개념 · 2026년 5월 19일

← 이전·다음 →

SHA-256이란 과연 무엇인가

64글자에 담기는 수학적 지문. 원문의 쉼표 하나만 바뀌어도 전체가 변합니다. 왜 이것을 디지털 인장(seal)이라고 부르는지 알아봅니다.

기술적인 이름 뒤에 숨겨진 단순한 아이디어

슬롯 하나와 화면 하나만 있는 기계가 있다고 상상해 보세요. 슬롯에 텍스트를 넣습니다. 단어 하나일 수도 있고, 문장일 수도 있고, 소설 전체일 수도 있습니다. 잠시 후 화면에는 정확히 64글자의 시퀀스가 나타납니다. 전문가들은 이를 '해시(hash)' 또는 '암호학적 요약'이라고 부르지만, 일반인들에게는 지문이 사람을 식별하듯 텍스트의 '수학적 지문'이라고 부를 수 있습니다.

같은 텍스트를 두 번 넣으면 기계는 두 번 다 똑같은 지문을 보여줍니다. 만약 텍스트를 아주 조금이라도 바꾸면(쉼표 하나를 옮기거나 대문자를 소문자로 바꾸면), 기계는 처음과 완전히 다른 지문을 보여줍니다. 비슷한 것이 아니라 완전히 다릅니다. 이 두 가지 성질(결정성과 민감성)이 바로 이 기술의 핵심입니다. SHA-256의 나머지 부분은 이 성질들이 잘 작동하도록 만드는 기계 장치일 뿐입니다.

먼저 이 기계가 '하지 않는 일'을 짚고 넘어가야 합니다. 이 기계는 텍스트를 암호화(숨김)하지 않습니다. 저장하지도 않습니다. 기계는 텍스트를 보고 지문을 계산한 뒤 텍스트를 잊어버립니다. 지문만 보고는 그것을 만든 원래 텍스트를 알아낼 수 없습니다. 오직 후보 텍스트가 주어졌을 때 그것이 원본과 일치하는지만 확인할 수 있습니다. 그래서 이를 '단방향' 요약이라고 부릅니다. 갈 수는 있지만 돌아올 수는 없습니다.

해시는 암호화와 다릅니다

자주 혼동되곤 하지만, 암호화와 해시는 서로 다른 작업입니다. 암호화는 열쇠를 가진 사람만이 원래대로 되돌릴 수 있도록 텍스트를 변형하는 것입니다. 해시는 열쇠가 있든 없든 원문을 절대 복구할 수 없는 지문을 생성하는 것입니다. 전자는 설계상 '가역적'이지만, 후자는 '불가역적'입니다.

이 차이는 실무에서 매우 중요합니다. 어떤 앱이 '비밀번호를 암호화해서 저장한다'고 한다면, 누군가(앱 자신 등)가 그것을 풀 수 있는 열쇠를 가지고 있다는 뜻입니다. 하지만 '비밀번호를 해시해서 저장한다'고 한다면, 앱 자신도 원래 비밀번호를 알 수 없습니다. 그저 당신이 입력한 비밀번호가 저장된 지문과 똑같은 지문을 만드는지 확인할 수 있을 뿐입니다. 제대로만 구현된다면 후자가 비밀번호 저장에 훨씬 안전합니다. 왜 '제대로' 구현하기 위해 SHA-256 이상의 장치가 필요한지는 뒤에서 설명하겠습니다.

암호학적 해시를 유용하게 만드는 4가지 성질

'암호학적'이라는 수식어를 붙일 만한 해시 함수는 다음 4가지 성질을 충족해야 합니다.

1. **결정성:** 같은 입력은 항상 같은 지문을 생성한다.
2. **쇄도 효과(Avalanche effect):** 입력의 아주 작은 변화가 이전과는 전혀 닮지 않은 완전히 다른 지문을 생성한다.
3. **역상 저항성:** 지문이 주어졌을 때 그것을 만든 원문을 찾아내는 것이 계산적으로 불가능하다.
4. **충돌 저항성:** 같은 지문을 생성하는 서로 다른 두 텍스트를 찾아내는 것이 계산적으로 불가능하다.

