

Cifrados... ¿secuenciales? (Stream ciphers)

Los cifrados simétricos pueden dividirse en 2:

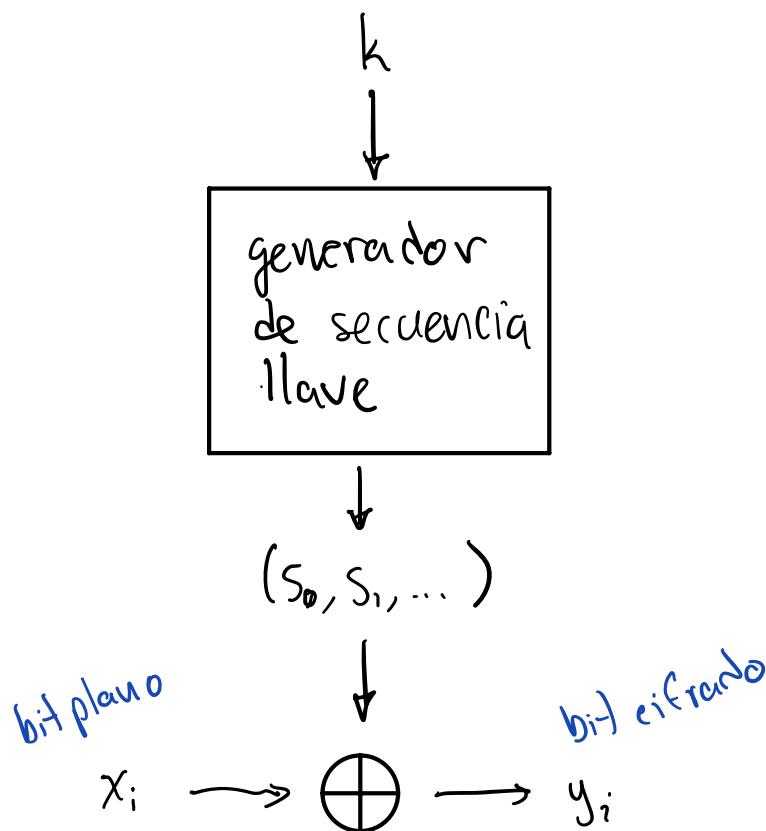
- Cifrado en bloque
- Cifrado en secuencia

Los cifrados de bloque cifran bloques enteros de texto de un solo jalón con la misma llave. Se puede pensar que el cifrado de un bit de texto plano **no es independiente** del cifrado del resto de bits en el mismo bloque

- En la práctica, la mayoría de los cifrados bloque tienen longitudes de bloque de 128 bits o 64 bits

Por otro lado, los cifrados en secuencia cifran **cada bit individualmente**, haciendo que el cifrado de un bit pueda ser independiente del cifrado de los otros bits. Esto se logra usando una **secuencia llave** en vez de una sola llave. Dicho esto, puede ser que haya una

sola llave k que es alimentada a una función generadora de secuencia llave



- Observaciones:
- 1) En práctica, en particular para la comunicación por Internet, es más común el uso de cifrados bloques
 - 2) Los cifrados en secuencia se usan más para operaciones de bajos recursos, pues tienden a ser más rápidos y pequeños. Aún así, llegan a ser usados para el cifrado de tráfico en Internet e.g. RC4

3) ¿Eficiencia?

Cifrado y descifrado

El proceso se realiza bit por bit: Sea x_i un bit de texto plano y s_i un bit de la secuencia llave. Luego, el bit de texto cifrado y_i se obtiene como $y_i = x_i + s_i \text{ mod } 2$



Ejemplo: $X = (10110001)$,

$$S = (01010101)$$

$$\Rightarrow Y = (11100100)$$

Observaciones: 1) La función de cifrado es la misma que la función de descifrado.

2) La suma mod 2 puede ser tratada como la función Booleana XOR (o exclusivo en inglés): La

tabla de verdad del XOR es la tabla de operación de $\mathbb{Z}/2\mathbb{Z}$

2.1) ¿Por qué usar XOR y no otra función Booleana?

