# ¿Cómo se construye una función de hash?

Los pasos de la construcción:

- Se define una función de hash para mensajes de largo fijo, la cual es llamada función de compresión
- Se define un método que permite utilizar de manera iterativa la función de compresión para construir una función de hash para mensajes de largo arbitrario

# Una función de compresión

Queremos construir una función de hash de largo fijo:

$$h : \{0,1\}^{2n} \to \{0,1\}^n$$

Para construir esta función suponemos que tenemos un esquema criptográfico $(Gen, Enc, Dec)$ sobre los espacios $\mathcal{M} = \mathcal{K} = \mathcal{C} = \{0,1\}^n$

- Tenemos que $Enc : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$

# Un primer intento

Dados $u, v \in \{0,1\}^n$, defina $h(u\|v) = Enc(u,v)$

- Recuerde que $u\|v$ es la concatenación de $u$ con $v$

Este es nuestro primer intento para definir una función de hash $h : \{0,1\}^{2n} \to \{0,1\}^n$

- Para ver si esta es una buena alternativa, vamos a considerar primero el caso $n = 128$ y AES

# AES en la práctica

```python
from Crypto.Cipher import AES

if __name__ == "__main__":
    k = b'Fgh64%djfg/fydhj'
    alg = AES.new(k, AES.MODE_ECB)

    m = b'Texto de prueba.'
    c = alg.encrypt(m)
    r = alg.decrypt(c)

    print(m)
    print(c)
    print(r)
```

Electronic codebook (ECB)

b'Texto de prueba.'

b'\x0fK\x8ec+\xaa\xeb\xcbE\x04\xf5\xf9\xdf(\xa5\x11'

b'Texto de prueba.'

# Nuestra primera función de compresión

```
def compresion(m: str) -> str:
    """
    Argumentos :
        m: str - mensaje
    Retorna :
        str: hash del mensaje usando AES
    """

    l = m[0:16]
    r = m[16:32]

    h = AES.new(l, AES.MODE_ECB)
    return h.encrypt(r)
```

# Los resultados del intento

```
if __name__ == "__main__":

    m1 = b'Texto de prueba para compresion.'
    m2 = b'TeXto de prueba para compresion.'
    m3 = b'Texto de prueba para compresion?'

    h1 = compresion(m1)
    h2 = compresion(m2)
    h3 = compresion(m3)

    print(h1.hex())
    print(h2.hex())
    print(h3.hex())
```

a4eb875d9a9e589d7dfaec66bb710441

98b1b377d5915e6a80a4db533fa416c7

bbb4aafcb004f5a4a5ede95da4f84395

# Pero tenemos un problema

Existe un algoritmo eficiente, en términos del parámetro de seguridad $1^n$, para construir preimagenes

- No solo para el caso de AES, sino que también para cualquier esquema criptográfico $(Gen, Enc, Dec)$ sobre los espacios $\mathcal{M} = \mathcal{K} = \mathcal{C} = \{0,1\}^n$

# La relación entre $Enc$ y $Dec$

Fije una llave $k \in \{0,1\}^n$, y defina las siguientes funciones de $\{0,1\}^n$ en $\{0,1\}^n$:

$$f(x) = Enc(k, x)$$
$$g(x) = Dec(k, x)$$

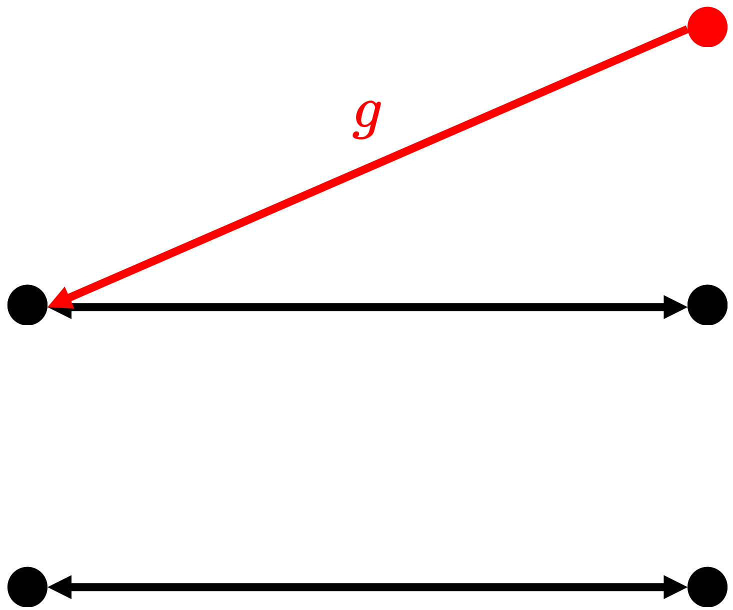
Tenemos que:

$$\forall x \in \{0,1\}^n : g(f(x)) = x$$

¿También tenemos que $f(g(x)) = x$?

# Cuidado con la respuesta

$g(f(x)) = x$




$f(g(x)) = x$

# Pero $\mathcal{M} = \mathcal{C} = \{0,1\}^n$

Tenemos que:

$$\forall x \in \{0,1\}^n : f(g(x)) = x$$

Concluimos que:

$$\forall k \in \mathcal{K}\, \forall m \in \mathcal{M} : Enc(k, Dec(k, m)) = m$$

# Ahora podemos demostrar que el primer intento no es resistente a preimagen

Dados $u, v \in \{0,1\}^n$, sea $x = h(u \| v) = Enc(u, v)$

Considere $u' \in \{0,1\}^n$ arbitrario, y defina $v' = Dec(u', x)$

Tenemos que: $h(u' \| v') = Enc(u', v') = Enc(u', Dec(u', x)) = x$

# En código ...

```python
if __name__ == "__main__":
    m = b'Texto de prueba para compresion.'
    h = compresion(m)

    k = b'1111111111111111'
    alg = AES.new(k, AES.MODE_ECB)
    c = alg.decrypt(h)
    p = k + c

    print(m.hex())
    print(h.hex())
    print(p.hex())
    print(compresion(p).hex())
```

546578746f20646520707275656261207061726120636f6d70726573696f6e2e

a4eb875d9a9e589d7dfaec66bb710441

3131313131313131313131313131313117894bb10a97cd21175266d33b1c25b4

a4eb875d9a9e589d7dfaec66bb710441

# Un segundo intento: la construcción de Davies-Meyer

$$h(u \| v) = Enc(u, v) \oplus v$$

Recuerde que el símbolo $\oplus$ representa al XOR, o suma en módulo 2, dado que estamos operando con bits

# La construcción de Davies–Meyer: formalización

Suponemos dado un esquema criptográfico $(Gen, Enc, Dec)$ sobre $\mathcal{M} = \mathcal{K} = \mathcal{C} = \{0,1\}^*$

Definimos una función de hash $(Gen', h')$ de largo fijo:

- $Gen'(1^n) = n$ para un parámetro de seguridad $1^n$
- $(h')^n : \{0,1\}^{2n} \to \{0,1\}^n$ tal que para cada $u, v \in \{0,1\}^n$ :

$$(h')^n(u \| v) = Enc(u, v) \oplus v$$

# La propiedad fundamental de la construcción de Davies–Meyer

Si $(Gen, Enc, Dec)$ es un esquema criptográfico *ideal,* entonces $(Gen', h')$ es resistente a colisiones

