SHA-256 वास्तव में क्या है

एक गणितीय छाप जो चौंसठ वर्णों में समा जाती है और मूल पाठ की एक भी कॉमा बदलने पर पूरी तरह बदल जाती है। हम इसे डिजिटल लाख की मुहर क्यों कहते हैं।

सीधे शब्दों में कहें तो: एक ऐसी मशीन की कल्पना करें जो किसी भी टेक्स्ट को पढ़ती है और 64 वर्णों का एक अनुक्रम देती है। अगर टेक्स्ट बिल्कुल समान है, तो अनुक्रम भी समान होगा। अगर आप एक कॉमा भी हटाते हैं, तो अनुक्रम पूरी तरह से अलग हो जाता है। वह अनुक्रम डिजिटल सीलिंग वैक्स है।

तकनीकी नाम के पीछे का सरल विचार

कल्पना करें कि एक मशीन है जिसमें एक स्लॉट और एक स्क्रीन है। स्लॉट के माध्यम से आप एक पाठ डालते हैं: एक शब्द, एक वाक्य, एक पूरा उपन्यास। स्क्रीन पर, कुछ क्षण बाद, ठीक चौंसठ वर्णों का एक क्रम दिखाई देता है। उस क्रम को, पेशेवर पाठक के लिए हम *hash* या *क्रिप्टोग्राफिक सारांश* कहते हैं; सामान्य पाठक के लिए, हम इसे अभी के लिए पाठ की गणितीय छाप कह सकते हैं, जैसे कि फिंगरप्रिंट किसी व्यक्ति की पहचान होती है।

यदि आप एक ही पाठ को दो बार डालते हैं, तो मशीन दोनों बार एक ही छाप दिखाती है। यदि आप थोड़ा अलग पाठ डालते हैं — एक बदली हुई कॉमा, एक बड़ा अक्षर जो छोटा हो जाता है — तो मशीन पहली वाली से पूरी तरह अलग छाप दिखाती है। मिलती-जुलती नहीं: अलग। ये दो गुण एक साथ — नियतत्ववाद (determinism) और संवेदनशीलता — सरल विचार हैं। SHA-256 की बाकी सभी चीजें वह मशीनरी हैं जो इन्हें सही ढंग से लागू करती हैं।

शुरुआत में ही यह कहना उचित है कि मशीन क्या नहीं करती है। यह पाठ को एन्क्रिप्ट नहीं करती है। इसे छिपाती नहीं है। इसे सहेजती नहीं है। मशीन पाठ को देखती है, छाप की गणना करती है, और पाठ को भूल जाती है। छाप से उस पाठ को फिर से बनाना संभव नहीं है जिसने इसे उत्पन्न किया था; यह केवल एक पाठ की जांच करने की अनुमति देती है कि वह मूल से मेल खाता है या नहीं। इसीलिए हम कहते हैं कि यह एक *एकतरफा* सारांश है: इसमें जाया जा सकता है, वापस नहीं आया जा सकता।

Hash एन्क्रिप्शन के समान नहीं है

भ्रम अक्सर होता है और इसे स्पष्ट करना उचित है: एन्क्रिप्शन और हैशिंग अलग-अलग ऑपरेशन हैं। एन्क्रिप्शन में पाठ को इस तरह से बदलना शामिल है कि केवल कुंजी का स्वामी ही उसे उसके मूल रूप में वापस ला सके। हैशिंग में पाठ की एक छाप बनाना शामिल है जिससे मूल पाठ को कभी भी पुनर्प्राप्त नहीं किया जा सकता, न तो कुंजी के साथ और न ही बिना कुंजी के। पहला डिज़ाइन द्वारा प्रतिवर्ती (reversible) है; दूसरा डिज़ाइन द्वारा अपरिवर्तनीय (irreversible) है।

इसका व्यावहारिक परिणाम महत्वपूर्ण है। जब कोई एप्लिकेशन कहता है «हम आपका पासवर्ड एन्क्रिप्टेड रखते हैं», तो कोई है जिसके पास उसे डिक्रिप्ट करने की कुंजी है — किसी भी मामले में एप्लिकेशन स्वयं। जब कोई एप्लिकेशन कहता है «हम आपका पासवर्ड हैश रखते हैं», तो एप्लिकेशन स्वयं भी मूल पासवर्ड नहीं पढ़ सकता चाहे वह चाहे; वह केवल यह जांच सकता है कि आप जो लिखते हैं वह फिर से वही छाप उत्पन्न करता है या नहीं। दूसरा मॉडल, यदि सही तरीके से किया जाए, तो पासवर्ड स्टोर करने के लिए पहले वाले की तुलना में कहीं अधिक बेहतर है। बाद में हम देखेंगे कि क्यों «सही तरीके से किए जाने» के लिए केवल SHA-256 से अधिक की आवश्यकता होती है।

वे चार गुण जो क्रिप्टोग्राफिक हैश को उपयोगी बनाते हैं

एक हैश फंक्शन जो *क्रिप्टोग्राफिक* विशेषण के योग्य है, चार गुणों को पूरा करता है:

1. **नियतत्ववाद (Determinism)।** समान इनपुट हमेशा समान छाप उत्पन्न करता है।
2. **एवलांच प्रभाव (Avalanche effect)।** इनपुट में एक छोटा सा बदलाव पूरी तरह से अलग छाप उत्पन्न करता है, जिसमें पिछले वाले से कोई स्पष्ट समानता नहीं होती।
3. **प्रतिलोमन प्रतिरोध (Resistance to inversion)।** एक छाप को देखते हुए, उस पाठ को खोजना कम्प्यूटेशनल रूप से व्यवहार्य नहीं है जिसने इसे उत्पन्न किया था।
4. **टकराव प्रतिरोध (Collision resistance)।** दो अलग-अलग पाठ खोजना कम्प्यूटेशनल रूप से व्यवहार्य नहीं है जो समान छाप उत्पन्न करते हों।

«कम्प्यूटेशनल रूप से व्यवहार्य नहीं» का मतलब यह नहीं है कि «यह गणितीय रूप से असंभव है»। इसका मतलब यह है कि इसे प्राप्त करने में लगने वाला समय, ऊर्जा और धन की लागत उचित रूप से उपलब्ध कुल कंप्यूटिंग क्षमता के परिमाण से कई गुना अधिक है। SHA-256 के लिए, वह सीमा हजारों ट्रिलियन वर्षों में मापी जाती है, यहाँ तक कि विशेष हार्डवेयर के साथ सबसे आशावादी दृष्टिकोणों के लिए भी। जो पाठक के व्यावहारिक उद्देश्यों के लिए «नहीं किया जा सकता» के समान है।

SHA-256, विशेष रूप से

नाम सब कुछ बताता है। SHA का अर्थ है *Secure Hash Algorithm*: सुरक्षित हैश एल्गोरिदम। संख्या 256 बिट्स में छाप के आकार को दर्शाती है: दो सौ छप्पन बिट्स, यानी बत्तीस बाइट्स, जो हेक्साडेसिमल में दिखाने पर वे चौंसठ वर्ण होते हैं जिन्हें पाठक पहले से पहचानता है। मानक को अमेरिकी NIST द्वारा प्रकाशित किया गया था, वह संस्था जो इस प्रकार के कार्यों को मानकीकृत करती है, 2001 में SHA-2 परिवार के हिस्से के रूप में; मानक का वर्तमान संस्करण, FIPS 180-4, 2015 का है।

उनके लिए जिन्हें अभी तक नहीं पता कि बिट्स और बाइट्स क्या हैं:

```
1 बिट     →  0 या 1          (एक स्विच: चालू या बंद)
1 बाइट    →  8 बिट्स         (256 संभावित संयोजन)
32 बाइट्स →  256 बिट्स      (SHA-256 छाप)
```

नाम के अंत में संख्या 256 बिट्स में छाप का आकार बताती है। हेक्साडेसिमल में — दस के बजाय सोलह प्रतीकों वाली एक संख्या प्रणाली — वे 256 बिट्स ठीक 64 वर्णों में समा जाते हैं। ये वे 64 वर्ण हैं जो आप प्रत्येक Cuaderno के नीचे देखते हैं।

आयामों पर एक पल के लिए विचार करना उचित है। दो सौ छप्पन बिट्स दो की घात दो सौ छप्पन अलग-अलग मानों की अनुमति देते हैं: अठहत्तर दशमलव अंकों वाली एक संख्या, जो अवलोकन योग्य ब्रह्मांड में परमाणुओं की अनुमानित संख्या से कई गुना अधिक है। दुनिया का हर पाठ — हर किताब, हर ईमेल, हर संदेश — इन मानों में से किसी एक पर गिरता है। संयोग से दो अलग-अलग पाठों के मेल खाने की संभावना व्यावहारिक रूप से शून्य के बराबर है।

कोड में यह कैसा दिखता है

Zig में, वह भाषा जिसमें हम Solo2 का समर्थन करने वाले हिस्से लिखते हैं, किसी पाठ के SHA-256 मुहर की गणना इस तरह दिखती है:

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

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

हमने अभी Zig की मानक लाइब्रेरी से उद्धरण चिह्नों के बीच के पाठ के SHA-256 की गणना करने के लिए कहा है। कॉल के बाद, वेरिएबल *resumen* में बत्तीस बाइट्स होते हैं जो मुहर को उसके कच्चे रूप में बनाते हैं; जब उन्हें स्क्रीन पर हेक्साडेसिमल में दिखाया जाता है, तो वे चौंसठ वर्ण होते हैं जो इस लेख के नीचे दिखाई देते हैं। यदि हम *Cuadernos Lacre* को *Cuadernos lacre* में बदलते हैं — एक बड़ा अक्षर कम — तो मुहर पूरी तरह से बदल जाएगी। पाँच पंक्तियों में, यह वह मुख्य गुण है जो बाकी सब को सहारा देता है। उनके लिए जो यह देखना चाहते हैं कि यह आंतरिक रूप से कैसे काम करता है, लेख के अंत में हम चरण-दर-चरण टिप्पणियों के साथ एल्गोरिदम का एक पठनीय संस्करण शामिल करते हैं।