Dado un x_i , hay prob. $\frac{1}{2}$ de que $y_i = 0$ y prob. $\frac{1}{2}$ de que $y_i = 1 \Rightarrow$ XOR está perfectamente balanceado... a diferencia de OR.

AND y NAND (y ni AND ni NAND) son invertibles)

3) La generación de la secuencia llave es clave para la seguridad del cifrado. Más aún, toda la seguridad del cifrado recae en la secuencia llave.

Por ende, la generación de la misma es de lo que tratan estos cífrados realmente: entre más "aleatorios" parezcan las secuencias, mejor.

4) En 1917 Gilbert Vernam inventa estos cífrados

Números aleatorios

Dada la importancia de la generación de secuencias llave, es de gran importancia el poder crear números aleatorios. La generación de estos números se puede dividir en 3 tipos:

True Random Number Generators (TRNG)

Son caracterizados por el hecho de que su output no puede ser reproducido. Ejemplo: Si se tiran 100 monedas, es virtualmente imposible volver a tirar las 100 monedas y obtener el mismo resultado.

Los TRNG's se basan en procesos físicos como:

- Tirar monedas
- Tirar dados
- Ruido blanco generado por semiconductores
- Decaimiento radioactivo.

Los TRNG's se usan más en la criptografía para otros procesos

Pseudorandom Number Generators (PRNG)

Los PRNG's computan sucesiones de números a partir de un valor (valores) inicial, y usualmente se obtienen de forma recursiva, es decir, se es "dado" y luego

$$H_i > 0 \quad s_i = f(s_{i-1})$$

Una generalización de esto último es cuando hay un $t \in \mathbb{Z}_+$ y $s_i = f(s_{i-1}, \dots, s_{i-t})$.

Ejemplo: Dados $A, B, M \in \mathbb{Z}$ fijos, se define

$$s_0 = \text{semilla} \quad s_i = As_{i-1} + B \pmod{M}$$

Observación: Los PRNG's no son en verdad aleatorios y en realidad son deterministas.

Tomar el ejemplo anterior y hacer $A = 1103515245$, $B = 12345$ y $M = 2^{31}$ es como funciona la función `rand()` en ANSI C

Lo que caracteriza a los PRNG's es que si bien son deterministas, sí poseen propiedades estadísticas cercanas a la verdadera aleatoriedad. Todo esto es medible.

El problema con los PRNGs es que al tener tantos usos fuera de la criptografía hacen que sí puedan ser predecibles, es decir existe un $N \geq 0$ tq si se conocen $s_i, s_{i+1}, \dots, s_{i+N}$ entonces se pueden determinar los siguientes s_{i+N+1}, \dots

Cryptographically Secure Random Number Generator (CSPRNG)

Estos son casos específicos de PRNGs que son "impredecibles".
Informalmente: Dados N bits s_i, \dots, s_{i+N} no es computablemente realizable el obtener los bits s_{i+N+1}, \dots

Más formalmente: Dados N bits consecutivos no existe un algoritmo que en tiempo polinomial pueda predecir el $N+1$ bit con una tasa de éxito mayor al 50%. Más aún, deben ser computablemente imposible calcular los bits precedentes también.

¿Por qué la seguridad depende de la aleatoriedad?

Dado un carácter de texto plano hay 256 formas posibles para cifrarlo:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
$s_1 \downarrow$	$s_2 \downarrow$	$s_3 \downarrow$	$s_4 \downarrow$	$s_5 \downarrow$	$s_6 \downarrow$	$s_7 \downarrow$	$s_8 \downarrow$
y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8

$$2^8 = 256$$

Uno podría pensar que entonces al repetir el carácter $N \geq 257$ veces tendríamos suficientes caracteres repetidos como para aplicar un ataque estadístico.

Sin embargo, como los bits se cifran de forma independiente la misma secuencia de 8 bits consecutivos puede dar un carácter en un punto del texto y otro carácter completamente diferente en otra parte del texto.