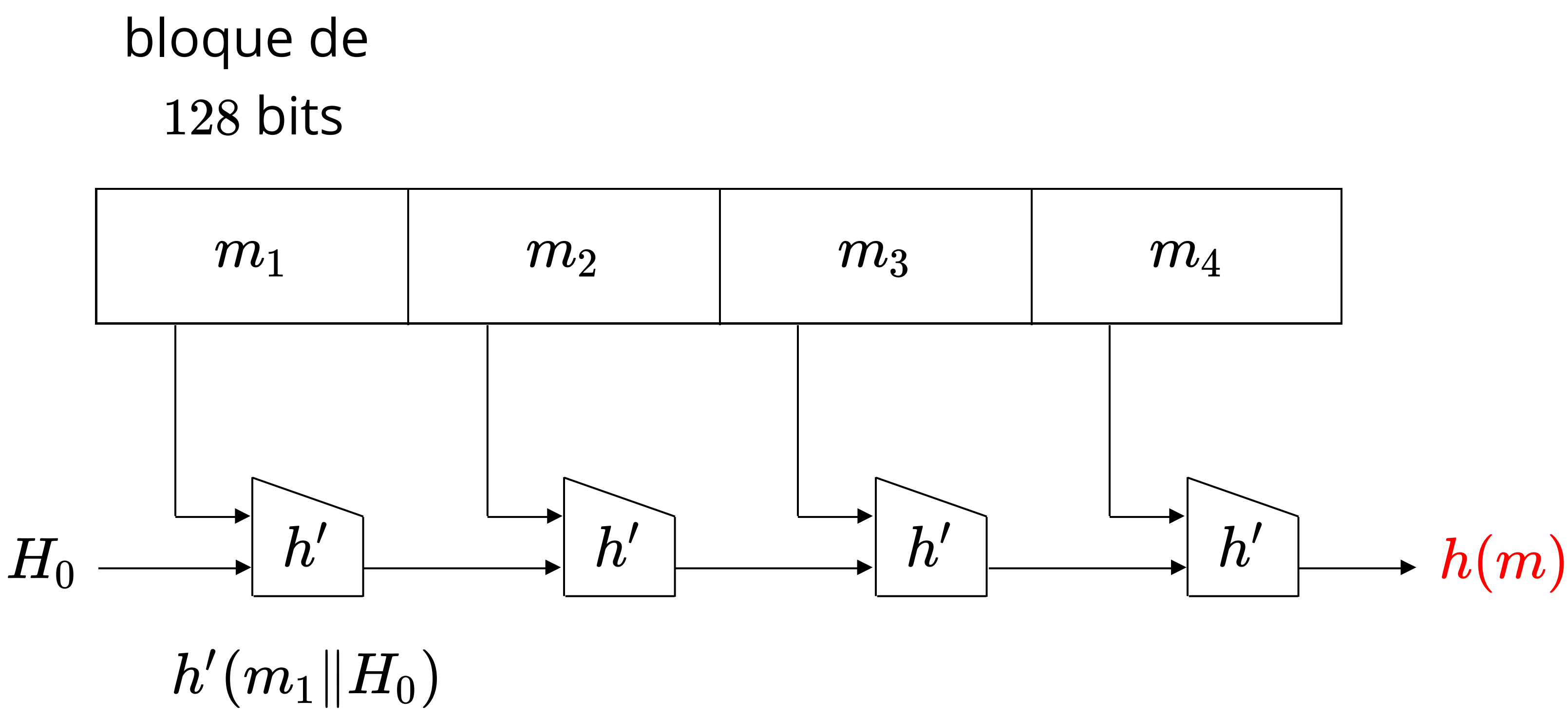
$(Gen', h')$ es una buena alternativa para una función de compresión

# La extensión a un largo arbitrario

Suponemos que tenemos una función de compresión $h' : \{0,1\}^{256} \to \{0,1\}^{128}$

$$\boxed{\quad m \quad}$$

# La extensión a un largo arbitrario



# ¿Cuál es la intuición detrás de esta idea?

Queremos demostrar que si $h'$ es resistente a colisiones y no se conoce una preimagen de $H_0$, entonces $h$ es resistente a colisiones

Piense en la idea de la demostración solo considerando mensajes cuyo largo es divisible por 128

- Y considere primero el caso donde los mensajes en una colisión tienen el mismo largo

