What SHA-256 Really Is

A mathematical fingerprint that fits into sixty-four characters and changes entirely if a single comma of the original text is moved. Why we call it a digital sealing wax.

The simple idea behind the technical name

Imagine there is a machine with a single slot and a single screen. Through the slot, you insert a text: a word, a sentence, an entire novel. On the screen, moments later, a sequence of exactly sixty-four characters appears. That sequence, to the professional reader, we call a *hash* or *cryptographic summary*; to the general reader, we can call it for now a mathematical fingerprint of the text, just as a fingerprint is for a person.

If you insert the same text twice, the machine shows the same fingerprint both times. If you insert a slightly different text —a single moved comma, an uppercase letter that becomes lowercase—the machine shows a fingerprint completely different from the first one. Not similar: different. These two properties together—determinism and sensitivity—are the simple idea. Everything else about SHA-256 is the machinery that makes them hold up well.

It's worth saying from the beginning what the machine does not do. It does not encrypt the text. It does not hide it. It does not save it. The machine looks at the text, calculates the fingerprint, and forgets the text. The fingerprint does not allow the reconstruction of the text that produced it; it only allows, given a candidate text, checking whether it matches the original or not. That's why we say it's a *one-way* summary: it goes out, it doesn't come back.

A hash is not the same as encrypting

Confusion is frequent and worth clearing up: encrypting and hashing are different operations. Encrypting consists of transforming a text so that only the holder of the key can return it to its original form. Hashing consists of producing a fingerprint of the text from which the original text can never be recovered, with or without a key. The first is reversible by design; the second, irreversible by design.

The practical consequence matters. When an application says "we save your password encrypted", there is someone who has the key to decrypt it—the application itself, in any case. When an application says "we save your password hashed", the application itself cannot read the original password even if it wanted to; it can only check if what you type produces the same fingerprint again. The second model, done well, is much preferable to the first for storing passwords. Later we will see why "done well" requires something more than just SHA-256.

The four properties that make a cryptographic hash useful

A hash function that deserves the adjective *cryptographic* meets four properties:

1. **Determinism.** The same input always produces the same fingerprint.
2. **Avalanche effect.** A small change in the input produces a completely different fingerprint, with no visible resemblance to the previous one.
3. **Pre-image resistance.** Given a fingerprint, it is not computationally feasible to find the text that produced it.
4. **Collision resistance.** It is not computationally feasible to find two different texts that produce the same fingerprint.

"Not computationally feasible" does not mean "it is mathematically impossible." It means that the cost in time, energy, and money of achieving it exceeds by orders of magnitude the sum of all reasonably available computing capacity. For SHA-

256, that threshold is measured in quadrillions of years even for the most optimistic approaches with specialized hardware. Which, for the reader's practical purposes, is the same as "it cannot be done."

SHA-256, specifically

The name says it all. SHA stands for *Secure Hash Algorithm*. The number 256 indicates the size of the fingerprint in bits: two hundred and fifty-six bits, that is, thirty-two bytes, which shown in hexadecimal are the sixty-four characters that the reader already recognizes. The standard was published by the US NIST, the body that normalizes this type of functions, in 2001 as part of the SHA-2 family; the current version of the standard, FIPS 180-4, is from 2015.

For those who still don't have in mind what bits and bytes are:

```
1 bit    →  0 or 1        (a switch: on or off)
1 byte   →  8 bits       (256 possible combinations)
32 bytes →  256 bits     (the SHA-256 fingerprint)
```

The number 256 at the end of the name tells the size of the fingerprint in bits. In hexadecimal—a numbering system with sixteen symbols instead of ten—those 256 bits fit into exactly 64 characters. Those are the 64 characters you see at the foot of each Cuaderno.

The dimensions deserve a moment. Two hundred and fifty-six bits allow two to the power of two hundred and fifty-six different values: a number with seventy-eight decimal digits, several orders of magnitude greater than the estimated number of atoms in the observable universe. Every text in the world—every book, every email, every message—falls on one of those values. The probability of two different texts coinciding by chance is, for practical purposes, indistinguishable from zero.

How it looks in code

In Zig, the language in which we write the pieces that sustain Solo2, calculating the SHA-256 seal of a text looks like this:

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

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

We have just asked the Zig standard library to calculate the SHA-256 of the text in quotes. After the call, the *summary* variable contains the thirty-two bytes that make up the seal in its raw form; when displayed on the screen in hexadecimal, they are the sixty-four characters that appear at the foot of this article. If we changed *Cuadernos Lacre* to *Cuadernos lacre* —one less capital letter—the seal would change entirely. That is, in five lines, the central property that sustains the rest. For those who want to see how it works internally, at the end of the article we include a readable version of the algorithm with step-by-step comments.

