ما هو SHA-256 حقاً

بصمة رياضية تتسع في أربعة وستين حرفاً وتتغير بالكامل إذا تحركت فاصلة واحدة في النص الأصلي. لماذا نسميه ختم الشمع الرقمي.

ببساطة: تخيل آلة تقرأ أي نص وتعيد تسلسلاً من 64 حرفًا. إذا دخل النص متطابقًا، يخرج التسلسل متطابقًا. إذا حركت فاصلة واحدة فقط، يكون التسلسل مختلفًا تمامًا. هذا التسلسل هو الختم الشمعي الرقمي.

الفكرة البسيطة وراء الاسم التقني

تخيل وجود آلة بفتحة واحدة وشاشة واحدة. عبر الفتحة تدخل نصاً: كلمة، جملة، أو رواية كاملة. يظهر على الشاشة، بعد لحظات، تسلسل من أربعة وستين حرفاً بالضبط. هذا التسلسل، نسميه للقارئ المحترف *hash* أو *ملخصاً تشفيرياً*؛ أما للقارئ العام، فيمكننا تسميته حالياً بصمة رياضية للنص، كما هي بصمة الإصبع للإنسان.

إذا أدخلت النص نفسه مرتين، ستعرض الآلة البصمة نفسها في المرتين. إذا أدخلت نصاً مختلفاً قليلاً —فاصلة واحدة مزاحة، حرف كبير أصبح صغيراً— ستعرض الآلة بصمة مختلفة تماماً عن الأولى. ليست متشابهة، بل مختلفة. هاتان الخاصيتان معاً —الحتمية والحساسية— هما الفكرة البسيطة. كل ما تبقى في SHA-256 هو الآلية التي تضمن تنفيذهما جيداً.

من الجيد توضيح ما لا تفعله الآلة منذ البداية. إنها لا تشفر النص، ولا تخفيه، ولا تحفظه. الآلة تنظر إلى النص، تحسب البصمة، وتنسى النص. البصمة لا تسمح بإعادة بناء النص الذي أنتجها؛ إنها تسمح فقط، عند توفر نص مرشح، بالتحقق مما إذا كان يطابق النص الأصلي أم لا. لهذا نقول إنه ملخص *أحادي الاتجاه*: تذهب منه ولا تعود إليه.

الهاش ليس هو نفسه التشفير

الارتباك شائع ومن الجيد توضيحه: التشفير والهاش (hashear) عمليتان مختلفتان. التشفير يتمثل في تحويل النص بحيث يمكن فقط لصاحب المفتاح إعادته إلى شكله الأصلي. أما الهاش فيتمثل في إنتاج بصمة للنص لا يمكن استرداد النص الأصلي منها أبداً، لا بمفتاح ولا بدونه. الأولى قابلة للعكس حسب التصميم؛ والثانية غير قابلة للعكس حسب التصميم.

النتيجة العملية مهمة. عندما يقول تطبيق «نحن نحفظ كلمة مرورك مشفرة»، فهناك شخص يملك المفتاح لفك تشفيرها — التطبيق نفسه في أي حال. عندما يقول تطبيق «نحن نحفظ كلمة مرورك مهشّرة (hasheada)»، فإن التطبيق نفسه لا يمكنه قراءة كلمة المرور الأصلية حتى لو أراد ذلك؛ يمكنه فقط التحقق مما إذا كان ما تكتبه ينتج البصمة نفسها مرة أخرى. النموذج الثاني، إذا نُفذ بشكل صحيح، هو الأفضل لتخزين كلمات المرور. سنرى لاحقاً لماذا يتطلب «التنفيذ بشكل صحيح» شيئاً أكثر من SHA-256 بمفرده.

الخصائص الأربع التي تجعل الهاش التشفيري مفيداً

دالة الهاش التي تستحق وصف *تشفيرية* تفي بأربع خصائص:

1. **الحتمية (Determinism).** المدخل نفسه ينتج دائماً البصمة نفسها.

