主题
ES
← Cuadernos Lacre
概念 · 2026年5月19日

← 上一页·下一页 →

SHA-256 究竟是什么

一个仅占六十四个字符的数学指纹，只要原始文本移动一个逗号，它就会完全改变。为什么我们称之为数字火漆印章。

通俗地说： 想象有一台机器可以读取任何文本并返回一个 64 位的字符序列。如果输入的文本完全相同，输出的序列就完全相同。如果你哪怕只是移动了一个逗号，序列就会完全不同。那个序列就是数字火漆。

技术名称背后的简单理念

想象一下有一台机器，只有一个插槽和一个屏幕。你从插槽输入一段文本：一个词、一句话，或者一整本小说。屏幕上随后会出现一段恰好六十四个字符的序列。这段序列，对于专业读者，我们称之为 *hash*（哈希）或 *加密摘要*；对于普通读者，我们可以暂时称之为文本的数学指纹，就像人的指纹一样。

如果你两次输入相同的文本，机器两次都会显示相同的指纹。如果你输入稍有不同的文本——一个逗号被移动，一个大写字母变成小写——机器显示的指纹将与第一个完全不同。不是相似：而是完全不同。这两项属性结合在一起——确定性和敏感性——就是这个简单的理念。SHA-256 的其他一切都是确保它们良好运行的机制。

有必要从一开始就说明这台机器不做什么。它不加密文本。它不隐藏它。它不保存它。机器查看文本，计算指纹，然后忘记该文本。指纹不允许重建产生它的文本；它只允许在给定候选文本的情况下，检查它是否与原始文本一致。这就是为什么我们说它是一个 *单向* 的摘要：有去无回。

哈希不等同于加密

这种混淆很常见，有必要澄清：加密 (cifrar) 和哈希 (hashear) 是不同的操作。加密涉及对文本进行转换，以便只有持有密钥的人才能将其恢复为原始形式。哈希涉及生成文本的指纹，无论是否有密钥，原始文本都永远无法从中恢复。前者在设计上是可逆的；后者在设计上是不可逆的。

实际后果很重要。当一个应用程序说“我们存储您的加密密码”时，总有人拥有解密它的密钥——无论如何，是应用程序本身。当一个应用程序说“我们存储您的哈希密码”时，应用程序本身即使想读也读不出原始密码；它只能检查你输入的密码是否再次产生相同的指纹。对于存储密码，如果操作得当，第二种模型比第一种模型要好得多。稍后我们将看到为什么“操作得当”需要的不仅仅是纯粹的 SHA-256。

使加密哈希函数发挥作用的四个属性

一个配得上 *加密*（cryptográfico）这个形容词的哈希函数必须满足四个属性：

1. **确定性。** 相同的输入总是产生相同的指纹。

2. **雪崩效应。** 输入的微小变化会产生完全不同的指纹，与之前的指纹没有明显的相似性。
3. **抗原像性（抗逆转性）。** 给定一个指纹，在计算上寻找产生它的文本是不可行的。
4. **抗碰撞性。** 在计算上寻找产生相同指纹的两个不同文本是不可行的。

“在计算上不可行”并不意味着“在数学上是不可能的”。它意味着实现这一目标的时间、能源和金钱成本超出了合理可用计算能力总和的几个数量级。对于 SHA-256，即使使用专用硬件的最乐观方案，该阈值也以数千万亿年计。这对于读者的实际目的来说，就等同于“不可能”。

具体到 SHA-256

名字说明了一切。SHA 是 *Secure Hash Algorithm* 的缩写：安全哈希算法。数字 256 表示指纹的大小（以位为单位）：256 位，即 32 字节，以十六进制显示时，就是读者已经识别出的 64 个字符。该标准由美国 NIST（对这类函数进行规范化的机构）于 2001 年作为 SHA-2 家族的一部分发布；该标准的现行版本 FIPS 180-4 是 2015 年发布的。

给尚不清楚什么是位（bit）和字节（byte）的人：

```
1 bit    →  0 或 1       (一个开关：开或关)
1 byte   →  8 bits      (256 种可能的组合)
32 bytes →  256 bits     (SHA-256 指纹)
```