Why we call it a sealing wax

In European correspondence from the fifteenth to the nineteenth centuries, sealing wax closed the letter. A drop of melted wax, a seal pressed on top, and the letter was marked in an unrepeatable way. It did not protect the content from the determined snooper—the paper could be read against the light, the wax could be broken—but it did evidence it. Any alteration of the closure was visible to the recipient even before opening the paper. The wax did not prevent the damage; it declared it.

The SHA-256 of the body of each Cuaderno serves the same function in its digital version. If a single word of the article changed between the time it was published and the time you read it, the hexadecimal seal at the foot of the text would no longer match the SHA-256 of the text in front of you. Any reader with five lines of code could check it. The publication cannot rewrite its history without the seal betraying it. It does not protect against damage; it makes it verifiable.

What a hash is not

Four uses are sometimes asked of SHA-256 that do not belong to it:

1. **Encrypting.** A hash summarizes; it does not hide. If you want the text not to be readable, you need to encrypt it, not hash it.
2. **Authenticating the author.** A hash does not say who wrote the text, only what text was hashed. To associate authorship, a cryptographic signature is needed on top of the hash, not the hash alone.
3. **Storing passwords.** There is a trap here that's worth understanding. SHA-256 is designed to be very fast—which is good for many things, but bad for this. An attacker with specialized hardware can test billions of passwords per second against a SHA-256 hash until finding yours. To store passwords, one must use deliberately slow key derivation functions like Argon2, scrypt, or bcrypt, combined with a *salt* (a unique random piece of data per user, which prevents two people with the same password from having the same hash).
4. **Reading the hash as an author identifier.** It is not. A hash identifies the content. If two people hash the word *hello* with SHA-256, both get the same summary—and that is the central property, not a defect: if they were different summaries, we could not check coincidence between what is published and what is received.

Where SHA-256 appears in your day-to-day life

Although you don't see it, SHA-256 sustains a good part of what you use daily on the internet. The Bitcoin blockchain is built by chaining the SHA-256 of each block to the next; altering a past block forces the recalculation of the entire subsequent chain. Git, the system with which half the world's code is versioned, identifies each commit by the SHA-256 (in recent versions) or by its predecessor SHA-1 (in older versions) of its full content. The HTTPS certificates that verify the identity of a website when you enter have an associated SHA-256 fingerprint. Software downloads are often accompanied by a SHA-256 published by the developer so you can verify that the file was not altered along the way. And, as we have said, at the foot of each Cuaderno Lacre.

For the professional reader

Four operational reminders for those who decide on or audit systems:

1. Hash is not encryption. If a provider confuses the two terms in their technical documentation, it's worth asking exactly what they mean.
2. For storing passwords, SHA-256 alone should never be used. SHA-256 is too fast for this task (see point 3 of *What a hash is not*). The current standard is **Argon2id**: slow by design, configurable according to the server's capacity, combined with a different random *salt* per user.
3. For document integrity—contracts, records, files—SHA-256 remains the reference standard. It is the one used by qualified time sealers in the EU.
4. For long-term preservation (decades), it is worth calculating and archiving also a SHA-3 or a SHA-512 alongside the SHA-256; cryptographic prudence recommends not relying on a single function during century-long archives.

Technically, this iterated structure —where the intermediate state is preserved between input blocks— is known as a **Merkle-Damgård** construction, the pattern on which SHA-1, SHA-2 (including SHA-256) and many other classic hash functions are based. SHA-3, on the other hand, abandons Merkle-Damgård in favor of a different architecture called *sponge*.

How SHA-256 works, step by step, in plain words

Imagine you have assembled the most elaborate domino circuit in the world: thousands of tiles, dozens of forks, mechanical bridges, and ramps crossing the entire room, carefully placed piece by piece.

If you give a tap to the first tile, the chain falls in a precise and repeatable sequence. Same assembly, same initial tap → identical final pattern of fallen tiles, over and over again.

Here's the interesting part: move **a single tile** half a centimeter to one side before starting and tap again. A ramp that should have activated remains inert, a bridge doesn't fall, a different fork is triggered. The final pattern of tiles on the floor is completely unrecognizable compared to the first one.

SHA-256 is mathematically this circuit. The text you write is the initial position of the tiles. The algorithm is the tap that releases the cascade. And the final result —what we call a *hash*— is the still photo of the floor when everything has stopped. Change a single comma in the original text and the photo will be radically different. As simple as that, and as drastic as that.

