SHA-256 คืออะไรกันแน่

รอยนิ้วมือทางคณิตศาสตร์ที่ยาวเพียง 64 ตัวอักษร แต่จะเปลี่ยนไปโดยสิ้นเชิงหากมีการขยับเครื่องหมายจุลภาคเพียงตัวเดียวในข้อความต้นฉบับ ทำไมเราถึงเรียกมันว่าตราประทับครั่งดิจิทัล

พูดง่ายๆ ก็คือ: ลองจินตนาการถึงเครื่องจักรที่อ่านข้อความใดๆ แล้วส่งรหัส 64 ตัวอักษรกลับมา ถ้าข้อความที่ใส่เข้าไปเหมือนเดิมเป๊ะ รหัสที่ออกมาก็จะเหมือนเดิมเป๊ะ แต่ถ้าคุณเลื่อนเพียงแค่ลูกน้ำตัวเดียว รหัสที่ได้จะเปลี่ยนไปอย่างสิ้นเชิง รหัสนั้นแหละคือตราประทับครั่งแบบดิจิทัล

แนวคิดเรียบง่ายเบื้องหลังชื่อทางเทคนิค

ลองจินตนาการว่ามีเครื่องจักรที่มีช่องรับข้อมูลเพียงช่องเดียวและหน้าจอเพียงหน้าจอเดียว คุณใส่ข้อความผ่านช่องนั้นลงไป ไม่ว่าจะเป็นคำเพียงคำเดียว ประโยคเดียว หรือนิยายทั้งเล่ม ในชั่วพริบตา หน้าจอจะแสดงลำดับตัวอักษรที่มีความยาวพอดี 64 ตัวอักษร ลำดับตัวอักษรนี้เหล่านักอ่านมืออาชีพเรียกว่า *Hash* หรือ *ค่าแฮช* หรือ *ผลสรุปเชิงรหัสลับ (Cryptographic Summary)* ส่วนสำหรับผู้อ่านทั่วไป ในตอนนี้เราอาจเรียกมันว่าเป็นรอยนิ้วมือทางคณิตศาสตร์ของข้อความ เช่นเดียวกับที่รอยนิ้วมือเป็นตัวแทนของบุคคลหนึ่งๆ

หากคุณใส่ข้อความเดิมสองครั้ง เครื่องจักรจะแสดงรอยนิ้วมือเดิมทั้งสองครั้ง หากคุณใส่ข้อความที่ต่างไปเพียงเล็กน้อย เช่น เลื่อนตำแหน่งเครื่องหมายจุลภาคเพียงตัวเดียว หรือเปลี่ยนจากตัวอักษรพิมพ์ใหญ่เป็นพิมพ์เล็ก เครื่องจักรจะแสดงรอยนิ้วมือที่ต่างไปจากเดิมอย่างสิ้นเชิง ไม่ใช่แค่คล้ายกัน แต่คือต่างกันไปเลย คุณสมบัติสองประการนี้ที่รวมกัน คือ ความแน่นอน (Determinism) และความฉับไวต่อการเปลี่ยนแปลง (Sensitivity) เป็นแนวคิดที่เรียกว่าเรียบง่าย ส่วนที่เหลือของ SHA-256 คือกลไกที่ทำให้มั่นใจว่าคุณสมบัติเหล่านี้ทำงานได้อย่างสมบูรณ์

ควรบอกไว้ตั้งแต่แรกว่าเครื่องจักรนี้ไม่ได้ทำอะไรบ้าง มันไม่ได้เข้ารหัส (Encrypt) ข้อความ มันไม่ได้ซ่อนมัน และมันไม่ได้บันทึกมัน เครื่องจักรเพียงแตมองดูข้อความ คำนวณรอยนิ้วมือ แล้วก็ลืมข้อความนั้นไป รอยนิ้วมือไม่สามารถใช้กู้คืนข้อความที่สร้างมันขึ้นมาได้ มันทำได้เพียงใช้ตรวจสอบว่าข้อความที่นำมาทดสอบนั้นตรงกับข้อความต้นฉบับหรือไม่ นั่นคือเหตุผลที่เราเรียกว่ามันเป็นการสรุปแบบ *ทิศทางเดียว*: ไปแล้วไม่กลับ

แฮช (Hash) ไม่ใช่สิ่งเดียวกับการเข้ารหัส (Encrypt)

ความสับสนมักเกิดขึ้นบ่อยครั้งและควรทำความเข้าใจให้ชัดเจน: การเข้ารหัสและการแฮชคือการทำงานที่ต่างกัน การเข้ารหัสคือการแปลงข้อความเพื่อให้ผู้ที่มีกุญแจเท่านั้นที่สามารถคืนค่าให้เป็นรูปแบบเดิมได้ ส่วนการแฮชคือการสร้างรอยนิ้วมือของข้อความซึ่งไม่สามารถกู้คืนข้อความต้นฉบับได้เลย ไม่ว่าจะใช้กุญแจหรือไม่ก็ตาม แบบแรกสามารถย้อนกลับได้โดยการออกแบบ (Reversible) ส่วนแบบหลังไม่สามารถย้อนกลับได้โดยการออกแบบ (Irreversible)

ผลลัพธ์ในทางปฏิบัติมีความสำคัญ เมื่อแอปพลิเคชันบอกว่า "เราเก็บรหัสผ่านของคุณแบบเข้ารหัส" แสดงว่ามีใครบางคนที่มีกุญแจในการถอดรหัส ซึ่งก็คือตัวแอปพลิเคชันเองในทุกกรณี แต่เมื่อแอปพลิเคชันบอกว่า "เราเก็บรหัสผ่านของคุณแบบแฮช" ตัว