'계산적으로 불가능하다'는 말은 '수학적으로 절대 불가능하다'는 뜻이 아닙니다. 그것을 해내는 데 드는 시간, 에너지, 비용이 현재 인류가 동원할 수 있는 모든 계산 능력을 수만 배 초과한다는 뜻입니다. SHA-256의 경우, 특수 하드웨어를 사용한 가장 낙관적인 추정으로도 그 한계는 수천조 년 단위입니다. 즉, 독자 여러분의 입장에서는 '불가능하다'는 말과 같습니다.

구체적으로 보는 SHA-256

이름에 모든 것이 담겨 있습니다. SHA는 'Secure Hash Algorithm(안전한 해시 알고리즘)'의 약자입니다. 256이라는 숫자는 지문의 크기가 256비트임을 나타냅니다. 이는 32바이트이며, 16진수로 표현하면 우리가 흔히 보는 64글자가 됩니다. 이 표준은 2001년 미국 NIST에서 SHA-2 제품군의 일부로 발표했으며, 현재 통용되는 표준 버전은 2015년의 FIPS 180-4입니다.

비트와 바이트가 아직 생소하신 분들을 위해:

```
1 비트    →  0 또는 1 (스위치: 켜짐 또는 꺼짐)
1 바이트   →  8 비트 (256가지 조합 가능)
32 바이트  →  256 비트 (SHA-256 지문)
```

이름 끝의 256은 비트 수를 말합니다. 10 대신 16개의 기호를 사용하는 16진법으로는 이 256비트가 정확히 64글자에 들어갑니다. 이것이 각 Cuaderno 하단에 보이는 64글자의 정체입니다.

그 규모를 생각해보세요. 256비트로는 2의 256승 개의 서로 다른 값을 가질 수 있습니다. 이는 10진수로 78자리에 달하는 숫자로, 관측 가능한 우주의 원자 수 추정치보다 훨씬 큽니다. 세상의 모든 텍스트는 이 값들 중 하나에 떨어집니다. 서로 다른 두 텍스트가 우연히 일치할 확률은 실질적으로 0에 가깝습니다.

코드에서의 모습

Solo2를 지탱하는 언어인 Zig에서 텍스트의 SHA-256 인장을 계산하는 코드는 다음과 같습니다.

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

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

방금 Zig 표준 라이브러리에 따옴표 안의 텍스트에 대한 SHA-256 계산을 요청했습니다. 호출 후 *resumen* 변수에는 인장을 구성하는 32바이트 원시 데이터가 담깁니다. 이를 화면에 16진수로 출력하면 이 기사 하단에 있는 64글자가 됩니다. 만약 *Cuadernos Lacre*를 *Cuadernos lacre*(대문자 하나를 소문자로)로 바꾸면 인장 전체가 바뀝니다. 이 5줄의 코드가 나머지 모든 것을 지탱하는 핵심 성질입니다. 내부 작동 방식이 궁금하신 분들을 위해 기사 끝에 단계별 주석이 달린 알고리즘 버전을 포함했습니다.

왜 '인장(seal)'이라고 부르는가

15세기에서 19세기 유럽의 서신 왕래에서 밀랍 인장은 편지를 봉인했습니다. 녹은 밀랍 한 방울을 떨어뜨리고 그 위에 도장을 누르면 편지에는 복제 불가능한 표시가 남았습니다. 이것이 엿보기를 막아주지는 못했지만(종이를 빛에 비춰 보거나

밀랍을 깨뜨릴 수 있었음), 조작 여부는 확실히 드러내 주었습니다. 봉인이 조금이라도 훼손되면 수신자는 편지를 열기도 전에 알 수 있었습니다. 인장은 피해를 막는 것이 아니라, 피해 사실을 '선언'했습니다.

각 Cuaderno 본문의 SHA-256은 디지털 세계에서 똑같은 역할을 합니다. 기사가 발행된 순간부터 당신이 읽는 순간 사이에 단 한 단어라도 바뀐다면, 하단의 16진수 인장은 당신 앞에 있는 텍스트의 SHA-256과 일치하지 않게 됩니다. 코딩을 조금이라도 아는 독자라면 누구든 이를 확인할 수 있습니다. 발행인은 인장에 들키지 않고 역사를 다시 쓸 수 없습니다. 인장은 피해를 막는 것이 아니라, 피해를 '검증 가능'하게 만듭니다.

