SHA-256 واقعاً چیست

اثر انگشت ریاضی که در شصت و چهار کاراکتر جا می‌شود و اگر تنها یک کامای متن اصلی جابه‌جا شود، کل آن تغییر می‌کند. چرا ما به آن مهر موم دیجیتال می‌گوییم.

ایده ساده پشت نام فنی

تصور کنید ماشینی وجود دارد که فقط یک شکاف و یک صفحه نمایش دارد. از شکاف، متنی را وارد می‌کنید: یک کلمه، یک جمله یا یک رمان کامل. لحظاتی بعد روی صفحه نمایش، دنباله‌ای دقیقاً شصت و چهار کاراکتری ظاهر می‌شود. ما به این دنباله، برای خواننده حرفه‌ای، *هش* یا *خلاصه رمزنگارانه* می‌گوییم؛ برای خواننده عمومی، فعلاً می‌توانیم آن را اثر انگشت ریاضی متن بنامیم، همان‌طور که اثر انگشت برای یک شخص است.

اگر یک متن را دو بار وارد کنید، ماشین هر دو بار همان اثر انگشت را نشان می‌دهد. اگر متنی را کمی متفاوت وارد کنید - یک کامای جابه‌جا شده یا تغییر یک حرف بزرگ به کوچک - ماشین اثر انگشتی کاملاً متفاوت از اولی نشان می‌دهد. نه شبیه، بلکه متفاوت. این دو ویژگی با هم - قطعیت و حساسیت - همان ایده ساده هستند. بقیه موارد SHA-256 ماشین‌آلاتی است که باعث می‌شود این ویژگی‌ها به خوبی اجرا شوند.

بهتر است از ابتدا بگوییم که ماشین چه کاری انجام نمی‌دهد. متن را رمزگذاری نمی‌کند. آن را پنهان نمی‌کند. آن را ذخیره نمی‌کند. ماشین به متن نگاه می‌کند، اثر انگشت را محاسبه می‌کند و متن را فراموش می‌کند. اثر انگشت اجازه بازسازی متنی که آن را تولید کرده را نمی‌دهد؛ فقط اجازه می‌دهد تا با داشتن یک متن کاندید، بررسی کنید که آیا با متن اصلی مطابقت دارد یا خیر. به همین دلیل است که می‌گوییم این یک خلاصه *یک‌طرفه* است: می‌رود و بر نمی‌گردد.

هش با رمزگذاری یکی نیست

سردرگمی در این مورد زیاد است و بهتر است برطرف شود: رمزگذاری و هش کردن عملیات متفاوتی هستند. رمزگذاری شامل تغییر دادن یک متن است به گونه‌ای که فقط دارنده کلید بتواند آن را به شکل اصلی‌اش بازگرداند. هش کردن شامل تولید اثر انگشتی از متن است که متن اصلی هرگز از آن قابل بازیابی نیست، نه با کلید و نه بدون آن. اولی طبق طراحی بازگشت‌پذیر است و دومی طبق طراحی بازگشت‌ناپذیر.

پیامد عملی این موضوع مهم است. وقتی یک اپلیکیشن می‌گوید «ما رمز عبور شما را به صورت رمزگذاری شده ذخیره می‌کنیم»، کسی هست که کلید رمزگشایی آن را دارد - در هر صورت خود اپلیکیشن. وقتی یک اپلیکیشن می‌گوید «ما رمز عبور شما را به صورت هش شده ذخیره می‌کنیم»، خود اپلیکیشن حتی اگر بخواهد هم نمی‌تواند رمز عبور اصلی را بخواند؛ فقط می‌تواند بررسی کند که آیا آنچه شما تایپ می‌کنید دوباره همان اثر انگشت را تولید می‌کند یا خیر. مدل دوم، اگر به درستی اجرا شود، برای ذخیره رمزهای عبور بسیار بهتر از اولی است. بعداً خواهیم دید که چرا «اجرای درست» به چیزی فراتر از صرفاً SHA-256 نیاز دارد.