हम इसे लाख की मुहर क्यों कहते हैं

पंद्रहवीं से उन्नीसवीं शताब्दी के यूरोपीय पत्राचार में, लाख (सीलिंग वैक्स) पत्र को बंद करता था। पिघले हुए मोम की एक बूंद, उस पर दबाई गई मुहर, और पत्र पर एक अनूठी छाप बन जाती थी। यह सामग्री को किसी दृढ़ व्यक्ति से सुरक्षित नहीं रखता था — कागज को रोशनी के सामने पढ़ा जा सकता था, लाख को तोड़ा जा सकता था — लेकिन यह इसका प्रमाण देता था। मुहर में कोई भी बदलाव कागज खोलने से पहले ही प्राप्तकर्ता को दिखाई दे जाता था। लाख नुकसान को रोकता नहीं था; यह उसकी घोषणा करता था।

प्रत्येक Cuaderno के मुख्य भाग का SHA-256 डिजिटल संस्करण में वही कार्य करता है। यदि लेख प्रकाशित होने और आपके द्वारा इसे पढ़ने के बीच लेख का एक भी शब्द बदल जाता है, तो पाठ के नीचे की हेक्साडेसिमल मुहर आपके सामने मौजूद पाठ के SHA-256 से मेल नहीं खाएगी। कोड की पाँच पंक्तियों वाला कोई भी पाठक इसकी जाँच कर सकता है। मुहर के बिना प्रकाशन अपने इतिहास को फिर से नहीं लिख सकता। यह नुकसान से बचाता नहीं है; यह इसे सत्यापन योग्य बनाता है।

हैश क्या नहीं है

कभी-कभी SHA-256 से चार ऐसे उपयोगों की अपेक्षा की जाती है जो इसके नहीं हैं:

1. **एन्क्रिप्ट करना।** हैश सारांश प्रस्तुत करता है; यह छिपाता नहीं है। यदि आप चाहते हैं कि पाठ पढ़ने योग्य न हो, तो आपको उसे एन्क्रिप्ट करने की आवश्यकता है, हैश करने की नहीं।
2. **लेखक को प्रमाणित करना।** हैश यह नहीं बताता कि पाठ किसने लिखा है, केवल यह कि कौन सा पाठ हैश किया गया था। लेखकत्व जोड़ने के लिए हैश के ऊपर एक क्रिप्टोग्राफिक हस्ताक्षर की आवश्यकता होती है, न कि केवल हैश की।
3. **पासवर्ड स्टोर करना।** यहाँ एक जाल है जिसे समझना उचित है। SHA-256 को बहुत तेज़ होने के लिए डिज़ाइन किया गया है — जो कई चीजों के लिए अच्छा है, लेकिन इसके लिए बुरा है। विशेष हार्डवेयर वाला एक हमलावर आपके पासवर्ड को खोजने के लिए SHA-256 हैश के विरुद्ध प्रति सेकंड अरबों पासवर्ड आज़मा सकता है। पासवर्ड सहेजने के लिए जानबूझकर धीमे की-डेरिवेशन फंक्शन जैसे कि Argon2, scrypt या bcrypt का उपयोग किया जाना चाहिए, जो एक *नमक* (salt - प्रति उपयोगकर्ता अद्वितीय यादृच्छिक डेटा, जो दो व्यक्तियों को एक ही पासवर्ड होने पर समान हैश होने से रोकता है) के साथ संयुक्त हो।
4. **लेखक की पहचान के रूप में हैश को पढ़ना।** यह पहचानकर्ता नहीं है। हैश सामग्री की पहचान करता है। यदि दो लोग SHA-256 के साथ *नमस्ते* शब्द को हैश करते हैं, तो दोनों को एक ही सारांश मिलता है — और यह मुख्य गुण है, कोई दोष नहीं: यदि वे अलग-अलग सारांश होते, तो हम प्रकाशित और प्राप्त के बीच मेल की जांच नहीं कर पाते।

आपके दैनिक जीवन में SHA-256 कहाँ दिखाई देता है

भले ही आप इसे न देखें, SHA-256 आपके द्वारा इंटरनेट पर प्रतिदिन उपयोग की जाने वाली अधिकांश चीजों का आधार है। बिटकॉइन की ब्लॉकचेन प्रत्येक ब्लॉक के SHA-256 को अगले से जोड़कर बनाई गई है; किसी पिछले ब्लॉक को बदलने से बाद की पूरी श्रृंखला की पुनर्गणना करनी पड़ती है। Git, वह सिस्टम जिसके साथ दुनिया के आधे हिस्से का कोड वर्जन किया जाता है, अपनी पूरी सामग्री के SHA-256 (हाल के संस्करणों में) या उसके पूर्ववर्ती SHA-1 (पुराने संस्करणों में) द्वारा प्रत्येक कमिट की पहचान करता है। HTTPS प्रमाणपत्र जो वेबसाइट की पहचान सत्यापित करते हैं, जब आप प्रवेश करते हैं, तो उनके पास एक संबद्ध SHA-256 छाप होती है। सॉफ़्टवेयर डाउनलोड के साथ अक्सर डेवलपर द्वारा प्रकाशित SHA-256 होता है ताकि आप यह सत्यापित कर सकें कि फ़ाइल रास्ते में नहीं बदली गई है। और, जैसा कि हमने कहा है, प्रत्येक Cuaderno Lacre के नीचे।