แอปพลิเคชันเองก็ไม่สามารถอ่านรหัสผ่านต้นฉบับได้แม้ว่าจะต้องการก็ตาม มันทำได้เพียงตรวจสอบว่ารหัสผ่านที่คุณพิมพ์เข้ามานั้นสร้างรอยนิ้วมือที่ตรงกันหรือไม่ รูปแบบหลังนี้ หากทำอย่างถูกต้อง (Done well) จะเหมาะสมกว่ามากสำหรับการเก็บรหัสผ่าน ซึ่งเราจะได้เห็นกันต่อไปว่าทำไมคำว่า "ทำอย่างถูกต้อง" ถึงต้องการอะไรที่มากกว่าแค่ SHA-256 เปล่าๆ

สี่คุณสมบัติที่ทำให้ค่าแฮชเชิงรหัสลับมีประโยชน์

ฟังก์ชันแฮชที่สมควรได้รับคำจำกัดความว่า *เชิงรหัสลับ (Cryptographic)* ต้องมีสี่คุณสมบัติดังนี้:

1. **ความแน่นอน (Determinism).** ข้อมูลนำเข้าเดิมจะให้รอยนิ้วมือเดิมเสมอ
2. **ปรากฏการณ์หิมะถล่ม (Avalanche Effect).** การเปลี่ยนแปลงเพียงเล็กน้อยในข้อมูลนำเข้าจะส่งผลให้รอยนิ้วมือเปลี่ยนไปโดยสิ้นเชิง โดยไม่มีความคล้ายคลึงกับของเดิมให้เห็น
3. **ความต้านทานต่อการย้อนกลับ (Resistance to inversion).** เมื่อได้รับรอยนิ้วมือมา การหาข้อความต้นฉบับที่สร้างมันขึ้นมานั้นเป็นไปไม่ได้ในทางคอมพิวเตอร์
4. **ความต้านทานต่อการซ้ำซ้อน (Resistance to collisions).** การหาข้อความสองข้อความที่ต่างกันแต่ให้รอยนิ้วมือเดียวกันนั้นเป็นไปไม่ได้ในทางคอมพิวเตอร์

"เป็นไปไม่ได้ในทางคอมพิวเตอร์" ไม่ได้หมายความว่า "เป็นไปไม่ได้ในทางคณิตศาสตร์" แต่มันหมายถึงต้นทุนด้านเวลา พลังงาน และเงินทองในการทำให้สำเร็จนั้นสูงกว่าความสามารถในการคำนวณทั้งหมดที่มีอยู่หลายเท่าตัว สำหรับ SHA-256 ขีดจำกัดนั้นวัดได้เป็นหลายล้านล้านปี แม้จะเป็นการคาดการณ์ในแง่ดีที่สุดโดยใช้ฮาร์ดแวร์เฉพาะทางก็ตาม ซึ่งในทางปฏิบัติสำหรับผู้อ่านทั่วไปแล้วมันก็มีความหมายเหมือนกับคำว่า "ทำไม่ได้" นั่นเอง

เจาะลึก SHA-256

ชื่อของมันบอกทุกอย่าง SHA ย่อมาจาก *Secure Hash Algorithm* หรืออัลกอริทึมแฮชที่ปลอดภัย ตัวเลข 256 ระบุขนาดของรอยนิ้วมือในหน่วยบิต: 256 บิต หรือ 32 ไบต์ ซึ่งเมื่อแสดงในรูปแบบเลขฐานสิบหกจะเท่ากับ 64 ตัวอักษรที่ผู้อ่านคุ้นเคย มาตรฐานนี้ถูกเผยแพร่โดย NIST ของสหรัฐอเมริกา ซึ่งเป็นองค์กรที่กำหนดมาตรฐานสำหรับฟังก์ชันประเภทนี้ ในปี 2001 โดยเป็นส่วนหนึ่งของตระกูล SHA-2 และเวอร์ชันปัจจุบันของมาตรฐานคือ FIPS 180-4 จากปี 2015

สำหรับใครที่ยังไม่คุ้นเคยว่าบิต (Bits) และไบต์ (Bytes) คืออะไร:

```
1 บิต    →  0 หรือ 1       (สวิตช์ไฟ: เปิด หรือ ปิด)
1 ไบต์   →  8 บิต          (ความเป็นไปได้ 256 รูปแบบ)
32 ไบต์  →  256 บิต        (รอยนิ้วมือ SHA-256)
```

ตัวเลข 256 ที่ท้ายชื่อบอกขนาดของรอยนิ้วมือในหน่วยบิต ในระบบเลขฐานสิบหก (Hexadecimal) ซึ่งใช้สัญลักษณ์ 16 ตัวแทน 10 ตัว ข้อมูล 256 บิตเหล่านั้นจะพอดีกับตัวอักษร 64 ตัว ซึ่งเป็นตัวอักษร 64 ตัวที่คุณเห็นที่ส่วนท้ายของทุก Cuaderno

มิติของมันควรค่าแก่การพิจารณาครู่หนึ่ง 256 บิต ช่วยให้เกิดค่าที่ต่างกันได้ถึงสองยกกำลังสองร้อยห้าสิบหกค่า ซึ่งเป็นตัวเลขที่มีหลักทศนิยมถึง 78 หลัก มากกว่าจำนวนอะตอมโดยประมาณในจักรวาลที่สังเกตได้หลายเท่าตัว ทุกข้อความในโลก ไม่ว่าจะเป็นหนังสือแต่ละเล่ม อีเมลแต่ละฉบับ หรือข้อความแต่ละข้อความ จะตกลงไปในค่าหนึ่งค่าใดจากบรรดาค่าเหล่านั้น ความน่าจะเป็นที่ข้อความที่ต่างกันสองข้อความจะให้ค่าตรงกันโดยบังเอิญนั้น ในทางปฏิบัติถือว่าเป็นศูนย์

รหัสตัวอย่าง

ในภาษา Zig ซึ่งเป็นภาษาที่เราใช้เขียนส่วนประกอบต่างๆ ของ Solo2 การคำนวณตราประทับ SHA-256 ของข้อความจะมีหน้าตาดังนี้:

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

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

เราเพิ่งสั่งให้ไลบรารีมาตรฐานของ Zig คำนวณ SHA-256 ของข้อความในเครื่องหมายอัญประกาศ หลังจากเรียกคำสั่งนี้ ตัวแปร *resumen* จะเก็บข้อมูล 32 ไบต์ซึ่งประกอบกันเป็นตราประทับในรูปแบบดิบ และเมื่อแสดงบนหน้าจอในรูปแบบเลขฐานสิบหก จะเป็นตัวอักษร 64 ตัวที่ปรากฏที่ส่วนท้ายของบทความนี้ หากเราเปลี่ยน *Cuadernos Lacre* เป็น *Cuadernos Lacre* (โดยมีการเปลี่ยนแปลงในข้อความ) ตราประทับจะเปลี่ยนไปโดยสิ้นเชิง นี่คือคุณสมบัติหลักที่มีความยาวเพียง 5 บรรทัด สำหรับใครที่ต้องการดูการทำงานภายใน เราได้รวมอัลกอริทึมเวอร์ชันที่อ่านง่ายพร้อมคำอธิบายทีละขั้นตอนไว้ที่ตอนท้ายของบทความ

ทำไมเราถึงเรียกว่าตราประทับครั่ง

ในการส่งจดหมายของยุโรปช่วงคริสต์ศตวรรษที่ 15 ถึง 19 ครั่งจะถูกใช้เพื่อปิดผนึกจดหมาย ครั่งที่หลอมละลายเพียงหยดเดียว ตราประทับที่กดทับลงไป และจดหมายจะถูกตีตราไว้ในแบบที่ทำซ้ำไม่ได้ มันไม่ได้ช่วยปกป้องเนื้อหาจากการแอบดูที่ตั้งใจ (กระดาษยังสามารถอ่านผ่านแสงได้ ครั่งสามารถแตกหักได้) แต่มันเป็นหลักฐาน การเปลี่ยนแปลงใดๆ ของรอยผนึกจะมองเห็นได้ชัดเจนโดยผู้รับก่อนที่จะเปิดจดหมายเสียอีก ครั่งไม่ได้ป้องกันความเสียหาย แต่มันฟ้องว่าเกิดความเสียหายขึ้น

SHA-256 ของเนื้อหาในทุก Cuaderno ทำหน้าที่แบบเดียวกันในเวอร์ชันดิจิทัล หากมีเพียงคำเดียวในบทความเปลี่ยนไป ระหว่างตอนที่เผยแพร่กับตอนที่คุณอ่าน ตราประทับเลขฐานสิบหกที่ส่วนท้ายของข้อความจะไม่ตรงกับ SHA-256 ของข้อความที่คุณมีอยู่ตรงหน้าอีกต่อไป ผู้อ่านคนใดที่มีโค้ดเพียง 5 บรรทัดก็สามารถตรวจสอบเรื่องนี้ได้ สำนักพิมพ์ไม่สามารถเขียนประวัติศาสตร์ซ้ำได้โดยที่ตราประทับไม่ฟ้องมัน มันไม่ได้ป้องกันความเสียหาย แต่มันทำให้ความเสียหายนั้นตรวจสอบได้

สิ่งที่ค่าแฮชไม่ใช่

การใช้งานสี่ประการที่มักจะถูกขอให้ SHA-256 ทำซึ่งไม่ใช่หน้าที่ของมัน:

1. **การเข้ารหัส (Encrypt).** แฮชคือการสรุป ไม่ใช่การซ่อน หากคุณต้องการไม่ให้ข้อความถูกอ่านได้ คุณต้องเข้ารหัสมัน ไม่ใช่แฮชมัน
2. **การยืนยันตัวตนผู้เขียน.** แฮชไม่ได้บอกว่าใครเป็นคนเขียนข้อความ มันบอกเพียงว่าข้อความใดถูกแฮช การเชื่อมโยงความเป็นเจ้าของผลงานต้องใช้ลายเซ็นดิจิทัล (Digital Signature) กำกับบนค่าแฮช ไม่ใช่ใช้ค่าแฮชเพียงอย่างเดียว
3. **การเก็บรหัสผ่าน.** มีกับดักที่ควรทำความเข้าใจที่นี่ SHA-256 ถูกออกแบบมาให้ทำงานเร็วมาก ซึ่งเป็นเรื่องดีสำหรับหลายสิ่ง แต่เป็นเรื่องแย่สำหรับสิ่งนี้ ผู้โจมตีที่มีฮาร์ดแวร์เฉพาะทางสามารถทดลองรหัสผ่านได้หลายพันล้านรหัสต่อวินาทีเทียบกับค่าแฮช SHA-256 จนกว่าจะพบรหัสของคุณ ในการเก็บรหัสผ่าน ต้องใช้ฟังก์ชันการแปลงกุญแจที่ถูกออกแบบมาให้ช้าอย่างจงใจ เช่น Argon2, scrypt หรือ bcrypt ร่วมกับ *Salt* (ข้อมูลสุ่มที่ไม่ซ้ำกันสำหรับผู้ใช้แต่ละราย เพื่อป้องกันไม่ให้คนสองคนที่มีรหัสผ่านเดียวกันมีค่าแฮชที่เหมือนกัน)
4. **การอ่านค่าแฮชเป็นตัวระบุตัวตนผู้เขียน.** มันไม่ใช่ แฮชใช้ระบุเนื้อหา หากคนสองคนแฮชคำว่า *สวัสดี* ด้วย SHA-256 ทั้งคู่จะได้ผลสรุปที่เหมือนกัน และนั่นคือคุณสมบัติหลัก ไม่ใช่ข้อบกพร่อง: หากมันให้ผลสรุปที่ต่างกัน เราก็จะไม่สามารถตรวจสอบความตรงกันระหว่างสิ่งที่เผยแพร่กับสิ่งที่ได้รับได้

