コンセプト · 2026年5月19日

← 前へ·次へ →

SHA-256とは一体何なのか

64文字に収まる数学的な指紋。元のテキストのコンマ一つが動くだけで、そのすべてが変化します。なぜデジタル封蝋（ふうろう）と呼ばれるのか。

かみ砕いて言うと： 任意のテキストを読み取って64文字の配列を返す機械を想像してください。テキストが全く同じなら、配列も全く同じになります。しかし、カンマ一つ動かしただけで、配列は完全に別物になります。この配列こそが「デジタルの封蝋」なのです。

テクニカルな名前の背後にあるシンプルなアイデア

スロットが一つ、画面が一つしかない機械を想像してみてください。スロットにテキストを入れます。それは単語、フレーズ、あるいは小説一冊かもしれません。すると画面に、正確に64文字の数列が表示されます。専門家はこの数列を「ハッシュ」または「暗号学的要約」と呼びますが、一般の方は、指紋が人のものであるように、テキストの「数学的な指紋」と呼んでいいでしょう。

同じテキストを2回入力すれば、機械は2回とも同じ指紋を表示します。もしテキストをわずかでも変えれば（コンマ一つを動かしたり、大文字を小文字にしたり）、機械は最初とは全く異なる指紋を表示します。似ているのではなく、完全に別物です。この2つの性質（決定性と感度）こそがシンプルなアイデアの正体です。SHA-256のそれ以外は、これらを確実に機能させるための仕組みに過ぎません。

まず、この機械が「しないこと」を明確にしておきましょう。テキストを暗号化（隠匿）はしません。保存もしません。機械はテキストを見て、指紋を計算し、テキストのことは忘れます。指紋から元のテキストを復元することは不可能です。できるのは、あるテキストが元のものと一致するかどうかを確認することだけです。そのため、これを「一方通行」の要約と呼びます。行くことはできても、戻ることはできません。

ハッシュと暗号化は別物

よく混同されますが、暗号化とハッシュ化は全く別の操作です。暗号化とは、鍵を持つ者だけが元の形に戻せるようにテキストを変換することです。ハッシュ化とは、鍵があろうとなかろうと、元のテキストを二度と復元できない指紋を生成することです。前者は設計上「可逆的」であり、後者は設計上「不可逆的」です。

この違いには実用的な意味があります。あるアプリが「パスワードを暗号化して保存しています」と言う場合、誰か（そのアプリ自体など）がそれを復号する鍵を持っています。しかし、アプリが「パスワードをハッシュ化して保存しています」と言う場合、アプリ側も元のパスワードを知ることはできません。あなたが入力したパスワードが、保存されている指紋と同じものを生成するかどうかを確認できるだけです。適切に行われれば、後者の方がパスワード保存において遥かに優れています。「適切に」というのがSHA-256単体以上のものを必要とする理由は、後ほど説明します。

暗号学的ハッシュを支える4つの性質

「暗号学的」ハッシュ関数と呼ばれるためには、以下の4つの性質を満たす必要があります：

1. **決定性：**同じ入力からは常に同じ指紋が生成される。
2. **雪崩効果：**入力のわずかな変化が、以前のものとは似ても似つかない全く異なる指紋を生成する。
3. **逆像耐性：**指紋から元のテキストを見つけ出すことは計算上不可能である。
4. **衝突耐性：**同じ指紋を生成する2つの異なるテキストを見つけ出すことは計算上不可能である。

「計算上不可能」とは、「数学的に不可能」という意味ではありません。それを達成するために必要な時間、エネルギー、コストが、利用可能な全計算能力の桁を遥かに超えていることを意味します。SHA-256の場合、その限界は、専用ハードウェアを用いた最も楽観的な見積もりでも数千兆年単位です。つまり、読者の皆さんにとっては実質的に「不可能」と同じことです。