چهار ویژگی که یک هش رمزنگارانه را مفید می‌کند

یک تابع هش که شایسته صفت *رمزنگارانه* باشد، چهار ویژگی را برآورده می‌کند:

1. **قطعیت (Determinism).** ورودی یکسان همیشه اثر انگشت یکسانی تولید می‌کند.
2. **اثر بهمنی (Avalanche effect).** تغییری کوچک در ورودی، اثر انگشتی کاملاً متفاوت تولید می‌کند، بدون هیچ شباهت ظاهری به قبلی.
3. **مقاومت در برابر معکوس‌سازی (Pre-image resistance).** با داشتن یک اثر انگشت، از نظر محاسباتی پیدا کردن متنی که آن را تولید کرده، امکان‌پذیر نیست.
4. **مقاومت در برابر تصادم (Collision resistance).** از نظر محاسباتی پیدا کردن دو متن متفاوت که اثر انگشت یکسانی تولید کنند، امکان‌پذیر نیست.

«از نظر محاسباتی امکان‌پذیر نیست» به معنای «از نظر ریاضی غیرممکن است» نیست. بلکه به این معناست که هزینه زمانی، انرژی و مالی برای دستیابی به آن، چندین برابر مجموع تمام توان محاسباتی در دسترس است. برای SHA-256، این حد حتی در خوش‌بینانه‌ترین حالت‌ها با سخت‌افزار تخصصی، میلیاردها میلیارد سال تخمین زده می‌شود. که برای اهداف کاربردی خواننده، همان «نمی‌شود» است.

به طور مشخص SHA-256

نام آن گویای همه چیز است. SHA مخفف *Secure Hash Algorithm* به معنای الگوریتم هش امن است. عدد ۲۵۶ اندازه اثر انگشت را به بیت نشان می‌دهد: دویست و پنجاه و شش بیت، یعنی سی و دو بایت، که در حالت هگزادسیمال همان شصت و چهار کاراکتری است که خواننده از قبل می‌شناسد. این استاندارد توسط NIST ایالات متحده، سازمانی که این نوع توابع را استانداردسازی می‌کند، در سال ۲۰۰۱ به عنوان بخشی از خانواده SHA-2 منتشر شد؛ نسخه فعلی استاندارد، FIPS 180-4، مربوط به سال ۲۰۱۵ است.

برای کسانی که هنوز به یاد ندارند بیت و بایت چیست:

```
۱ بیت   ← ۰ یا ۱          (یک کلید: روشن یا خاموش)
۱ بایت  ← ۸ بیت           (۲۵۶ ترکیب ممکن)
۳۲ بایت ← ۲۵۶ بیت         (اثر انگشت SHA-256)
```

عدد ۲۵۶ در انتهای نام، اندازه اثر انگشت را به بیت می‌گوید. در هگزادسیمال - یک سیستم عددنویسی با شانزده نماد به جای ده - این ۲۵۶ بیت دقیقاً در ۶۴ کاراکتر جا می‌شوند. این‌ها همان ۶۴ کاراکتری هستند که در پایین هر Cuaderno می‌بینید.

ابعاد آن ارزش یک لحظه تأمل را دارد. دویست و پنجاه و شش بیت، اجازه ۲ به توان ۲۵۶ مقدار متفاوت را می‌دهد: عددی با هفتاد و هشت رقم اعشار، چندین برابر بزرگتر از تعداد تخمینی اتم‌ها در جهان قابل مشاهده. هر متن در جهان - هر کتاب، هر ایمیل، هر پیام - روی یکی از این مقادیر می‌افتد. احتمال اینکه دو متن متفاوت به طور تصادفی با هم یکی شوند، برای اهداف کاربردی، صفر است.

در کد چگونه دیده می‌شود

در Zig، زبانی که قطعات نگهدارنده Solo2 را با آن می‌نویسیم، محاسبه مهر SHA-256 یک متن به این صورت است:

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

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

ما به تازگی از کتابخانه استاندارد Zig خواستیم که SHA-256 متن داخل گیومه را محاسبه کند. بعد از فراخوانی، متغیر *resumen* شامل سی و دو بایتی است که مهر را در شکل خام آن تشکیل می‌دهند؛ وقتی به صورت هگزادسیمال روی صفحه نمایش داده می‌شوند، همان شصت و چهار کاراکتری هستند که در پایین این مقاله ظاهر می‌شوند. اگر *Cuadernos Lacre* را به *Cuadernos lacre* تغییر دهیم - یک حرف بزرگ کمتر - کل مهر تغییر می‌کند. این، در پنج خط، ویژگی مرکزی است که بقیه موارد بر آن تکیه دارند. برای کسانی که می‌خواهند ببینند در داخل چگونه کار می‌کند، در انتهای مقاله نسخه‌ای خوانا از الگوریتم را با توضیحات مرحله به مرحله آورده‌ایم.