2. **تأثير الانهيار (Avalanche effect).** أي تغيير بسيط في المدخل ينتج بصمة مختلفة تماماً، دون تشابه مرئي مع السابقة.
3. **مقاومة العكس (Pre-image resistance).** عند توفر بصمة، ليس من الممكن حاسوبياً العثور على النص الذي أنتجها.
4. **مقاومة التصادم (Collision resistance).** ليس من الممكن حاسوبياً العثور على نصين مختلفين ينتجان البصمة نفسها.

«ليس من الممكن حاسوبياً» لا تعني «أنه مستحيل رياضياً». بل تعني أن التكلفة من حيث الوقت والطاقة والمال لتحقيق ذلك تتجاوز بمرات مضاعفة مجموع كل قدرات الحوسبة المتاحة بشكل معقول. بالنسبة لـ SHA-256، تُقاس هذه الحدود بآلاف المليارات من السنين حتى بالنسبة لأكثر التوقعات تفاؤلاً باستخدام عتاد متخصص. وهو ما يعني، للأغراض العملية للقارئ، أنه «غير ممكن».

SHA-256، على وجه التحديد

الاسم يقول كل شيء. SHA هي اختصار لـ *Secure Hash Algorithm*: خوارزمية الهاش الآمنة. يشير الرقم 256 إلى حجم البصمة بالبتات: مائتان وستة وخمسون بتاً، أي اثنان وثلاثون بايت، والتي تظهر في النظام الست عشري كأربعة وستين حرفاً يعرفها القارئ بالفعل. نشر المعيار المعهد الوطني للمعايير والتقنية (NIST) الأمريكي، وهو الهيئة التي تنظم هذا النوع من الدوال، في عام 2001 كجزء من عائلة SHA-2؛ والنسخة الحالية من المعيار، FIPS 180-4، تعود لعام 2015.

لمن لا يزال لا يدرك ما هي البتات والبايتات:

```
بت واحد    ← 0 أو 1          (مفتاح: تشغيل أو إيقاف)
بايت واحد  ← 8 بتات         (256 تشكيلة ممكنة)
32 بايت    ← 256 بت          (بصمة SHA-256)
```

الرقم 256 في نهاية الاسم يخبرنا بحجم البصمة بالبتات. في النظام الست عشري —وهو نظام عد يستخدم ستة عشر رمزاً بدلاً من عشرة— تتسع هذه الـ 256 بت في 64 حرفاً بالضبط. هذه هي الـ 64 حرفاً التي تراها في أسفل كل Cuaderno.

الأبعاد تستحق وقفة. مائتان وستة وخمسون بتاً تسمح باثنين أس مائتين وستة وخمسين قيمة مختلفة: رقم مكون من ثمانية وسبعين خانة عشرية، وهو أكبر بعدة مرات من عدد الذرات المقدر في الكون المرئي. كل نص في العالم —كل كتاب، كل بريد إلكتروني، كل رسالة— يقع على واحدة من تلك القيم. احتمال تطابق نصين مختلفين بالصدفة هو، للأغراض العملية، لا يختلف عن الصفر.

كيف يبدو في الكود

في لغة Zig، وهي اللغة التي نكتب بها الأجزاء التي تدعم Solo2، يبدو حساب ختم SHA-256 لنص ما كما يلي:

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

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

لقد طلبنا للتو من مكتبة Zig القياسية حساب SHA-256 للنص الموجود بين علامتي الاقتباس. بعد الاستدعاء، يحتوي المتغير *resumen* على الاثنين وثلاثين بايتاً التي تشكل الختم في شكله الخام؛ وعندما تُعرض على الشاشة بالنظام الست عشري، تكون هي الأحرف الأربعة والستين التي تظهر في أسفل هذا المقال. إذا قمنا بتغيير *Cuadernos Lacre* إلى *Cuadernos lacre* —تغيير حرف كبير واحد— سيتغير الختم بالكامل. هذه هي، في خمسة أسطر، الخاصية المركزية التي تدعم الباقي. لمن يريد رؤية كيف يعمل داخلياً، قمنا بإدراج نسخة مقروءة من الخوارزمية مع تعليقات خطوة بخطوة في نهاية المقال.