SHA-256 ปรากฏอยู่ที่ใดในชีวิตประจำวันของคุณ

แม้คุณจะไม่เห็นมัน แต่ SHA-256 รองรับการทำงานเกือบทุกอย่างที่คุณใช้บนอินเทอร์เน็ตในแต่ละวัน บล็อกเชนของ Bitcoin ถูกสร้างขึ้นโดยการเชื่อมโยง SHA-256 ของแต่ละบล็อกเข้ากับบล็อกถัดไป การแก้ไขบล็อกในอดีตทำให้ต้องคำนวณห่วงโซ่ที่ตามมาทั้งหมดใหม่ Git ระบบที่ใช้จัดการเวอร์ชันของโค้ดที่คนค่อนโลกใช้ ระบุการยืนยันการเปลี่ยนแปลง (Commit) แต่ละครั้งด้วย SHA-

256 (ในเวอร์ชันใหม่ๆ) หรือรุ่นก่อนหน้าอย่าง SHA-1 (ในเวอร์ชันเก่า) ของเนื้อหาทั้งหมด ใบรับรอง HTTPS ที่ยืนยันตัวตนของเว็บไซต์เมื่อคุณเข้าชมก็มีรอยนิ้วมือ SHA-256 กำกับอยู่ การดาวน์โหลดซอฟต์แวร์มักจะมาพร้อมกับค่า SHA-256 ที่ผู้พัฒนาเผยแพร่เพื่อให้คุณตรวจสอบว่าไฟล์ไม่ถูกแก้ไขระหว่างทาง และตามที่เราได้บอกไป มันอยู่ที่ส่วนท้ายของทุก Cuadernos Lacre

สำหรับผู้อ่านมืออาชีพ

สี่ข้อเตือนใจในการปฏิบัติงานสำหรับผู้ที่ตัดสินใจหรือตรวจสอบระบบ:

1. แฮชไม่ใช่การเข้ารหัส หากผู้ให้บริการสับสนระหว่างสองคำนี้ในเอกสารทางเทคนิค ควรตั้งคำถามว่าพวกเขาหมายถึงอะไรกันแน่
2. สำหรับการเก็บรหัสผ่าน ไม่ควรใช้ SHA-256 เพียงอย่างเดียว SHA-256 เร็วเกินไปสำหรับงานนี้ (ดูข้อ 3 ใน *สิ่งที่ค่าแฮชไม่ใช่*) มาตรฐานปัจจุบันคือ **Argon2id**: ซึ่งช้าโดยการออกแบบ สามารถปรับค่าตามความสามารถของเซิร์ฟเวอร์ และทำงานร่วมกับ *Salt* ที่สุ่มต่างกันในแต่ละผู้ใช้
3. สำหรับความสมบูรณ์ของเอกสาร ไม่ว่าจะเป็นสัญญา สำนวนคดี หรือไฟล์ข้อมูล SHA-256 ยังคงเป็นมาตรฐานอ้างอิง และเป็นมาตรฐานที่ผู้ให้บริการประทับรับรองเวลา (Time-stamping) ในสหภาพยุโรปใช้
4. สำหรับการเก็บรักษาในระยะยาว (หลายทศวรรษ) ควรคำนวณและเก็บค่า SHA-3 หรือ SHA-512 ควบคู่ไปกับ SHA-256 ด้วย ความรอบคอบเชิงรหัสลับแนะนำว่าไม่ควรยึดติดกับฟังก์ชันเพียงฟังก์ชันเดียวสำหรับการเก็บข้อมูลในระดับศตวรรษ

ในทางเทคนิค โครงสร้างแบบวนซ้ำนี้ — ซึ่งสถานะกลางจะถูกเก็บไว้ระหว่างบล็อกข้อมูลขาเข้า — เรียกว่าโครงสร้างแบบ **Merkle-Damgård** ซึ่งเป็นรูปแบบที่เป็นพื้นฐานของ SHA-1, SHA-2 (รวมถึง SHA-256) และฟังก์ชันแฮชคลาสสิกอื่นๆ อีกมากมาย ในทางตรงกันข้าม SHA-3 ได้ละทิ้ง Merkle-Damgård และหันไปใช้สถาปัตยกรรมที่แตกต่างออกไปซึ่งเรียกว่า *sponge*

การทำงานของ SHA-256 ทีละขั้นตอน ในภาษาที่เข้าใจง่าย

ลองจินตนาการว่าคุณได้สร้างวงจรโดมิโนที่ซับซ้อนที่สุดในโลก: มีตัวโดมิโนนับพันตัว ทางแยกนับสิบ สะพานกลไก และทางลาดที่ทอดยาวไปทั่วทั้งห้อง ซึ่งทั้งหมดถูกวางไว้อย่างพิถีพิถันทีละชิ้น

หากคุณผลักโดมิโนตัวแรก โซ่จะล้มลงตามลำดับที่แม่นยำและทำซ้ำได้ การติดตั้งเหมือนเดิม การผลักเริ่มต้นเหมือนเดิม → รูปแบบสุดท้ายของโดมิโนที่ล้มลงจะเหมือนเดิมทุกครั้ง

นี่คือสิ่งที่น่าสนใจ: หากคุณขยับ **โดมิโนเพียงตัวเดียว** ออกไปด้านข้างเพียงครึ่งเซนติเมตรก่อนที่จะเริ่มแล้วผลักใหม่ ทางลาดที่ควรจะทำงานก็จะหยุดนิ่ง สะพานไม่ตกลงมา ทางแยกที่แตกต่างออกไปจะถูกเปิดใช้งาน รูปแบบสุดท้ายของโดมิโนบนพื้นจะแตกต่างไปจากเดิมอย่างสิ้นเชิงจนจำไม่ได้

