Что такое SHA-256 на самом деле

Математический отпечаток, который помещается в шестьдесят четыре символа и полностью меняется, если сдвинуть хотя бы одну запятую в оригинальном тексте. Почему мы называем его цифровой сургучной печатью.

Проще говоря: Представьте себе машину, которая считывает любой текст и выдает последовательность из 64 символов. Если текст на входе идентичен, последовательность на выходе будет идентичной. Если вы переставите хоть одну запятую, последовательность будет совершенно другой. Эта последовательность — цифровой сургуч.

Простая идея за техническим названием

Представьте, что существует машина с одной прорезью и одним экраном. В прорезь вы вводите текст: слово, фразу, целый роман. На экране мгновением позже появляется последовательность ровно из шестидесяти четырех символов. Эту последовательность профессиональные читатели называют *хешем* или *криптографическим дайджестом*; для обычного читателя мы пока можем назвать ее математическим отпечатком текста, как отпечаток пальца у человека.

Если ввести один и тот же текст дважды, машина оба раза покажет одинаковый отпечаток. Если ввести слегка измененный текст — сдвинута одна запятая, заглавная буква изменена на строчную, — машина покажет отпечаток, совершенно отличный от первого. Не похожий: другой. Эти два свойства вместе — детерминированность и чувствительность — и есть простая идея. Все остальное в SHA-256 — это механизмы, которые заставляют их хорошо работать.

С самого начала стоит сказать, чего машина не делает. Она не шифрует текст. Она не прячет его. Она его не сохраняет. Машина смотрит на текст, вычисляет отпечаток и забывает текст. Отпечаток не позволяет восстановить породивший его текст; он позволяет лишь, при наличии текста-кандидата, проверить, совпадает он с оригиналом или нет. Поэтому мы говорим, что это дайджест *одностороннего действия*: туда можно, обратно — нет.

Хеш — это не то же самое, что шифрование

Эта путаница встречается часто, и ее стоит прояснить: шифрование и хеширование — разные операции. Шифрование заключается в преобразовании текста таким образом, чтобы только владелец ключа мог вернуть его к первоначальному виду. Хеширование заключается в создании отпечатка текста, из которого исходный текст не может быть восстановлен никогда, ни с ключом, ни без него. Первое обратимо по задумке; второе необратимо по задумке.

Практическое следствие имеет значение. Когда приложение говорит «мы храним ваш пароль зашифрованным», это значит, что кто-то имеет ключ для его расшифровки — как минимум, само приложение. Когда приложение говорит «мы храним ваш пароль хешированным», само приложение не может прочитать исходный пароль, даже если захочет; оно может только проверить, дает ли введенный вами пароль тот же отпечаток. Вторая модель, при правильной реализации, гораздо предпочтительнее первой для хранения паролей. Позже мы увидим, почему «при правильной реализации» требует чего-то большего, чем просто SHA-256.

Четыре свойства, делающие криптографический хеш полезным

Хеш-функция, заслуживающая прилагательного *криптографическая*, обладает четырьмя свойствами:

1. **Детерминированность.** Одинаковый ввод всегда дает одинаковый отпечаток.

2. **Лавинный эффект.** Небольшое изменение во вводе дает совершенно другой отпечаток, без видимого сходства с предыдущим.
3. **Необратимость.** Имея отпечаток, вычислительно невозможно найти породивший его текст.
4. **Стойкость к коллизиям.** Вычислительно невозможно найти два разных текста, которые дадут одинаковый отпечаток.

«Вычислительно невозможно» не означает «математически невозможно». Это означает, что затраты времени, энергии и денег на достижение цели на порядки превышают сумму всех разумно доступных вычислительных мощностей. Для SHA-256 этот предел измеряется тысячами миллиардов лет даже при самых оптимистичных подходах со специализированным оборудованием. Что для практических целей читателя означает то же самое, что «невозможно».