لماذا نسميه ختم الشمع

في المراسلات الأوروبية من القرن الخامس عشر إلى التاسع عشر، كان الشمع يغلق الرسالة. قطرة من الشمع المذاب، وختم يُضغط فوقها، فتصبح الرسالة معلمة بشكل لا يتكرر. لم يكن يحمي المحتوى من المتطفل المصمم —فالورق يمكن قراءته مقابل الضوء، والشمع يمكن كسره— لكنه كان يثبت التلاعب. أي تغيير في الإغلاق كان مرئياً للمستلم حتى قبل فتح الورقة. الشمع لم يمنع الضرر، بل كان يعلنه.

يؤدي SHA-256 الخاص بمتن كل Cuaderno الوظيفة نفسها في نسخته الرقمية. إذا تغيرت كلمة واحدة في المقال بين لحظة نشره ولحظة قراءتك له، فإن الختم الست عشري في أسفل النص لن يطابق SHA-256 للنص الذي أمامك. يمكن لأي قارئ بخمسة أسطر من الكود التحقق من ذلك. لا يمكن للمنشور إعادة كتابة تاريخه دون أن يفضحه الختم. إنه لا يحمي من الضرر، بل يجعله قابلاً للتحقق.

ما ليس عليه الهاش

هناك أربعة استخدامات تُطلب أحياناً من SHA-256 وهي ليست من اختصاصه:

1. **التشفير.** الهاش يلخص؛ ولا يخفي. إذا كنت تريد ألا يُقرأ النص، فأنت بحاجة لتشفيره، وليس تهشيره.
2. **توثيق المؤلف.** الهاش لا يخبرنا بمن كتب النص، بل يخبرنا فقط ما هو النص الذي تم تهشيره. لربط التأليف، هناك حاجة لتوقيع تشفيري فوق الهاش، وليس الهاش بمفرده.
3. **تخزين كلمات المرور.** هنا يوجد فخ من الجيد فهمه. تم تصميم SHA-256 ليكون سريعاً جداً —وهو أمر جيد لأشياء كثيرة، لكنه سيء لهذا الغرض. يمكن للمهاجم باستخدام عتاد متخصص تجربة مليارات كلمات المرور في الثانية مقابل هاش SHA-256 حتى يعثر على كلمتك. لحفظ كلمات المرور، يجب استخدام دوال اشتقاق مفاتيح بطيئة عمداً مثل Argon2 أو scrypt أو bcrypt، مدمجة مع *sal* (بيانات عشوائية فريدة لكل مستخدم، تمنع حصول شخصين يملكان كلمة المرور نفسها على الهاش نفسه).
4. **قراءة الهاش كمعرف للمؤلف.** ليس كذلك. الهاش يحدد المحتوى. إذا قام شخصان بتهشير كلمة *مرحباً* باستخدام SHA-256، فسيحصل كلاهما على الملخص نفسه — وهذه هي الخاصية المركزية، وليست عيباً: لو كانت الملخصات مختلفة، لما تمكنا من التحقق من التطابق بين ما نُشر وما استُلم.

أين يظهر SHA-256 في حياتك اليومية

حتى لو لم تره، فإن SHA-256 يدعم جزءاً كبيراً مما تستخدمه يومياً على الإنترنت. تُبنى سلسلة كتل بيتكوين (Bitcoin) بربط SHA-256 لكل كتلة بالكتلة التالية؛ تغيير كتلة سابقة يجبر على إعادة حساب السلسلة اللاحقة بالكامل. Git، النظام الذي تُدار به إصدارات الكود لنصف العالم، يحدد كل عملية تأكيد (commit) بواسطة SHA-256 (في الإصدارات الحديثة) أو بسلفه SHA-1 (في الإصدارات الأقدم) لمحتواها الكامل. شهادات HTTPS التي تتحقق من هوية موقع ويب عند دخوله تحمل بصمة SHA-256 مرتبطة بها. غالباً ما تُرفق تنزيلات البرامج بـ SHA-256 ينشره المطور لتتحقق من أن الملف لم يتغير في الطريق. وكما قلنا، في أسفل كل Cuadernos Lacre.