SHA-256 คือวงจรนี้ในทางคณิตศาสตร์ ข้อความที่คุณเขียนคือตำแหน่งเริ่มต้นของโดมิโน อัลกอริทึมคือการผลักที่ปลดปล่อยการล้มแบบต่อเนื่อง และผลลัพธ์สุดท้าย — ที่เราเรียกว่า *แฮช (hash)* — คือภาพนิ่งของพื้นที่หยุดนิ่งแล้ว เปลี่ยนเพียงคอมมาตัวเดียวในข้อความต้นฉบับ ภาพที่ได้จะแตกต่างไปอย่างสิ้นเชิง เรียบง่ายแต่ทรงพลังเช่นนั้นเอง

ขั้นตอนที่ 1 แปลงข้อความเป็นชิ้นส่วนไบนารี คอมพิวเตอร์ไม่เข้าใจตัวอักษร พวกเขาต้องแปลงเป็นตัวเลข (ASCII) ก่อน และจากตัวเลขเป็นเลขฐานสอง (ศูนย์และหนึ่ง) ตัวอักษรแต่ละตัวจะกลายเป็นชิ้นส่วนสีขาวหรือดำ 8 ชิ้น: ตัว *A* คือ `01000001`, *B* คือ `01000010`, เว้นวรรคคือ `00100000` ข้อความทั้งหมดของคุณ — ไม่ว่าจะเป็นคำเดียว สัญญา หรือนวนิยาย — จะกลายเป็นแถวโดมิโนสีขาวและดำที่ยาวเหยียด

ขั้นตอนที่ 2 เติมข้อมูลให้ได้ขนาดมาตรฐาน วงจรจะประมวลผลแถวข้อมูลเป็น *ช่วง (chunks)* ช่วงละ 512 ชิ้นพอดี หากข้อความของคุณยาวไม่ถึงผลคูณของ 512 จะมีการเพิ่มชิ้นส่วนตัวทำเครื่องหมาย (ค่า `10000000`) ต่อท้ายข้อความทันที และตามด้วยเลขศูนย์

จนครบช่วง 64 ตำแหน่งสุดท้ายของแต่ละช่วงจะถูกจองไว้เพื่อบันทึกความยาวเดิมของข้อความ วิธีนี้ทำให้วงจรทราบเสมอว่าเนื้อหาจริงสิ้นสุดที่ใดและส่วนที่เติมเริ่มจากที่ใด

ขั้นตอนที่ 3 วางโดมิโนหลักแปดตัว ก่อนเริ่มต้น เราวาง **โดมิโนหลักแปดตัว** ลงบนโต๊ะในตำแหน่งเริ่มต้นที่แม่นยำ โดมิโนแปดตัวนี้ไม่ใช่ความลับ ค่าเริ่มต้นของพวกมันถูกกำหนดโดยกฎคณิตศาสตร์สาธารณะ (รากที่สองของเลขจำนวนเฉพาะแปดตัวแรก — 2, 3, 5, 7, 11, 13, 17, 19 — และบิตแรกของส่วนทศนิยมของแต่ละราก) ทุกคนในทุกมุมโลกจะเริ่มต้นด้วยโดมิโนหลักแปดตัวเดิมในตำแหน่งเดียวกัน เป้าหมายของพวกมันคือการถูกผลักและเปลี่ยนแปลงโดยการล้มแบบต่อเนื่อง

ขั้นตอนที่ 4 การล้มแบบต่อเนื่องครั้งใหญ่: การผลักหกสิบสี่รอบ นี่คือจุดเริ่มต้นของการแสดง ข้อมูลช่วงแรก 512 ชิ้นของข้อความของคุณจะถูกผลักให้ชนกับโดมิโนหลักแปดตัว แต่พวกมันจะไม่ล้มในทันที กลไกจะทำงาน **หกสิบสี่รอบติดต่อกัน** ในแต่ละรอบจะมีการดำเนินการสามอย่างกับโดมิโน:

- **ม้าหมุน (การหมุน)** โดมิโนจะเคลื่อนที่วนเป็นวงกลม: ตัวที่อยู่ขวาจะย้ายไปซ้าย ไม่มีโดมิโนตัวใดหายไปหรือเพิ่มขึ้น เพียงแค่จัดลำดับใหม่โดยการหมุนม้าหมุนให้ครบหนึ่งรอบ นี่เป็นวิธีที่ประหยัดและย้อนกลับได้ในการกระจายข้อมูล
- **กรวยตรรกะ (XOR)** โดมิโนจะผ่านกรวยที่เปรียบเทียบพวกมันทีละคู่: หากทั้งสองตัวมีสีเดียวกัน จะออกมาเป็นสีขาว หากต่างกันจะออกมาเป็นสีดำ เป็นการดำเนินการที่ง่ายที่สุดของตรรกะไบนารี แต่เมื่อรวมกับการหมุนของม้าหมุนแล้ว มันจะมีพลังอย่างมหาศาลในการผสมข้อมูลโดยไม่สูญหาย
- **การล้น (การบวกแบบโมดูลาร์)** ผลลัพธ์จะถูกนำมาบวกกับ *โดมิโนแรงผลักคงที่* ที่นำมาจากรายการสาธารณะของค่าคงที่หกสิบสี่ตัว (รากที่สามของเลขจำนวนเฉพาะหกสิบสี่ตัวแรก) หากการบวกสร้างโดมิโนส่วนเกินที่ไม่พอดีกับพื้นที่ 32 ชิ้นที่เตรียมไว้ โดมิโนส่วนเกินเหล่านั้นจะถูกทิ้งไป โต๊ะมีพื้นที่สำหรับโดมิโนเพียง 32 ตัวเท่านั้น ไม่มากไปกว่านี้

เมื่อสิ้นสุดรอบที่หกสิบสี่ โดมิโนแต่ละตัวในช่วงข้อความของคุณจะมีอิทธิพลต่อตำแหน่งของโดมิโนหลักแปดตัว พลังงานจากการผลักจะเดินทางไปทั่วทั้งวงจร