Entonces, siempre que (s_0, \dots) sea "suficientemente" aleatorio, la revisión de repeticiones no es un ataque efectivo.

OTP

Un criptosistema es **incondicionalmente seguro / informáticamente teóricamente seguro** si no puede romperse incluso con recursos

computacionales infinitos.

Para entender la dificultad de esto, considérese un criptosistema con llave k de longitud 10,000 bits tq la única forma de romperlo es fuerza bruta.

¿Es incondicionalmente seguro?

¡No! Un atacante podría tener 2^{10000} computadoras tales que cada una revise una posible llave \Rightarrow De un julen se obtiene la llave necesaria.

\Rightarrow El criptosistema es computacionalmente seguro, pero no incondicionalmente.

Definición: Un cifrado en secuencia tq

- 1) la secuencia llave s_0, s_1, \dots , es generada por un TRNG,
- 2) la llave es conocida solo a las partes que se van a comunicar,

- 3) cada secuencia llave (s_i) es usada una única vez, es llamado un **one-time pad**

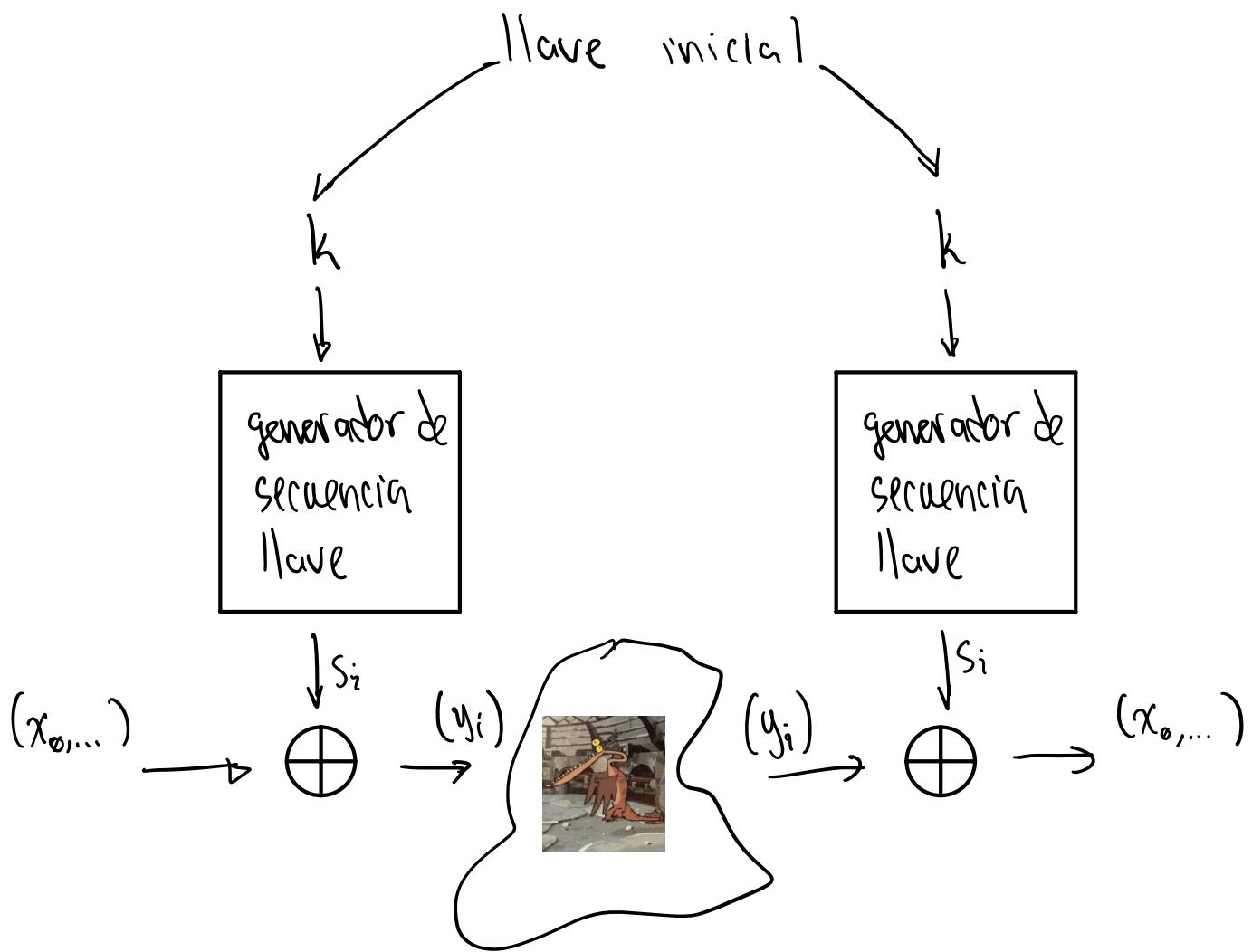
Los OTPs son incondicionalmente seguros. ¿Por qué?