للقارئ المحترف

أربع تذكيرات تشغيلية لمن يتخذ القرار أو يدقق في الأنظمة:

1. الهاش ليس تشفيراً. إذا خلط مزود بين المصطلحين في توثيقه التقني، فمن الجيد الاستفسار عما يقصده بالضبط.
2. لتخزين كلمات المرور، يجب ألا يُستخدم SHA-256 بمفرده أبداً. SHA-256 سريع جداً لهذه المهمة (انظر النقطة 3 في *ما ليس عليه الهاش*). المعيار الحالي هو **Argon2id**: بطيء حسب التصميم، وقابل للتهيئة وفقاً لقدرة الخادم، ومدمج مع *sal* عشوائي مختلف لكل مستخدم.
3. لنزاهة المستندات —العقود، السجلات، الملفات— لا يزال SHA-256 هو المعيار المرجعي. وهو ما يستخدمه موثقو الطوابع الزمنية المؤهلون في الاتحاد الأوروبي.
4. للحفظ طويل الأمد (عقود)، من الجيد حساب وأرشفة SHA-3 أو SHA-512 بجانب SHA-256؛ الحذر التشفيري ينصح بعدم الاعتماد على دالة واحدة خلال الأرشفة لقرون.

من الناحية التقنية، تُعرف هذه البنية المتكررة — حيث يتم الحفاظ على الحالة المتوسطة بين كتل الإدخال — باسم بناء **Merkle-Damgård**، وهو النمط الذي تعتمد عليه SHA-1 و SHA-2 (بما في ذلك SHA-256) والعديد من دوال التجزئة الكلاسيكية الأخرى. على النقيض من ذلك، تتخلى SHA-3 عن Merkle-Damgård لصالح بنية مختلفة تسمى *sponge* (الإسفنجة).

كيف يعمل SHA-256، خطوة بخطوة، بكلمات بسيطة

تخيل أنك قمت ببناء أكثر دائرة دومينو تعقيداً في العالم: آلاف القطع، عشرات التفرعات، جسور ميكانيكية ومنحدرات تعبر الغرفة بأكملها، موضوعة بعناية قطعة بقطعة.

إذا لمست القطعة الأولى، تسقط السلسلة في تتابع دقيق وقابل للتكرار. نفس التركيب، نفس اللمسة الأولى ← نفس النمط النهائي للقطع الساقطة، مراراً وتكراراً.

هنا مكمن الإثارة: حرك **قطعة واحدة فقط** لمسافة نصف سنتيمتر إلى الجانب قبل البدء والمسها مرة أخرى. المنحدر الذي كان يجب أن يتفعل يظل خاملاً، الجسر لا يسقط، وتفرع مختلف تماماً ينطلق. النمط النهائي للقطع على الأرض لا يمكن التعرف عليه تماماً مقارنة بالأول.

SHA-256 هو رياضياً هذه الدائرة. النص الذي تكتبه هو الوضع الأولي للقطع. الخوارزمية هي اللمسة التي تطلق الشلال. والنتيجة النهائية — ما نسميه *hash* (التجزئة) — هي الصورة الثابتة للأرض عندما يتوقف كل شيء. غيّر فاصلة واحدة في النص الأصلي وستكون الصورة مختلفة جذرياً. بهذه البساطة، وبهذه القوة.

الخطوة 1. ترجمة النص إلى قطع ثنائية. أجهزة الكمبيوتر لا تفهم الحروف؛ بل تترجمها أولاً إلى أرقام (ASCII) والأرقام إلى ثنائي (واحد وأصفار). يتم تحويل كل حرف إلى 8 قطع بيضاء أو سوداء: حرف *A* هو `01000001`، وحرف *B* هو `01000010`، والمسافة هي `00100000`. نصك بالكامل — كلمة، عقد، رواية — يصبح صفاً طويلاً من القطع البيضاء والسوداء.