名称末尾的数字 256 表示指纹的大小（以位为单位）。在十六进制（hexadecimal）中——一种使用十六个符号而不是十个符号的计数系统——这 256 位恰好可以容纳在 64 个字符中。这就是你在每本 Cuaderno 底部看到的 64 个字符。

这些维度值得关注一下。256 位允许 2 的 256 次方个不同的值：这是一个包含 78 位十进制数字的数字，比可观测宇宙中估计的原子数量还要大几个数量级。世界上的每一段文本——每一本书、每一封电子邮件、每一条消息——都会落入这些值之一。两个不同文本偶然重合的概率，在实际应用中，与零无异。

代码中的体现

在 Zig 语言（我们编写 Solo2 支撑组件所使用的语言）中，计算文本的 SHA-256 印章如下所示：

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

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

我们刚刚请求 Zig 标准库计算引号内文本的 SHA-256。调用结束后，变量 *resumen* 包含组成印章原始形式的 32 个字节；当以十六进制形式显示在屏幕上时，它们就是出现在本文末尾的 64 个字符。如果我们将 *Cuadernos Lacre* 更改为 *Cuadernos lacre*（少了一个大写字母），印章就会完全改变。这五行代码展示了支撑其余部分的核心属性。对于想了解内部运作的人，我们在文章末尾附带了一个带有逐步注释的易读版算法实现。

为什么我们称之为火漆印章

在 15 至 19 世纪的欧洲通信中，火漆（lacre）被用来封信。一滴熔化的蜡，上面压着一个印章，信件就被打上了不可重复的标记。它并不能防止有心人的偷窥——信纸可以透光阅读，火漆可以被破坏——但它确实能提供证据。在打开信纸之前，收信人就能看到封口是否有改动。火漆并不能阻止破坏，它只是揭示了破坏。

每本 Cuaderno 正文的 SHA-256 在数字版本中起着同样的作用。如果在发布到你阅读之间，文章中的一个词发生了变化，文本底部的十六进制印章将不再与你面前文本的 SHA-256 匹配。任何读者只要写五行代码就能验证。出版物无法在不被印章识破的情况下重写历史。它不防止破坏，但它使破坏可验证。

哈希函数不是什么

有时人们要求 SHA-256 执行四个不属于它的任务：

1. **加密。** 哈希是摘要，不是隐藏。如果你希望文本无法被读取，你需要加密它，而不是哈希它。
2. **验证作者。** 哈希并不说明谁编写了文本，只说明哈希了什么文本。要关联作者身份，需要在哈希之上进行加密签名，而不仅仅是哈希本身。
3. **存储密码。** 这里有一个陷阱需要理解。SHA-256 设计得非常快——这对许多事情都有好处，但对这件事却很糟糕。使用专用硬件的攻击者每秒可以针对 SHA-256 哈希值测试数十亿个密码，直到找到你的密码。存储密码必须使用故意设计的慢速密钥派生函数，如 Argon2、scrypt 或 bcrypt，并结合“盐”（salt，每个用户唯一的随机数据，可防止两个拥有相同密码的人产生相同的哈希值）。
4. **将哈希值读作作者标识符。** 它不是。哈希值标识内容。如果两个人使用 SHA-256 哈希单词 *hola*，他们都会得到相同的摘要——这是核心属性，而不是缺陷：如果摘要不同，我们就无法检查发布内容与接收内容是否一致。

SHA-256 出现在你日常生活的哪里

虽然你看不到，但 SHA-256 支撑着你在互联网上日常使用的很大一部分功能。比特币区块链通过将每个区块的 SHA-256 链接到下一个区块来构建；更改过去的区块将迫使重新计算之后的所有链。Git（世界上大部分代码版本管理的系统）通过其完整内容的 SHA-256（在最近版本中）或其前身 SHA-1（在较旧版本中）来标识每次提交。当你访问网站时，用于验证网站身份的 HTTPS 证书带有一个关联的 SHA-256 指纹。软件下载通常附带开发者发布的 SHA-256，以便你验证文件在传输过程中未被篡改。而且，正如我们所说，出现在每本 Cuadernos Lacre 的底部。