해시가 아닌 것

때때로 SHA-256에 기대하지만 실제로는 그 역할이 아닌 4가지 경우가 있습니다.

1. **암호화:** 해시는 요약일 뿐 숨기는 것이 아닙니다. 텍스트를 읽지 못하게 하려면 해시가 아니라 암호화가 필요합니다.
2. **저자 인증:** 해시는 누가 썼는지 말해주지 않고, 어떤 텍스트가 해시되었는지만 말해줍니다. 저자를 연결하려면 해시 자체가 아니라 해시 위에 암호학적 서명이 필요합니다.
3. **비밀번호 저장:** 여기에는 함정이 있습니다. SHA-256은 매우 빠르게 작동하도록 설계되었습니다. 이는 많은 경우 장점이지만, 이 경우에는 단점입니다. 공격자는 특수 하드웨어를 사용해 초당 수십억 개의 비밀번호를 대조하며 당신의 비밀번호를 찾아낼 수 있습니다. 비밀번호 저장에는 Argon2, scrypt, bcrypt처럼 의도적으로 느리게 만든 키 유도 함수를 솔트(salt, 사용자별 고유 랜덤 데이터)와 함께 사용해야 합니다.
4. **해시를 저자 식별자로 사용:** 그렇지 않습니다. 해시는 내용을 식별합니다. 두 사람이 똑같이 'hola'라는 단어를 SHA-256으로 해시하면 둘 다 똑같은 결과를 얻습니다. 이것은 결함이 아니라 핵심 성질입니다. 결과가 다르다면 발행된 내용과 수신된 내용의 일치 여부를 확인할 수 없을 것입니다.

일상 속의 SHA-256

보이지 않지만 SHA-256은 우리가 매일 사용하는 인터넷의 상당 부분을 지탱합니다. 비트코인 블록체인은 각 블록을 다음 블록에 SHA-256으로 연결하여 구축됩니다. 과거 블록을 수정하려면 이후의 모든 체인을 다시 계산해야 합니다. 전 세계 코드 버전 관리 시스템인 Git은 각 커밋을 내용 전체의 해시(최근 버전은 SHA-256, 구버전은 SHA-1)로 식별합니다. 웹사이트 접속 시 신원을 확인하는 HTTPS 인증서에도 SHA-256 지문이 포함됩니다. 소프트웨어 다운로드 시 파일이 중간에 변조되지 않았는지 확인할 수 있도록 개발자가 SHA-256을 공개하는 경우도 많습니다. 그리고 물론, 각 Cuaderno Lacre 하단에도 있습니다.

전문가 독자를 위해

시스템을 결정하거나 감사하는 분들을 위한 4가지 운영상 유의점:

1. 해시는 암호화가 아닙니다. 공급업체가 기술 문서에서 두 용어를 혼동한다면 정확히 무엇을 의미하는지 물어볼 필요가 있습니다.
2. 비밀번호 저장에 SHA-256만 단독으로 사용해서는 안 됩니다. SHA-256은 이 작업에 너무 빠릅니다. 현재 표준은 **Argon2id**입니다. 설계상 느리고, 서버 능력에 따라 설정 가능하며, 사용자별 고유 솔트와 결합됩니다.
3. 계약서, 문서, 파일의 무결성을 위해서는 SHA-256이 여전히 표준입니다. EU의 공인 타임스탬프 서비스에서도 이를 사용합니다.
4. 수십 년 단위의 장기 보존을 위해서는 SHA-256과 함께 SHA-3나 SHA-512도 계산해서 보관하는 것이 좋습니다. 세기를 넘기는 아카이브에서는 단일 함수에만 의존하지 않는 것이 암호학적으로 신중한 태도입니다.

