Resumen Teórica 5 : Criptografía (parte 2)

Tomás F. Melli

August 2025

Índice

1 Números aleatorios

En la clase pasada arrancamos a ver sobre la criptografía moderna, esta se apoya en la impredecibilidad. Si un atacante puede adivinar una clave, un reto de autenticación o el próximo bit de un flujo cifrado, la seguridad colapsa. Por eso, los números aleatorios son fundamentales, ya que :

- Se usan para generar claves secretas,
- Para construir vectores de inicialización en cifrados de bloques,
- Para producir el flujo de claves en cifrados de flujo.
- Y para protocolos como el challenge–response de autenticación.

El tema es que generar números realmente aleatorios no es trivial. O sea, es difícil encontrar fuentes aleatorias verdaderas (no predecibles ni reproducibles). En la práctica distinguimos dos fuentes :

1. **Aleatorios verdaderos (TRNG)** : basados en fenómenos físicos impredecibles. Como el ruido físico (radioactividad, ruido térmico en un semiconductor, ruido de un micrófono), estado de una computadora (reloj, posición del mouse, actividad de la red). Lo bueno es que son no predecibles (no hay patrón oculto). No reproducibles (si hago la misma medición otra vez, el resultado es distinto). El problema es que son lentos, o sea, producen pocos bits por segundo. Muchas veces se necesita post-procesamiento (por ejemplo, un hash) para eliminar sesgos.
2. **Pseudo-aleatorios (PRNG)** : generados por algoritmos determinísticos que simulan aleatoriedad. Parecen aleatorios estadísticamente (pasan tests de aleatoriedad). Son reproducibles, misma semilla implica misma secuencia. Son rápidos, producen millones de bits por segundo. El tema es que si la semilla no es secreta o es predecible, el atacante puede regenerar toda la secuencia.

2 RC4 (Rivest Cipher 4)

El RC4 fue diseñado en 1987 por Ron Rivest, uno de los grandes referentes de la criptografía moderna. Durante muchos años fue uno de los cifradores de flujo más utilizados en el mundo, aunque en la actualidad ya no se recomienda su uso debido a vulnerabilidades descubiertas con el tiempo. En 1994, la descripción del algoritmo se filtró en Internet bajo el nombre Arcfour (Alleged RC4), y a partir de ahí empezó a implementarse de manera abierta. Fue ampliamente utilizado en protocolos como TLS/SSL (para conexiones seguras en la web) y en WEP (cifrado de redes Wi-Fi). Su popularidad se debió principalmente a que era muy eficiente en software, especialmente en computadoras de la época.

2.1 Funcionamiento

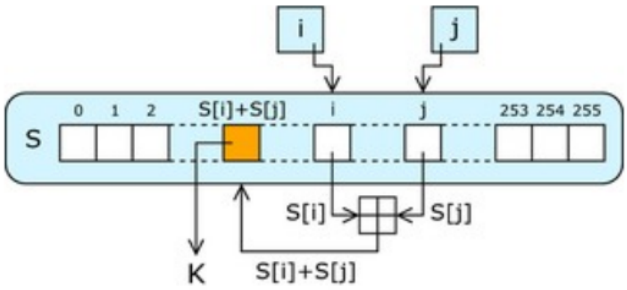
RC4 es un cifrador de flujo sincrónico que trabaja a nivel de bytes. El algoritmo genera un flujo de bytes pseudo-aleatorios, llamado *keystream*. Este flujo se combina con el texto plano mediante la operación XOR ($\oplus$). El resultado es el texto cifrado. El proceso de descifrado es idéntico: se aplica XOR entre el *keystream* y el texto cifrado, y se recupera el mensaje original.

$$m \oplus k \oplus k = m$$

donde m es el mensaje y k el *keystream*. El núcleo de RC4 se basa en un estado interno secreto y dos algoritmos principales:

1. **Key Scheduling Algorithm (KSA)**
   - Es la fase de programación de la clave.
   - Toma como entrada una clave de entre 40 y 256 bits.
   - Inicializa una permutación de los números del 0 al 255 en una tabla llamada *S-box*.
   - A partir de la clave, mezcla los valores de la *S-box* de manera compleja.
   - El resultado es un estado inicial secreto, dependiente de la clave.
2. **Pseudo-Random Generation Algorithm (PRGA)**
   - Es la fase de generación de *keystream*.
   - A partir del estado inicial de la *S-box*, va produciendo byte a byte una secuencia pseudo-aleatoria.
   - En cada iteración:
     (a) Actualiza índices internos.
     (b) Intercambia valores dentro de la *S-box*.
     (c) Selecciona un byte de salida como parte del *keystream*.
   - Este proceso es determinístico: con la misma clave inicial, se genera siempre la misma secuencia.

Esquema



3 DES – Data Encryption Standard

El Data Encryption Standard (DES) es uno de los algoritmos de cifrado más influyentes en la historia de la criptografía moderna. Fue el primer estándar oficial de cifrado adoptado en los Estados Unidos y, durante décadas, se convirtió en la base para la protección de información en sistemas comerciales y gubernamentales.

3.1 Historia

- En 1973, el National Bureau of Standards (NBS) (hoy conocido como NIST) lanzó un concurso público para definir un algoritmo criptográfico estándar de uso general.
- En 1974, la NSA (National Security Agency) declaró desierto el primer concurso y publicó una nueva especificación. Como resultado, eligió el algoritmo Lucifer, desarrollado por IBM, pero con algunas modificaciones introducidas por la propia NSA.
- Finalmente, en 1976, el algoritmo fue adoptado oficialmente bajo el nombre de DES y se autorizó su uso en las comunicaciones no clasificadas del gobierno de los EE. UU.

3.2 Características principales

3.2.1 Cifrado en bloque

- DES es un cifrador por bloques.
- Trabaja con bloques de 64 bits de texto plano.
- Utiliza una clave de 64 bits (8 bytes). Sin embargo, uno de los bits de cada byte se reserva para paridad, por lo que la clave efectiva es de 56 bits útiles.
- El resultado de la operación es un bloque cifrado de 64 bits.

3.2.2 Cifrado por producto

- DES es un ejemplo de cifrado por producto, es decir, combina de manera iterativa operaciones de sustitución y permutación (transposición) a nivel de bits.
- La unidad básica de procesamiento es el bit, lo que permite un control muy fino sobre la estructura del mensaje.

3.2.3 Rondas de cifrado

- El diseño de DES sigue una estructura de Feistel(*).
- Consiste en 16 rondas (iteraciones).
- En cada ronda:
  - Se toma una subclave generada a partir de la clave maestra original.
  - Se aplica una combinación de sustituciones, permutaciones y operaciones lógicas (principalmente XOR).
- Este esquema garantiza **difusión** y **confusión**, principios fundamentales de la criptografía moderna.

Red de Feistel (*)

Una red de Feistel es un diseño o esquema general para construir cifradores en bloque. Fue inventada por Horst Feistel (IBM) en los años 70, y se hizo famosa porque DES la adoptó como base. La idea es dividir el bloque de datos en dos mitades y transformarlo en varias rondas, aplicando funciones relativamente simples.
Supongamos un bloque de entrada de 64 bits. Lo dividimos en dos mitades de 32 bits:

- Mitad izquierda: L_0
- Mitad derecha: R_0

Cada ronda de Feistel aplica la siguiente transformación:

$$L_{i+1} = R_i$$

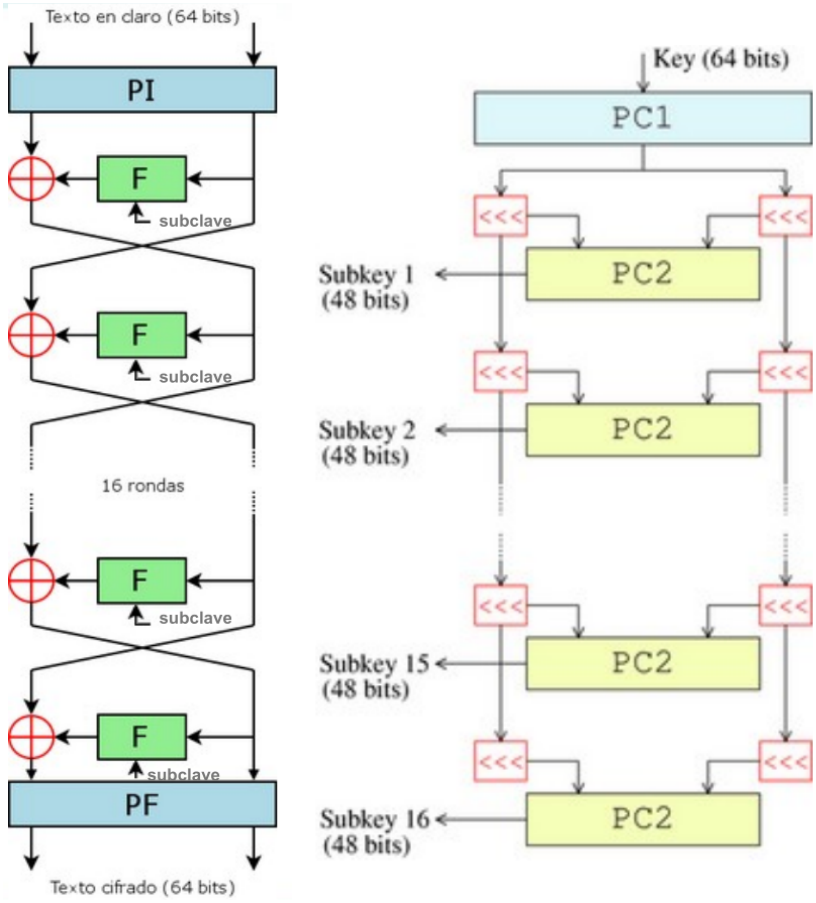
$$R_{i+1} = L_i \oplus F(R_i, K_i)$$

Donde:

- F es una función de cifrado que combina la mitad derecha con una subclave K_i.
- $\oplus$ representa la operación XOR.
- i es el número de ronda.

Después de un número suficiente de rondas (en el caso de DES son 16), se obtiene el bloque cifrado.

3.3 Esquema general



1. **Permutaciones inicial y final**
   - Antes de comenzar las rondas, se aplica una permutación inicial (PI) al bloque de 64 bits de entrada.
   - Al finalizar todas las rondas, se aplica una permutación final (PF) para obtener el bloque cifrado definitivo.
2. **16 rondas de Feistel**

- Cada bloque pasa por 16 iteraciones con la siguiente estructura:
  - El bloque se divide en dos mitades (izquierda y derecha).
  - Una mitad se procesa mediante la función F de Feistel**(**)**, que combina sustituciones, permutaciones y la subclave de la ronda.
  - Los resultados se mezclan con la otra mitad usando XOR, garantizando difusión y confusión.
  - Finalmente, ambas mitades se rotan para preparar la siguiente ronda.

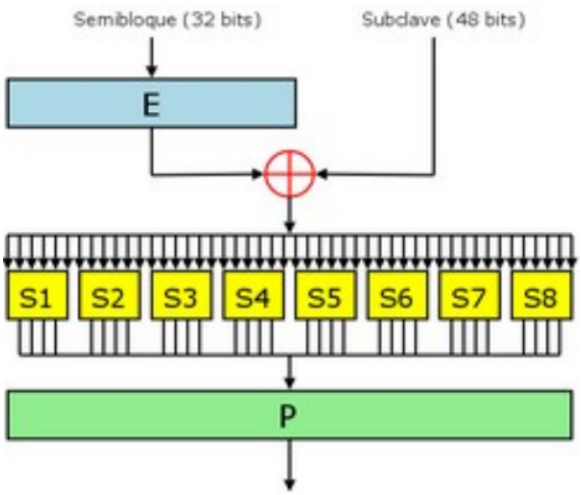
3. **Generación de subclaves**
   - Se permutan y eligen 56 bits de la clave inicial mediante la permutación PC1.
   - Los 8 bits restantes se descartan o se usan como bits de paridad.
   - Los 56 bits se dividen en dos mitades de 28 bits.
   - En cada ronda, ambas mitades se desplazan hacia la izquierda uno o dos bits (dependiendo de la ronda) y se seleccionan los 48 bits de la subclave permutando cada mitad (PC2) y seleccionando 24 bits de cada una.

Función F de Feistel (**)

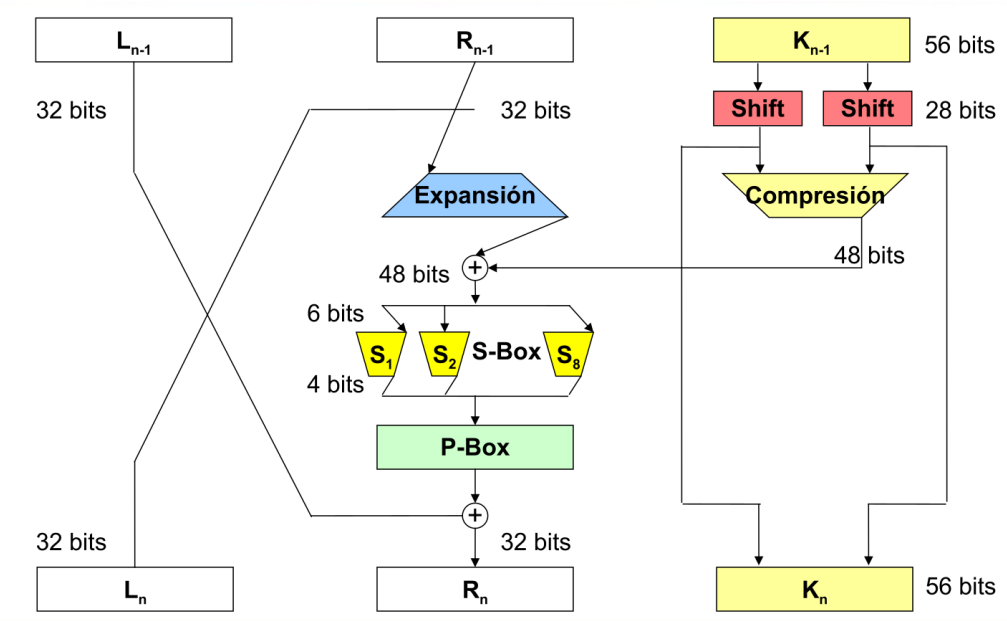
Dentro del esquema de Feistel de DES, la función F es el núcleo que transforma cada mitad del bloque durante cada ronda. Trabaja con bloques de 32 bits y consta de cuatro pasos fundamentales:

1. **Expansión**
   - El bloque de 32 bits se expande a 48 bits mediante la permutación de expansión.
   - Este proceso duplica algunos bits para preparar la mezcla con la subclave.
2. **Mezcla con subclave**
   - El bloque expandido se combina con la subclave de 48 bits mediante una operación XOR.
   - Esto asegura que la transformación dependa tanto del mensaje como de la clave.
3. **Sustitución (S-Box)**
   - El bloque resultante se divide en ocho trozos de 6 bits.
   - Cada trozo pasa por una S-Box (caja de sustitución), que reemplaza sus 6 bits de entrada por 4 bits de salida.
   - Cada S-Box aplica una transformación no lineal, definida mediante una tabla de búsqueda, introduciendo confusión en el cifrado.
4. **Permutación (P-Box)**
   - Finalmente, las 32 salidas de las S-Box se reordenan según una permutación fija, llamada P-Box.
   - Esto asegura una difusión uniforme, mezclando los bits para la siguiente ronda.