Step 1. Translate the text into binary tiles. Computers don't understand letters; they translate them first into numbers (ASCII) and numbers into binary (ones and zeros). Each letter becomes 8 white or black tiles: *A* is `01000001`, *B* is `01000010`, the space is `00100000`. Your entire text —a word, a contract, a novel— becomes a long row of white and black tiles.

Step 2. Pad to standard size. The circuit processes the row in *stretches* of exactly 512 tiles. If your message doesn't reach a multiple of 512, a marker tile (the one with the value `10000000`) is added right after the text and then zeros until the stretch is complete. The last 64 positions of each stretch are reserved to record the original length of the text. Thus the circuit always knows where the real content ended and where the padding began.

Step 3. Place the eight master tiles. Before starting, we place **eight master tiles** on the table in a precise initial position. These eight tiles are no secret: their initial value is fixed by a public mathematical rule (the square roots of the first eight prime numbers —2, 3, 5, 7, 11, 13, 17, 19— and the first bits of the decimal part of each root). Everyone, in any corner of the planet, starts with the same eight master tiles in the same position. Their destiny is to be pushed and transformed by the avalanche.

Step 4. The big avalanche: sixty-four rounds of pushes. Here the show begins. The first 512-tile stretch of your text is made to collide against the eight master tiles. But they don't fall all at once: the mechanism executes **sixty-four consecutive rounds**. In each round it performs three operations with the tiles:

- **The Tiovivo** (rotation). The tiles move in a circle: the ones on the right pass to the left. No tile is lost or added; they are simply rearranged by making a full turn on the tiovivo. It is a cheap and reversible way to redistribute information.
- **The Logical Funnel** (XOR). The tiles pass through a funnel that compares them two by two: if both are the same color, a white one comes out; if they are different, a black one comes out. It is the simplest operation of binary logic, but combined with the rotations of the tiovivo it becomes extremely powerful for mixing information without losing it.
- **The Overflow** (modular addition). The result is added with a *constant push tile* brought from a public list of sixty-four constants (the cube roots of the first sixty-four prime numbers). If the sum generates extra tiles that don't fit in the provided 32-tile space, those surplus tiles are discarded. The table only has space for 32 tiles, not one more.

At the end of round sixty-four, each of the tiles from your text's stretch has influenced the position of the eight master tiles. The energy of the push has traveled through the entire circuit.

Step 5. Add the next stretch (without resetting). If your text was long and there is another 512-tile stretch to process, **the circuit does not reset**. The eight master tiles stay exactly as the first avalanche left them, and the second stretch is launched against them to activate another sixty-four rounds. It's like adding a new room full of dominoes at the end of the one that just fell: the disorder of the first one entirely conditions how the second one will fall.

Step 6. Take the final photo. When there are no more stretches to process, the avalanche stops. We look at the final position in which the eight master tiles have remained. We translate their configuration into a code of letters and numbers in hexadecimal system. The result is a string of exactly sixty-four characters: that is your SHA-256 seal.

Four properties fall by themselves from how the circuit is assembled:

1. **Determinism.** The same text always produces the same final photo, on any computer in the world. Zero randomness, zero surprises.
2. **Avalanche effect.** A comma added, a capital letter changed, an accent forgotten: the photo turns out completely unrecognizable. This is the extreme sensitivity we already described at the beginning.
3. **One-way.** Given the final photo, you cannot reconstruct the original text. Rotations, funnels, and overflows destroy all directional information about *where each bit came from* and preserve only *what was added in total*.
4. **Collision resistance.** In twenty-five years of public cryptanalysis, no one has managed to find two different texts whose final photos match. And the difficulty of doing so is beyond the computational reach of any reasonably imaginable civilization.

The code appendix that follows implements exactly these six steps in Zig. Now you can read it knowing what each bit operation means, instead of accepting the manipulations blindly.

Technical glossary

For the reader who wants to understand what each operation does. Feel free to skip it: the article can still be understood without it.