ขั้นตอนที่ 5 เพิ่มช่วงถัดไป (โดยไม่เริ่มใหม่) หากข้อความของคุณยาวและยังมีช่วง 512 ชิ้นที่ต้องประมวลผลต่อ **วงจรจะไม่เริ่มใหม่** โดมิโนหลักแปดตัวจะยังคงอยู่ในตำแหน่งที่การล้มครั้งแรกทิ้งไว้ และช่วงที่สองจะถูกผลักใส่พวกมันเพื่อเปิดใช้งานอีกหกสิบสี่รอบ เหมือนกับการเพิ่มห้องใหม่ที่เต็มไปด้วยโดมิโนต่อท้ายห้องที่เพิ่งล้มไป: ความยุ่งเหยิงของห้องแรกจะกำหนดรูปแบบการล้มของห้องที่สองอย่างสมบูรณ์

ขั้นตอนที่ 6 ถ่ายภาพสุดท้าย เมื่อไม่มีช่วงข้อมูลให้ประมวลผลต่อ การล้มแบบต่อเนื่องจะหยุดลง เราดูตำแหน่งสุดท้ายที่โดมิโนหลักแปดตัวหยุดอยู่ เราแปลการจัดวางของพวกมันเป็นรหัสตัวอักษรและตัวเลขในระบบเลขฐานสิบหก ผลลัพธ์คือสตริงที่มีความยาว 64 ตัวอักษรพอดี: นั่นคือตราประทับ SHA-256 ของคุณ

คุณสมบัติสี่ประการเกิดขึ้นเองตามธรรมชาติจากการติดตั้งวงจรนี้:

1. **ความเป็นเหตุเป็นผล (Determinism)** ข้อความเดียวกันจะให้ภาพสุดท้ายเหมือนเดิมเสมอ บนคอมพิวเตอร์เครื่องใดก็ได้ในโลก ไม่มีความสุ่ม ไม่มีความประหลาดใจ
2. **ผลกระทบของการล้มแบบต่อเนื่อง (Avalanche effect)** การเพิ่มคอมมาหนึ่งตัว การเปลี่ยนตัวพิมพ์ใหญ่ หรือการลืมสระ: ภาพที่ได้จะจำไม่ได้เลย นี่คือความละเอียดอ่อนสูงสุดที่เราได้อธิบายไว้ตั้งแต่ต้น
3. **ทางเดียว (One-way)** จากภาพสุดท้าย คุณไม่สามารถสร้างข้อความต้นฉบับขึ้นมาใหม่ได้ การหมุน กรวย และการล้นทำลายข้อมูลทิศทางทั้งหมดว่า *แต่ละบิตมาจากที่ใด* และเก็บไว้เพียง *ผลรวมทั้งหมดที่ถูกบวกเข้าไป*
4. **การทนทานต่อการชนกัน (Collision resistance)** ในยี่สิบห้าปีของการวิเคราะห์การเข้ารหัสลับแบบสาธารณะ ไม่มีใครสามารถหาข้อความที่แตกต่างกันสองข้อความที่ให้ภาพสุดท้ายตรงกันได้ และความยากในการทำเช่นนั้นอยู่นอกเหนือความสามารถในการคำนวณของอารยธรรมใดๆ ที่พอจะจินตนาการได้

ภาคผนวกโค้ดด้านล่างนี้ใช้วิธีดำเนินการหกขั้นตอนเหล่านี้ในภาษา Zig ตอนนี้คุณสามารถอ่านมันได้โดยเข้าใจความหมายของการดำเนินการบิตแต่ละอย่าง แทนที่จะยอมรับการจัดการข้อมูลอย่างมืดบอด

อภิธานศัพท์ทางเทคนิค

สำหรับผู้อ่านที่ต้องการเข้าใจหน้าที่ของการดำเนินการแต่ละอย่าง สามารถข้ามส่วนนี้ไปได้หากต้องการ: บทความยังคงสามารถเข้าใจได้โดยไม่ต้องอ่านส่วนนี้