چرا ما به آن مهر موم می‌گوییم

در نامه‌نگاری‌های اروپایی قرن پانزدهم تا نوزدهم، موم نامه را می‌بست. یک قطره موم ذوب شده، مهری که روی آن فشار داده می‌شد و نامه به شکلی تکرارنشدنی علامت‌گذاری می‌شد. این کار محتوا را از کنجکاوهای مصمم محافظت نمی‌کرد - کاغذ را می‌شد در مقابل نور خواند، موم را می‌شد شکست - اما آن را آشکار می‌کرد. هرگونه تغییر در بسته‌بندی برای گیرنده، حتی قبل از باز کردن کاغذ، قابل مشاهده بود. موم از آسیب جلوگیری نمی‌کرد؛ بلکه آن را اعلام می‌کرد.

مقدار SHA-256 بدنه هر Cuaderno همان عملکرد را در نسخه دیجیتال خود ایفا می‌کند. اگر تنها یک کلمه از مقاله بین لحظه انتشار و لحظه خواندن شما تغییر کند، مهر هگزادسیمال پایین متن دیگر با SHA-256 متنی که در مقابل دارید مطابقت نخواهد داشت. هر خواننده‌ای با پنج خط کد می‌تواند این را بررسی کند. ناشر نمی‌تواند تاریخ خود را بدون اینکه مهر آن را لو دهد، بازنویسی کند. از آسیب محافظت نمی‌کند؛ بلکه آن را قابل تأیید می‌کند.

آنچه یک هش نیست

گاهی اوقات از SHA-256 انتظارهایی می‌رود که مربوط به آن نیست:

1. **رمزگذاری.** یک هش خلاصه می‌کند، پنهان نمی‌کند. اگر می‌خواهید متن قابل خواندن نباشد، باید آن را رمزگذاری کنید، نه هش.
2. **احراز هویت نویسنده.** یک هش نمی‌گوید چه کسی متن را نوشته است، فقط می‌گوید چه متنی هش شده است. برای مرتبط کردن نویسندگی، به یک امضای رمزنگارانه روی هش نیاز است، نه صرفاً خود هش.
3. **ذخیره رمزهای عبور.** اینجا تله‌ای وجود دارد که بهتر است درک شود. SHA-256 طوری طراحی شده که بسیار سریع باشد - که برای خیلی چیزها خوب است، اما برای این کار بد. یک مهاجم با سخت‌افزار تخصصی می‌تواند میلیاردها رمز عبور را در ثانیه در مقابل یک هش SHA-256 آزمایش کند تا به رمز شما برسد. برای ذخیره رمزهای عبور، باید از توابع اشتقاق کلید (Key Derivation Functions) که عمداً کند هستند مانند scrypt، Argon2 یا bcrypt استفاده کرد، همراه با یک *نمک* (salt) (یک داده تصادفی منحصر به فرد برای هر کاربر که مانع از آن می‌شود که دو نفر با رمز عبور یکسان، هش یکسانی داشته باشند).
4. **خواندن هش به عنوان شناسه نویسنده.** این‌طور نیست. یک هش محتوا را شناسایی می‌کند. اگر دو نفر کلمه *سلام* را با SHA-256 هش کنند، هر دو خلاصه یکسانی به دست می‌آورند - و این ویژگی اصلی است، نه یک نقص: اگر خلاصه‌ها متفاوت بودند، نمی‌توانستیم مطابقت بین آنچه منتشر شده و آنچه دریافت شده را بررسی کنیم.

SHA-256 در زندگی روزمره شما کجا ظاهر می‌شود

اگرچه آن را نمی‌بینید، اما SHA-256 بخش بزرگی از آنچه را که روزانه در اینترنت استفاده می‌کنید، پشتیبانی می‌کند. بلاک‌چین بیت‌کوین با زنجیر کردن SHA-256 هر بلاک به بلاک بعدی ساخته می‌شود؛ تغییر یک بلاک در گذشته، محاسبه مجدد کل زنجیره بعدی را اجباری می‌کند. Git، سیستمی که نیمی از کدهای دنیا با آن نسخه‌بندی می‌شوند، هر کامیت را با SHA-256 (در نسخه‌های جدید) یا با نسخه قبلی آن یعنی SHA-1 (در نسخه‌های قدیمی) از کل محتوای آن شناسایی می‌کند. گواهی‌های HTTPS که هویت یک وب‌سایت را هنگام ورود شما تأیید می‌کنند، یک اثر انگشت SHA-256 همراه خود دارند. دانلودهای نرم‌افزار اغلب با یک SHA-256 منتشر شده توسط توسعه‌دهنده همراه هستند تا بتوانید بررسی کنید که فایل در طول مسیر تغییر نکرده است. و همان‌طور که گفتیم، در پایین هر Cuaderno Lacre.