Esquema



3.3.1 Funcionamiento



3.4 Propiedades no deseadas y ataques

A pesar de ser un estándar ampliamente utilizado, DES posee ciertas propiedades que lo hacen vulnerable frente a ataques avanzados:

- **Claves débiles**
  - Existen 4 claves débiles, las cuales son su propia inversa.
  - Esto significa que cifrar dos veces con la misma clave devuelve el mensaje original, lo cual reduce la seguridad.
- **Claves semi-débiles**
  - Se han identificado 6 pares de claves semi-débiles.
  - Cada clave de un par tiene como inversa a la otra clave del par, lo que permite ciertos ataques simples sobre el cifrado doble.
- **Propiedad complementaria**
  - Existe una relación de simetría conocida como propiedad complementaria:

    $$\text{DES}_k(m) = c \implies \text{DES}_{k'}(m') = c'$$

  - Esto significa que el cifrado del complemento de un mensaje con la clave complementaria produce el complemento del texto cifrado original, introduciendo patrones predecibles.
- **Irregularidades en las S-Box**
  - Las S-Box de DES no son completamente uniformes.
  - La distribución de los números pares e impares no es aleatoria.
  - La salida de la cuarta S-Box depende de la entrada de la tercera S-Box, lo que genera correlaciones no deseadas.

Ataques conocidos a DES

A lo largo de los años, se han desarrollado varios ataques que explotan estas debilidades:

- **Fuerza bruta**
  - Consiste en probar una por una todas las posibles claves hasta encontrar la correcta.
  - Con las capacidades computacionales actuales, esto se puede hacer en un tiempo relativamente corto para claves de 56 bits.

- **Criptoanálisis diferencial (finales de los '80)**
  - Basado en analizar pares de textos en claro elegidos y observar cómo evolucionan durante las rondas con la misma clave.
  - Para ser efectivo, requiere aproximadamente 2^{47} textos en claro elegidos.
- **Criptoanálisis lineal (1993)**
  - Explota correlaciones lineales aproximadas entre bits del texto plano, la clave y el texto cifrado.
  - Para realizar este ataque, se necesitan alrededor de 2^{43} textos en claro elegidos.

3.5 Modos de operación

Los **modos de operación** definen cómo se aplica un cifrador en bloque (como DES) a mensajes más largos que un solo bloque. Cada modo introduce distintas propiedades de seguridad y resistencia a ataques.

- **Electronic Code Book (ECB)**
  - Cada bloque se cifra de manera independiente.
  - Es sencillo de implementar, pero los bloques idénticos de texto plano producen bloques idénticos de texto cifrado, revelando patrones en los datos.
- **Cipher Block Chaining (CBC)**
  - Cada bloque de texto plano se combina mediante XOR con el bloque cifrado anterior antes de cifrarlo.
  - Introduce dependencia entre bloques, haciendo que bloques idénticos de texto plano produzcan textos cifrados diferentes.
  - El primer bloque se combina con un **vector de inicialización (IV)** para asegurar aleatoriedad en la primera iteración.

Modos extendidos para aumentar la seguridad

- **Encrypt-Decrypt-Encrypt (EDE)**
  - Utiliza dos claves, k y k', con un tamaño total de 112 bits.
  - Si $k = k'$, este modo equivale a DES simple.
  - El cifrado se realiza como:

$$c = \mathrm{DES}_k\big(\mathrm{DES}_{k'}^{-1}\big(\mathrm{DES}_k(m)\big)\big)$$

  - Este esquema aumenta la seguridad frente a ataques de fuerza bruta y se conoce como **Triple DES de dos claves**.
- **Encrypt-Encrypt-Encrypt (Triple DES o 3DES)**
  - Utiliza tres claves, k, k' y k'', con un tamaño total de 168 bits.
  - El cifrado se realiza como:

$$c = \mathrm{DES}_k\big(\mathrm{DES}_{k'}\big(\mathrm{DES}_{k''}(m)\big)\big)$$

  - Triple DES es significativamente más seguro que DES simple y fue ampliamente utilizado antes de la adopción de AES.

3.6 Más historia de DES

El **Data Encryption Standard (DES)** ha sido un hito en la historia de la criptografía, pero su evolución refleja la necesidad constante de adaptarse a nuevas amenazas.

- **Certificación por NIST**
  - En 1987 y nuevamente en 1993, el **NIST** (National Institute of Standards and Technology, antes NBS) certificó oficialmente DES como estándar de cifrado.

- **Desafíos públicos (DES Challenge)**
  - Entre 1997 y 1999, DES fue puesto a prueba mediante **tres desafíos públicos**, conocidos como DES Challenge, impulsados y promocionados por la compañía RSA.
  - Estos desafíos demostraron que, con la capacidad computacional disponible, DES podía ser quebrado por fuerza bruta, evidenciando la necesidad de un cifrado más robusto.
- **Búsqueda de un nuevo estándar**
  - En 1997, NIST decidió no certificar más DES y lanzó un **concurso internacional** para encontrar un nuevo algoritmo de cifrado que reemplazara a DES.
- **Selección de AES**
  - En 2001, NIST seleccionó el algoritmo **Rijndael** como sucesor de DES, y lo nombró **Advanced Encryption Standard (AES)**.
  - AES fue diseñado específicamente para resistir los ataques que habían sido efectivos contra DES, ofreciendo mayor seguridad y flexibilidad para distintas longitudes de clave.

4 AES - Advanced Encryption Standard

El **AES (Advanced Encryption Standard)** es un **cifrador de bloque** y un **cifrador por producto**, diseñado para ser seguro, eficiente y flexible frente a distintas necesidades de seguridad.

- **Bloques y claves de longitud variable**
  - AES opera con bloques y claves de longitudes variables que pueden ser especificadas de manera independiente: 128, 192 o 256 bits.
  - Las combinaciones posibles permiten una adaptación a futuras necesidades de seguridad, y el diseño se puede extender fácilmente a múltiplos de 32 bits, fortaleciendo su resistencia ante ataques avanzados.
- **Implementación eficiente**
  - AES permite implementaciones rápidas y eficientes tanto en software como en hardware.
  - Esto lo hace adecuado para prácticamente cualquier tipo de aplicación actual, desde comunicaciones seguras hasta almacenamiento cifrado.
- **Cifrador por producto**
  - Al igual que DES, AES se basa en la idea de un cifrado por producto, combinando iterativamente sustituciones, permutaciones y mezclas de bytes.
  - Presenta mejoras significativas en seguridad y eficiencia frente a su predecesor DES.

4.1 Funcionamiento

AES es un cifrador de bloque que opera sobre bloques de 128 bits (16 bytes), con claves de 128, 192 o 256 bits. Su diseño se basa en un cifrado por sustitución-permutación (SPN) y se realiza en varias rondas.

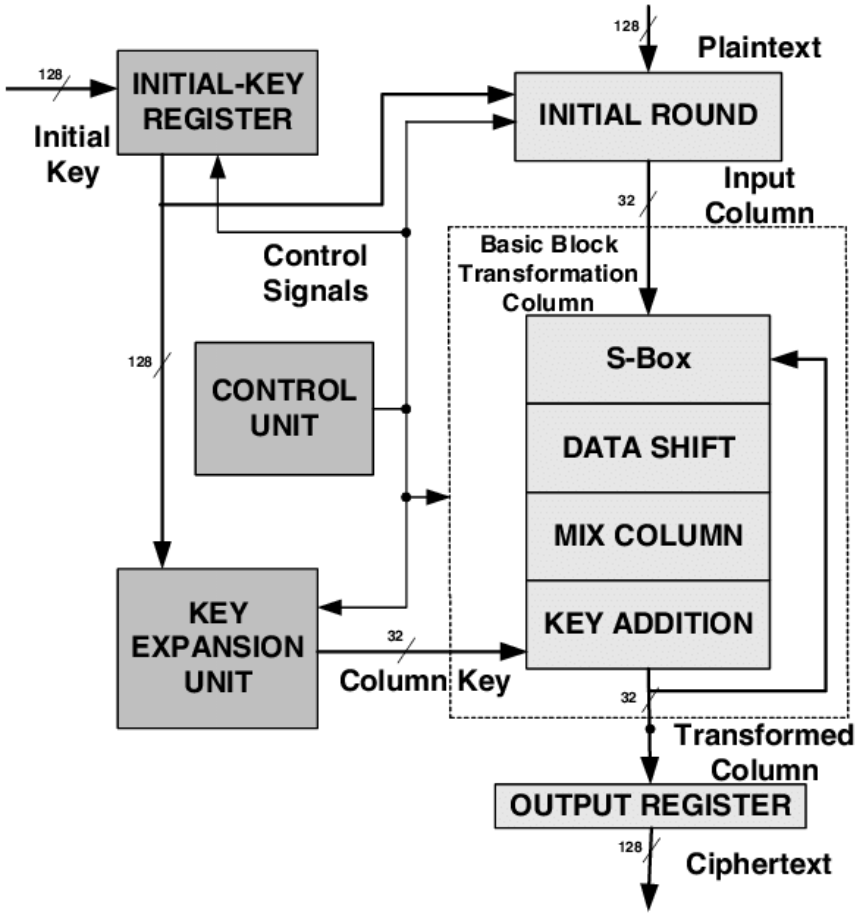
- **Estructura general**
  - El bloque de 128 bits se representa como una matriz de 4x4 bytes llamada **State**.
  - Cada operación transforma esta matriz mediante sustituciones y permutaciones, asegurando confusión y difusión.
  - Número de rondas según longitud de clave:
    - 128 bits: 10 rondas
    - 192 bits: 12 rondas
    - 256 bits: 14 rondas
- **Etapas de cada ronda**
  1. **SubBytes (Sustitución de bytes)**
     - Cada byte del State se reemplaza mediante una S-Box no lineal.