В частности, SHA-256

Название говорит само за себя. SHA — это аббревиатура от *Secure Hash Algorithm*: алгоритм безопасного хеширования. Число 256 указывает размер отпечатка в битах: двести пятьдесят шесть бит, то есть тридцать два байта, которые в шестнадцатеричном формате представляют собой те самые шестьдесят четыре символа, которые читатель уже узнает. Стандарт был опубликован американским институтом NIST, органом, который стандартизирует такие функции, в 2001 году как часть семейства SHA-2; текущая версия стандарта, FIPS 180-4, от 2015 года.

Для тех, кто еще не знает, что такое биты и байты:

```
1 бит    →  0 или 1      (переключатель: включен или выключен)
1 байт   →  8 бит        (256 возможных комбинаций)
32 байта →  256 бит      (отпечаток SHA-256)
```

Число 256 в конце названия указывает размер отпечатка в битах. В шестнадцатеричной системе — системе счисления с шестнадцатью символами вместо десяти — эти 256 бит умещаются ровно в 64 символа. Это те самые 64 символа, которые вы видите внизу каждого выпуска Cuadernos Lacre.

Размеры заслуживают минуты внимания. Двести пятьдесят шесть бит дают два в степени двести пятьдесят шесть различных значений: число с семьюдесятью восемью десятичными цифрами, на несколько порядков превышающее оценочное количество атомов в наблюдаемой Вселенной. Каждый текст в мире — каждая книга, каждое электронное письмо, каждое сообщение — попадает на одно из этих значений. Вероятность того, что два разных текста совпадут случайно, в практических целях неотличима от нуля.

Как это выглядит в коде

В Zig, языке, на котором мы пишем компоненты, поддерживающие Solo2, вычисление печати SHA-256 текста выглядит так:

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

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

Мы только что попросили стандартную библиотеку Zig вычислить SHA-256 текста в кавычках. После вызова переменная *resumen* содержит тридцать два байта, составляющие печать в сыром виде; при выводе на экран в шестнадцатеричном формате это шестьдесят четыре символа, которые появляются внизу этой статьи. Если бы мы изменили *Cuadernos Lacre* на *Cuadernos lacre* — на одну заглавную букву меньше, — вся печать изменилась бы. Это и есть, в пяти строках, центральное свойство, на котором держится все остальное. Для тех, кто хочет увидеть, как это работает внутри, в конце статьи мы приводим читаемую версию алгоритма с пошаговыми комментариями.

Почему мы называем это сургучной печатью

В европейской переписке с пятнадцатого по девятнадцатый век письмо запечатывали сургучом. Капля расплавленного воска, прижатая сверху печать, и письмо было отмечено неповторимым образом. Это не защищало

содержимое от решительного шпиона — бумагу можно было прочитать на просвет, сургуч можно было сломать, — но это делало вмешательство очевидным. Любое изменение закрытия было заметно получателю еще до вскрытия бумаги. Сургучная печать не предотвращала ущерб; она заявляла о нем.

SHA-256 тела каждого выпуска Cuadernos выполняет ту же функцию в цифровой версии. Если изменится хоть одно слово статьи между моментом ее публикации и моментом, когда вы ее читаете, шестнадцатеричная печать внизу текста больше не будет совпадать с SHA-256 текста перед вами. Любой читатель с пятью строками кода мог бы это проверить. Публикация не может переписать свою историю так, чтобы печать этого не выдала. Она не защищает от повреждений; она делает их проверяемыми.

Чем хеш не является

Иногда от SHA-256 требуют четырех функций, которые ему не свойственны:

1. **Шифровать.** Хеш резюмирует; он не скрывает. Если вы хотите, чтобы текст невозможно было прочитать, вам нужно его зашифровать, а не хешировать.
2. **Устанавливать авторство.** Хеш не говорит, кто написал текст, только какой текст был хеширован. Чтобы связать авторство, поверх хеша нужна криптографическая подпись, а не просто сам хеш.
3. **Хранить пароли.** Здесь кроется ловушка, которую стоит понять. Алгоритм SHA-256 задуман как очень быстрый — что хорошо для многих задач, но плохо для этой. Злоумышленник со специализированным оборудованием может проверять миллиарды паролей в секунду против хеша SHA-256, пока не найдет ваш. Для хранения паролей следует использовать намеренно медленные функции деривации ключа, такие как Argon2, scrypt или bcrypt, в комбинации с *солью* (уникальным случайным значением для каждого пользователя, которое предотвращает совпадение хешей у двух людей с одинаковым паролем).
4. **Служить идентификатором автора.** Это не так. Хеш идентифицирует контент. Если два человека хешируют слово *привет* с помощью SHA-256, они оба получат одинаковый дайджест — и это главное свойство, а не недостаток: если бы дайджесты были разными, мы не могли бы проверить соответствие между опубликованным и полученным.