برای خواننده حرفه‌ای

چهار یادآوری عملیاتی برای کسانی که در مورد سیستم‌ها تصمیم می‌گیرند یا آن‌ها را ممیزی می‌کنند:

1. هش، رمزگذاری نیست. اگر یک ارائه‌دهنده این دو اصطلاح را در اسناد فنی خود اشتباه بگیرد، بهتر است بپرسید منظورش دقیقاً چیست.

2. برای ذخیره رمزهای عبور هرگز نباید از SHA-256 به تنهایی استفاده کرد. SHA-256 برای این کار بسیار سریع است (به نقطه ۳ در بخش *آنچه یک هش نیست* مراجعه کنید). استاندارد فعلی **Argon2id** است: در طراحی کند، قابل تنظیم بر اساس توان سرور، و همراه با یک *نمک* تصادفی متفاوت برای هر کاربر.
3. برای یکپارچگی اسناد - قراردادها، پرونده‌ها، فایل‌ها - SHA-256 همچنان استاندارد مرجع است. این همان چیزی است که مهرهای زمانی معتبر در اتحادیه اروپا از آن استفاده می‌کنند.
4. برای نگهداری طولانی‌مدت (چندین دهه)، بهتر است در کنار SHA-256، یک SHA-3 یا SHA-512 را نیز محاسبه و آرشیو کنید؛ احتیاط رمزنگارانه توصیه می‌کند در آرشیوهای صدساله به یک تابع واحد تکیه نکنید.

از نظر فنی، این ساختار تکرار شونده - که در آن حالت میانی بین بلوک‌های ورودی حفظ می‌شود - به عنوان ساختار **Merkle-Damgård** شناخته می‌شود؛ الگویی که SHA-1، SHA-2 (شامل SHA-256) و بسیاری دیگر از توابع درهم‌ساز (hash) کلاسیک بر پایه آن بنا شده‌اند. در مقابل، SHA-3 ساختار Merkle-Damgård را به نفع معماری متفاوتی به نام *اسفنجی* (sponge) کنار می‌گذارد.

SHA-256 چگونه کار می‌کند؛ گام‌به‌گام و به زبان ساده

تصور کنید پیچیده‌ترین مدار دومینوی جهان را چیده‌اید: هزاران مهره، ده‌ها دوشاخه، پل‌های مکانیکی و رمپ‌هایی که کل اتاق را در بر می‌گیرند و قطعه‌به‌قطعه با دقت چیده شده‌اند.

اگر ضربه‌ای به اولین مهره بزنید، زنجیره با توالی دقیق و تکرارپذیری فرو می‌ریزد. چیدمان یکسان، ضربه اولیه یکسان ← الگوی نهایی یکسان از مهره‌های فرو ریخته، بارها و بارها.

بخش جالب اینجاست: قبل از شروع، **فقط یک مهره** را نیم سانتی‌متر به یک طرف جابه‌جا کنید و دوباره ضربه بزنید. رمپی که قرار بود فعال شود ثابت می‌ماند، پلی فرو نمی‌ریزد، دوشاخه متفاوتی فعال می‌شود. الگوی نهایی مهره‌ها روی زمین در مقایسه با الگوی اول کاملاً غیرقابل تشخیص است.

SHA-256 از نظر ریاضی همین مدار است. متنی که می‌نویسید، موقعیت اولیه مهره‌هاست. الگوریتم، همان ضربه‌ای است که بهمن را رها می‌کند. و نتیجه نهایی - چیزی که ما به آن *هش* (hash) می‌گوییم - عکس ثابتی از زمین است، وقتی همه چیز متوقف شده باشد. تنها یک کاما را در متن اصلی تغییر دهید و عکس به کلی متفاوت خواهد شد. به همین سادگی و به همین شدت.