# ¿Cómo podemos estar seguros que nadie conoce una preimagen de $H_0$?

¿Podríamos pedir que $H_0$ sea un número *sacado al azar?*

Nos pueden engañar: el adversario saca un mensaje $m_0 \in \{0,1\}^{256}$ al azar y nos dice que $H_0 = h'(m_0)$

- $H_0$ se ve como un número sacado al azar, pero el adversario conoce una preimagen $m_0$

# ¿Cómo podemos estar seguros que nadie conoce una preimagen de $H_0$?

Lo que necesitamos es un *nothing-up-my-sleeve number*

# ¿Cómo podemos estar seguros que nadie conoce una preimagen de $H_0$?

Lo que necesitamos es un *nothing-up-my-sleeve number*

Por ejemplo: considere los primeros 128 bits de la representación de $\pi$ en binario

- ¿Cree que alguien conoce la preimagen de este valor de $H_0$?

# Aún por responder: ¿Qué hacemos si el largo de $m$ no es divisible por 128?

Tenemos que hacer *padding*

| | | | |
|---|---|---|---|
| $m_1$ | $m_2$ | $m_3$ | $m_4$ |

# Aún por responder: ¿Qué hacemos si el largo de $m$ no es divisible por 128?

Tenemos que hacer *padding*

| | | | |
|---|---|---|---|
| $m_1$ | $m_2$ | $m_3$ | $m_4 \; 00\cdots0$ |

Aunque esta no es una buena forma de hacer padding

- Es fácil encontrar colisiones: $h(m) = h(m0)$ si $|m|$ no es divisible por 128

# ¿Qué debe cumplir una buena función de padding?

Considere una función $Pad(\cdot)$ para bloques de largo $n$

- En el ejemplo anterior, $n = 128$

Dado un mensaje $m \in \{0,1\}^*$, se debe tener que $|Pad(m)| \geq n$ y $|Pad(m)|$ es divisible por $n$

# Padding: los axiomas fundamentales

- $m$ es un prefijo de $Pad(m)$
- si $|m_1| = |m_2|$, entonces $|Pad(m_1)| = |Pad(m_2)|$
- si $|m_1| \neq |m_2|$, entonces el último bloque de $Pad(m_1)$ es distinto del último bloque de $Pad(m_2)$

# Poniendo todo junto: la construcción de Merkle–Damgård

Suponga dados:

- La función de compresión $(Gen', h')$ de Davies-Meyer
- Para cada $n \in \mathbb{N}$, una función de padding $Pad_n$ que considera bloques con $n$ elementos y satisface los axiomas fundamentales

Vamos a definir una función de hash $(Gen, h)$ para mensajes de largo arbitrario

# La construcción de Merkle–Damgård

Considere un parámetro de seguridad $1^n$

Tenemos que $Gen(1^n) = s$, y

$$h^s : \{0,1\}^* \to \{0,1\}^n$$

El valor del vector de inicialización $H_0$ está contenido en $s$

# La construcción de Merkle–Damgård

Dado $m \in \{0,1\}^*$, calculamos $h(m)$ de la siguiente forma:

1. Suponga que $Pad_n(m) = m_1 m_2 \cdots m_\ell$, donde el largo de cada bloque $m_i$ es $n$
2. Para cada $i \in \{1, \ldots, \ell\}$:
$$H_i := (h')^n(m_i \| H_{i-1})$$
3. $h(m) := H_\ell$

# La construcción es resistente a colisiones

Demuestre que $(Gen, h)$ es resistente a colisiones, dado que $(Gen', h')$ es resistente a colisiones y cada función de padding $Pad_n$ satisface los axiomas fundamentales

# Algunos comentarios sobre la construcción

- Podemos reemplazar la construcción de Davies-Meyer por cualquier función de compresión resistente a colisiones
- Podemos considerar funciones de compresión de la forma $h' : \{0,1\}^{p(n)} \to \{0,1\}^n$ con $p(n)$ un polinomio tal que $p(n) > n$