기술적으로 이 반복 구조(중간 상태가 입력 블록 간에 보존되는 구조)는 **Merkle-Damgård**(머클-담가드) 구조로 알려져 있습니다. 이는 SHA-1, SHA-2(SHA-256 포함) 및 기타 여러 고전적인 해시 함수가 기반을 둔 패턴입니다. 반대로 SHA-3는 머클-담가드를 버리고 *스펀지*(sponge)라고 불리는 다른 아키텍처를 채택했습니다.

SHA-256 작동 원리: 단계별 안내 (쉬운 설명)

세상에서 가장 정교한 도미노 회로를 만들었다고 상상해 보세요. 수천 개의 말판, 수십 개의 갈림길, 기계식 다리, 방 전체를 가로지르는 경사로가 하나하나 정성스럽게 배치되어 있습니다.

첫 번째 도미노를 살짝 건드리면, 체인은 정확하고 반복 가능한 순서로 넘어집니다. 동일한 배치, 동일한 첫 충격 → 반복해서 수행해도 넘어지는 마지막 도미노 패턴은 동일합니다.

여기 흥미로운 점이 있습니다. 시작하기 전에 **단 하나의 도미노**를 옆으로 0.5cm 옮긴 다음 다시 건드려 보세요. 작동해야 할 경사로가 가만히 있고, 다리는 넘어지지 않으며, 다른 갈림길이 작동합니다. 바닥에 남은 마지막 도미노 패턴은 처음 것과 완전히 다릅니다.

수학적으로 SHA-256은 바로 이 회로입니다. 여러분이 작성하는 텍스트는 도미노의 초기 위치입니다. 알고리즘은 연쇄 반응을 일으키는 충격입니다. 그리고 최종 결과(우리가 *해시*라고 부르는 것)는 모든 것이 멈췄을 때 바닥을 찍은 정지 사진입니다. 원본 텍스트에서 콤마 하나만 바꿔도 그 사진은 완전히 달라집니다. 이렇게 단순하면서도 극적입니다.

1단계. 텍스트를 이진 도미노로 변환하기. 컴퓨터는 문자를 이해하지 못합니다. 먼저 숫자(ASCII)로 바꾸고, 그 숫자를 이진수(0과 1)로 바꿉니다. 각 문자는 8개의 흰색 또는 검은색 도미노로 바뀝니다. 예를 들어 *A*는 `01000001`, *B*는 `01000010`, 공백은 `00100000`입니다. 여러분의 전체 텍스트(단어, 계약서, 소설 등)는 흰색과 검은색 도미노의 긴 줄이 됩니다.

2단계. 표준 크기까지 채우기. 회로는 도미노 줄을 정확히 512개씩 *블록* 단위로 처리합니다. 메시지가 512의 배수에 도달하지 못하면 텍스트 바로 뒤에 표시 도미노(값이 `10000000`인 것)를 추가하고 블록이 완성될 때까지 0을 채웁니다. 각 블록의 마지막 64개 자리는 원본 텍스트의 길이를 기록하기 위해 예약됩니다. 이를 통해 회로는 항상 실제 내용이 어디서 끝났고 어디서부터 채우기가 시작되었는지 알 수 있습니다.

3단계. 8개의 마스터 도미노 배치하기. 시작하기 전에 테이블 위에 **8개의 마스터 도미노**를 정확한 초기 위치에 배치합니다. 이 8개는 비밀이 아닙니다. 초기값은 공개된 수학적 규칙(첫 8개 소수 2, 3, 5, 7, 11, 13, 17, 19의 제곱근과 각 제곱근 소수 부분의 첫 번째 비트)에 의해 고정되어 있습니다. 전 세계 누구나 똑같은 위치에 있는 똑같은 8개의 마스터 도미노로 시작합니다. 이들의 운명은 연쇄 반응에 의해 밀리고 변형되는 것입니다.

4단계. 거대한 연쇄 반응: 64라운드의 충격. 여기서부터 본격적인 쇼가 시작됩니다. 텍스트의 첫 512개 도미노 블록이 8개의 마스터 도미노와 충돌합니다. 하지만 한꺼번에 넘어지는 것이 아닙니다. 메커니즘은 **64번의 연속된 라운드**를 실행합니다. 각 라운드에서 도미노에 세 가지 연산을 수행합니다.