گام ۱. ترجمه متن به مهره‌های باینری. کامپیوترها حروف را نمی‌فهمند؛ آن‌ها ابتدا حروف را به اعداد (ASCII) و اعداد را به باینری (صفر و یک) ترجمه می‌کنند. هر حرف به ۸ مهره سفید یا سیاه تبدیل می‌شود: حرف *A* می‌شود `01000001` ، حرف *B* می‌شود `01000010` و فاصله می‌شود `00100000`. کل متن شما - یک کلمه، یک قرارداد، یک رمان - به ردیف طویلی از مهره‌های سفید و سیاه تبدیل می‌شود.

گام ۲. پر کردن تا اندازه استاندارد. مدار، ردیف را در **بخش‌های** دقیقاً ۵۱۲ مهره‌ای پردازش می‌کند. اگر پیام شما به مضربی از ۵۱۲ نرسد، یک مهره نشانگر (با مقدار `10000000`) درست بعد از متن اضافه می‌شود و سپس صفرهایی تا کامل شدن بخش اضافه می‌گردد. ۶۴ جایگاه آخر هر بخش برای ثبت طول اصلی متن رزرو شده است. به این ترتیب، مدار همیشه می‌داند محتوای واقعی کجا تمام شده و پرکننده از کجا شروع شده است.

گام ۳. چیدن هشت مهره اصلی. قبل از شروع، **هشت مهره اصلی** (master tiles) را در موقعیت اولیه دقیقی روی میز قرار می‌دهیم. این هشت مهره رازی ندارند: مقدار اولیه آن‌ها با یک قاعده ریاضی عمومی تعیین شده است (ریشه دوم هشت عدد اولِ نخست - ۲، ۳، ۵، ۷، ۱۱، ۱۳، ۱۷، ۱۹ - و اولین بیت‌های بخش اعشاری هر ریشه). همه در هر گوشه از سیاره، با همان هشت مهره اصلی در همان موقعیت شروع می‌کنند. سرنوشت آن‌ها این است که توسط بهمن هل داده شوند و تغییر شکل دهند.

گام ۴. بهمن بزرگ: شصت و چهار دور هل دادن. اینجا نمایش شروع می‌شود. اولین بخش ۵۱۲ مهره‌ای متن شما با هشت مهره اصلی برخورد می‌کند. اما آن‌ها یکباره فرو نمی‌ریزند: مکانیزم، **شصت و چهار دور متوالی** را اجرا می‌کند. در هر دور، سه عملیات روی مهره‌ها انجام می‌دهد:

- **چرخ‌وفلک** (چرخش). مهره‌ها به صورت دایره‌ای حرکت می‌کنند: مهره‌های سمت راست به سمت چپ می‌روند. هیچ مهره‌ای گم یا اضافه نمی‌شود؛ آن‌ها صرفاً با یک دور کامل در چرخ‌وفلک بازآرایی می‌شوند. این

یک راه ارزان و برگشت‌پذیر برای توزیع مجدد اطلاعات است.
- **قیف منطقی** (XOR). مهره‌ها از قیفی عبور می‌کنند که آن‌ها را دوبه‌دو مقایسه می‌کند: اگر هر دو همرنگ باشند، یک مهره سفید خارج می‌شود؛ اگر متفاوت باشند، یک مهره سیاه خارج می‌شود. این ساده‌ترین عملیات در منطق باینری است، اما در ترکیب با چرخش‌های چرخ‌وفلک، برای مخلوط کردن اطلاعات بدون از دست دادن آن‌ها بسیار قدرتمند می‌شود.
- **سرریز** (جمع پیمانه‌ای). نتیجه با یک *مهره هل‌دهنده ثابت* که از لیست عمومیِ شصت و چهار ثابت (ریشه سوم شصت و چهار عدد اولِ نخست) آورده شده، جمع می‌شود. اگر حاصل جمع مهره‌های اضافی تولید کند که در فضای ۳۲ مهره‌ای پیش‌بینی شده جا نشوند، آن مهره‌های اضافی دور ریخته می‌شوند. میز فقط برای ۳۲ مهره جا دارد، نه یکی بیشتر.

در پایان دور شصت و چهارم، هر یک از مهره‌های بخشِ متن شما بر موقعیت هشت مهره اصلی تأثیر گذاشته است. انرژیِ هل دادن در کل مدار سفر کرده است.