पेशेवर पाठक के लिए

सिस्टम का निर्णय लेने या ऑडिट करने वालों के लिए चार परिचालन अनुस्मारक:

1. हैश एन्क्रिप्शन नहीं है। यदि कोई प्रदाता अपने तकनीकी दस्तावेजों में इन दो शब्दों को मिलाता है, तो यह पूछना उचित है कि उसका वास्तव में क्या मतलब है।
2. पासवर्ड स्टोर करने के लिए कभी भी केवल SHA-256 का उपयोग नहीं किया जाना चाहिए। SHA-256 इस कार्य के लिए बहुत तेज़ है (देखें *हैश क्या नहीं है* का बिंदु 3)। वर्तमान मानक **Argon2id** है: डिज़ाइन द्वारा धीमा, सर्वर की क्षमता के अनुसार विन्यास योग्य, प्रति उपयोगकर्ता अलग यादृच्छिक *नमक* के साथ संयुक्त।
3. दस्तावेजों की अखंडता के लिए — अनुबंध, फाइलें, अभिलेख — SHA-256 संदर्भ मानक बना हुआ है। यह वह है जिसका उपयोग यूरोपीय संघ में योग्य टाइम स्टैम्प प्रदाताओं द्वारा किया जाता है।
4. दीर्घकालिक (दशकों) संरक्षण के लिए SHA-256 के साथ SHA-3 या SHA-512 की गणना और संग्रह करना भी उचित है; क्रिप्टोग्राफिक विवेक सदियों पुराने अभिलेखों के लिए केवल एक फंक्शन पर भरोसा न करने की सलाह देता है।

तकनीकी रूप से, यह इट्रेटेड संरचना (iterated structure) — जहाँ मध्यवर्ती अवस्था (intermediate state) को इनपुट ब्लॉक के बीच संरक्षित किया जाता है — एक **Merkle-Damgård** निर्माण के रूप में जानी जाती है, वह पैटर्न जिस पर SHA-1, SHA-2 (SHA-256 सहित) और कई अन्य क्लासिक हैश फंक्शन आधारित हैं। इसके विपरीत, SHA-3 Merkle-Damgård को छोड़कर *स्पंज (sponge)* नामक एक अलग आर्किटेक्चर को अपनाता है।

SHA-256 कैसे काम करता है, कदम-दर-कदम, सरल शब्दों में

कल्पना कीजिए कि आपने दुनिया का सबसे विस्तृत डोमिनो सर्किट बनाया है: हजारों गोटियां (fichas), दर्जनों शाखाएं, यांत्रिक पुल और रैंप जो पूरे कमरे में फैले हुए हैं, और हर एक टुकड़े को सावधानीपूर्वक लगाया गया है।

यदि आप पहली गोटी को हल्का सा धक्का देते हैं, तो पूरी श्रृंखला एक सटीक और दोहराव योग्य क्रम में गिरती है। वही सेटअप, वही शुरुआती धक्का → बार-बार गिराई गई गोटियों का बिल्कुल वही अंतिम पैटर्न।

यहाँ दिलचस्प बात यह है: शुरू करने से पहले **एक ही गोटी** को आधा सेंटीमीटर एक तरफ खिसका दें और फिर से धक्का दें। एक रैंप जिसे सक्रिय होना चाहिए था वह स्थिर रहता है, एक पुल नहीं गिरता, एक अलग शाखा सक्रिय हो जाती है। फर्श पर गोटियों का अंतिम पैटर्न पहले की तुलना में पूरी तरह से अपरिचित होगा।

SHA-256 गणितीय रूप से यही सर्किट है। आपके द्वारा लिखा गया टेक्स्ट गोटियों की शुरुआती स्थिति है। एल्गोरिथ्म वह धक्का है जो इस झरने (cascade) को मुक्त करता है। और अंतिम परिणाम — जिसे हम *हैश (hash)* कहते हैं — वह फर्श की एक स्थिर फोटो है जब सब कुछ रुक जाता है। मूल टेक्स्ट का एक भी कोमा बदलें और फोटो मौलिक रूप से अलग होगी। इतना सरल, और इतना प्रभावशाली।

चरण 1. टेक्स्ट को बाइनरी गोटियों (fichas) में अनुवाद करना। कंप्यूटर अक्षरों को नहीं समझते हैं; वे उन्हें पहले संख्याओं (ASCII) में और संख्याओं को बाइनरी (एक और शून्य) में अनुवाद करते हैं। प्रत्येक अक्षर 8 सफेद या काली गोटियों में बदल जाता है: *A* है `01000001`, *B* है `01000010`, स्पेस है `00100000`। आपका पूरा टेक्स्ट — एक शब्द, एक अनुबंध, एक उपन्यास — सफेद और काली गोटियों की एक लंबी कतार बन जाता है।