- Introduce confusión, haciendo la relación entre texto plano y cifrado no lineal.

2. **ShiftRows (Desplazamiento de filas)**
   - Cada fila se desplaza a la izquierda un número de bytes según su fila:
     * Fila 0: sin desplazamiento
     * Fila 1: 1 byte
     * Fila 2: 2 bytes
     * Fila 3: 3 bytes
   - Aumenta la difusión entre columnas.
3. **MixColumns (Mezcla de columnas)**
   - Cada columna se transforma combinando sus 4 bytes mediante operaciones lineales en GF(2ˆ8).
   - Mezcla los bytes dentro de cada columna.
   - La última ronda no incluye este paso.
4. **AddRoundKey (Suma de subclave)**
   - Se combina el State con la subclave de la ronda mediante XOR.
   - Las subclaves se generan a partir de la clave original usando el Key Schedule de AES.

- **Flujo completo del cifrado**
  - **AddRoundKey inicial**: se aplica la primera subclave al State antes de las rondas.
  - **Rondas principales**: SubBytes → ShiftRows → MixColumns → AddRoundKey (excepto MixColumns en la última ronda).
  - **Ronda final**: SubBytes → ShiftRows → AddRoundKey.
  - **Salida**: el State final se convierte en el texto cifrado de 128 bits.

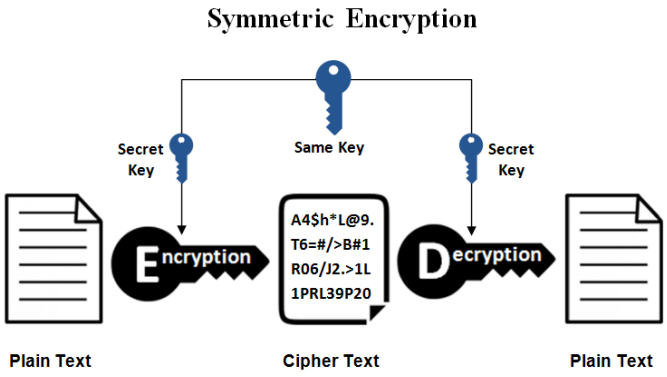
- **Descifrado**
  - Invierte cada operación con funciones inversas: InvSubBytes, InvShiftRows, InvMixColumns, AddRoundKey.
  - Se aplican las subclaves en orden invertido.



5 Criptografía simétrica

La criptografía simétrica se basa en el uso de una misma clave tanto para cifrar como para descifrar la información. Si bien es eficiente y rápida, presenta problemas importantes en su aplicación práctica:

- **Distribución de las claves**
  - El receptor debe conocer la clave que se va a utilizar para poder descifrar el mensaje.
  - No es posible enviar la clave a través de medios inseguros, ya que si un atacante la intercepta, toda la comunicación queda comprometida.
- **Complejidad en la gestión de las claves**
  - A medida que aumenta el número de usuarios o sistemas que necesitan comunicarse, el número de claves necesarias crece rápidamente.
  - Mantener, actualizar y proteger todas estas claves se vuelve una tarea compleja y costosa, especialmente en entornos grandes o distribuidos.

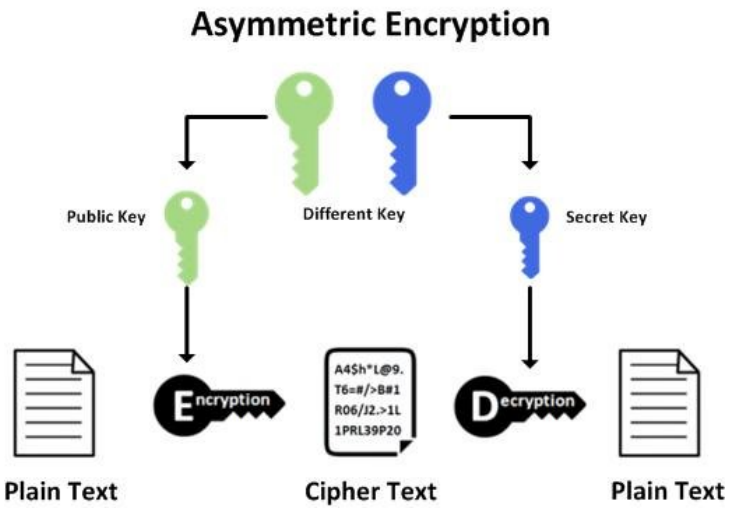


6 Criptografía asimétrica

La **criptografía asimétrica**, también conocida como **criptografía de clave pública**, fue introducida por **Diffie y Hellman en 1976**. Surge como una solución al problema de la distribución de claves en criptografía simétrica, permitiendo que dos partes puedan establecer una clave en común incluso a través de **canales inseguros**.

A diferencia de la criptografía simétrica, en la criptografía asimétrica **el cifrado y el descifrado se realizan con claves diferentes**:

- **Claves utilizadas**
  - **Clave pública**: disponible para cualquier persona.
  - **Clave privada**: conocida únicamente por el propietario.
- **Idea de uso**
  - **Confidencialidad**:
    - El remitente cifra el mensaje usando la **clave pública** del destinatario.
    - El destinatario descifra el mensaje usando su **clave privada**.
  - **Integridad y autenticación**:
    - El emisor cifra el mensaje usando su **clave privada**.
    - Cualquier receptor puede verificar la autenticidad descifrándolo con la **clave pública** del emisor.
- **Condiciones fundamentales de seguridad**
  - Dada la clave apropiada, debe ser computacionalmente fácil cifrar y descifrar un mensaje.
  - Debe ser computacionalmente imposible derivar la clave privada a partir de la clave pública.
  - Debe ser computacionalmente imposible determinar la clave privada incluso a partir de un ataque de texto en claro elegido.



7 Diffie-Hellman – Intercambio de claves

El algoritmo **Diffie-Hellman** permite a dos partes establecer una clave secreta compartida sobre un canal inseguro sin necesidad de intercambiar previamente claves. Fue introducido por **Whitfield Diffie** y **Martin Hellman** en 1976.

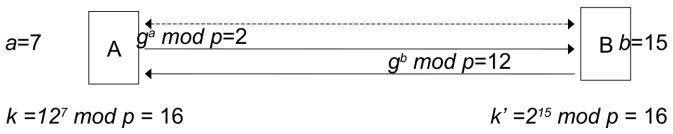
7.1 Funcionamiento conceptual

1. **Elección de parámetros públicos:**
   - Alice y Bob acuerdan dos números: p (un número primo) y g (un entero tal que $2 \leq g \leq p-1$).
2. **Generación de claves privadas:**
   - Alice elige un número secreto a.
   - Bob elige un número secreto b.
3. **Intercambio de claves públicas:**
   - Alice calcula $A = g^a \mod p$ y lo envía a Bob.
   - Bob calcula $B = g^b \mod p$ y lo envía a Alice.
4. **Cálculo de la clave compartida:**
   - Alice calcula $k = B^a \mod p$.
   - Bob calcula $k' = A^b \mod p$.
5. **Resultado:**
   - Se cumple que $k = k' = g^{ab} \mod p$, siendo k la clave secreta compartida.