گام ۵. اضافه کردن بخش بعدی (بدون بازنشانی). اگر متن شما طولانی بود و بخش ۵۱۲ مهره‌ای دیگری برای پردازش باقی مانده باشد، **مدار بازنشانی نمی‌شود**. هشت مهره اصلی دقیقاً به همان صورتی که اولین بهمن آن‌ها را رها کرده باقی می‌مانند و بخش دوم علیه آن‌ها پرتاب می‌شود تا شصت و چهار دور دیگر را فعال کند. این مثل اضافه کردنِ یک اتاق جدید پر از دومینو به انتهای اتاقی است که تازه فرو ریخته: بی‌نظمیِ اولی کاملاً تعیین می‌کند که دومی چگونه فرو بریزد.

گام ۶. گرفتن عکس نهایی. وقتی دیگر بخشی برای پردازش باقی نمانده باشد، بهمن متوقف می‌شود. به موقعیت نهایی که هشت مهره اصلی در آن قرار گرفته‌اند نگاه می‌کنیم. پیکربندی آن‌ها را به کدی از حروف و اعداد در سیستم هگزادسیمال ترجمه می‌کنیم. نتیجه، رشته‌ای دقیقاً شصت و چهار کاراکتری است: این همان مهر SHA-256 شماست.

چهار ویژگی به خودیِ خود از نحوه چیدمان مدار حاصل می‌شوند:

1. **قطعیت (Determinism).** یک متن یکسان همیشه عکس نهایی یکسانی را در هر کامپیوتری در جهان تولید می‌کند. بدون تصادفی بودن، بدون غافلگیری.
2. **اثر بهمنی.** اضافه شدن یک کاما، تغییر یک حرف بزرگ، فراموش کردن یک علامت: عکس نهایی کاملاً غیرقابل تشخیص می‌شود. این همان حساسیت شدیدی است که در ابتدا شرح دادیم.
3. **یک‌طرفه بودن.** با داشتن عکس نهایی، نمی‌توانید متن اصلی را بازسازی کنید. چرخش‌ها، قیف‌ها و سرریزها تمام اطلاعات جهت‌دار درباره اینکه *هر بیت از کجا آمده* را از بین می‌برند و فقط حفظ می‌کنند که *در مجموع چه چیزی جمع شده است*.
4. **مقاومت در برابر تصادم.** در بیست و پنج سال تحلیل عمومی رمزنگاری، هیچ‌کس موفق نشده دو متن متفاوت پیدا کند که عکس‌های نهایی آن‌ها یکسان باشد. و دشواریِ انجام این کار فراتر از توان محاسباتیِ هر تمدنِ قابل تصوری است.

ضمیمه کدی که در ادامه می‌آید، دقیقاً این شش مرحله را در زبان Zig پیاده‌سازی می‌کند. اکنون می‌توانید آن را با دانستن معنای هر عملیات بیتی بخوانید، به جای اینکه دستکاری‌ها را کورکورانه بپذیرید.

واژه‌نامه فنی

برای خواننده‌ای که می‌خواهد بداند هر عملیات چه کاری انجام می‌دهد. می‌توانید آزادانه از آن بگذرید: مقاله بدون آن هم قابل درک است.

ASCII و Unicode - چگونه حروف به عدد تبدیل می‌شوند. کامپیوترها حروف را نمی‌بینند؛ آن‌ها اعداد را می‌بینند. استانداردی به نام **ASCII** (*American Standard Code for Information Interchange*، محصول ۱۹۶۳) به هر کاراکتر کیبورد یک عدد خاص اختصاص می‌دهد: *A* برابر ۶۵ است، *B* برابر ۶۶، *a* برابر ۹۷، *0* برابر ۴۸، فاصله برابر ۳۲ و کاما برابر ۴۴ است. سیستم‌های مدرن آن را با **Unicode** گسترش می‌دهند که به هر کاراکتر از هر الفبای جهان (سیریلیک، عربی، چینی، ژاپنی و حتی ایموجی‌ها) یک عدد اختصاص می‌دهد. وقتی کاراکتری را تایپ می‌کنید یا یک فایل متنی را باز می‌کنید، کامپیوتر عددِ زیرین را می‌خواند، نه شکل روی صفحه را. SHA-256 روی این اعداد کار