चरण 2. मानक आकार तक भरना (Padding)। सर्किट कतार को ठीक 512 गोटियों के *हिल्लों (tramos)* में संसाधित करता है। यदि आपका संदेश 512 के गुणक (multiple) तक नहीं पहुँचता है, तो टेक्स्ट के ठीक बाद एक मार्कर गोटी (वैल्यू `10000000` वाली) जोड़ी जाती है और फिर हिस्से को पूरा करने के लिए शून्य जोड़े जाते हैं। प्रत्येक हिस्से के अंतिम 64 स्थान टेक्स्ट की मूल लंबाई को नोट करने के लिए आरक्षित हैं। इस तरह सर्किट को हमेशा पता रहता है कि वास्तविक सामग्री कहाँ समाप्त हुई और पैडिंग (relleno) कहाँ से शुरू हुई।

चरण 3. आठ मुख्य गोटियों (fichas maestras) को रखना। शुरू करने से पहले, हम मेज पर एक सटीक शुरुआती स्थिति में **आठ मुख्य गोटियां** रखते हैं। ये आठ गोटियां कोई रहस्य नहीं हैं: उनका प्रारंभिक मान एक सार्वजनिक गणितीय नियम (पहले आठ अभाज्य संख्याओं — 2, 3, 5, 7, 11, 13, 17, 19 — के वर्गमूल और प्रत्येक वर्गमूल के दशमलव भाग के पहले बिट्स) द्वारा तय किया जाता है। दुनिया के किसी भी कोने में हर कोई उन्हीं आठ मुख्य गोटियों के साथ उसी स्थिति में शुरू करता है। उनका भाग्य झरने (avalancha) द्वारा धकेले जाना और परिवर्तित होना है।

चरण 4. विशाल झरना (avalancha): धक्कों के चौंसठ राउंड। यहाँ खेल शुरू होता है। आपके टेक्स्ट की 512 गोटियों के पहले हिस्से को आठ मुख्य गोटियों (fichas maestras) से टकराया जाता है। लेकिन वे एक साथ नहीं गिरती हैं: तंत्र **लगातार चौंसठ राउंड** निष्पादित करता है। प्रत्येक राउंड में वह गोटियों के साथ तीन ऑपरेशन करता है:

- **हिंडोला (Tiovivo)** (रोटेशन)। गोटियां एक घेरे में चलती हैं: दाईं ओर वाली बाईं ओर चली जाती हैं। कोई भी गोटी खोती नहीं है और न ही जुड़ती है; वे बस हिंडोले में एक पूरा चक्कर लगाकर फिर से व्यवस्थित हो जाती हैं। यह जानकारी को पुनर्वितरित करने का एक सस्ता और प्रतिवर्ती (reversible) तरीका है।
- **तार्किक कीप (Embudo Lógico)** (XOR)। गोटियां एक कीप से होकर गुजरती हैं जो उनकी दो-दो करके तुलना करती है: यदि दोनों एक ही रंग की हैं, तो एक सफेद गोटी निकलती है; यदि वे अलग-अलग हैं, तो एक काली गोटी निकलती है। यह बाइनरी लॉजिक का सबसे सरल ऑपरेशन है, लेकिन हिंडोले (tiovivo) के रोटेशन के साथ मिलकर यह जानकारी को बिना खोए मिलाने के लिए बहुत शक्तिशाली हो जाता है।
- **ओवरफ्लो (Desborde)** (मॉड्यूलर जोड़)। परिणाम को साठ सार्वजनिक स्थिरांकों (constants) की सूची से लाई गई एक *स्थिर पुश गोटी (ficha de empuje constante)* के साथ जोड़ा जाता है (पहले चौंसठ अभाज्य संख्याओं के घनमूल)। यदि जोड़ अतिरिक्त गोटियां उत्पन्न करता है जो 32 गोटियों के निर्धारित स्थान में नहीं आती हैं, तो उन बची हुई गोटियों को छोड़ दिया जाता है। मेज पर केवल 32 गोटियों के लिए जगह है, उससे एक भी ज्यादा नहीं।

चौंसठवें राउंड के अंत में, आपके टेक्स्ट के हिस्से की प्रत्येक गोटी ने आठ मुख्य गोटियों (fichas maestras) की स्थिति को प्रभावित किया है। धक्के की ऊर्जा पूरे सर्किट में फैल गई है।

चरण 5. अगला हिस्सा जोड़ना (बिना रीसेट किए)। यदि आपका टेक्स्ट लंबा था और संसाधित करने के लिए 512 गोटियों का एक और हिस्सा बचा है, तो **सर्किट रीसेट नहीं होता है**। आठ मुख्य गोटियाँ (fichas maestras) वैसी ही रहती हैं जैसी उन्हें पहले झरने (avalancha) ने छोड़ा था, और दूसरा हिस्सा उनके खिलाफ अन्य चौंसठ राउंड सक्रिय करने के लिए लॉन्च किया जाता है। यह अभी गिरी हुई कमरे के अंत में डोमिनोज़ से भरा एक नया कमरा जोड़ने जैसा है: पहले का विकार पूरी तरह से यह तय करता है कि दूसरा कैसे गिरेगा।