Supongamos que Alice y Bob acuerdan los siguientes valores: $p = 23$, $g = 3$

- Alice elige: $a = 7$ y envía $g^a \mod p$, es decir: $3^7 \mod 23 = 2$
- Bob elige: $b = 15$ y envía $g^b \mod p$, es decir: $3^{15} \mod 23 = 12$
- Alice calcula la clave compartida: $k = 12^7 \mod 23 = 16$
- Bob calcula la clave compartida: $k' = 2^{15} \mod 23 = 16$

Ambos obtienen la misma clave secreta compartida: $k = k' = 16$.



7.2 Características

- Se basa en operaciones de exponenciación modular y en la dificultad del problema del logaritmo discreto.
- Requiere el uso de números grandes para garantizar la seguridad.
- No proporciona autenticación, lo que lo hace vulnerable a ataques *Man-in-the-Middle*.
- Solo las claves privadas a y b, y el valor $g^{ab} \mod p$, deben mantenerse en secreto.

Más información

Para detalles técnicos y especificaciones, consultar el RFC 2631.

8 RSA – Cifrado y firma de mensajes

El algoritmo **RSA** fue creado en 1977 por **Ron Rivest, Adi Shamir y Len Adleman** del MIT. RSA es un algoritmo de **criptografía asimétrica** que puede utilizarse tanto para **cifrado** como para **firma de mensajes**. Su seguridad se basa en la dificultad de **factorizar números muy grandes**, y se clasifica como un cifrador **exponencial**.
Más información: RFC 2313 (RSA v1.5).

Fundamentos matemáticos

- **Función $\varphi(n)$ (totiente de Euler):** Número de enteros positivos menores que n que son **coprimos** con n. Dos números son coprimos si no comparten factores primos.
- **Ejemplos:**
  - $\varphi(10) = 4$ porque 1, 3, 7, 9 son coprimos con 10.
  - $\varphi(21) = 12$ porque 1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20 son coprimos con 21.

8.1 Funcionamiento de RSA

1. **Elegir números primos grandes:** Seleccionar p y q tal que $p \neq q$.
2. **Calcular el módulo n:**

   $$n = p \cdot q$$

   La función totiente se calcula como:

   $$\varphi(n) = (p-1)(q-1)$$

3. **Elegir el exponente público e:** Seleccionar $e < n$ tal que $\gcd(e, \varphi(n)) = 1$, es decir, e es coprimo con $\varphi(n)$.
4. **Calcular el exponente privado d:** Resolver la ecuación:

   $$e \cdot d \mod \varphi(n) = 1$$

   Esta solución existe y es única.
5. **Definición de claves:**
   - **Clave pública:** (e, n)
   - **Clave privada:** d

Cifrado y descifrado

- **División en bloques:** Dado un mensaje m, se lo divide en bloques de longitud menor a n.
- **Cifrado de cada bloque:** Para cada bloque m_i se calcula:

  $$c_i = m_i^e \mod n$$

- **Descifrado de cada bloque:** Para cada bloque cifrado c_i se calcula:

  $$m_i = c_i^d \mod n$$

8.2 Confidencialidad con RSA

RSA permite garantizar la **confidencialidad** de los mensajes, de modo que solo el receptor que posee la clave privada pueda leer la información enviada.

8.2.1 Generación de claves (Ejemplo)

Tomemos números primos pequeños para ilustrar el proceso:

$$p = 7, \quad q = 11$$

1. Calcular el módulo n:

   $$n = p \cdot q = 7 \cdot 11 = 77$$

2. Calcular la función totiente:

   $$\varphi(n) = (p-1)(q-1) = 6 \cdot 10 = 60$$

3. Elegir el exponente público e y calcular el exponente privado d:

   $$e = 17, \quad d = 53 \quad \text{porque } e \cdot d \mod \varphi(n) = 1$$

4. Claves de Alice:

   Clave pública: $(e, n) = (17, 77)$, Clave privada: $d = 53$

8.2.2 Cifrado del mensaje (Ejemplo)

Bob quiere enviar a Alice el mensaje secreto **HELLO**, codificado como números:

H E L L O = 07 04 11 11 14

Para cifrar, Bob usa la clave pública de Alice y calcula:

$$c_i = m_i^e \mod n$$

Los cálculos para cada bloque son:

$$07^{17} \mod 77 = 28$$
$$04^{17} \mod 77 = 16$$
$$11^{17} \mod 77 = 44$$
$$11^{17} \mod 77 = 44$$
$$14^{17} \mod 77 = 42$$

El mensaje cifrado que Bob envía es:

28 16 44 44 42

8.2.3 Descifrado del mensaje (Ejemplo)

Alice recibe el mensaje cifrado 28 16 44 44 42 y utiliza su clave privada $d = 53$ para descifrar cada bloque:

$$m_i = c_i^d \mod n$$

Los cálculos son:

$$28^{53} \mod 77 = 07 \text{ (H)}$$
$$16^{53} \mod 77 = 04 \text{ (E)}$$
$$44^{53} \mod 77 = 11 \text{ (L)}$$
$$44^{53} \mod 77 = 11 \text{ (L)}$$
$$42^{53} \mod 77 = 14 \text{ (O)}$$

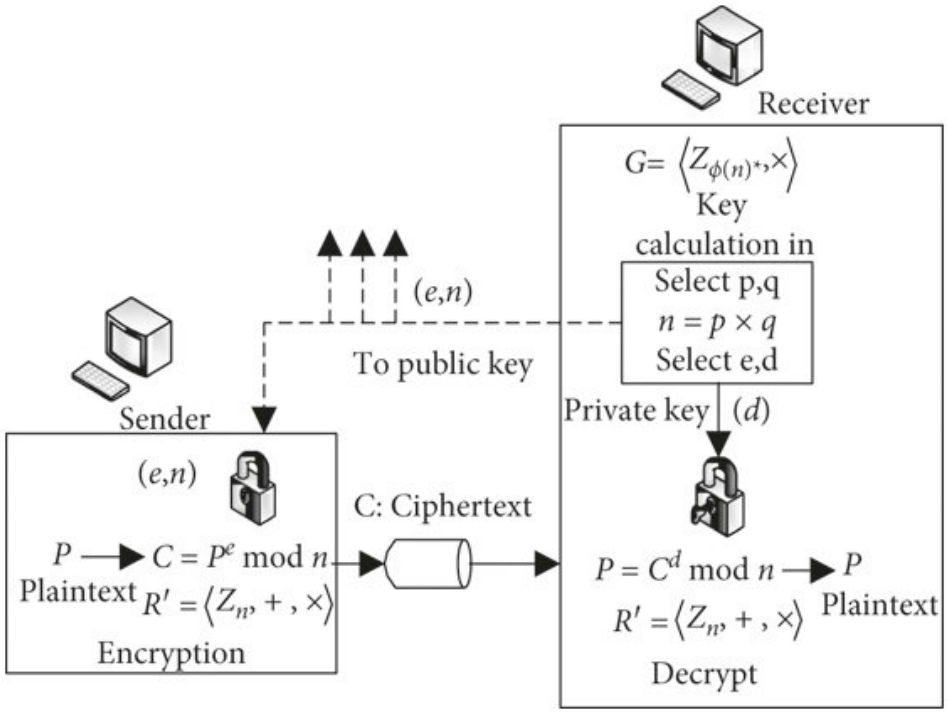
Alice recupera correctamente el mensaje original **HELLO**.

Conclusión (Ejemplo)

Gracias al uso de la clave pública para cifrar y la clave privada para descifrar:

- Nadie que no sea Alice puede leer el mensaje de Bob.
- Se garantiza la **confidencialidad** de la comunicación.