- **회전목마**(회전). 도미노들이 원을 그리며 이동합니다. 오른쪽에 있는 도미노가 왼쪽으로 넘어갑니다. 손실되거나 추가되는 도미노 없이 단순히 회전목마를 한 바퀴 돌려 재배열하는 것입니다. 이는 정보를 섞는 저렴하고 가역적인 방법입니다.
- **논리 깔때기**(XOR). 도미노들이 두 개씩 비교하는 깔때기를 통과합니다. 두 개가 같은 색이면 흰색이 나오고, 다른 색이면 검은색이 나옵니다. 이진 논리에서 가장 간단한 연산이지만 회전목마의 회전과 결합하면 정보를 잃지 않고 섞는 데 매우 강력한 힘을 발휘합니다.
- **오버플로**(모듈로 덧셈). 결과값은 공개된 64개 상수 목록(첫 64개 소수의 세제곱근)에서 가져온 *일정한 충격 도미노*와 더해집니다. 만약 덧셈 결과가 예정된 32개 도미노 공간을 넘어서는 추가 도미노를 생성하면, 그 넘치는 부분은 버려집니다. 테이블에는 32개의 도미노 공간만 있으며 더 이상은 들어갈 수 없습니다.

64라운드가 끝날 때쯤이면 텍스트 블록의 각 도미노가 8개의 마스터 도미노 위치에 영향을 미쳤습니다. 충격의 에너지가 회로 전체를 훑고 지나갔습니다.

5단계. 다음 블록 추가하기(초기화 없이). 만약 텍스트가 길어서 처리해야 할 또 다른 512개 블록이 남아 있다면, **회로는 초기화되지 않습니다**. 8개의 마스터 도미노는 첫 번째 연쇄 반응이 끝난 상태 그대로 남고, 두 번째 블록이 그들을 향해 발사되어 또 다른 64라운드가 시작됩니다. 이는 방금 넘어진 도미노 방 끝에 새로운 방을 추가하는 것과 같습니다. 첫 번째 방의 무질서가 두 번째 방이 어떻게 넘어질지를 완전히 결정합니다.

6단계. 최종 사진 찍기. 더 이상 처리할 블록이 없으면 연쇄 반응이 멈춥니다. 8개의 마스터 도미노가 최종적으로 어떤 위치에 남았는지 확인합니다. 그 구성을 16진수 코드(문자와 숫자 조합)로 변환합니다. 결과는 정확히 64자리의 문자열입니다. 이것이 바로 여러분의 SHA-256 인장입니다.

회로가 구성된 방식으로부터 네 가지 속성이 자연스럽게 도출됩니다.

1. **결정성.** 전 세계 어떤 컴퓨터에서도 동일한 텍스트는 항상 동일한 최종 사진을 만듭니다. 무작위성 제로, 예외 제로입니다.
2. **쇄도 효과(Avalanche effect).** 콤마 하나 추가, 대문자 변경, 악센트 기호 생략만으로도 사진은 완전히 알아볼 수 없게 바뀝니다. 이것이 바로 처음에 설명한 극한의 민감성입니다.
3. **단방향성.** 최종 사진으로부터 원본 텍스트를 재구성할 수 없습니다. 회전, 깔때기, 오버플로는 *각 비트가 어디서 왔는지*에 대한 방향성 정보를 모두 파괴하고 *총합이 무엇인지*만 보존합니다.
4. **충돌 저항성.** 25년간의 공개 암호 분석 과정에서 최종 사진이 일치하는 두 개의 서로 다른 텍스트를 찾아낸 사람은 아무도 없습니다. 그리고 이를 찾아내는 난이도는 우리가 상상할 수 있는 어떤 문명의 계산 능력도 넘어섭니다.

이어지는 코드 부록은 Zig 언어로 이 6단계를 정확하게 구현한 것입니다. 이제 각 비트 연산이 무엇을 의미하는지 알고서, 맹목적으로 받아들이는 대신 내용을 이해하며 읽을 수 있을 것입니다.

기술 용어집

각 연산이 무엇을 하는지 이해하고 싶은 독자를 위해 준비했습니다. 건너뛰어도 기사 내용을 이해하는 데 지장이 없습니다.