चरण 6. अंतिम फोटो लेना। जब संसाधित करने के लिए कोई और हिस्से नहीं बचते हैं, तो झरना (avalancha) रुक जाता है। हम उस अंतिम स्थिति को देखते हैं जिसमें आठ मुख्य गोटियाँ (fichas maestras) रह गई हैं। हम उनके विन्यास (configuration) को हेक्साडेसिमल सिस्टम में अक्षरों और संख्याओं के कोड में अनुवाद करते हैं। परिणाम ठीक चौंसठ वर्णों (characters) की एक स्ट्रिंग है: यही आपका SHA-256 मुहर (sello) है।

सर्किट कैसे बनाया गया है, इससे चार गुण अपने आप स्पष्ट हो जाते हैं:

1. **नियतत्ववाद (Determinismo)।** एक ही टेक्स्ट हमेशा दुनिया के किसी भी कंप्यूटर पर एक ही अंतिम फोटो तैयार करता है। शून्य यादृच्छिकता (randomness), शून्य आश्चर्य।
2. **एवलांच प्रभाव (Efecto avalancha)।** एक जोड़ा गया कोमा, एक बदला हुआ बड़ा अक्षर, एक भूला हुआ चिह्न: फोटो पूरी तरह से अपरिचित हो जाती है। यह वह चरम संवेदनशीलता है जिसका वर्णन हमने शुरुआत में ही किया है।
3. **एक तरफा दिशा (Una sola dirección)।** अंतिम फोटो को देखते हुए, आप मूल टेक्स्ट का पुनर्निर्माण नहीं कर सकते। रोटेशन, कीप (embudos) और ओवरफ्लो (desbordes) इस बारे में सभी दिशात्मक जानकारी को नष्ट कर देते हैं कि *प्रत्येक बिट कहाँ से आया है* और केवल यह सुरक्षित रखते हैं कि *कुल मिलाकर क्या जोड़ा गया था*।
4. **टकराव प्रतिरोध (Resistencia a colisiones)।** सार्वजनिक क्रिप्टोएनालिसिस के पच्चीस वर्षों में, कोई भी दो अलग-अलग टेक्स्ट खोजने में सफल नहीं हुआ है जिनकी अंतिम फोटो मेल खाती हो। और ऐसा करने की कठिनाई किसी भी उचित रूप से कल्पनीय सभ्यता की कम्प्यूटेशनल पहुंच से बाहर है।

निम्नलिखित कोड अपेंडिक्स (appendice) Zig में इन छह चरणों को ठीक वैसे ही लागू करता है। अब आप प्रत्येक बिट ऑपरेशन का अर्थ जानते हुए इसे पढ़ सकते हैं, इसके बजाय कि हेरफेर को आँख मूंदकर स्वीकार करें।

तकनीकी शब्दावली

उस पाठक के लिए जो यह समझना चाहता है कि प्रत्येक ऑपरेशन क्या करता है। इसे बेझिझक छोड़ दें: लेख इसके बिना भी समझ में आता है।