می‌کند و با هر متنی به عنوان توالی طویلی از ارقام برخورد می‌کند. به همین دلیل است که می‌تواند یک مقاله اسپانیایی، یک شعر ژاپنی و یک فایل باینری را با همان الگوریتم مهروموم کند.
XOR - مقایسه‌گر بیت‌به‌بیت. XOR (مخفف *exclusive or* یا «یا-یِ انحصاری») یکی از ساده‌ترین عملیاتی است که یک کامپیوتر می‌تواند با دو عدد باینری انجام دهد. این عملیات، دو بیت را جایگاه‌به‌جایگاه مقایسه می‌کند و بازمی‌گرداند: **1** اگر دقیقاً یکی از آن دو ۱ باشد (یکی ولی نه هر دو)، **0** اگر هر دو یکسان باشند (هر دو ۰ یا هر دو ۱). مثال: `1010` XOR `1100` می‌شود `0110`. این عملیات ویژگی قابل توجهی دارد: برگشت‌پذیر است - اگر دو بار با همان کلید XOR انجام دهید، به مقدار اصلی بازمی‌گردید. به همین دلیل است که اسبِ بارکشِ رمزنگاری است: بیت‌ها را بدون از دست دادن اطلاعات مخلوط می‌کند، اما نتیجه هیچ چیزی از ورودی‌ها را فاش نمی‌کند مگر اینکه یکی از آن‌ها را بدانید.
هگزادسیمال - شمارش در مبنای ۱۶. تقریباً تمام اعداد روزمره از ده رقم (0-9) استفاده می‌کنند. هگزادسیمال از شانزده رقم استفاده می‌کند: ارقام معمول 0-9 به علاوه شش حرف که مقادیر بعدی را نشان می‌دهند: `A` = 10، `B` = 11، `C` = 12، `D` = 13، `E` = 14، `F` = 15. چرا شانزده؟ چون کامپیوترها در گروه‌های چهار بیتی فکر می‌کنند و چهار بیت می‌تواند دقیقاً شانزده مقدار متفاوت را نشان دهد - بنابراین، یک کاراکتر هگزادسیمال دقیقاً با چهار بیت متناظر است. یک اثر انگشت SHA-256 دارای ۲۵۶ بیت است که دقیقاً می‌شود **۶۴ کاراکتر هگزادسیمال**. اگر آن را در سیستم دهدهیِ معمولی می‌نوشتیم، حدود ۷۸ رقم اشغال می‌کرد و دشوارتر می‌شد. این انتخاب، زیبایی‌شناختی و فشرده است؛ عدد زیرین همان است.
چرخش بیت - چرخ‌وفلکِ باینری. ردیفی از هفت لامپ را تصور کنید که برخی روشن (1) و برخی خاموش (0) هستند: `1 0 1 1 0 0 1`. چرخش به راست به اندازه یک جایگاه، شامل برداشتن لامپِ سمت راستِ مطلق، بردن آن به منتهی‌الیه سمت چپ و جابه‌جا کردن بقیه به اندازه یک جایگاه به راست است: `1 1 0 1 1 0 0`. هیچ لامپی گم یا اضافه نمی‌شود؛ آن‌ها صرفاً به صورت دایره‌ای می‌رقصند. SHA-256 صدها بار در هر محاسبه از چرخش بیت استفاده می‌کند؛ این راهی ارزان و بدون اتلاف برای توزیع مجدد اطلاعات در داخل حالت (state) است.
ثابت‌های «*بدون پنهان‌کاری*» - چرا از اعداد اول می‌آیند. هشت مهره اصلی و شصت و چهار ثابتِ دور (round constants) در SHA-256 به صورت تصادفی انتخاب نشده‌اند. آن‌ها از ریشه دوم و سوم اولین اعداد اول گرفته شده‌اند. چرا؟ چون طراحان آن‌ها ثابت‌های **«بدون پنهان‌کاری»** (nothing-up-my-sleeve) می‌خواستند: مقادیری که هر کسی بتواند منشأ آن‌ها را تأیید کند. اگر کسی به شما می‌گفت *«به من اعتماد کن: از این عدد تصادفی ۳۲ بیتی استفاده کن»*، شما منطقاً به یک نقطه ضعف پنهان یا یک دربِ پشتی شک می‌کردید. اما هر کسی با یک ماشین حساب می‌تواند بررسی کند که ۳۲ بیت اولِ ریشه دوم عدد ۲، همان `0x6a09e667` است. مقادیرْ ریاضی، عمومی و بازتولیدپذیر هستند: هیچ حیله پنهانی نمی‌تواند وارد این دستورالعمل شود.