ASCII และ Unicode — วิธีที่ตัวอักษรกลายเป็นตัวเลข คอมพิวเตอร์ไม่เห็นตัวอักษร แต่เห็นตัวเลข มาตรฐานที่เรียกว่า **ASCII** (ค.ศ. 1963) กำหนดตัวเลขเฉพาะให้กับอักขระแต่ละตัวบนคีย์บอร์ด: *A* คือ 65, *B* คือ 66, *a* คือ 97, *0* คือ 48, เว้นวรรคคือ 32, คอมมาคือ 44 ระบบสมัยใหม่ขยายขอบเขตด้วย **Unicode** ซึ่งกำหนดตัวเลขให้กับอักขระทุกตัวของทุกตัวอักษรในโลก: ซิริลลิก อาหรับ จีน ญี่ปุ่น และแม้แต่เอโมจิ เมื่อคุณพิมพ์อักขระหรือเปิดไฟล์ข้อความ คอมพิวเตอร์จะอ่านตัวเลขที่อยู่เบื้องหลัง ไม่ใช่รูปทรงบนหน้าจอ SHA-256 ทำงานกับตัวเลขเหล่านี้ โดยถือว่าข้อความใดๆ เป็นลำดับตัวเลขที่ยาวเหยียด นั่นคือเหตุผลที่มันสามารถประทับตราบทความภาษาสเปน บทกวีภาษาญี่ปุ่น และไฟล์ไบนารีด้วยอัลกอริทึมเดียวกันได้
XOR — ตัวเปรียบเทียบบิตต่อบิต XOR (ย่อมาจาก *exclusive or*) คือหนึ่งในการดำเนินการที่ง่ายที่สุดที่คอมพิวเตอร์สามารถทำได้กับเลขฐานสองสองตัว มันเปรียบเทียบบิตสองบิตในตำแหน่งเดียวกันและส่งคืนค่า: **1** หากมีเพียงหนึ่งในสองตัวที่เป็น 1 (ตัวใดตัวหนึ่งแต่ไม่ใช่ทั้งคู่), **0** หากทั้งสองตัวเหมือนกัน (เป็น 0 ทั้งคู่หรือ 1 ทั้งคู่) ตัวอย่าง: XOR ของ `1010` และ `1100` คือ `0110` มันมีคุณสมบัติที่โดดเด่นคือ: มันย้อนกลับได้ — หากคุณทำ XOR สองครั้งด้วยรหัสเดิม คุณจะกลับไปที่ค่าเดิม นั่นคือเหตุผลที่มันเป็นหัวใจสำคัญของการเข้ารหัสลับ: มันผสมบิตโดยไม่สูญเสียข้อมูล แต่ผลลัพธ์จะไม่เปิดเผยข้อมูลใดๆ เกี่ยวกับข้อมูลขาเข้าหากคุณไม่ทราบค่าใดค่าหนึ่ง
เลขฐานสิบหก — การนับในฐาน 16 ตัวเลขเกือบทั้งหมดในชีวิตประจำวันใช้สิบหลัก (0-9) เลขฐานสิบหกใช้สิบหกหลัก: คือ 0-9 ตามปกติ บวกกับตัวอักษรหกตัวที่แทนค่าถัดไป: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15 ทำไมต้องสิบหก? เพราะคอมพิวเตอร์คิดเป็นกลุ่มละสี่บิต และสี่บิตสามารถแทนค่าที่แตกต่างกันได้สิบหกค่าพอดี — ดังนั้น อักขระเลขฐานสิบหกหนึ่งตัวจึงสอดคล้องกับสี่บิตได้อย่างลงตัว ตราประทับ SHA-256 วัดได้ 256 บิต ซึ่งมีค่าเท่ากับ **64 อักขระเลขฐานสิบหก** พอดี หากเราเขียนเป็นเลขฐานสิบปกติ มันจะกินพื้นที่ประมาณ 78 หลักและใช้งานไม่สะดวก การเลือกนี้เป็นเรื่องของความสวยงามและความกะทัดรัด แต่ค่าเบื้องหลังนั้นเหมือนกัน
การหมุนบิต — ม้าหมุนไบนารี ลองจินตนาการถึงแถวของหลอดไฟเจ็ดดวง บางดวงเปิด (1) และบางดวงปิด (0): `1 0 1 1 0 0 1` การหมุนไปทางขวาหนึ่งตำแหน่งประกอบด้วยการนำหลอดไฟขวาสุดไปไว้ที่ปลายซ้ายสุด และเลื่อนหลอดไฟดวงอื่นๆ ไปทางขวาหนึ่งตำแหน่ง: `1 1 0 1 1 0 0` ไม่มีหลอดไฟดวงใดหายไปหรือเพิ่มขึ้น เพียงแค่พวกมันเต้นระบำเป็นวงกลม SHA-256 ใช้การหมุนบิตหลายร้อยครั้งในการคำนวณแต่ละครั้ง เป็นวิธีที่ประหยัดและไม่มีการสูญเสียในการกระจายข้อมูลภายในสถานะ
ค่าคงที่ "nothing-up-my-sleeve" — ทำไมจึงมาจากเลขจำนวนเฉพาะ โดมิโนหลักแปดตัวและค่าคงที่รอบทั้งหกสิบสี่ตัวของ SHA-256 ไม่ได้ถูกเลือกมาแบบสุ่ม พวกมันมาจากรากที่สองและรากที่สามของเลขจำนวนเฉพาะตัวแรกๆ ทำไม? เพราะผู้ออกแบบต้องการค่าคงที่ที่ *"ไม่มีอะไรซ่อนอยู่ในแขนเสื้อ"*: ค่าที่ทุกคนสามารถตรวจสอบที่มาได้ หากใครบางคนบอกคุณว่า *"เชื่อใจฉันสิ: ใช้เลขสุ่ม 32 บิตนี้"* คุณอาจสงสัยได้อย่างมีเหตุผลว่ามีจุดอ่อนที่ซ่อนอยู่หรือมีประตูหลัง (backdoor) แต่ใครก็ตามที่มีเครื่องคิดเลขสามารถตรวจสอบได้ว่า 32 บิตแรกของรากที่สองของ 2 คือ `0x6a09e667` ค่าเหล่านี้เป็นค่าทางคณิตศาสตร์ เป็นสาธารณะ และทำซ้ำได้: ไม่มีกับดักที่ซ่อนอยู่สามารถเล็ดลอดเข้าไปในสูตรได้

ภาคผนวก: SHA-256 ในโค้ดที่อ่านง่าย

ภาคผนวกนี้จัดทำขึ้นสำหรับผู้อ่านที่ต้องการดูอัลกอริทึมจากภายใน นี่คือการทำงานในภาษา Zig เพื่อการศึกษาซึ่งอ้างอิงตามข้อกำหนด FIPS 180-4 มันไม่ใช่เวอร์ชันที่ Solo2 ใช้งานจริง (เวอร์ชันจริงอยู่ใน `std.crypto.hash.sha2.Sha256` ของไลบรารีมาตรฐาน Zig ซึ่งได้รับการปรับแต่งและตรวจสอบแล้ว) แต่อัลกอริทึมเป็นสิ่งเดียวกัน สิ่งที่เห็นที่นี่คือขั้นตอนที่เกิดขึ้นเมื่อมีการเรียกคำสั่งเพียงไม่กี่ตัวอักษรนั้นทำงาน

```
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
}
```

การเขียนซ้ำในภาษาอื่นที่ใช้โครงสร้างเดียวกัน ไม่ว่าจะเป็นค่าคงที่เริ่มต้น การขยายตาราง การทำงาน 64 รอบ และการสะสมผล จะให้ผลลัพธ์เดียวกัน อัลกอริทึมนี้ไม่มีความลับ คุณค่าของมันอยู่ที่คุณสมบัติต่างๆ ที่กล่าวไว้ข้างต้นยังคงยึดถือได้หลังจากผ่านการวิเคราะห์รหัสลับมานานกว่าสองทศวรรษภายใต้สายตาของผู้คนนับพัน