Где SHA-256 встречается в вашей повседневной жизни

Даже если вы этого не видите, SHA-256 поддерживает большую часть того, чем вы ежедневно пользуетесь в интернете. Блокчейн Bitcoin строится путем связывания SHA-256 каждого блока со следующим; изменение прошлого блока заставляет пересчитывать всю последующую цепочку. Git, система контроля версий кода, используемая половиной мира, идентифицирует каждый коммит по SHA-256 (в новых версиях) или по его предшественнику SHA-1 (в старых) от его полного содержимого. HTTPS-сертификаты, подтверждающие личность веб-сайта при входе, имеют связанный с ними отпечаток SHA-256. Загрузки программного обеспечения часто сопровождаются SHA-256, публикуемым разработчиком, чтобы вы могли убедиться, что файл не был изменен по пути. И, как мы уже говорили, внизу каждого выпуска Cuadernos Lacre.

Для профессионального читателя

Четыре оперативных напоминания для тех, кто принимает решения или проводит аудит систем:

1. Хеш — это не шифрование. Если провайдер путает эти два термина в своей технической документации, стоит спросить, что именно он имеет в виду.
2. Для хранения паролей никогда не следует использовать SHA-256 в чистом виде. SHA-256 слишком быстр для этой задачи (см. пункт 3 в *Чем хеш не является*). Современным стандартом является **Argon2id**: медленный по дизайну, настраиваемый в зависимости от мощности сервера, в сочетании с разной случайной *солью* для каждого пользователя.
3. Для контроля целостности документов — контрактов, досье, архивов — SHA-256 остается эталонным стандартом. Именно он используется для квалифицированных электронных меток времени в ЕС.
4. Для долгосрочного хранения (десятилетия) рекомендуется также вычислять и сохранять SHA-3 или SHA-512 вместе с SHA-256; криптографическая осмотрительность рекомендует не полагаться на одну функцию для вековых архивов.

Технически эта итерационная структура, в которой промежуточное состояние сохраняется между входными блоками, известна как конструкция **Меркла — Дамгора** (Merkle-Damgård) — модель, на которой основаны SHA-1, SHA-2 (включая SHA-256) и многие другие классические хеш-функции. SHA-3, напротив, отказывается от конструкции Меркла — Дамгора в пользу иной архитектуры, называемой *губкой*.

Как работает SHA-256 шаг за шагом простыми словами

Представьте, что вы собрали самую сложную в мире цепочку из домино: тысячи костяшек, десятки разветвлений, механические мостики и пандусы через всю комнату, тщательно расставленные деталь за деталью.

Стоит подтолкнуть первую костяшку, и вся цепь обрушится в строго определенной и повторяемой последовательности. Та же сборка, тот же первый толчок → идентичный финальный рисунок упавших костяшек, раз за разом.

И вот что интересно: сдвиньте **всего одну костяшку** на полсантиметра в сторону перед началом и снова подтолкните. Пандус, который должен был сработать, останется неподвижным, мостик не упадет, сработает другое разветвление. Финальный узор костяшек на полу будет совершенно не похож на первый.

SHA-256 — это математический аналог такой цепочки. Текст, который вы пишете, — это начальное положение костяшек. Алгоритм — это толчок, запускающий каскад. А конечный результат — то, что мы называем *хешем*, — это стоп-кадр пола, когда всё остановилось. Измените хоть одну запятую в исходном тексте, и кадр будет радикально другим. Вот так просто и так решительно.