具体的にはSHA-256とは

名前がすべてを物語っています。SHAは「Secure Hash Algorithm（安全なハッシュアルゴリズム）」の略です。256という数字は指紋のサイズが256ビットであることを示しています。これは32バイトであり、16進数で表示するとお馴染みの64文字になります。この規格は、2001年に米国のNIST（国立標準技術研究所）によってSHA-2ファミリーの一部として公開されました。現在の現行規格であるFIPS 180-4は2015年のものです。

ビットとバイトがまだピンとこない方へ：

```
1ビット    →  0 または 1 （スイッチのオン/オフ）
1バイト    →  8ビット （256通りの組み合わせ）
32バイト   →  256ビット （SHA-256の指紋）
```

名前の最後にある256はビット数です。10の代わりに16の記号を使う16進法では、この256ビットは正確に64文字に収まります。これが、各Cuadernoの末尾に表示されている64文字の正体です。

その規模を想像してみてください。256ビットでは、2の256乗通りの値が可能です。これは10進数で78桁の数字であり、観測可能な宇宙にある原子の推定数よりも数桁大きいです。世界中のあらゆるテキストが、これらの値のいずれかに割り当てられます。2つの異なるテキストが偶然一致する確率は、実質的にゼロと見なせます。

コードでの見え方

Solo2の基盤となっている言語Zigでは、テキストのSHA-256を計算するのは以下のようになります：

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

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

ここでは、Zigの標準ライブラリに引用符内のテキストのSHA-256を計算させています。呼び出し後、変数 *resumen* には生の32バイトの指紋が格納されます。これを画面に16進数で表示すると、この記事の末尾にある64文字になります。もし *Cuadernos Lacre* を *Cuadernos lacre* （Lを小文字に）と変えれば、指紋は完全に変わります。このわずか5行のコードが、他を支える中心的な性質です。内部構造を知りたい方のために、記事の最後にステップごとのコメント付きのアルゴリズムを掲載しています。

なぜ「封蝋（ふうろう）」と呼ぶのか

15世紀から19世紀のヨーロッパの通信では、封蝋が手紙を封じていました。溶けた蝋を垂らし、その上に印章（シール）を押し当てることで、手紙に複製不可能な印が刻まれました。これは覗き見を防ぐものではありませんでしたが（紙は透かして読め、蝋は壊せた）、改ざんの証拠にはなりました。封印が少しでも乱れていれば、受取人は手紙を開く前に異変に気づけました。封蝋は被害を防ぐのではなく、それを「宣言」したのです。

各Cuadernoの本文のSHA-256は、デジタル版の同じ役割を果たします。記事が公開された瞬間からあなたが読むまでの間に、一文字でも変わっていれば、末尾の指紋は目の前にあるテキストのSHA-256とは一致しなくなります。5行のコードを書ける読者なら誰でもそれを検証できます。発行者は、指紋に露見することなく歴史を書き換えることはできません。これは被害を防ぐのではなく、それを「検証可能」にするのです。

ハッシュが「しない」こと

SHA-256に期待されがちですが、本来の役割ではない4つのケースがあります：

1. **暗号化：**ハッシュは要約であり、隠匿ではありません。テキストを読ませたくない場合は、ハッシュ化ではなく暗号化が必要です。
2. **著者の認証：**ハッシュは誰が書いたかは教えず、どのテキストがハッシュ化されたかだけを教えます。著者を結びつけるには、ハッシュ単体ではなく、その上に暗号署名が必要です。
3. **パスワードの保存：**ここには罠があります。SHA-256は非常に高速に設計されています。これは多くの場合メリットですが、パスワード保存にはデメリットです。攻撃者は専用ハードウェアを使い、1秒間に数十億のパスワードを試して、あなたのハッシュに一致するものを見つけ出せてしまいます。パスワード保存には、Argon2、scrypt、bcryptなどの意図的に遅い鍵導出関数を、ソルト（ユーザーごとの固有のランダムデータ）と組み合わせて使うべきです。
4. **著者の識別子としてのハッシュ：**ハッシュは著者を識別しません。内容を識別します。もし二人の人が同じ「hola」という言葉をハッシュ化すれば、二人とも同じ結果を得ます。これは中心的な性質であり、欠陥ではありません。もし結果が異なれば、公開されたものと受信したものが一致するか確認できなくなってしまいます。