# Una pregunta pendiente: ¿qué función de padding utilizamos?

Consideramos bloques de largo $n$

| | | | |
|---|---|---|---|
| $m$ | $m_1$ | $m_2$ | $m_3$ |

# Una pregunta pendiente: ¿qué función de padding utilizamos?

Consideramos bloques de largo $n$

| | | | |
|---|---|---|---|
| $Pad(m)$ | $m_1$ | $m_2$ | $m_3$ |

# Una pregunta pendiente: ¿qué función de padding utilizamos?

Consideramos bloques de largo $n$

| | | | |
|---|---|---|---|
| $Pad(m)$ | $m_1$ | $m_2$ | $m_3 \ 10 \cdots 0$ |

# Una pregunta pendiente: ¿qué función de padding utilizamos?

Consideramos bloques de largo $n$

| | | | | |
|---|---|---|---|---|
| $Pad(m)$ | $m_1$ | $m_2$ | $m_3 \; 10\cdots0$ | $\lvert m \rvert \bmod 2^n$ |

# ¿Cuáles axiomas satisface esta función de padding?

Satisface los dos primeros axiomas fundamentales, pero pueden existir mensajes $m_1$ y $m_2$ tales que $|m_1| \neq |m_2|$ y los últimos bloques de $m_1$ y $m_2$ son iguales

Se debe tener que: $|m_1| \equiv |m_2| \mod 2^n$

- Si $|m_1| < |m_2|$, entonces $|m_2| > 2^n$

# ¿Cuáles axiomas satisface esta función de padding?

Si $n = 128$, entonces $|m_2| > 2^{128}$, y $m_2$ debe tener al menos $10^{38}$ dígitos

¿Es posible escribir un número de este tamaño?

- No: Cisco estima que el tráfico de Internet en 2025 será de $175$ zettabytes, vale decir $175 \cdot 10^{21}$ bytes

# Funciones de hash en la práctica: SHA-2

SHA-2 (Secure Hash Algorithm 2) es una familia de funciones de hash

- Se definen utilizando la construcción de Merkle–Damgård
- Las funciones de compresión son definidas utilizando la construcción de Davies–Meyer sobre un block cipher propio (no AES)

# SHA-256

El número 256 se refiere al largo del hash

- SHA-256 es una función de $\{0, 1\}^*$ en $\{0, 1\}^{256}$

SHA-256 puede considerarse como el resultado de instanciar el parámetro de seguridad en el valor 256

- También son utilizadas las funciones de hash SHA-224, SHA-384 y SHA-512

SHA-256 es utilizada en Bitcoin

# SHA-256: bloques y estados internos

SHA-256 considera bloques de 512 bits, y sus estados internos $H_i$ son de 256 bits

La función de compresión es de la forma

$$h' : \{0,1\}^{512} \times \{0,1\}^{256} \to \{0,1\}^{256}$$

# SHA-256: función de padding

Se define utilizando las ideas descritas anteriormente, pero reservando los últimos 64 bits para el largo del mensaje $m$

# SHA-256: función de padding

Se realiza los siguiente pasos sobre el mensaje $m$:

1. Se agrega un símbolo 1
2. Se agregan $\ell$ símbolos 0, donde $\ell$ es el menor número natural tal que $|m| + 1 + \ell \equiv 448 \mod 512$
3. Se agrega $|m| \mod 2^{64}$

# SHA-256: vector de inicialización

$H_0$ se define como $H_0^1 H_0^2 H_0^3 H_0^4 H_0^5 H_0^6 H_0^7 H_0^8$, donde

- $H_0^i$ tiene los primeros 32 bits de la parte decimal de la raíz cuadrada del $i$-ésimo número primo

Por ejemplo, $H_0^1$ tiene los primeros 32 bits de la parte decimal de $\sqrt{2}$:

$$H_0^2 = 01101010000010011110011001100111$$