Шаг 1. Перевод текста в бинарные фишки. Компьютеры не понимают букв; сначала они переводят их в числа (ASCII), а числа — в двоичный код (единицы и нули). Каждая буква превращается в 8 белых или черных фишек: *A* — это `01000001`, *B* — `01000010`, пробел — `00100000`. Весь ваш текст — слово, контракт, роман — превращается в длинный ряд белых и черных фишек.

Шаг 2. Заполнение до стандартного размера. Механизм обрабатывает ряд *блоками* ровно по 512 фишек. Если ваше сообщение не кратно 512, сразу после текста добавляется маркерная фишка (со значением `10000000`), а затем нули до конца блока. Последние 64 позиции каждого блока резервируются для записи исходной длины текста. Так механизм всегда знает, где закончилось реальное содержимое и где началась «набивка».

Шаг 3. Расстановка восьми мастер-фишек. Перед началом мы выкладываем на стол **восемь мастер-фишек** в строго определенном начальном положении. Эти восемь фишек — не секрет: их начальные значения зафиксированы общедоступным математическим правилом (квадратные корни первых восьми простых чисел — 2, 3, 5, 7, 11, 13, 17, 19 — и первые биты дробной части каждого корня). Любой человек в любом уголке планеты начинает с теми же восьми мастер-фишками в том же положении. Их судьба — быть сдвинутыми и измененными лавиной.

Шаг 4. Великая лавина: шестьдесят четыре раунда толчков. Здесь начинается самое интересное. Первый блок из 512 фишек вашего текста сталкивается с восемью мастер-фишками. Но они не падают разом: механизм выполняет **шестьдесят четыре последовательных раунда**. В каждом раунде с фишками производятся три операции:

- **Карусель** (ротация). Фишки движутся по кругу: те, что справа, переходят налево. Фишки не теряются и не добавляются — они просто переупорядочиваются, совершая полный круг на карусели. Это дешевый и обратимый способ перераспределить информацию.
- **Логическая воронка** (XOR). Фишки проходят через воронку, которая сравнивает их попарно: если обе одного цвета, на выходе белая; если разные — черная. Это простейшая операция двоичной логики, но в сочетании с вращением карусели она становится мощнейшим инструментом перемешивания информации без её потери.
- **Переполнение** (модульное сложение). Результат складывается с *фишкой постоянного толчка*, взятой из открытого списка шестидесяти четырех констант (кубические корни первых шестидесяти четырех простых чисел). Если при сложении возникают лишние фишки, не помещающиеся в отведенное пространство из 32 фишек, они отбрасываются. На столе есть место только для 32 фишек, и ни одной больше.

По окончании шестьдесят четвертого раунда каждая фишка из блока вашего текста повлияла на положение восьми мастер-фишек. Энергия толчка прошла через всю цепь.

Шаг 5. Добавление следующего блока (без сброса). Если ваш текст длинный и остался еще один блок из 512 фишек, **механизм не перезапускается**. Восемь мастер-фишек остаются в том виде, в котором их оставила первая лавина, и на них обрушивается второй блок, запуская еще шестьдесят четыре раунда. Это как пристроить новую комнату с домино в конце той, что только что упала: беспорядок в первой полностью определяет то, как упадет вторая.

Шаг 6. Финальный кадр. Когда блоков больше не осталось, лавина останавливается. Мы смотрим на финальное положение восьми мастер-фишек. Переводим их конфигурацию в код из букв и цифр в шестнадцатеричной системе. Результат — строка ровно из шестидесяти четырех символов: это и есть ваша печать SHA-256.

Четыре свойства вытекают сами собой из того, как устроена эта цепь:

1. **Детерминизм.** Один и тот же текст всегда дает один и тот же финальный кадр на любом компьютере в мире. Ноль случайности, ноль сюрпризов.
2. **Эффект лавины.** Лишняя запятая, измененная заглавная буква, забытое ударение — и кадр становится совершенно неузнаваемым. Это та самая экстремальная чувствительность, о которой мы говорили в начале.
3. **В одну сторону.** Имея финальный кадр, невозможно восстановить исходный текст. Вращения, воронки и переполнения уничтожают всю информацию о том, *откуда взялся каждый бит*, сохраняя лишь то, *что получилось в итоге*.
4. **Устойчивость к коллизиям.** За двадцать пять лет публичного криптоанализа никому не удалось найти два разных текста, финальные кадры которых совпали бы. И сложность этой задачи выходит за рамки вычислительных возможностей любой вообразимой цивилизации.