الخطوة 2. التكملة حتى الحجم القياسي. تعالج الدائرة الصف في *مقاطع* طول كل منها بالضبط 512 قطعة. إذا لم تصل رسالتك إلى مضاعف 512، يتم إضافة قطعة علامة (قيمتها `10000000`) مباشرة بعد النص ثم أصفار حتى يكتمل المقطع. يتم حجز آخر 64 موضعاً من كل مقطع لتسجيل الطول الأصلي للنص. هكذا تعرف الدائرة دائماً أين انتهى المحتوى الحقيقي وأين بدأ الحشو.

الخطوة 3. وضع القطع الثمانية الرئيسية. قبل البدء، نضع على الطاولة **ثماني قطع رئيسية** في وضع أولي دقيق. هذه القطع الثمانية ليست سراً: يتم تحديد قيمتها الأولية بواسطة قاعدة رياضية عامة (الجذور التربيعية لأول ثمانية أرقام أولية — 2، 3، 5، 7، 11، 13، 17، 19 — والبتات الأولى من الجزء العشري لكل جذر). الجميع، في أي ركن من أركان الكوكب، يبدأ بنفس القطع الثمانية الرئيسية في نفس الوضع. قدرها هو أن يتم دفعها وتحويلها بواسطة الانهيار.

الخطوة 4. الانهيار الكبير: أربعة وستون جولة من الدفع. هنا يبدأ العرض. المقطع الأول المكون من 512 قطعة من نصك يصطدم بالقطع الثمانية الرئيسية. لكنها لا تسقط دفعة واحدة: تنفذ الآلية **أربعة وستين جولة متتالية**. في كل جولة تقوم بثلاث عمليات بالقطع:

- **الدوامة** (الدوران). تتحرك القطع في دائرة: القطع الموجودة على اليمين تنتقل إلى اليسار. لا تفقد أي قطعة ولا تضاف أي قطعة؛ يتم ببساطة إعادة ترتيبها من خلال دورة كاملة. إنها طريقة رخيصة وقابلة للعكس لإعادة توزيع المعلومات.
- **القمع المنطقي** (XOR). تمر القطع عبر قمع يقارن بينها اثنتين اثنتين: إذا كانت القطعتان من نفس اللون، تخرج قطعة بيضاء؛ وإذا كانتا مختلفتين، تخرج قطعة سوداء. إنها أبسط عملية في المنطق الثنائي، ولكن عند دمجها مع دورات الدوامة تصبح قوية جداً لمزج المعلومات دون فقدانها.
- **الفيض** (الجمع المعياري). تُجمع النتيجة مع *قطعة دفع ثابتة* مأخوذة من قائمة عامة مكونة من أربع وستين ثابتاً (الجذور التكعيبية لأول أربعة وستين رقماً أولياً). إذا أدى الجمع إلى قطع إضافية لا تتسع لها مساحة الـ 32 قطعة المقررة، يتم استبعاد تلك القطع الزائدة. الطاولة تتسع لـ 32 قطعة فقط، لا أكثر.

في نهاية الجولة الرابعة والستين، تكون كل قطعة من مقطع نصك قد أثرت في وضع القطع الثمانية الرئيسية. لقد انتقلت طاقة الدفع عبر الدائرة بأكملها.

الخطوة 5. إضافة المقطع التالي (بدون إعادة ضبط). إذا كان نصك طويلاً وبقي مقطع آخر من 512 قطعة لمعالجته، **فإن الدائرة لا تعيد ضبط نفسها.** تظل القطع الثمانية الرئيسية كما تركتها الانهيار الأول، ويتم إطلاق المقطع الثاني ضدها لتفعيل أربعة وستين جولة أخرى. الأمر يشبه إضافة غرفة جديدة مليئة بالدومينو في نهاية الغرفة التي سقطت للتو: الفوضى في الأولى تحدد تماماً كيف ستسقط الثانية.