ASCII and Unicode — how letters become numbers. Computers don't see letters; they see numbers. A standard called **ASCII** (*American Standard Code for Information Interchange*, from 1963) assigns a specific number to each keyboard character: *A* is 65, *B* is 66, *a* is 97, *0* is 48, space is 32, comma is 44. Modern systems extend this with **Unicode**, which assigns a number to every character in every alphabet in the world: Cyrillic, Arabic, Chinese, Japanese, and even emojis. When you type a character or open a text file, the computer reads the underlying number, not the shape on the screen. SHA-256 works on these numbers, treating any text as a long sequence of digits. That's why it can seal an article in Spanish, a poem in Japanese, and a binary file with the same algorithm.
XOR — the bitwise comparator. XOR (pronounced *"ex-or"*, from *exclusive or*) is one of the simplest operations a computer can do with two binary numbers. It compares two bits position by position and returns: **1** if exactly one of the two is 1 (one but not both), **0** if both are the same (both 0 or both 1). Example: XOR of `1010` and `1100` is `0110`. It has a notable property: it is reversible —if you XOR twice with the same key, you go back to the original. That's why it is the workhorse of cryptography: it mixes bits without losing information, but the result reveals nothing about the inputs if you don't know one of them.
Hexadecimal — counting in base 16. Almost all day-to-day numbers use ten digits (0-9). Hexadecimal uses sixteen: the usual 0-9 plus six letters representing the following values: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Why sixteen? Because computers think in groups of four bits, and four bits can represent exactly sixteen different values —thus, one hexadecimal character maps cleanly to four bits. A SHA-256 fingerprint measures 256 bits, which are exactly **64 hexadecimal characters**. If we wrote it in ordinary decimal, it would take about 78 digits and be more awkward. The choice is aesthetic and compact; the underlying number is the same.
Bit rotation — the binary tiovivo. Imagine a row of seven light bulbs, some on (1) and some off (0): `1 0 1 1 0 0 1`. Rotating to the right by one position consists of taking the bulb on the far right, moving it to the far left, and shifting the others one place to the right: `1 1 0 1 1 0 0`. No bulb is lost or added: they simply dance in a circle. SHA-256 uses bit rotation hundreds of times in each calculation; it is a cheap and lossless way to redistribute information within the state.
Nothing-up-my-sleeve constants — why they come from prime numbers. The eight master tiles and the sixty-four round constants of SHA-256 were not chosen at random. They come from the square and cube roots of the first prime numbers. Why? Because their designers wanted **"nothing-up-my-sleeve"** constants: values whose origin anyone can verify. If someone told you *"trust me: use this 32-bit random number"*, you would reasonably suspect a hidden weakness or a back door. But anyone with a calculator can check that the first 32 bits of the square root of 2 are `0x6a09e667`. The values are mathematical, public, and reproducible: no hidden trick can sneak into the recipe.

Appendix: SHA-256 in readable code

This appendix is for the reader who wants to see the algorithm from the inside. It is a didactic implementation in Zig that follows the FIPS 180-4 specification. It is not the version that Solo2 uses—the real one is in `std.crypto.hash.sha2.Sha256` in the Zig standard library, optimized and audited. But the algorithm is the same: what you see here is, step by step, what happens when that five-character call performs its work.

```
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
}
```

Any rewrite in another language that follows the same structure—initial constants, schedule expansion, sixty-four rounds, accumulation—produces the same result. The algorithm has no secrets: its value resides in the fact that the properties listed above continue to hold up after two decades of public cryptanalysis by thousands of eyes.

If you go back to the foot of this article, you will see a sixty-four character hexadecimal seal. It is the SHA-256 of the text you have just read, in this language. If we translated the article, the seal would be different; if a word of the Spanish version changed, the Spanish seal would change. The seal does not protect the content—for that there are other tools—but it identifies it uniquely. And that, as modest as it sounds, is enough so that no step in the editorial chain can alter what has been said without it being noticed. Everything else—encrypting, signing, identifying—is built on top of this simple idea.

Sources and further reading

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, August 2015. Official specification of the SHA-2 family, including SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, May 2011. Normative version for implementers.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Chapters 5 and 6 cover hash functions and their legitimate and illegitimate uses.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Practical example of using SHA-256 to chain blocks in a structure immutable by construction.
- Regulation (EU) 910/2014 (eIDAS) — framework for qualified time sealers. SHA-256 is the reference function for qualified electronic signatures and seals issued in the EU.
- Reference implementation in Zig: `std.crypto.hash.sha2.Sha256` in the language's official repository (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). It is the optimized and audited version that Solo2 actually uses. Useful for contrasting with the didactic implementation in the appendix.

← PreviousCUADERNOS_LIST_SCHREMS_TITLENext →CUADERNOS_LIST_KILLSWITCH_TITLE

Recent readings

- CUADERNOS_LIST_PREGUNTAS_TITLE
- CUADERNOS_LIST_SELFHOST_TITLE
- CUADERNOS_LIST_IDENTIDAD_TITLE

Take this article wherever you need it.

↓ Markdown ↓ Plain text ↓ PDF

The file is downloaded to your device. From there, you can save it, import it into Solo2, or share it as you wish. Cuadernos does not decide the destination for you.

Wax seal · SHA-256 d1fcec2c0c6a6605c94a697a60ca566f8f2c5fb2b9298d006ed4c71946a88241

Cuadernos Lacre · A publication of Menzuri Gestión S.L. ·
written by R.Eugenio · edited by the team of Solo2.