Код в приложении ниже реализует именно эти шесть шагов на языке Zig. Теперь вы можете читать его, понимая значение каждой битовой операции, а не слепо принимая эти манипуляции.

Технический глоссарий

Для читателя, который хочет понять суть каждой операции. Можно смело пропустить: статья будет понятна и без этого.

ASCII и Unicode — как буквы становятся числами. Компьютеры не видят букв, они видят числа. Стандарт под названием **ASCII** (*American Standard Code for Information Interchange*, 1963 г.) присваивает каждому символу клавиатуры определенное число: *A* — 65, *B* — 66, *a* — 97, *0* — 48, пробел — 32, запятая — 44. Современные системы расширяют его с помощью **Unicode**, который присваивает число каждому символу каждого алфавита в мире: кириллицы, арабского, китайского, японского и даже эмодзи. Когда вы пишете символ или открываете текстовый файл, компьютер считывает фоновое число, а не экранную форму. SHA-256 работает с этими числами, обрабатывая любой текст как длинную последовательность цифр. Поэтому он может запечатать статью на испанском, стихотворение на японском и бинарный файл одним и тем же алгоритмом.
XOR — побитовое сравнение. XOR (произносится *«эксор»*, от англ. *exclusive or*, «исключающее ИЛИ») — одна из простейших операций, которую компьютер может выполнить с двумя двоичными числами. Она сравнивает два бита позиция за позицией и возвращает: **1**, если ровно один из двух равен 1 (один, но не оба), и **0**, если оба одинаковы (оба 0 или оба 1). Пример: XOR для `1010` и `1100` — это `0110`. Она обладает замечательным свойством: она обратима — если выполнить XOR дважды с одним и тем же ключом, вы вернетесь к оригиналу. Вот почему это рабочая лошадка криптографии: она смешивает биты без потери информации, но результат ничего не говорит о входных данных, если вы не знаете одну из них.
Шестнадцатеричная система — счет по основанию 16. Почти все числа в повседневной жизни используют десять цифр (0–9). В шестнадцатеричной системе используется шестнадцать: привычные 0–9 плюс шесть букв для следующих значений: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Почему шестнадцать? Потому что компьютеры думают группами по четыре бита, а четыре бита могут представлять ровно шестнадцать различных значений — таким образом, один шестнадцатеричный символ четко соответствует четырем битам. Отпечаток SHA-256 имеет размер 256 бит, что составляет ровно **64 шестнадцатеричных символа**. Если бы мы записали его обычными десятичными числами, он занял бы около 78 цифр и был бы менее удобным. Этот выбор обусловлен эстетикой и компактностью; само число остается прежним.
Ротация бит — бинарная карусель. Представьте ряд из семи лампочек, одни из которых горят (1), а другие нет (0): `1 0 1 1 0 0 1`. Ротация вправо на одну позицию заключается в том, чтобы взять крайнюю правую лампочку, перенести её в крайнее левое положение и сдвинуть остальные на одно место вправо: `1 1 0 1 1 0 0`. Ни одна лампочка не теряется и не добавляется — они просто «танцуют» по кругу. SHA-256 использует ротацию бит сотни

раз при каждом вычислении; это дешевый и безошибочный способ перераспределения информации внутри состояния.

Константы *«nothing-up-my-sleeve»* — почему они происходят от простых чисел. Восемь мастер-фишек и шестьдесят четыре константы раунда SHA-256 не были выбраны случайно. Они получены из квадратных и кубических корней первых простых чисел. Почему? Потому что его создатели хотели получить константы *«без туза в рукаве»* (*«nothing-up-my-sleeve»*): значения, происхождение которых может проверить любой желающий. Если бы кто-то сказал вам: *«поверь мне: используй это случайное 32-битное число»*, вы бы резонно заподозрили скрытую уязвимость или бэкдор. Но любой человек с калькулятором может убедиться, что первые 32 бита квадратного корня из 2 — это `0x6a09e667`. Эти значения математически обоснованы, публичны и воспроизводимы: ни одна скрытая ловушка не сможет прокрасться в этот рецепт.