日常生活の中のSHA-256

目には見えませんが、SHA-256はインターネットの日常を支えています。ビットコインのブロックチェーンは、各ブロックを次のブロックにSHA-256で繋ぐことで構築されています。過去のブロックを改ざんするには、それ以降のすべてのチェーンを再計算する必要があります。Git（世界中のコード管理システム）は、各コミットをその内容全体のハッシュ（最近はSHA-256、以前はSHA-1）で識別します。ウェブサイトのHTTPS証明書にもSHA-256が関連付けられています。ソフトウェアのダウンロード時には、ファイルが途中で改ざんされていないか確認できるように、開発者がSHA-256を公開していることがよくあります。そしてもちろん、各Cuaderno Lacreの末尾にも。

プロフェッショナルな読者のために

システムを決定または監査する人のための4つの運用上の注意点：

1. ハッシュは暗号化ではありません。もしベンダーが技術文書でこの二つを混同していたら、正確にはどういう意味か問い質すべきです。
2. パスワード保存にSHA-256単体を使ってはいけません。前述の通り、この用途には速すぎます。現在の標準は **Argon2id** です。意図的に遅く、サーバーの能力に合わせて調整可能で、ユーザーごとのランダムなソルトと組み合わせます。
3. 契約書やファイルなどの文書の完全性（インテグリティ）には、SHA-256が依然として参照基準です。EUの認定タイムスタンプでも使用されています。
4. 数十年単位の長期保存には、SHA-256に加えてSHA-3やSHA-512も計算してアーカイブしておくのが賢明です。100年単位のアーカイブでは、単一の関数のみに依存しないのが暗号学的な慎重さです。

技術的には、この反復構造（中間状態が入力ブロック間で保持される構造）は、**Merkle-Damgård**（メルクル・ダムガード）構成として知られています。これは、SHA-1、SHA-2（SHA-256を含む）、および他の多くの古典的なハッシュ関数が基づいているパターンです。対照的に、SHA-3はメルクル・ダムガードを廃止し、*スポンジ*構造と呼ばれる異なるアーキテクチャを採用しています。

SHA-256の仕組み：ステップ・バイ・ステップ、平易な言葉で

世界で最も精巧なドミノ倒しの回路を組み立てたと想像してみてください。何千もの駒、数十の分岐、機械的なブリッジ、そして部屋中を横切るスロープ。それらすべてが、一つ一つ慎重に配置されています。

最初の駒を軽く叩くと、連鎖は正確で再現可能な順序で倒れていきます。同じ配置、同じ最初の衝撃 → 何度繰り返しても、倒れた駒の最終的なパターンは同一になります。

ここが面白いところです。始める前に、**たった一つの駒**を横に5ミリ動かしてから、もう一度叩いてみてください。作動するはずのスロープは動かず、ブリッジは倒れず、別の分岐が作動します。床に残された駒の最終的なパターンは、最初のものとは全く見分けがつかないものになります。

SHA-256は数学的にこの回路です。あなたが書くテキストは駒の初期位置です。アルゴリズムは、連鎖を解き放つ衝撃です。そして最終的な結果（私たちがハッシュと呼ぶもの）は、すべてが停止した瞬間の床の静止画です。元のテキストのコンマ一つを変えるだけで、その写真は劇的に異なります。それほど単純で、それほど劇的なのです。

ステップ1：テキストをバイナリの駒に変換する。 コンピュータは文字を理解しません。まず数字（ASCII）に変換し、その数字をバイナリ（0と1）に変換します。各文字は8つの白または黒の駒に変わります。たとえば、*A*は`01000001`、*B*は`01000010`、スペースは`00100000`です。あなたのテキスト全体（単語、契約書、小説など）は、白と黒の駒の長い列になります。

