den Artikel übersetzen würden, wäre das Siegel ein anderes; wenn sich ein Wort in der deutschen Version ändern würde, würde sich das deutsche Siegel ändern. Das Siegel schützt den Inhalt nicht —dafür gibt es andere Werkzeuge—, sondern identifiziert ihn eindeutig. Und das reicht, so bescheiden es auch klingen mag, aus, damit kein Schritt in der Redaktionskette das Gesagte ändern kann, ohne dass es bemerkt wird. Alles andere —Verschlüsseln, Signieren, Identifizieren— baut auf dieser einfachen Idee auf.

Anmerkung der Redaktion: Wenn diese Cuadernos Unternehmen oder Produkte nennen, geschieht dies nicht, um sie anzuklagen. Diejenigen, die sie entwickeln, leisten Arbeit, die Millionen von Menschen nutzen und schätzen. Was wir aufzeigen, ist struktureller Natur — das Modell, nicht die Marke. Marken erscheinen als Beispiele, weil der Leser sie erkennt.

Quellen und weiterführende Literatur

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, August 2015. Offizielle Spezifikation der SHA-2-Familie, einschließlich SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, Mai 2011. Normative Version für Implementierer.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Die Kapitel 5 und 6 behandeln Hash-Funktionen und ihre legitimen und illegitimen Verwendungen.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Praktisches Beispiel für die Verwendung von SHA-256 zur Verkettung von Blöcken in einer konstruktionsbedingt unveränderlichen Struktur.
- Verordnung (EU) 910/2014 (eIDAS) — Rahmen für qualifizierte Zeitstempel. SHA-256 ist die Referenzfunktion für qualifizierte elektronische Signaturen und Siegel, die in der EU ausgestellt werden. (Konform mit der DSGVO für Datenschutzaspekte).
- Referenzimplementierung in Zig: `std.crypto.hash.sha2.Sha256` im offiziellen Repository der Sprache (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). Es ist die optimierte und auditierte Version, die Solo2 tatsächlich verwendet. Nützlich zum Vergleich mit der didaktischen Implementierung im Anhang.

← ZurückSchrems II, fünf Jahre späterWeiter →Kill Switch und institutionelle Vereinnahmung

Aktuelle Lektüre

- Analyse · 18. Mai 2026 Echte vs. scheinbare Privatsphäre: Die Fragen, die man sich stellen sollte

- Analyse · 18. Mai 2026 Self-Hosting als berufliche Praxis
- Konzept · 18. Mai 2026 Die 24 Wörter: Was eine kryptografische Identität ist

Nehmen Sie diesen Artikel mit, wohin Sie ihn brauchen.

↓ Markdown ↓ Klartext ↓ PDF

Die Datei wird auf Ihr Gerät heruntergeladen. Von dort aus können Sie sie speichern, in Solo2 importieren oder teilen, wo immer Sie möchten. Cuadernos entscheidet nicht über den Zielort für Sie.

Siegellack-Siegel · SHA-256 65c8dea2021d067b78342dd76aebf395e7fe5061fd5d03b906549abc1ca7c3cd

Cuadernos Lacre · Eine Publikation von Menzuri Gestión S.L. ·
geschrieben von R.Eugenio · herausgegeben vom Team von Solo2.