Приложение: SHA-256 в читаемом коде

Это приложение для читателя, желающего посмотреть на алгоритм изнутри. Это дидактическая реализация на Zig, следующая спецификации FIPS 180-4. Это не та версия, которую использует Solo2 — настоящая находится в `std.crypto.hash.sha2.Sha256` стандартной библиотеки Zig, оптимизированная и проверенная. Но алгоритм тот же самый: то, что вы здесь видите, шаг за шагом показывает, что происходит при выполнении этой функции из пяти символов.

```
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
}
```

Любое переписывание на другом языке, следующее той же структуре — начальные константы, расширение расписания, шестьдесят четыре раунда, аккумуляция, — даст тот же результат. Алгоритм не имеет секретов: его ценность в том, что перечисленные выше свойства продолжают сохраняться после двух десятилетий публичного криптоанализа тысячами глаз.

Если вы вернетесь в конец этой статьи, то увидите шестнадцатеричную печать из шестидесяти четырех символов. Это SHA-256 текста, который вы только что прочитали, на этом языке. Если бы мы перевели статью, печать была бы другой; если бы изменилось одно слово в испанской версии, испанская печать изменилась бы. Печать не защищает контент — для этого есть другие инструменты, — но она однозначно его идентифицирует. И этого, как бы скромно это ни звучало, достаточно, чтобы ни один этап издательской цепочки не мог незаметно изменить сказанное. Все остальное — шифрование, подпись, идентификация — строится поверх этой простой идеи.

От редакции: когда в этих Cuadernos упоминаются компании или продукты, это делается не для того, чтобы кого-то обвинить. Те, кто их создает, делают работу, которой пользуются и которую ценят миллионы людей. Мы указываем на структурную проблему — модель, а не бренд. Бренды появляются в качестве примера, потому что они узнаваемы для читателя.

Источники и дополнительная литература

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, август 2015 г. Официальная спецификация семейства SHA-2, включая SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, май 2011 г. Нормативная версия для разработчиков.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Главы 5 и 6 посвящены хеш-функциям и их легитимному и нелегитимному использованию.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Практический пример использования SHA-256 для связывания блоков в неизменяемую по определению структуру.
- Регламент (ЕС) 910/2014 (eIDAS) — основа для квалифицированных электронных меток времени. SHA-256 является эталонной функцией для квалифицированных электронных подписей и печатей, выдаваемых в ЕС.
- Эталонная реализация на Zig: `std.crypto.hash.sha2.Sha256` в официальном репозитории языка (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). Это оптимизированная и проверенная версия, которую на самом деле использует Solo2. Полезна для сравнения с дидактической реализацией в приложении.

← ПредыдущийSchrems II, пять лет спустяСледующий →Kill switch и институциональный захват

Недавние материалы

- Анализ · 18 мая 2026 г. Реальная vs мнимая конфиденциальность: вопросы, которые стоит себе задать
- Анализ · 18 мая 2026 г. Self-hosting как профессиональная практика

- Концепция · 18 мая 2026 г. 24 слова: что такое криптографическая идентичность

Возьмите эту статью с собой туда, где она вам понадобится.

↓ Markdown ↓ Простой текст ↓ PDF

Файл будет загружен на ваше устройство. Оттуда вы можете сохранить его, импортировать в Solo2 или поделиться им где угодно. Cuadernos не решает место назначения за вас.

Сургучная печать · SHA-256 e5e0dcd42d084c763382242b21c7f13ae94da9d80869f8c342acd94daeda7b2a

Cuadernos Lacre · Издание Menzuri Gestión S.L. ·
текст R.Eugenio · под редакцией команды Solo2.

Этот сайт не использует куки и не загружает сторонние ресурсы. Используется анонимный счетчик посещений (Umami, на нашем европейском сервере) и минимум JavaScript, необходимый для двух элементов управления в шапке: светлой или темной темы и выбора языка. Без трекеров, без профилирования, без передачи данных. Если вы хотите следить за нами: RSS.