ステップ2：標準サイズまで補充する。 回路は、駒の列を正確に512個ずつの*ブロック*で処理します。メッセージが512の倍数に満たない場合は、テキストの直後にマーカーの駒（値が`10000000`のもの）が追加され、その後ブロックが完成するまでゼロが補充されます。各ブロックの最後の64ポジションは、元のテキストの長さを記録するために予約されています。これにより、回路は常にどこで実際のコンテンツが終わり、どこから補充が始まったかを知ることができます。

ステップ3：8つのマスター駒を配置する。 始める前に、テーブルの上に**8つのマスター駒**を正確な初期位置に配置します。この8つの駒は秘密ではありません。その初期値は、公開されている数学的ルール（最初の8つの素数 2, 3, 5, 7, 11, 13, 17, 19 の平方根と、各平方根の小数部分の最初のビット）によって固定されています。世

界中の誰もが、同じ位置にある同じ8つのマスター駒から始めます。それらの運命は、連鎖によって押し流され、変容することにあります。

ステップ4：大連鎖：64ラウンドの衝撃。 ここからが本番です。テキストの最初の512個の駒が、8つのマスター駒に衝突します。しかし、一気に倒れるわけではありません。メカニズムは**64回の連続したラウンド**を実行します。各ラウンドで、駒に対して3つの操作を行います。

- **メリーゴーラウンド**（回転）。駒は円を描くように動きます。右側の駒が左側に移ります。駒が失われたり追加されたりすることはありません。ただ単に、メリーゴーラウンドを一回転させて並べ替えるだけです。これは情報を再分配するための安価で可逆的な方法です。
- **論理漏斗**（XOR）。駒は、2つずつ比較する漏斗を通過します。2つが同じ色なら白が出てき、異なる色なら黒が出てきます。これはバイナリ論理の中で最も単純な操作ですが、メリーゴーラウンドの回転と組み合わせることで、情報を失うことなく混合するための非常に強力な手段となります。
- **オーバーフロー**（モジュロ加算）。結果は、64個の定数の公開リスト（最初の64個の素数の立方根）から取得された*一定の衝撃の駒*と加算されます。加算によって予定されていた32個の駒のスペースに収まらない余分な駒が発生した場合、それらは破棄されます。テーブルには32個の駒のスペースしかなく、それ以上は入りません。

第64ラウンドが終わる頃には、テキストブロックの各駒が8つのマスター駒の位置に影響を与えています。衝撃のエネルギーは回路全体を駆け巡りました。

ステップ5：次のブロックを追加する（リセットなし）。 テキストが長く、処理すべき別の512個のブロックが残っている場合、**回路はリセットされません**。8つのマスター駒は最初の連鎖が終わった状態のまま維持され、第2のブロックがそれらに向かって放たれ、さらに64ラウンドが開始されます。これは、倒れたばかりのドミノの部屋の最後に新しい部屋を追加するようなものです。最初の部屋の無秩序さが、次の部屋がどのように倒れるかを完全に決定づけます。

ステップ6：最終的な写真を撮る。 処理すべきブロックがなくなると、連鎖が止まります。8つのマスター駒が最終的にどの位置に残ったかを確認します。その構成を16進法の英数字コードに変換します。その結果、正確に64文字の文字列が出来上がります。これがあなたのSHA-256シールです。

回路の組み立て方から、4つの特性が自然と導き出されます。

1. **決定性。** 世界中のどのコンピュータでも、同じテキストからは常に同じ最終的な写真が作成されます。ランダム性はゼロ、驚きもゼロです。
2. **雪崩効果。** コンマ一つ追加されたり、大文字が変わったり、アクセント記号を忘れたりするだけで、写真は全く見分けがつかなくなります。これは、冒頭で説明した極端な感度です。
3. **一方向性。** 最終的な写真から元のテキストを復元することはできません。回転、漏斗、オーバーフローは、*各ビットがどこから来たか*という方向性情報をすべて破壊し、*合計で何が加算されたか*という情報だけを保持します。
4. **衝突耐性。** 25年間にわたる公開暗号解読の歴史の中で、最終的な写真が一致する2つの異なるテキストを見つけられた人は誰もいません。そして、それを見つける難しさは、合理的に想像可能なあらゆる文明の計算能力を超えています。