الخطوة 6. التقاط الصورة النهائية. عندما لا يتبقى أي مقاطع أخرى للمعالجة، يتوقف الانهيار. ننظر إلى الوضع النهائي الذي استقرت عليه القطع الثمانية الرئيسية. نترجم تكوينها إلى كود من الحروف والأرقام بنظام الست عشري. النتيجة هي سلسلة مكونة بالضبط من أربعة وستين حرفاً: هذا هو ختم SHA-256 الخاص بك.

أربع خصائص تظهر تلقائياً من كيفية بناء الدائرة:

1. **الحتمية.** ينتج نفس النص دائماً نفس الصورة النهائية، على أي جهاز كمبيوتر في العالم. صفر عشوائية، صفر مفاجآت.
2. **تأثير الانهيار.** فاصلة مضافة، حرف كبير تم تغييره، علامة نطق منسية: الصورة الناتجة لا يمكن التعرف عليها تماماً. هذه هي الحساسية القصوى التي وصفناها بالفعل في البداية.
3. **اتجاه واحد فقط.** بالنظر إلى الصورة النهائية، لا يمكنك إعادة بناء النص الأصلي. الدورات والأقماع والفيض تدمر كل المعلومات الاتجاهية حول *من أين أتى كل بت* وتحافظ فقط على *ما تم جمعه في المجموع*.
4. **مقاومة التصادم.** في خمسة وعشرين عاماً من تحليل التشفير العام، لم يتمكن أحد من العثور على نصين مختلفين تتطابق صورهما النهائية. وصعوبة القيام بذلك خارج نطاق القدرة الحسابية لأي حضارة يمكن تخيلها بشكل معقول.

ملحق الكود التالي ينفذ بالضبط هذه الخطوات الست في Zig. الآن يمكنك قراءته وأنت تعرف معنى كل عملية بتات، بدلاً من قبول التلاعبات بشكل أعمى.

قاموس المصطلحات التقنية

للقارئ الذي يريد فهم ما تفعله كل عملية. يمكنك تخطيه بحرية: المقال يظل مفهوماً بدونه.

ASCII و Unicode — كيف تصبح الحروف أرقاماً. لا ترى أجهزة الكمبيوتر الحروف؛ بل ترى الأرقام. معيار يسمى **ASCII** (*الرمز الأمريكي القياسي لتبادل المعلومات*، من عام 1963) يخصص لكل حرف لوحة مفاتيح رقماً معيناً: حرف *A* هو 65، وحرف *B* هو 66، وحرف *a* هو 97، والرقم *0* هو 48، والمسافة هي 32، والفاصلة هي 44. الأنظمة الحديثة توسع ذلك بـ **Unicode**، الذي يخصص رقماً لكل حرف من كل أبجدية في العالم: السيريلية، العربية، الصينية، اليابانية، وحتى الرموز التعبيرية (emojis). عندما تكتب حرفاً أو تفتح ملفاً نصياً، يقرأ الكمبيوتر الرقم الأساسي، وليس الشكل على الشاشة. يعمل SHA-256 على هذه الأرقام، حيث يعامل أي نص كسلسلة طويلة من الأرقام. لهذا السبب يمكنه ختم مقال بالإسبانية، وقصيدة باليابانية، وملف ثنائي بنفس الخوارزمية.
XOR — المقارن بت بـ بت. XOR (تُنطق «*إكس أور*»، من الإنجليزية *exclusive or*، «أو الحصرية») هي واحدة من أبسط العمليات التي يمكن للكمبيوتر القيام بها مع رقمين ثنائيين. يقارن بتين في نفس الموضع ويعيد: **1** إذا كان واحد فقط من الاثنين هو 1 (واحد ولكن ليس كلاهما)، و **0** إذا كان الاثنان متساويين (كلاهما 0 أو كلاهما 1). مثال: XOR لـ `1010` و `1100` هو `0110`. لها خاصية ملحوظة: هي قابلة للعكس — إذا قمت بعمل XOR مرتين بنفس المفتاح، فستعود إلى الأصل. لهذا السبب هي العمود الفقري للتشفير: تمزج البتات دون فقدان المعلومات، لكن النتيجة لا تكشف شيئاً عن المدخلات إذا لم تكن تعرف أحدها.
Hexadecimal — العد على أساس 16. تستخدم جميع أرقام الحياة اليومية تقريباً عشرة أرقام (0-9). يستخدم النظام الست عشري ستة عشر رقماً: الأرقام المعتادة 0-9 بالإضافة إلى ستة حروف تمثل القيم التالية: `A` = 10، `B` = 11، `C` = 12، `D` = 13، `E` = 14، `F` = 15. لماذا ستة عشر؟ لأن أجهزة الكمبيوتر تفكر في مجموعات من أربعة بتات، وأربعة بتات يمكن أن تمثل بالضبط ستة عشر قيمة مختلفة — وبالتالي، حرف ست عشري واحد يتوافق بوضوح مع أربعة بتات. يبلغ طول ختم SHA-256 حوالي 256 بت، وهو ما يمثل بالضبط **64 حرفاً ست عشرياً**. إذا كتبناه بالنظام العشري العادي، فسيشغل حوالي 78 رقماً وسيكون أقل راحة. الاختيار جمالي ومدمج؛ الرقم الأساسي هو نفسه.
دوران البتات — الدوامة الثنائية. تخيل صفاً من سبعة مصابيح، بعضها مضاء (1) وبعضها مطفأ (0): `1 0 1 1 0 0 1`. يتكون الدوران إلى اليمين بمقدار موضع واحد من أخذ المصباح الموجود في أقصى اليمين، ونقله إلى أقصى اليسار وإزاحة المصابيح الأخرى موضعاً واحداً إلى اليمين: `1 1 0 1 1 0 0`. لا يضيع أي مصباح ولا يضاف أي مصباح:

ببساطة ترقص في دائرة. يستخدم SHA-256 دوران البتات مئات المرات في كل عملية حسابية؛ إنها طريقة رخيصة وبدون فقدان لإعادة توزيع المعلومات داخل الحالة.
ثوابت *«nothing-up-my-sleeve»* — لماذا تأتي من أرقام أولية. لم يتم اختيار القطع الثمانية الرئيسية والثوابت الأربعة والستين لجولات SHA-256 بشكل عشوائي. إنها تأتي من الجذور التربيعية والتكعيبية لأول أرقام أولية. لماذا؟ لأن مصمميها أرادوا ثوابت *«بدون أي شيء مخفي»*: قيم يمكن لأي شخص التحقق من أصلها. إذا قال لك شخص ما *«ثق بي: استخدم هذا الرقم العشوائي المكون من 32 بت»*، فستشك بشكل معقول في وجود ضعف مخفي أو باب خلفي. ولكن يمكن لأي شخص لديه آلة حاسبة التحقق من أن أول 32 بت من الجذر التربيعي لـ 2 هي `0x6a09e667`. القيم رياضية وعامة وقابلة للتكرار: لا يمكن لأي خدعة مخفية أن تتسلل إلى الوصفة.

ملحق: SHA-256 في كود مقروء

هذا الملحق مخصص للقارئ الذي يريد رؤية الخوارزمية من الداخل. إنه تطبيق تعليمي بلغة Zig يتبع مواصفات FIPS 180-4. ليس هو الإصدار الذي يستخدمه Solo2 — الإصدار الحقيقي موجود في `std.crypto.hash.sha2.Sha256` من مكتبة Zig القياسية، وهو محسن ومدقق—. لكن الخوارزمية هي نفسها: ما تراه هنا هو، خطوة بخطوة، ما يحدث عندما ينفذ ذلك الاستدعاء المكون من خمسة أحرف مهمته.

```
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

                  Expansión del schedule: 16 palabras → 64. Las nuevas se obtienen .1 //
                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
                                                                              {

                                    .Variables de trabajo: copia del estado actual .2 //
                ;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]

                                                .rondas de mezcla no lineal 64 .3 //
                                .'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
                                                                              {

                                    .Acumular las variables de trabajo en el estado .4 //
                           ;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
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
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 //
                                                                        {
```

أي إعادة كتابة بلغة أخرى تتبع الهيكل نفسه —الثوابت الأولية، توسيع الجدول، أربع وستون جولة، التراكم— تنتج النتيجة نفسها. الخوارزمية ليس لها أسرار: قيمتها تكمن في أن الخصائص المذكورة أعلاه لا تزال صامدة بعد عقدين من تحليل التشفير العام من قبل آلاف العيون.