¿Cuáles son las implicaciones de un OTP?

- 1) Se necesita un dispositivo capaz de crear un TRNG.
- 2) La primera persona debe comunicar los s_i de forma segura a la segunda persona.
- 3) Los s_i solo pueden ser usados una vez.

Esto hace que los OTPs no sean tan prácticos.

En camino a cifrados secuenciales prácticos



Definición: Un criptosistema es computacionalmente seguro si el mejor algoritmo conocido para romperlo requiere al menos t operaciones.

Observación: Esto no es algo definitivo: RSA.

Generación de secuencias llave

El problema con PRNGs es que se pueden romper de diferentes formas:

$$\text{PRNG: } S_0 = \text{semilla} \quad \text{con } m \text{ de 100 bits}$$
$$S_{i+1} \equiv AS_i + B \pmod{m} \quad \text{público}$$

$$\Rightarrow S_i, A, B \in \{0, \dots, m\}$$

→ El secreto es A, B y S_0 ... a lo más 300 bits

→ Alice tiene (x_0, \dots) → $y_i \equiv x_i + s_i \pmod{2}$
con s_i los bits de S_i en base 2.

Problema: Si el atacante conoce 300 bits de texto plano y su correspondiente texto cifrado, ent. conoce que

$$s_i \equiv y_i + x_i \pmod{m}$$

$$\Rightarrow S_1 = (s_1, \dots, s_{100}), S_2 = (s_{101}, \dots, s_{200}), S_3 = (s_{201}, \dots, s_{300})$$

$$\rightsquigarrow S_2 \equiv AS_1 + B \pmod{m}$$

$$S_3 \equiv AS_2 + B \pmod{m}$$

$$\Rightarrow A \equiv \frac{(S_2 - S_3)}{(S_1 - S_2)} \pmod{m}$$

$$B \equiv S_2 - \frac{S_1(S_2 - S_3)}{(S_1 - S_2)} \pmod{m}$$

\Rightarrow Hay finitas ($< m$) posibles soluciones ✓

Seguridad de DES

Hay en esencia dos tipos de ataques:

- 1) Búsqueda exhaustiva de llaves (Fuerza bruta)
- 2) Ataques analíticos

Como bien lo describe el nombre, para (1) la idea es aplicar fuerza bruta para encontrar la llave.

Por otro lado, para (2) es necesario entender el funcionamiento del criptosistema para poder aplicar ataques especialmente diseñados.

Tras la propuesta de DES, inmediatamente saltaron a la vista dos problemas:

- 1) El espacio de llaves es muy pequeño, entonces es vulnerable a fuerza bruta
- 2) Las S-cajas se mantuvieron en secreto, llevando a la idea de que pudieron haber ataques/propiiedades matemáticas efectivas para romper DES que solo conocen

los diseñadores de DES

Hasta la fecha del libro, es más fácil y eficiente aplicar fuerza bruta y romper DES en cuestión de horas/días

Búsqueda exhaustiva de llaves

Algo a destacar es que IBM propuso una longitud de llaves de 128 bits, lo cual fue sospechosamente reducido a 56 bits. La motivación oficial es sospechosa.