给专业读者

给决策或审计系统的人员的四个操作提醒：

1. 哈希不是加密。如果供应商在其技术文档中混淆了这两个术语，有必要询问其确切含义。
2. 对于存储密码，绝不应直接使用 SHA-256。SHA-256 对于这项任务来说太快了（参见 *哈希函数不是什么* 的第 3 点）。当前的标准是 **Argon2id**：设计上很慢，可根据服务器能力进行配置，并结合每个用户不同的随机“盐”。
3. 对于文件（合同、档案、文件）的完整性，SHA-256 仍然是参考标准。它是欧盟合格时间戳服务机构所使用的标准。
4. 对于长期保存（数十年），建议在 SHA-256 的基础上同时计算并存档 SHA-3 或 SHA-512；加密审慎性建议在长达一个世纪的存档中不要仅依赖单个函数。

从技术角度来看，这种中间状态在输入块之间保持不变的迭代结构被称为 **Merkle-Damgård** 结构，它是 SHA-1、SHA-2（包括 SHA-256）和许多其他经典哈希函数的基础模式。相比之下，SHA-3 放弃了 Merkle-Damgård，转而采用一种称为 *海绵 (sponge)* 的不同架构。

SHA-256 是如何工作的，通俗易懂地分步解析

想象一下，你搭建了世界上最精细的多米诺骨牌线路：成千上万块骨牌、数十个分支、机械桥梁和横跨整个房间的斜坡，每一块都是精心摆放的。

如果你轻触第一块骨牌，骨牌链会按照精确且可重复的顺序倒下。相同的摆放方式，相同的初始触碰 → 骨牌倒下的最终图案也会一次又一次地完全相同。

有趣的地方在这里：在开始前，将 **仅仅一块骨牌** 向旁边移动半厘米，然后再次触碰。原本应该激活的斜坡保持不动，桥梁没有倒下，触发了不同的分支。地面上骨牌的最终图案与第一次相比，将完全无法辨认。

SHA-256 在数学上就是这条线路。你输入的文本就是骨牌的初始位置。算法就是释放连锁反应的那次触碰。而最终结果——我们称之为 *哈希 (hash)* ——就是一切停止后地面的固定照片。改变原始文本中的一个逗号，照片就会截然不同。就是这么简单，也这么彻底。

步骤 1. 将文本翻译成二进制骨牌。 计算机不懂字母；它们首先将其翻译成数字 (ASCII)，再将数字翻译成二进制（1 和 0）。每个字母变成 8 块白色或黑色的骨牌：字母 *A* 是 `01000001`，字母 *B* 是 `01000010`，空格是 `00100000`。你的整个文本——一个单词、一份合同、一部小说——变成了一长排白色和黑色的骨牌。

步骤 2. 填充至标准尺寸。 该线路以正好 512 块骨牌为 *段 (chunks)* 来处理这一长排骨牌。如果你的消息没有达到 512 的倍数，会在文本后面立即添加一块标记骨牌（值为 `10000000`），然后填充零直到补满该段。每一段的最后 64 个位置预留用于记录文本的原始长度。这样，线路始终知道实际内容在哪里结束，填充在哪里开始。

步骤 3. 摆放八块主骨牌。 在开始之前，我们在桌子上精确的初始位置摆放 **八块主骨牌**。这八块骨牌并不是秘密：它们的初始值由公开的数学规则确定（前八个质数——2、3、5, 7、11、13、17、19——的平方根，以及每个平方根小数部分的前几位）。地球上任何角落的任何人，都从相同位置的这八块相同主骨牌开始。它们的命运是被连锁反应推动和转化。

步骤 4. 大连锁反应：六十四轮推动。 表演在这里开始。你文本的第一段 512 块骨牌被撞向八块主骨牌。但它们不会一次性倒下：机械装置会执行 **六十四轮连续操作**。在每一轮中，它对骨牌进行三种操作：

- **旋转木马**（循环移位）。骨牌循环移动：右边的移动到左边。没有骨牌丢失或增加；它们只是在旋转木马上转了一整圈后重新排列。这是一种低成本且可逆的信息重新分配方式。
- **逻辑漏斗** (XOR)。骨牌通过一个成对比较它们的漏斗：如果两块颜色相同，输出白色；如果颜色不同，输出黑色。这是二进制逻辑中最简单的操作，但结合旋转木马的旋转，它在不丢失信息的情况下混合信息的能力变得异常强大。
- **溢出**（模加法）。结果与来自六十四个常量公开列表中的一个 *恒定推动骨牌* 相加（前六十四个质数的立方根）。如果相加产生了超出预设 32 块骨牌空间的额外骨牌，这些多余的骨牌就会被丢弃。桌面上只有 32 块骨牌的空间，一块也不多。