ASCII와 유니코드 — 글자가 숫자가 되는 법. 컴퓨터는 글자를 보지 않습니다. 숫자가 봅니다. 1963년에 만들어진 **ASCII**(American Standard Code for Information Interchange) 표준은 키보드의 각 문자에 특정 숫자를 부여합니다. 예를 들어 *A*는 65, *B*는 66, *a*는 97, *0*은 48, 공백은 32, 콤마는 44입니다. 현대 시스템은 이를 **유니코드**로 확장하여 키릴 문자, 아랍어, 중국어, 한국어, 일본어, 심지어 이모지까지 전 세계 모든 알파벳의 문자에 숫자를 부여합니다. 여러분이 문자를 쓰거나 텍스트 파일을 열 때 컴퓨터는 화면상의 형태가 아니라 그 이면의 숫자를 읽습니다. SHA-256은 이러한 숫자를 바탕으로 작동하므로 스페인어 기사, 일본어 시, 바이나리 파일을 모두 동일한 알고리즘으로 봉인할 수 있습니다.
XOR — 비트 단위 비교기. XOR(익스클루시브 오어, 배타적 논리합)은 컴퓨터가 두 이진수 사이에서 수행할 수 있는 가장 간단한 연산 중 하나입니다. 두 비트를 자리별로 비교하여, **1**(두 비트 중 정확히 하나만 1인 경우), 또는 **0**(두 비트가 같은 경우, 즉 둘 다 0이거나 둘 다 1인 경우)을 반환합니다. 예: `1010`과 `1100`의 XOR은 `0110`입니다. 주목할 만한 특성은 가역적이라는 점입니다. 동일한 키로 XOR을 두 번 하면 원래대로 돌아옵니다. 이것이 암호학의 핵심 도구인 이유입니다. 정보를 잃지 않고 비트를 섞지만, 입력값 중 하나를 모르면 결과로부터 아무것도 알아낼 수 없습니다.
16진수 — 16을 밑으로 하여 세기. 일상의 거의 모든 숫자는 10개의 숫자(0-9)를 사용합니다. 16진수는 16개를 사용합니다. 일반적인 0-9에 다음 값을 나타내는 6개의 글자를 더합니다: `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. 왜 16일까요? 컴퓨터는 4비트 그룹으로 생각하는데, 4비트는 정확히 16개의 서로 다른 값을 나타낼 수 있기 때문입니다. 즉, 16진수 한 자리는 4비트에 깔끔하게 대응됩니다. SHA-256 지문은 256비트이며, 이는 정확히 **64자리의 16진수**가 됩니다. 만약 이를 일

반 10진수로 쓴다면 약 78자리가 되어 훨씬 불편할 것입니다. 16진수를 선택한 것은 미학적이고 압축적이기 때문이며, 이면의 숫자는 동일합니다.

비트 회전 — 이진 회전목마. 7개의 전구가 일렬로 늘어서 있고 어떤 것은 켜져(1) 있고 어떤 것은 꺼져(0) 있다고 상상해 보세요: `1 0 1 1 0 0 1`. 오른쪽으로 한 칸 회전시키는 것은 맨 오른쪽 전구를 빼서 맨 왼쪽 끝으로 옮기고 나머지를 오른쪽으로 한 칸씩 밀어내는 것입니다: `1 1 0 1 1 0 0`. 전구가 사라지거나 추가되지 않고 단순히 원을 그리며 춤을 추는 것입니다. SHA-256은 각 계산에서 비트 회전을 수백 번 사용합니다. 이는 상태 내에서 정보를 손실 없이 재분배하는 저렴한 방법입니다.

'Nothing-up-my-sleeve(숨긴 것 없음)' 상수 — 소수에서 유래한 이유. SHA-256의 8개 마스터 도미노와 64개 라운드 상수는 무작위로 선택된 것이 아닙니다. 첫 번째 소수들의 제곱근과 세제곱근에서 유래했습니다. 왜일까요? 설계자들이 누구나 그 기원을 검증할 수 있는 *'숨긴 것 없는'* 값을 원했기 때문입니다. 만약 누군가 *"나를 믿고 이 32비트 난수를 쓰세요"*라고 한다면, 여러분은 합리적으로 숨겨진 취약점이나 백도어를 의심할 것입니다. 하지만 계산기가 있는 사람이라면 누구나 2의 제곱근의 첫 32비트가 `0x6a09e667`임을 확인할 수 있습니다. 이 값들은 수학적이고 공개적이며 재현 가능하므로, 레시피에 어떤 함정도 숨어들 수 없습니다.