続くコードの付録では、これら6つのステップをZigで正確に実装しています。ビット操作の一つ一つが何を意味しているかを知った上で、盲目的に操作を受け入れるのではなく、読み進めることができるはずです。

技術用語集

各操作が何をしているかを理解したい読者のために。自由にスキップしてください。これなしでも記事の内容は理解できます。

ASCIIとUnicode — 文字がいかにして数字になるか。 コンピュータは文字を見ているのではなく、数字を見ています。1963年に作られた**ASCII**（American Standard Code for Information Interchange）という標準は、キーボードの各文字に特定の数字を割り当てています。たとえば、*A*は65、*B*は66、*a*は97、*0*は48、スペースは32、コンマは44です。現代のシステムはこれを**Unicode**で拡張し、キリル文字、アラビア文字、中国語、日本語、さらには絵文字まで、世界中のあらゆる文字に数字を割り当てています。文字を入力したりテキストファイルを開いたりするとき、コンピュータは画面上の形ではなく、背後にある数字を読み取ります。SHA-256はこれらの数字に対して動作するため、スペイン語の記事、日本語の詩、バイナリファイルを同じアルゴリズムで封印することができるのです。

XOR（排他的論理和） — ビット単位の比較器。 XOR（「エグゾア」と読みます）は、コンピュータが2つのバイナリ数字に対して行える最も単純な操作の一つです。2つのビットを位置ごとに比較し、**1**（2つのうち正確に一方が1の場合、つまり両方1ではない場合）、または**0**（両方が同じ場合、つまり両方0または両方1の場合）を返します。例：`1010`と`1100`のXORは`0110`になります。これには注目すべき特性があります。それは可逆的であるということです。同じキーで2回XORを行うと、元の状態に戻ります。そのため、暗号学の主力として使われています。情報を失うことなくビットを混合しますが、入力の一方を知らなければ結果から何も明かされることはありません。

16進法 — 16を基数として数える。 日常生活のほとんどの数字は10個の数字（0-9）を使います。16進法は16個の記号を使います。通常の0-9に加えて、次の値を表す6つの文字を使います：`A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15。なぜ16でしょうか？ それは、コンピュータが4ビットのグループで考えるためであり、4ビットは正確に16種類の異なる値を表すことができるからです。つまり、1つの16進文字はきれいに4ビットに対応します。SHA-256の指紋は256ビットで、これは正確に**64文字の16進文字**になります。もしこれを通常の10進法で書くと約78桁になり、扱いにくくなります。16進法の選択は美的でコンパクトなものであり、背後にある数字自体は同じです。

ビット回転 — バイナリのメリーゴーラウンド。 7つの電球が並んでいて、点灯（1）しているものと消灯（0）しているものがあると想像してください：`1 0 1 1 0 0 1`。右に1つ回転させるとは、右端の電球を取り出し、左端に持ってきて、他の電球を右に1つずらすことです：`1 1 0 1 1 0 0`。電球が失われたり追加されたりすることはありません。ただ円を描いて踊っているだけです。SHA-256は各計算の中でビット回転を何百回も使用します。これは、状態内の情報を再分配するための、低コストで損失のない方法です。