在第六十四轮结束时，你文本段中的每一块骨牌都影响了八块主骨牌的位置。推动的能量已经传遍了整个线路。

步骤 5. 添加下一段（无需重启）。 如果你的文本很长，还有另一段 512 块骨牌需要处理，**线路不会重启**。八块主骨牌保持在第一次连锁反应结束时的状态，第二段骨牌被撞向它们以激活另外六十四轮。这就像是在刚倒下的房间末尾增加一个装满多米诺骨牌的新房间：第一个房间的混乱程度完全决定了第二个房间将如何倒下。

步骤 6. 拍摄最终照片。 当没有更多段需要处理时，连锁反应停止。我们观察八块主骨牌最终停留的位置。我们将它们的布局翻译成十六进制系统的字母和数字代码。结果是一个正好六十四个字符的字符串：这就是你的 SHA-256 印章。

由于线路的搭建方式，四种特性自然显现：

1. **确定性。** 相同的文本在世界上任何一台计算机上总是产生相同的最终照片。零随机性，零意外。
2. **雪崩效应。** 增加一个逗号、改变一个大写字母、忘记一个重音符号：最终的照片将完全无法辨认。这就是我们在开头描述过的极度敏感性。
3. **单向性。** 给定最终照片，你无法还原原始文本。旋转、漏斗和溢出破坏了关于 *每个比特来自哪里* 的所有方向性信息，仅保留了 *总共累加了什么*。
4. **抗碰撞性。** 在二十五年的公开密码分析中，没有人能够找到两段最终照片相同的不同文本。而做到这一点的难度超出了任何可以合理想象的文明的计算能力。

随后的代码附录在 Zig 中实现了这六个步骤。现在你可以在了解每个位操作含义的基础上阅读它，而不是盲目地接受这些数据处理。

技术词汇表

供想要了解每个操作具体作用的读者参考。可以自由跳过：不看这些也不影响对文章的理解。

ASCII 和 Unicode —— 字母如何变成数字。 计算机看不见字母；它们看到的是数字。一种名为 **ASCII**（1963 年）的标准为键盘上的每个字符分配了一个特定数字：*A* 是 65，*B* 是 66，*a* 是 97，*0* 是 48，空格是 32，逗号是 44。现代系统通过 **Unicode** 对其进行了扩展，Unicode 为世界上每种语言的每个字符都分配了一个数字：西里尔字母、阿拉伯字母、中文、日文，甚至包括表情符号 (emoji)。当你输入一个字符或打开一个文本文件时，计算机读取的是底层的数字，而不是屏幕上的形状。SHA-256 作用于这些数字，将任何文本视为一长串数字。这就是为什么它可以用相同的算法为西班牙语文章、日语诗歌和二进制文件加盖印章。
XOR —— 逐位比较器。 XOR（英文 *exclusive or* 的缩写，意为“异或”）是计算机可以对两个二进制数执行的最简单操作之一。它逐位比较两个比特并返回：如果两个比特中恰好有一个为 1（是一个但不是两个都是），则返回 **1**；如果两个比特相同（都是 0 或都是 1），则返回 **0**。例如：`1010` 和 `1100` 的 XOR 结果是 `0110`。它有一个显著的特性：它是可逆的——如果你用相同的密钥执行两次 XOR，就会回到原始值。这就是为什么它是密码学的基石：它在不丢失信息的情况下混合比特，但如果你不知道其中一个输入，结果就不会透露任何关于输入的信息。
十六进制 —— 以 16 为基数计数。 日常生活中几乎所有的数字都使用十个数字 (0-9)。十六进制使用十六个：通常的 0-9 加上六个代表后续值的字母：`A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15。为什么要用十六？因为计算机以四个比特为一组进行思考，而四个比特正好可以代表十六个不同的值——因此，一个十六进制字符可以清晰地对应四个比特。一个 SHA-256 印章长度为 256 比特，正好是 **64 个十六进制字符**。如果我们用普通的十进制来写，它将占约 78 位数字且非常不便。这种选择兼顾了美感和简洁；底层的数字是相同的。
位旋转 —— 二进制旋转木马。 想象一排七个灯泡，有些亮着 (1)，有些熄灭 (0)：`1 0 1 1 0 0 1`。向右旋转一个位置包括取最右边的灯泡，将其移到最左端，并将其余灯泡向右移动一个位置：`1 1 0 1 1 0 0`。没有灯泡丢失或增加：它们只是在循环舞动。SHA-256 在每次计算中都会使用数百次位旋转；这是一种在状态内重新分配信息的低成本且无损的方式。
"Nothing-up-my-sleeve" 常量 —— 为什么它们来自质数。 SHA-256 的八块主骨牌和六十四个轮常量并非随机选择。它们来自前几个质数的平方根和立方根。为什么？因为设计者想要 *“袖子里没有任何猫腻”* (nothing up my sleeve) 的常量：任何人都可以验证其来源的值。如果有人告诉你 *“相信我：使用这个随机的 32 位数字”*，你理所