8.2.4 Esquema



8.3 Integridad y Autenticación con RSA

RSA no solo sirve para garantizar **confidencialidad**, sino también para asegurar **integridad** y **autenticación** de los mensajes. Esto permite que el receptor verifique que el mensaje proviene del emisor legítimo y que no ha sido modificado durante la transmisión.

Generación de claves (Ejemplo)

Tomamos los mismos números primos que en el ejemplo de confidencialidad:

$$p = 7, \quad q = 11$$

Calcular el módulo n:

$$n = p \cdot q = 7 \cdot 11 = 77$$

Calcular la función totiente:

$$\varphi(n) = (p-1)(q-1) = 6 \cdot 10 = 60$$

Elegir el exponente público e y calcular el exponente privado d:

$$e = 17, \quad d = 53 \quad \text{porque } e \cdot d \mod \varphi(n) = 1$$

Claves de Alice:

$$\text{Clave pública: } (e, n) = (17, 77), \quad \text{Clave privada: } d = 53$$

Firma del mensaje (Ejemplo)

Alice quiere enviar el mensaje **HELLO**, codificado como:

H E L L O = 07 04 11 11 14

Para asegurar que Bob sepa que el mensaje proviene de Alice y que no ha sido alterado, Alice utiliza su clave privada y calcula:

$$c_i = m_i^d \mod n$$

Los cálculos de cada bloque son:

$$07^{53} \mod 77 = 35$$
$$04^{53} \mod 77 = 09$$
$$11^{53} \mod 77 = 44$$
$$11^{53} \mod 77 = 44$$
$$14^{53} \mod 77 = 49$$

Alice envía el mensaje firmado:

35 09 44 44 49

Verificación del mensaje por el receptor (Ejemplo)

Bob recibe el mensaje firmado 35 09 44 44 49 y utiliza la clave pública de Alice para verificarlo:

$$m_i = c_i^e \mod n$$

Los cálculos son:

$$35^{17} \mod 77 = 07 \text{ (H)}$$
$$09^{17} \mod 77 = 04 \text{ (E)}$$
$$44^{17} \mod 77 = 11 \text{ (L)}$$
$$44^{17} \mod 77 = 11 \text{ (L)}$$
$$49^{17} \mod 77 = 14 \text{ (O)}$$

Bob recupera correctamente el mensaje original **HELLO**.

Conclusión (Ejemplo)

- La verificación confirma que Alice fue quien envió el mensaje, ya que solo ella conoce la clave privada utilizada para firmarlo.
- Si algún bloque del mensaje hubiera sido alterado durante la transmisión, el descifrado no produciría los valores correctos, garantizando así la **integridad** del mensaje.

8.4 Ataques a RSA

La seguridad de RSA depende completamente del problema de **factorizar números grandes**. Si un atacante lograra factorizar el módulo $n = p \cdot q$, podría reconstruir la clave privada y comprometer tanto la confidencialidad como la autenticidad de los mensajes. Existen varios tipos de ataques que pueden explotarse bajo ciertas condiciones:

- **Texto cifrado elegido (firma de textos aleatorios):** El atacante solicita firmas de mensajes escogidos por él y utiliza esta información para deducir la clave privada o para falsificar firmas.
- **Módulo común:** Si diferentes usuarios utilizan el mismo módulo n pero exponentes e distintos y coprimos, un atacante podría combinar los mensajes cifrados para recuperar información del texto original.
- **Exponente de cifrado bajo:** Si el exponente público e es demasiado pequeño, se pueden realizar ataques matemáticos que permiten recuperar el mensaje original sin necesidad de factorizar el módulo.
- **Exponente de descifrado bajo:** Claves privadas demasiado pequeñas facilitan el cálculo de d mediante métodos de análisis de congruencias o factorización parcial, comprometiendo la seguridad.

En resumen, para mantener la seguridad de RSA es crucial elegir números primos grandes y exponentes adecuados, evitando valores pequeños y condiciones que puedan facilitar estos ataques.

9 ECC (Elliptic-curve cryptography)

9.1 ¿Qué es una curva elíptica?

En criptografía, una curva elíptica no es un óvalo común, sino un conjunto de puntos (x, y) que cumplen una ecuación de la forma:

$$y^2 = x^3 + ax + b$$

donde a y b son constantes que definen la curva. Además, estas operaciones se realizan sobre un campo finito, normalmente con números enteros módulo un primo grande p. Esto limita los puntos posibles y garantiza que las operaciones sean seguras y finitas.
Lo interesante es que sobre estos puntos se define una operación de suma de puntos que tiene propiedades matemáticas útiles: es asociativa, tiene un elemento neutro y cada punto tiene un inverso. Estas propiedades permiten construir sistemas criptográficos robustos y eficientes.
El fundamento de la seguridad de las curvas elípticas se basa en el **problema del logaritmo discreto en curvas elípticas (ECDLP)**. En términos sencillos, dado un punto P de la curva y otro punto Q que es un múltiplo de P, es decir,

$$Q = kP,$$

resulta extremadamente difícil calcular el valor de k. Este número k es secreto y su dificultad de cálculo garantiza que operaciones como el intercambio de claves o la firma digital sean seguras. En otras palabras, aunque P y Q sean públicos, encontrar k sería computacionalmente inviable, lo que hace que los sistemas de criptografía de curva elíptica sean eficientes y seguros incluso con claves relativamente cortas.

9.2 Cómo se usa en criptografía ?

La idea clave de ECC es que es fácil multiplicar un punto por un número entero (operación repetida de suma de puntos), pero es prácticamente imposible encontrar el número entero si sólo se conoce el resultado. Este es el **problema del logaritmo discreto sobre curvas elípticas**, que garantiza la seguridad de ECC.
Supongamos que hay un punto P en la curva:

- Alice elige un número secreto k_A y calcula $Q_A = k_A P$.
- Bob elige un número secreto k_B y calcula $Q_B = k_B P$.

Para generar una clave compartida:

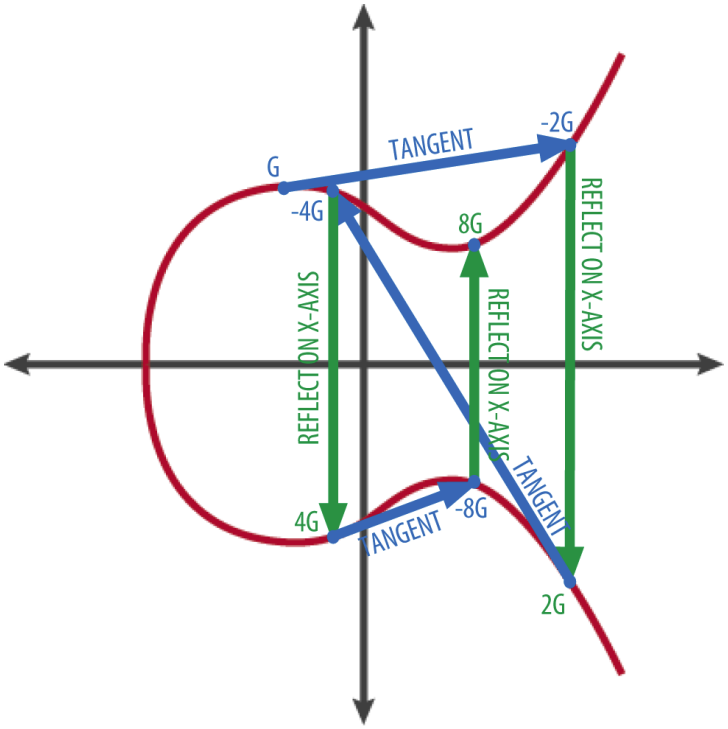
- Alice calcula $k_A Q_B = k_A(k_B P)$.
- Bob calcula $k_B Q_A = k_B(k_A P)$.

Ambos obtienen el mismo punto $k_A k_B P$, que puede convertirse en una clave simétrica para cifrar mensajes. Este procedimiento se llama intercambio de claves basado en ECC, y es conceptualmente equivalente a Diffie-Hellman, pero utilizando operaciones sobre curvas elípticas en lugar de multiplicaciones módulo primo.