پیوست: SHA-256 در کد خوانا

این پیوست برای خواننده‌ای است که می‌خواهد الگوریتم را از داخل ببیند. این یک پیاده‌سازی آموزشی در Zig است که از مشخصات FIPS 180-4 پیروی می‌کند. این نسخه‌ای نیست که Solo2 استفاده می‌کند - نسخه واقعی در `std.crypto.hash.sha2.Sha256` در کتابخانه استاندارد Zig قرار دارد که بهینه شده و ممیزی شده است. اما الگوریتم همان است: آنچه در اینجا می‌بینید، مرحله به مرحله، اتفاقی است که وقتی آن فراخوانی پنج کاراکتری کار خود را انجام می‌دهد، رخ می‌دهد.

```
;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 //
                                        .(19 ,17 ,13 ,11 ,7 ,5 ,3 ,2) //
                                              }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
```

```
8a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5
07aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174
9b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da
3e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967
b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85
bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070
a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3
8f82ee, 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 //
                                                                                {
```

هرگونه بازنویسی به زبان دیگر که از همان ساختار پیروی کند - ثابت‌های اولیه، گسترش زمان‌بندی (schedule expansion)، شصت و چهار دور، انباشت - نتیجه یکسانی تولید می‌کند. الگوریتم هیچ رازی ندارد: ارزش آن در این است که ویژگی‌های ذکر شده در بالا پس از دو دهه تحلیل رمز عمومی توسط هزاران چشم، همچنان پابرجا هستند.

اگر به پایین این مقاله برگردید، یک مهر هگزادسیمال شصت و چهار کاراکتری خواهید دید. این SHA-256 متنی است که به این زبان خواندید. اگر مقاله را ترجمه می‌کردیم، مهر متفاوت می‌بود؛ اگر یک کلمه از نسخه اسپانیایی تغییر می‌کرد، مهر اسپانیایی تغییر می‌کرد. مهر از محتوا محافظت نمی‌کند - ابزارهای دیگری برای آن وجود دارد - بلکه آن را به طور منحصر به فرد شناسایی می‌کند. و این، هرچند ساده به نظر برسد، کافی است تا هیچ مرحله‌ای از زنجیره تحریریه نتواند آنچه گفته شده را بدون اینکه متوجه شوید، تغییر دهد. بقیه موارد - رمزگذاری، امضا، شناسایی - بر روی این ایده ساده ساخته می‌شوند.

منابع و مطالعه بیشتر

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*، اوت ۲۰۱۵. مشخصات رسمی خانواده SHA-2، شامل SHA-256.
- IETF، RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*، مه ۲۰۱۱. نسخه هنجاری برای پیاده‌سازان.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). فصل‌های ۵ و ۶ توابع هش و استفاده‌های مشروع و نامشروع آن‌ها را پوشش می‌دهند.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). نمونه عملی استفاده از SHA-256 برای زنجیره بلاک‌ها در ساختاری که ذاتاً تغییرناپذیر است.
- مقررات (EU) 910/2014 (eIDAS) — چارچوب مهرهای زمانی معتبر. SHA-256 تابع مرجع برای امضاها و مهرهای الکترونیکی معتبری است که در اتحادیه اروپا صادر می‌شوند.
- پیاده‌سازی مرجع در Zig: `std.crypto.hash.sha2.Sha256` در مخزن رسمی زبان (*github.com/ziglang/zig* ← `lib/std/crypto/sha2.zig`). این نسخه بهینه شده و ممیزی شده‌ای است که Solo2 در واقع از آن استفاده می‌کند. مفید برای مقایسه با پیاده‌سازی آموزشی در پیوست.

← السابق CUADERNOS_LIST_SCHREMS_TITLE التالي → CUADERNOS_LIST_KILLSWITCH_TITLE

قراءات حديثة

- CUADERNOS_LIST_PREGUNTAS_TITLE
- CUADERNOS_LIST_SELFHOST_TITLE
- CUADERNOS_LIST_IDENTIDAD_TITLE

این مقاله را هر کجا که نیاز دارید همراه خود ببرید.

↓ مارک‌داون ↓ متن ساده ↓ PDF

فایل در دستگاه شما دانلود خواهد شد. از آنجا می‌توانید آن را ذخیره کنید، به Solo2 وارد کنید یا در هر کجا که می‌خواهید به اشتراک بگذارید. Cuadernos مقصد را برای شما تعیین نمی‌کند.

ختم شمعي · SHA-256 1bc45f83aa6537cb0f641d73524677fc691be8e95cc470cd38353fa537aef5a0

Cuadernos Lacre · نشريّة من Menzuri Gestión S.L. ·
كتبها R.Eugenio · حرّرها فريق Solo2.