ASCII और Unicode — अक्षर संख्या कैसे बनते हैं। कंप्यूटर अक्षर नहीं देखते; वे संख्याएं देखते हैं। **ASCII** (*American Standard Code for Information Interchange*, 1963) नामक एक मानक प्रत्येक कीबोर्ड वर्ण को एक विशिष्ट संख्या प्रदान करता है: *A* 65 है, *B* 66 है, *a* 97 है, *0* 48 है, स्पेस 32 है, कोमा 44 है। आधुनिक सिस्टम इसे **Unicode** के साथ विस्तारित करते हैं, जो दुनिया के प्रत्येक वर्णमाला के प्रत्येक वर्ण को एक संख्या प्रदान करता है: सिरिलिक, अरबी, चीनी, जापानी और यहाँ तक कि इमोजी भी। जब आप कोई वर्ण लिखते हैं या कोई टेक्स्ट फ़ाइल खोलते हैं, तो कंप्यूटर बैकग्राउंड नंबर पढ़ता है, स्क्रीन पर दिखने वाला रूप नहीं। SHA-256 इन नंबरों पर काम करता है, किसी भी टेक्स्ट को अंकों की एक लंबी श्रृंखला के रूप में मानता है। इसीलिए यह उसी एल्गोरिथ्म के साथ स्पेनिश में एक लेख, जापानी में एक कविता और एक बाइनरी फ़ाइल को सील कर सकता है।
XOR — बिट-दर-बिट तुलनाकर्ता (comparador bit a bit)। XOR (उच्चारण *«exor»*, अंग्रेजी *exclusive or* से) सबसे सरल ऑपरेशनों में से एक है जो कंप्यूटर दो बाइनरी नंबरों के साथ कर सकता है। यह दो बिट्स की स्थिति-दर-स्थिति तुलना करता है और देता है: **1** यदि दोनों में से ठीक एक 1 है (एक लेकिन दोनों नहीं), **0** यदि दोनों समान हैं (दोनों 0 या दोनों 1)। उदाहरण: `1010` और `1100` का XOR `0110` है। इसमें एक उल्लेखनीय गुण है: यह प्रतिवर्ती (reversible) है — यदि आप एक ही कुंजी के साथ दो बार XOR करते हैं, तो आप मूल पर वापस आ जाते हैं। इसीलिए यह क्रिप्टोग्राफी का मुख्य आधार (caballo de batalla) है: यह जानकारी खोए बिना बिट्स को मिलाता है, लेकिन यदि आप उनमें से एक को नहीं जानते हैं तो परिणाम इनपुट के बारे में कुछ भी प्रकट नहीं करता है।
हेक्साडेसिमल — बेस 16 में गिनती। रोजमर्रा की जिंदगी में लगभग सभी संख्याएं दस अंकों (0-9) का उपयोग करती हैं। हेक्साडेसिमल सोलह का उपयोग करता है: सामान्य 0-9 और छह अक्षर जो निम्नलिखित मानों का प्रतिनिधित्व करते हैं: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15। सोलह क्यों? क्योंकि कंप्यूटर चार बिट्स के समूहों में सोचते हैं, और चार बिट्स ठीक सोलह अलग-अलग मानों का प्रतिनिधित्व कर सकते हैं — इस प्रकार, एक हेक्साडेसिमल वर्ण स्पष्ट रूप से चार बिट्स से मेल खाता है। एक SHA-256 हैश (huella) की लंबाई 256 बिट्स होती है,

जो ठीक **64 हेक्साडेसिमल वर्ण** होते हैं। यदि हम इसे सामान्य दशमलव (decimal) में लिखते हैं, तो यह लगभग 78 अंक ले लेगा और अधिक असुविधाजनक होगा। चुनाव सौंदर्यपूर्ण और संक्षिप्त है; बैकग्राउंड नंबर वही है।

बिट्स का रोटेशन — बाइनरी हिंडोला (tiovivo binario)। कल्पना कीजिए कि सात बल्बों की एक कतार है, कुछ चालू (1) और कुछ बंद (0) हैं: `1 0 1 1 0 0 1`। एक स्थान दाईं ओर घुमाने (rotate) में सबसे दाईं ओर का बल्ब लेना, उसे सबसे बाईं ओर ले जाना और अन्य को एक स्थान दाईं ओर खिसकाना शामिल है: `1 1 0 1 1 0 0`। कोई भी बल्ब खोता नहीं है और न ही जुड़ती है: वे बस एक घेरे में नाचते हैं। SHA-256 प्रत्येक गणना में सैकड़ों बार बिट रोटेशन का उपयोग करता है; यह स्थिति (state) के भीतर जानकारी को पुनर्वितरित करने का एक सस्ता और दोषरहित (lossless) तरीका है।

स्थिरांक *«nothing-up-my-sleeve»* — वे अभाज्य संख्याओं से क्यों आते हैं। SHA-256 की आठ मुख्य गोटियां (fichas maestras) और चौंसठ राउंड स्थिरांक (constants) यादृच्छिक रूप से नहीं चुने गए थे। वे पहले अभाज्य संख्याओं के वर्गमूल और घनमूल से आते हैं। क्यों? क्योंकि उनके डिज़ाइनर *«बिना कुछ छुपाए» (nothing-up-my-sleeve)* स्थिरांक चाहते थे: ऐसे मान जिनका मूल कोई भी सत्यापित कर सके। यदि कोई आपसे कहता *«मुझ पर भरोसा करें: इस 32-बिट रैंडम नंबर का उपयोग करें»*, तो आप उचित रूप से छिपी हुई कमजोरी या पिछले दरवाजे (backdoor) का संदेह करेंगे। लेकिन कैलकुलेटर वाला कोई भी व्यक्ति यह जांच सकता है कि 2 के वर्गमूल के पहले 32 बिट `0x6a09e667` हैं। मान गणितीय, सार्वजनिक और प्रतिलिपि प्रस्तुत करने योग्य (reproducible) हैं: नुस्खा (recipe) में कोई छिपा हुआ जाल नहीं फंस सकता।

परिशिष्ट: पठनीय कोड में SHA-256

यह परिशिष्ट उस पाठक के लिए है जो एल्गोरिदम को अंदर से देखना चाहता है। यह Zig में एक शैक्षिक कार्यान्वयन है जो FIPS 180-4 विनिर्देश का पालन करता है। यह वह संस्करण नहीं है जिसे Solo2 उपयोग करता है — वास्तविक संस्करण Zig की मानक लाइब्रेरी के `std.crypto.hash.sha2.Sha256` में है, जो अनुकूलित और ऑडिटेड है। लेकिन एल्गोरिदम वही है: जो आप यहाँ देखते हैं, वह चरण-दर-चरण है, जो तब होता है जब पाँच वर्णों की वह कॉल अपना काम करती है।

```
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
}
```