Ventajas de ECC

- **Claves más cortas:** Una clave de 256 bits en ECC ofrece un nivel de seguridad comparable a una clave de 3072 bits en RSA, reduciendo tiempo de cálculo y requerimientos de almacenamiento.
- **Eficiencia:** ECC es muy adecuado para dispositivos con recursos limitados, como teléfonos móviles, tarjetas inteligentes y sistemas embebidos.
- **Versatilidad:** ECC se puede usar para cifrado, firma digital y establecimiento de claves compartidas, ofreciendo confidencialidad, integridad y autenticación.
- **Seguridad:** La dificultad del problema del logaritmo discreto en curvas elípticas hace que ECC sea resistente a ataques de fuerza bruta y criptoanálisis directo, incluso con claves relativamente pequeñas.

9.2.1 Esquema



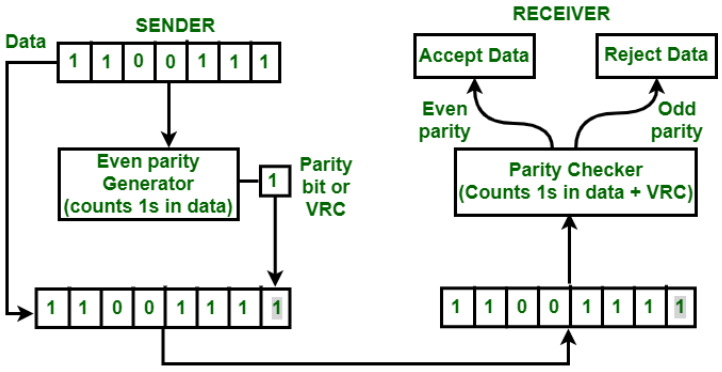
Checksums: detección de errores en datos Los **checksums** son valores de verificación que se utilizan para detectar errores en la transmisión o almacenamiento de datos. Funcionan como un mecanismo simple que permite verificar si los datos han sido alterados o corrompidos, sin necesidad de técnicas criptográficas complejas.

9.3 Bit de paridad

El **bit de paridad** es un dígito binario añadido a un conjunto de bits con el propósito de indicar si la cantidad de bits en 1 es *par o impar*. Esta técnica permite detectar errores simples, como un solo bit invertido.

- Por ejemplo, en **DES**, cada clave de 8 bytes reserva el 8º bit de cada byte para paridad.
- Esto significa que de los 64 bits totales, 56 se usan efectivamente para la clave, y los 8 restantes sirven para controlar errores simples durante la manipulación de la clave.

Esquema



9.4 Cyclic Redundancy Check (CRC)

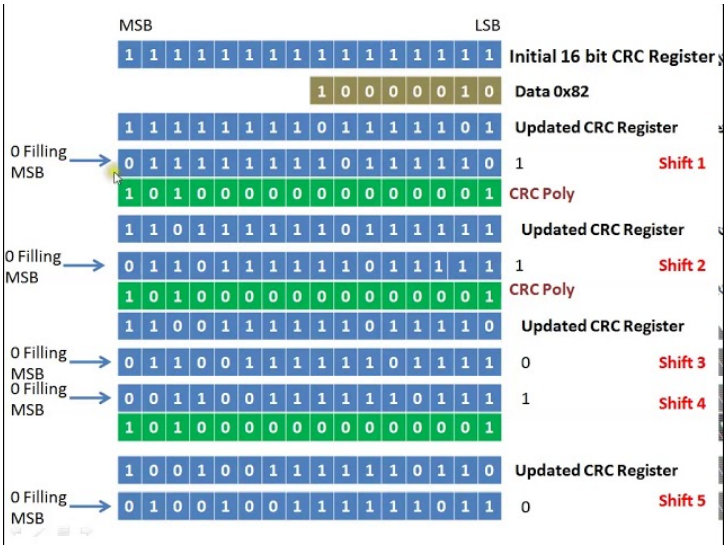
Los **CRC** son un método más robusto que el bit de paridad. Se basan en operaciones sobre polinomios: el mensaje se interpreta como un polinomio y se divide entre un polinomio generador $g(x)$. El *resto* de esta división se utiliza como código de verificación.

- Por ejemplo, un **CRC-16** puede usar el polinomio generador:

$$g(x) = 1 + x^2 + x^{15} + x^{16}$$

- Al transmitir el mensaje, el remitente añade este resto; al recibirlo, el receptor realiza la misma operación y verifica si el resto coincide, indicando que los datos no han sido alterados.

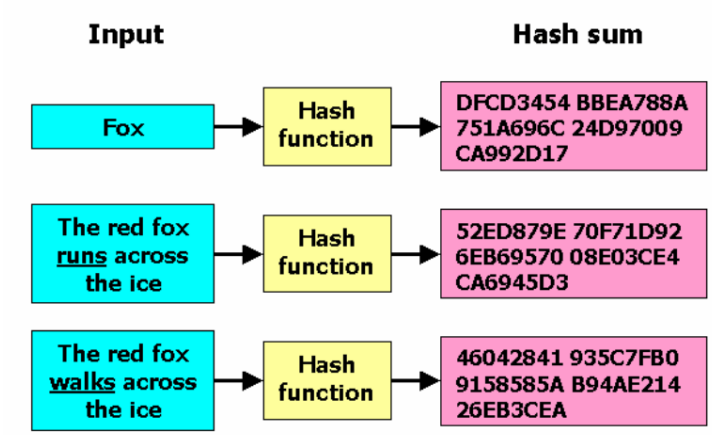
Esquema



En resumen, tanto los bits de paridad como los CRC son mecanismos fundamentales para asegurar la integridad básica de los datos, antes de aplicar técnicas criptográficas más avanzadas.

10 Funciones de Hashing

Las **funciones de hashing** son algoritmos que transforman un mensaje de entrada M, de longitud variable, en una cadena de salida $H(M)$ de longitud fija. Deben ser fáciles de computar en una dirección: dado un mensaje, calcular su hash es rápido, mientras que invertir este proceso o encontrar colisiones debe ser extremadamente difícil.

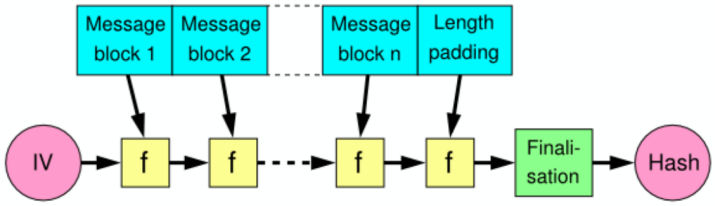


10.1 Propiedades de las funciones de hashing criptográficas

Las funciones de hashing criptográficas poseen propiedades especiales que las hacen útiles para la seguridad informática:

- **Resistencia a preimágenes:** dado un valor z, es computacionalmente no factible hallar un mensaje x tal que $h(x) = z$. Esto asegura que, conociendo un hash, no se pueda deducir el mensaje original.
- **Resistencia a segundas preimágenes:** dado un mensaje x, es prácticamente imposible encontrar otro mensaje $x' \neq x$ que produzca el mismo hash: $h(x) = h(x')$. Esto protege contra la suplantación de mensajes.
- **Resistencia a colisiones:** es computacionalmente no factible hallar dos mensajes distintos x y x' que generen el mismo hash. Si una función cumple esta propiedad junto con las anteriores, se denomina **función fuerte de una vía** (Collision-Resistant Hash Function, CRHF). Si solo cumple la resistencia a preimágenes y segundas preimágenes, se llama **función débil de una vía** (One-Way Hash Function, OWHF).

La longitud típica de salida de estas funciones suele ser de 128, 160, 256 o 512 bits. Entre las funciones más utilizadas se encuentran: MD5, SHA, SHA-1, RIPEMD160, SHA-256, SHA-512 y SHA-3. Muchos de estos algoritmos siguen la construcción de **Merkle–Damgård**, como MD5, SHA-1 y SHA-2.