當然会怀疑其中隐藏了弱点或后门。但任何拥有计算器的人都可以验证 2 的平方根的前 32 位是 `0x6a09e667`。这些值是数学上的、公开的且可重现的：配方中不可能混入任何隐藏的陷阱。

附录：易读代码中的 SHA-256

本附录供希望从内部了解算法的读者参考。这是一个遵循 FIPS 180-4 规范的 Zig 语言教学实现。这不是 Solo2 使用的版本——真实版本位于 Zig 标准库的 `std.crypto.hash.sha2.Sha256` 中，经过优化和审计。但算法是一样的：你在这里看到的，就是那五行代码执行时逐步发生的过程。

```
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
}
```

任何遵循相同结构（初始常量、调度表扩展、六十四轮迭代、累加）的其他语言重写都会产生相同的结果。该算法没有秘密：其价值在于，在经历了二十年的公众密码分析和成千上万双眼睛的审视之后，上述属性依然成立。

如果你回到本文末尾，你会看到一个六十四字符的十六进制印章。它是你刚才阅读的文本（以该语言）的 SHA-256。如果我们翻译这篇文章，印章就会不同；如果西班牙语版本改动了一个词，西班牙语印章就会改变。印章并不保护内容——那是其他工具的作用——但它唯一地标识了内容。尽管这听起来很微不足道，但它足以确保编辑链中的任何环节都无法在不被察觉的情况下篡改内容。其他一切——加密、签名、身份验证——都建立在这个简单的理念之上。

编者按： 当本 Cuadernos 提及公司或产品名称时，并非为了指责。这些产品的开发者所做的工作被数百万人使用和欣赏。我们指出的是结构性问题——是模式问题，而非品牌问题。品牌作为例子出现，是因为读者熟悉它们。

来源及延伸阅读

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, 2015年8月。SHA-2 家族的官方规范，包括 SHA-256。
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, 2011年5月。开发者的规范版本。
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010)。第 5 章和第 6 章涵盖了哈希函数及其合法与非法用途。
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008)。在不可变结构中链接区块使用 SHA-256 的实际案例。
- 欧盟第 910/2014 号条例 (eIDAS) — 合格时间戳服务框架。SHA-256 是欧盟颁发的合格电子签名和电子印章的参考函数。
- Zig 语言参考实现：该语言官方仓库中的 `std.crypto.hash.sha2.Sha256` (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`)。这是 Solo2 实际使用的优化和审计版本。有助于与附录中的教学实现进行对比。

← 上一页Schrems II，五年后的现状下一页 → 自毁开关 (Kill switch) 与机构俘获

最近阅读

- 分析 · 2026年5月18日 真实隐私 vs 表象隐私：你应该问自己的问题
- 分析 · 2026年5月18日 作为专业实践的自托管 (Self-hosting)
- 概念 · 2026年5月18日 24 个单词：什么是加密身份

随身携带本文，以备不时之需。

↓ Markdown ↓ 纯文本 ↓ PDF

文件将下载到您的设备中。您可以从那里将其保存、导入 Solo2 或在任何地方共享。Cuadernos 不会为您决定文件的去向。

火漆印章 · SHA-256 1a935a0bcccfe4a8fb2b09dce9479d218fcfdf2700c5cf3b0f540152b9fc0243

Cuadernos Lacre · Menzuri Gestión S.L. 的出版物 ·
由 R.Eugenio 撰写 · 由 Solo2 团队编辑。