إذا عدت إلى أسفل هذا المقال، سترى ختماً ست عشرياً مكوناً من أربعة وستين حرفاً. إنه SHA-256 للنص الذي قرأته للتو، بهذه اللغة. إذا قمنا بترجمة المقال، فسيكون الختم مختلفاً؛ وإذا تغيرت كلمة واحدة من النسخة العربية، سيتغير الختم العربي. الختم لا يحمي المحتوى —فلذلك توجد أدوات أخرى— بل يحدده بشكل فريد. وهذا، مهما بدا متواضعاً، يكفي لضمان عدم تمكن أي خطوة في السلسلة التحريرية من تغيير ما قيل دون أن يلاحظ أحد. الباقي — التشفير، التوقيع، التعريف— يُبنى فوق هذه الفكرة البسيطة.

ملاحظة هيئة التحرير: عندما تذكر هذه الـ Cuadernos شركات أو منتجات، فليس ذلك للاتهام. فالذين يبنونها يقومون بأعمال يستخدمها الملايين ويقدرونها. ما نشير إليه هو هيكلي — النموذج، وليس العلامة التجارية. تظهر العلامات التجارية كأمثلة لأنها التي يتعرف عليها القارئ.

المصادر ومزيد من القراءة

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*، أغسطس 2015. المواصفات الرسمية لعائلة SHA-2، بما في ذلك SHA-256.
- IETF، RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*، مايو 2011. الإصدار المعياري للمنفذين.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). الفصول 5 و 6 تغطي دوال الهاش واستخداماتها المشروعة وغير المشروعة.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). مثال عملي على استخدام SHA-256 لربط الكتل في هيكل غير قابل للتغيير حسب البناء.
- لائحة (الاتحاد الأوروبي) 910/2014 (eIDAS) — إطار عمل موثقي الطوابع الزمنية المؤهلين. SHA-256 هو الدالة المرجعية للتوقيعات والأختام الإلكترونية المؤهلة الصادرة في الاتحاد الأوروبي.

- التطبيق المرجعي في Zig: `std.crypto.hash.sha2.Sha256` في المستودع الرسمي للغة (*github.com/ziglang/zig* ← `lib/std/crypto/sha2.zig`). إنه الإصدار المحسن والمدقق الذي يستخدمه في الواقع Solo2. مفيد للمقارنة مع التطبيق التعليمي للملحق.

← السابقSchrems II، بعد مرور خمس سنواتالتالي → Kill switch والاستحواذ المؤسسي

قراءات حديثة

- تحليل · ١٨ مايو ٢٠٢٦ الخصوصية الحقيقية مقابل الظاهرية: الأسئلة التي ينبغي طرحها
- تحليل · ١٨ مايو ٢٠٢٦ الاستضافة الذاتية كممارسة مهنية
- مفهوم · ١٨ مايو ٢٠٢٦ الـ 24 كلمة: ما هي الهوية التشفيرية

خذ هذا المقال معك أينما احتجت إليه.

↓ ماركداون ↓ نص عادي ↓ PDF

سيتم تنزيل الملف على جهازك. من هناك يمكنك حفظه، أو استيراده إلى Solo2 أو مشاركته أينما تريد. Cuadernos لا تقرر الوجهة نيابة عنك.

ختم شمعي · SHA-256 f30872d258f16511bba085a33d74277077174e690a1908c269fe59dee33cf5c8

Cuadernos Lacre · منشور لشركة Menzuri Gestión S.L. ·
بقلم R.Eugenio · تحرير فريق Solo2.