Merkle–Damgård hash construction: MD5, SHA1, SHA2.

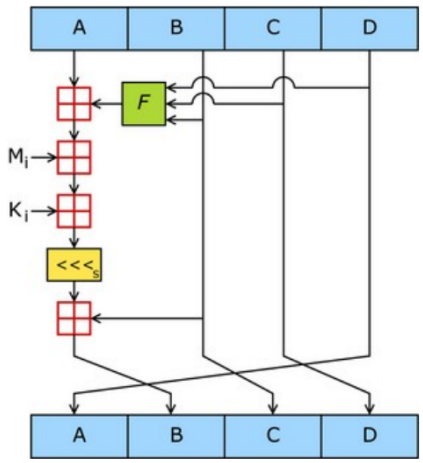
10.2 MD5: Modo de operación

MD5 es un algoritmo de hashing que transforma un mensaje de longitud variable en un hash de 128 bits. Su funcionamiento se organiza en varios pasos:

- El mensaje se divide en bloques de 512 bits.
- El último bloque se completa mediante relleno: se agrega un bit '1' y tantos ceros como sean necesarios, dejando 64 bits al final para la longitud original del mensaje.
- El algoritmo opera sobre un estado de 128 bits, dividido en cuatro palabras de 32 bits: A, B, C, D. Estas palabras se inicializan con valores constantes.
- Cada bloque de 512 bits modifica el estado a través de **4 rondas**, cada una compuesta por **16 operaciones** similares:
  - Suma módulo 2^{32}
  - Rotación a izquierda de s bits
  - Aplicación de una función no lineal F, diferente en cada ronda
- En cada operación, M_i representa un bloque de 32 bits del mensaje de entrada y K_i es una constante de 32 bits específica para esa operación.

Gracias a esta estructura, incluso cambios mínimos en el mensaje producen un hash completamente diferente, garantizando integridad y detección de manipulaciones. Para más detalles técnicos, se puede consultar el estándar RFC 1321 que describe MD5.

Esquema



$$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$$
$$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$$
$$H(X,Y,Z) = X \oplus Y \oplus Z$$
$$I(X,Y,Z) = Y \oplus (X \vee \neg Z)$$

11 Confianza en funciones de hash

En la actualidad, no todas las funciones de hash se consideran seguras. Algoritmos como MD5 y SHA-1 ya no son recomendables para nuevas aplicaciones, debido a vulnerabilidades que permiten ataques de colisión; un ejemplo de estas vulnerabilidades puede consultarse en https://shattered.io/. Por ello, es habitual usar variantes más seguras de la familia SHA-2, como SHA-256, SHA-384 o SHA-512, que ofrecen mayor resistencia frente a ataques modernos. Más recientemente, tras una competencia internacional para elegir un nuevo estándar de hash, se seleccionó la función Keccak como SHA-3, establecida oficialmente en agosto de 2015 mediante el estándar FIPS 202 (http://keccak.noekeon.org/). SHA-3 representa una alternativa robusta y moderna para garantizar la integridad de la información.

11.1 Usos y mal usos de funciones de hash

Las funciones de hash son herramientas muy útiles cuando se emplean correctamente. Su objetivo principal es verificar la integridad de los datos: si se distribuye de forma segura el valor hash $H(x)$ y se recibe un valor x' por un canal inseguro, se puede comprobar que x' es efectivamente igual a x sin revelar información adicional.
No obstante, existen formas incorrectas de utilizarlas:

- Distribuir $H(x)$ y x por el mismo canal inseguro. En este caso, un atacante podría modificar tanto x como $H(x)$, comprometiendo la verificación.
- Usar $H(x)$ como una firma digital, ya que cualquier persona que conozca x puede calcular $H(x)$ y hacerse pasar por el remitente legítimo.

El uso adecuado de funciones de hash requiere entender sus limitaciones y elegir algoritmos seguros según el contexto, garantizando la integridad y la confianza en la información transmitida.

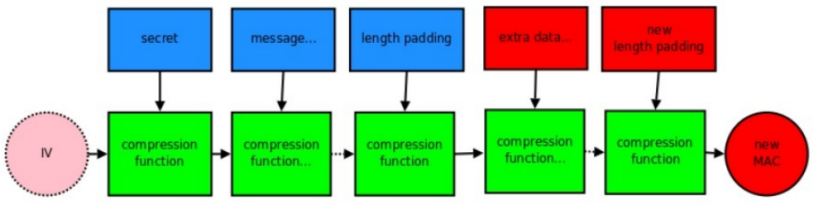
12 HMAC (Hash-based Message Authentication Code)

Las funciones de hash tradicionales, como MD5 y SHA-1, no fueron diseñadas para la autenticación, ya que carecen de una clave secreta. Esto significa que cualquiera puede calcular el hash de un mensaje, pero no garantiza que el mensaje provenga del remitente legítimo ni que no haya sido alterado. Para resolver esta limitación, se utiliza un **Message Authentication Code (MAC)**, que emplea una clave secreta para mapear mensajes de cualquier tamaño a salidas de longitud fija de forma que sea extremadamente difícil generar un par válido $(x, MAC_k(x))$ sin conocer la clave, incluso si se han visto pares válidos previamente.

A diferencia de una función de hash tradicional, conocer $MAC_k(x)$ no permite calcular $MAC_k(y)$ para otro mensaje y. Combinando una función de hash con una clave secreta se obtiene un **HMAC**, que se puede usar para detectar manipulación de contenido. HMAC es muy rápido y eficiente, lo que lo hace ideal para sistemas que requieren autenticación de mensajes en tiempo real.

Vulnerabilidades y construcción segura

No todas las funciones de hash son igualmente seguras para construir HMAC. Funciones que utilizan la construcción Merkle–Damgård, como MD5 y SHA-1, son vulnerables a ataques de extensión de longitud (*Length Extension Attack*),



mientras que SHA-3 no presenta esta vulnerabilidad. Una solución para estas funciones vulnerables es aplicar un doble hashing, o mejor aún, usar HMAC directamente:

$$HMAC_k(m) = h\big((k \oplus \text{opad}) \,\|\, h((k \oplus \text{ipad}) \,\|\, m)\big)$$

donde opad $= 0x5c5c5c\ldots5c$ y ipad $= 0x363636\ldots36$. Ejemplos de HMAC incluyen HMAC-MD5 y HMAC-SHA1, ampliamente utilizados en protocolos de seguridad como SSL e IPSEC para detectar cambios (RFC 2104).

Usos correctos e incorrectos

El uso correcto de HMAC garantiza que no existan dos mensajes diferentes que generen la misma entrada protegida por el HMAC. Por ejemplo, en AWS Signature V1, la firma de un Query String HTTP se realiza de la siguiente manera:

1. Partir el query string en pares clave-valor usando los caracteres '&' y '='.
2. Ordenar los pares según las claves.
3. Concatenar: $key_1 + value_1 + key_2 + value_2 + \ldots$
4. Calcular HMAC-SHA1 de la concatenación.

Más detalles sobre la inseguridad de la versión V1 pueden consultarse en este enlace. Además, es importante considerar ataques por canales laterales relacionados con el tiempo: si la función de comparación de la firma aborta al detectar el primer byte diferente, un atacante puede adivinar iterativamente los bytes de la firma.

No reinventar la rueda

No se debe implementar HMAC de forma improvisada. Por ejemplo:

- Usar $HMAC_k(m) = h(k\|m)$ es vulnerable a *Length Extension Attack* (véase la API de Flickr: Flickr API Signature Forgery).
- Usar $HMAC_k(m) = h(m\|k)$ evita el ataque de extensión de longitud, pero aún puede ser vulnerable si se encuentra una colisión en la función hash subyacente. Por ejemplo, en MD5, generar una colisión de prefijo elegido puede tomar menos de un minuto.

En resumen, HMAC combina la velocidad de las funciones de hash con la seguridad de una clave secreta, proporcionando autenticación y protección contra manipulación de datos de manera eficiente y confiable.

Esquema