หากคุณกลับไปดูที่ส่วนท้ายของบทความนี้ คุณจะเห็นตราประทับเลขฐานสิบหกที่มีความยาว 64 ตัวอักษร มันคือ SHA-256 ของข้อความที่คุณเพิ่งอ่านในภาษานี้ หากเราแปลบทความนี้ ตราประทับก็จะเปลี่ยนไป หากมีการเปลี่ยนคำเพียงคำเดียวในเวอร์ชันภาษาไทย ตราประทับภาษาไทยก็จะเปลี่ยนไป ตราประทับไม่ได้ปกป้องเนื้อหา (นั่นคือหน้าที่ของเครื่องมืออื่น) แต่มันระบุตัวตนของเนื้อหาได้อย่างเป็นเอกลักษณ์ และนั่นก็เพียงพอแล้วที่จะทำให้ขั้นตอนใดๆ ในกระบวนการบรรณาธิการไม่สามารถแก้ไขสิ่งที่พูดไว้ได้โดยไม่ถูกสังเกตเห็น ส่วนที่เหลือ ไม่ว่าจะเป็นการเข้ารหัส การลงนาม การยืนยันตัวตน ล้วนสร้างขึ้นบนแนวคิดที่เรียบง่ายนี้

หมายเหตุจากบรรณาธิการ: เมื่อ Cuadernos เหล่านี้เอ่ยชื่อบริษัทหรือผลิตภัณฑ์ ไม่ใช่เพื่อการกล่าวหา ผู้ที่สร้างสรรค์สิ่งเหล่านั้นได้ทำงานที่คนนับล้านได้ใช้และชื่นชอบ สิ่งที่เรากำลังชี้ให้เห็นคือเรื่องเชิงโครงสร้าง — รูปแบบธุรกิจ ไม่ใช่แบรนด์ แบรนด์ต่างๆ ถูกยกมาเป็นตัวอย่างเพราะเป็นสิ่งที่ผู้อ่านรู้จัก

แหล่งข้อมูลและการอ่านเพิ่มเติม

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, สิงหาคม 2015. ข้อกำหนดอย่างเป็นทางการของตระกูล SHA-2 รวมถึง SHA-256
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, พฤษภาคม 2011. เวอร์ชันมาตรฐานสำหรับผู้พัฒนา
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). บทที่ 5 และ 6 ครอบคลุมถึงฟังก์ชันแฮชและการใช้งานที่ถูกต้องและไม่ถูกต้อง
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). ตัวอย่างการใช้งาน SHA-256 ในเชิงปฏิบัติเพื่อเชื่อมโยงบล็อกในโครงสร้างที่ไม่สามารถแก้ไขได้โดยการออกแบบ
- ข้อบังคับ (สหภาพยุโรป) 910/2014 (eIDAS) — กรอบการทำงานสำหรับผู้ให้บริการประทับรับรองเวลาที่ผ่านการรับรอง SHA-256 เป็นฟังก์ชันอ้างอิงสำหรับลายเซ็นดิจิทัลและตราประทับอิเล็กทรอนิกส์ที่ออกในสหภาพยุโรป
- ตัวอย่างอ้างอิงในภาษา Zig: `std.crypto.hash.sha2.Sha256` ในพื้นที่เก็บข้อมูลหลักของภาษา (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`) ซึ่งเป็นเวอร์ชันที่ Solo2 ใช้งานจริง มีประโยชน์สำหรับการเปรียบเทียบกับตัวอย่างเพื่อการศึกษาในภาคผนวก

← ก่อนหน้าSchrems II ห้าปีให้หลังถัดไป →Kill switch และการยึดกุมโดยสถาบัน

บทความล่าสุด

- การวิเคราะห์ · 18 พฤษภาคม 2026 ความเป็นส่วนตัวที่แท้จริง vs ความเป็นส่วนตัวที่ฉาบฉวย: คำถามที่คุณควรตั้งกับตัวเอง
- การวิเคราะห์ · 18 พฤษภาคม 2026 Self-hosting ในฐานะการปฏิบัติทางวิชาชีพ
- แนวคิด · 18 พฤษภาคม 2026 คำ 24 คำ: อัตลักษณ์การเข้ารหัสคืออะไร

ดาวน์โหลดบทความนี้เก็บไว้เพื่อใช้งานได้ทุกที่ที่คุณต้องการ

↓ Markdown ↓ ข้อความธรรมดา ↓ PDF

ไฟล์จะถูกดาวน์โหลดลงในอุปกรณ์ของคุณ คุณสามารถบันทึก นำเข้าสู่ Solo2 หรือแชร์ได้ทุกที่ตามต้องการ Cuadernos จะไม่กำหนดปลายทางแทนคุณ

ตราประทับครั่ง · SHA-256 11dd19b708ac3bfaf5509a553019cbd35d3ae2a855596af052b5135c5718eae9

Cuadernos Lacre · สิ่งพิมพ์ของ Menzuri Gestión S.L. ·
เขียนโดย R.Eugenio · เรียบเรียงโดยทีมงาน Solo2

เว็บไซต์นี้ไม่ใช้คุกกี้และไม่โหลดทรัพยากรจากบุคคลภายนอก ใช้ตัวนับการเข้าชมแบบไม่ระบุตัวตนที่โฮสต์เอง (Umami บนเซิร์ฟเวอร์ยุโรปของเรา) และ JavaScript ขั้นต่ำที่จำเป็นสำหรับส่วนควบคุมสองอย่างในส่วนหัว: ธีมสว่างหรือมืด และตัวเลือกภาษา ไม่มีเครื่องมือติดตาม ไม่มีการสร้างโปรไฟล์ ไม่มีการแชร์ข้อมูล หากคุณต้องการติดตามเรา: RSS