「Nothing-up-my-sleeve（種も仕掛けもない）」定数 — なぜ素数に由来するのか。 SHA-256の8つのマスター駒と64のラウンド定数は、ランダムに選ばれたのではありません。それらは最初の素数の平方根と立方根に由来しています。なぜでしょうか？ 設計者たちは、誰でもその起源を確認できる「種も仕掛けもない（nothing-up-my-sleeve）」値を求めたからです。もし誰かが*「私を信じて、この32ビットのランダムな数字を使ってください」*と言ったら、あなたは当然、隠された脆弱性やバックドアを疑うでしょう。しかし、電卓があれば誰でも、2の平方根の最初の32ビットが`0x6a09e667`であることを確認できます。値は数学的、公開、かつ再現可能であり、レシピに隠された罠が入り込む余地はありません。

付録：読みやすいコードで見るSHA-256

この付録は、アルゴリズムの内部を見たい読者のためのものです。FIPS 180-4仕様に従ったZigでの教育用実装です。Solo2が実際に使っているのは、Zig標準ライブラリの `std.crypto.hash.sha2.Sha256` であり、最適化と監

査が済んだものですが、アルゴリズム自体は同じです。ここにあるのは、あの5文字の呼び出しが実行している作業のステップバイステップです。

```
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
}
```

同じ構造（初期定数、スケジュール拡張、64ラウンド、累積）に従えば、他の言語で書き直しても同じ結果が得られます。アルゴリズムに秘密はありません。その価値は、20年にわたる数千の目による公開クリプタナリシスの後でも、前述の性質が維持され続けていることにあります。

この記事の末尾に戻ると、64文字の16進数の印が見えるはずです。それは、あなたが今読んだこの言語のテキストのSHA-256です。もし記事を翻訳すれば、印は変わります。スペイン語版の一文字を変えれば、スペイン語版の印も変わります。印は内容を保護するものではありませんが（それには別のツールがあります）、内容を一意に識別します。控えめに聞こえるかもしれませんが、それだけで、編集チェーンのどの段階でも、気づかれずに内容を書き換えることができなくなります。暗号化、署名、識別といった他のすべては、このシンプルなアイデアの上に構築されています。

編集後記： この「Cuadernos」で企業名や製品名を挙げているのは、決して非難するためではありません。それらを構築している人々は、何百万人もの人々に利用され愛される素晴らしい仕事をしています。私たちが指摘しているのは構造的な問題であり、ブランドではなくモデルの問題です。読者に馴染みがあるため、例としてブランド名を挙げているに過ぎません。

参考文献および関連資料

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, 2015年8月。SHA-256を含むSHA-2ファミリーの公式仕様。
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, 2011年5月。実装者のための規範的バージョン。
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). 第5章と第6章でハッシュ関数とその正当・不当な使用法を扱っています。
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). 構造的に不変なチェーンを作るためにSHA-256を使用した実践的な例。
- 規則 (EU) 910/2014 (eIDAS) — 認定タイムスタンプの枠組み。SHA-256はEUで発行される認定電子署名およびシールの参照関数です。
- Zigでの参照実装：公式リポジトリの `std.crypto.hash.sha2.Sha256` (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`)。Solo2が実際に使用している、最適化され監査されたバージョンです。付録の教育用実装との比較に役立ちます。

← 前へシュレムスII、5年後の現状次へ →キルスイッチと制度的キャプチャ

最近の記事

- 分析 · 2026年5月18日 真のプライバシー vs 表向きのプライバシー：問い直すべきこと
- 分析 · 2026年5月18日 専門的実践としてのセルフホスティング
- コンセプト · 2026年5月18日 24個の単語：暗号学的アイデンティティとは何か

この記事をダウンロードして、必要な場所で活用してください。

↓ Markdown ↓ テキスト形式 ↓ PDF

ファイルはお使いのデバイスにダウンロードされます。そこから保存、Solo2 へのインポート、または任意の場所での共有が可能です。Cuadernos が送信先を決定することはありません。

封蝋 · SHA-256 9a92b51b3386196444fadef5dfdc91b50ea4a3cefc2944e7696ee77fdeafc9eb

Cuadernos Lacre · Menzuri Gestión S.L. による刊行物 ·
著者：R.Eugenio · 編集：Solo2 チーム