부록: 읽기 쉬운 코드로 보는 SHA-256

이 부록은 알고리즘 내부를 들여다보고 싶은 독자들을 위한 것입니다. FIPS 180-4 명세를 따르는 Zig 언어의 교육용 구현체입니다. Solo2가 실제로 사용하는 것은 Zig 표준 라이브러리의 `std.crypto.hash.sha2.Sha256`으로, 최적화와 감사가 완료된 버전이지만 알고리즘 자체는 동일합니다. 여기서 보시는 것은 5글자의 호출이 실행될 때 일어나는 단계별 과정입니다.

```
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
}
```

초기 상수, 스케줄 확장, 64번의 라운드, 누적 등 똑같은 구조를 따르는 다른 언어의 구현체도 똑같은 결과를 냅니다. 알고리즘에는 비밀이 없습니다. 그 가치는 수만 명의 눈이 지켜보는 공개된 암호 해독 분석 속에서도 20년 넘게 앞서 언급한 성질들이 유지되고 있다는 데 있습니다.

이 기사 하단으로 돌아가면 64글자의 16진수 인장을 볼 수 있습니다. 그것은 당신이 방금 읽은 이 언어로 된 텍스트의 SHA-256입니다. 기사를 번역하면 인장이 달라질 것이고, 스페인어 원문의 한 단어라도 바뀐다면 스페인어 인장도 바뀔 것입니다. 인장은 내용을 보호하지는 않지만(그것은 다른 도구들의 역할입니다), 내용을 고유하게 식별합니다. 그리고 그것만으로도 편집 체인의 그 어떤 단계에서도 말한 내용을 흔적 없이 고칠 수 없게 만듭니다. 암호화, 서명, 식별과 같은 나머지 모든 것은 이 단순한 아이디어 위에 세워집니다.

참고 문헌 및 관련 자료

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, 2015년 8월. SHA-256을 포함한 SHA-2 제품군의 공식 명세.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, 2011년 5월. 구현자를 위한 규범적 버전.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). 5장과 6장에서 해시 함수와 정당한 사용법 및 오용 사례를 다룹니다.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). 구조적으로 불변인 체인에서 블록을 연결하기 위해 SHA-256을 사용한 실례.
- 유럽 연합 규정 910/2014 (eIDAS) — 공인 타임스탬프 프레임워크. SHA-256은 EU에서 발행되는 공인 전자 서명 및 인장의 참조 함수입니다.
- Zig 참조 구현체: 공식 리포지토리의 `std.crypto.hash.sha2.Sha256` (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). Solo2가 실제로 사용하는 최적화 및 감사된 버전입니다. 부록의 교육용 구현체와 대조해 보기 좋습니다.

← 이전CUADERNOS_LIST_SCHREMS_TITLE다음 →CUADERNOS_LIST_KILLSWITCH_TITLE

최근 읽은 글

- CUADERNOS_LIST_PREGUNTAS_TITLE
- CUADERNOS_LIST_SELFHOST_TITLE
- CUADERNOS_LIST_IDENTIDAD_TITLE

이 기사를 다운로드하여 필요한 곳에서 활용하십시오.

↓ 마크다운 ↓ 텍스트 형식 ↓ PDF

파일이 기기에 다운로드됩니다. 해당 위치에서 저장하거나 Solo2로 가져오거나 원하는 곳에 공유할 수 있습니다. Cuadernos는 전송 대상을 결정하지 않습니다.

봉인 · SHA-256 57dfe27f26edd210e6cc3f506fa98234b9cdab9a0258fbbf111a555d0355b6fd

Cuadernos Lacre · Menzuri Gestión S.L.의 간행물 ·
저자: R.Eugenio · Solo2 팀 편집