किसी अन्य भाषा में कोई भी पुनर्लेखन जो उसी संरचना का पालन करता है — प्रारंभिक स्थिरांक, शेड्यूल विस्तार, चौंसठ राउंड, संचय — वही परिणाम उत्पन्न करता है। एल्गोरिदम में कोई रहस्य नहीं है: इसका मूल्य इस तथ्य में निहित है कि ऊपर सूचीबद्ध गुण हजारों आँखों के सामने सार्वजनिक क्रिप्टएनालिसिस के दो दशकों के बाद भी बने हुए हैं।

यदि आप इस लेख के नीचे वापस जाते हैं, तो आपको चौंसठ वर्णों की एक हेक्साडेसिमल मुहर दिखाई देगी। यह उस पाठ का SHA-256 है जिसे आपने अभी-अभी इस भाषा में पढ़ा है। यदि हम लेख का अनुवाद करते हैं, तो मुहर अलग होगी; यदि हिंदी संस्करण का एक भी शब्द बदलता है, तो हिंदी मुहर बदल जाएगी। मुहर सामग्री की रक्षा नहीं करती है — उसके लिए अन्य उपकरण हैं — बल्कि यह विशिष्ट रूप से उसकी पहचान करती है। और यह, भले ही मामूली लगे, यह सुनिश्चित करने के लिए पर्याप्त है कि संपादकीय श्रृंखला का कोई भी कदम जो कहा गया है उसे बिना ध्यान दिए नहीं बदल सकता। बाकी सब कुछ — एन्क्रिप्ट करना, हस्ताक्षर करना, पहचान करना — इसी सरल विचार पर बनाया गया है।

संपादकीय नोट: जब ये Cuadernos कंपनियों या उत्पादों का नाम लेते हैं, तो यह आरोप लगाने के लिए नहीं है। जो लोग इन्हें बनाते हैं, वे ऐसा काम करते हैं जिसका लाखों लोग उपयोग करते हैं और सराहना करते हैं। हम जो इशारा कर रहे हैं वह संरचनात्मक है — मॉडल, न कि ब्रांड। ब्रांड उदाहरण के रूप में दिखाई देते हैं क्योंकि वे ही हैं जिन्हें पाठक पहचानता है।

स्रोत और आगे पढ़ने के लिए

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, अगस्त 2015। SHA-256 सहित SHA-2 परिवार का आधिकारिक विनिर्देश।
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, मई 2011। कार्यान्वयनकर्ताओं के लिए मानक संस्करण।
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010)। अध्याय 5 और 6 हैश फंक्शन और उनके वैध और अवैध उपयोगों को कवर करते हैं।
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008)। निर्माण द्वारा अपरिवर्तनीय संरचना में ब्लॉक को जोड़ने के लिए SHA-256 के उपयोग का व्यावहारिक उदाहरण।
- विनियमन (EU) 910/2014 (eIDAS) — योग्य टाइम स्टैम्प प्रदाताओं का ढांचा। SHA-256 यूरोपीय संघ में जारी योग्य इलेक्ट्रॉनिक हस्ताक्षरों और मुहरों के लिए संदर्भ फंक्शन है।
- Zig में संदर्भ कार्यान्वयन: भाषा के आधिकारिक रिपॉजिटरी में `std.crypto.hash.sha2.Sha256` (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`)। यह वह अनुकूलित और ऑडिटेड संस्करण है जिसे वास्तव में Solo2 उपयोग करता है। परिशिष्ट के शैक्षिक कार्यान्वयन के साथ तुलना करने के लिए उपयोगी।

← पिछलाSchrems II, पांच साल बादअगला →Kill switch और संस्थागत कब्जा

हाल की रीडिंग

- विश्लेषण · 18 मई, 2026 वास्तविक बनाम आभासी गोपनीयता: वे प्रश्न जिन्हें आपको खुद से पूछना चाहिए
- विश्लेषण · 18 मई, 2026 पेशेवर अभ्यास के रूप में सेल्फ-होस्टिंग
- अवधारणा · 18 मई, 2026 वे 24 शब्द: क्रिप्टोग्राफिक पहचान क्या है

इस लेख को अपने साथ ले जाएं जहाँ भी आपको इसकी आवश्यकता हो।

↓ मार्कडाउन ↓ सादा टेक्स्ट ↓ PDF

फ़ाइल आपके डिवाइस पर डाउनलोड हो जाएगी। वहां से आप इसे सहेज सकते हैं, Solo2 में आयात कर सकते हैं या जहां चाहें साझा कर सकते हैं। Cuadernos आपके लिए गंतव्य तय नहीं करता है।

मोहरबंद · SHA-256 2d42e7752fda8ff9762748df4214b4c129f2b36488ce812f9242b4daf632562a

Cuadernos Lacre · Menzuri Gestión S.L. का एक प्रकाशन ·
R.Eugenio द्वारा लिखित · Solo2 की टीम द्वारा संपादित।