Ahora, ¿a qué nos referimos con búsqueda exhaustiva de llaves?

Definición (3.5.1) Búsqueda exhaustiva de llaves:

Input: Una pareja (x, y) de texto plano x y texto cifrado y .

Output: Llave k tq $DES_k(x) = y$

Ataque: Probar las 2^{56} posibles llaves.

Observación: ¡Se puede encontrar una llave incorrecta!
(Probabilidad de $\frac{1}{2^{56}}$)

Una computadora regular (2010) no es realmente apta para la búsqueda exhaustiva, pero se pueden hacer máquinas especializadas (a un precio).

1977 Whitfield Diffie, Martin Hellman

Propuesta no construida

20 millones de 1977-USD $\rightsquigarrow \sim 104.2$ millones de USD

Tiempo no especificado

1993 Michael Wiener

Propuesta no construida

1 millón de 1993-USD $\rightsquigarrow \sim 2.16$ millones de USD

Tiempo de ~ 1.5 días.

1998 Electronic Frontier Foundation

Máquina Deep Crack (~ 1800 c.i. de 24 test c/u)

$< 250,000$ 1998-USD $\rightsquigarrow \sim 485,000$ USD

Tiempo récord de 56 h pero promedio de 15 días

2006 Universidades de Bochum y Kiel (Alemania)

COPACOBANA (Cost-Optimized Parallel Code-Breaker)

$\sim 10,000$ 2006 USD $\rightarrow \sim 15,700$ USD

Tiempo promedio 7 días

\Rightarrow 56 bits es una longitud de clave muy chira.

\Rightarrow DES no es más "efectivo" para usos cortos (horas) o cuando los datos encriptados son de bajo valor.

Ataques analíticos

En 1990 Eli Biham y Adi Shamir descubren el criptoanálisis diferencial (DC)

En 1993 Mitsuru Matsui publica el ataque llamado criptoanálisis lineal (LC)

En ambos casos, estos son ataques en general aplicables (y eficientes) para cifrados de bloque.

Sin embargo, estos no son eficientes para DES. En el caso de DC, al parecer por diseño.

DC $\rightarrow 2^{47}$ parejas conocidas escogidas o 2^{55} aleatorias

LC $\rightarrow 2^{43}$ parejas escogidas

¿Por qué no es práctico?

- 1) Números muy grandes
- 2) Toma mucho tiempo recolectar estas parejas
- 3) Tras todo esto se obtiene una sola llave.

Resumen:

1990 Biham-Shamir DC $\sim 2^{47}$ escogidas

1993 Matsui LC $\sim 2^{43}$ escogidas

Implementaciones en Software y Hardware

Software: Implementaciones en computadoras de escritorio, celulares, etc.

Hardware: Implementaciones en circuitos integrados e.g. ASIC
o FPGA.

Software

Una implementación directa es poco práctica por las permutaciones involucradas, las cuales son lentas en software

Además, las S-rajas pequeñas si bien son prácticas en hard

ware, sólo son moderadamente eficientes CPU's (2010). Para "darle la vuelta" a esto se suelen usar tablas de las operaciones DES

Una implementación optimizada requiere aprox. 240 ciclos por bloque en un CPU de 32-bits. Tomando un CPU de 2GHz \rightarrow 533 Mbits/s

Sin optimizar nos deja alrededor de 100 Mbits/s.

Eli Brigham desarrolló bit slicing, lo cual puede aplicarse para mejorar considerablemente el desempeño. Sin embargo, tiene desventajas dependiendo de la aplicación.

Hardware

Uno de los criterios de DES era la eficiencia de implementación por hardware. Las permutaciones implementadas son fáciles en hardware a través de cableado sin necesidad de funciones lógicas. Luego, las S-rajitas pequeñas son fáciles de implementar con lógica Booleana. Una implementación eficiente por área se puede hacer con menos de 30000 puertas. Caben en chips de identifica-

ción de frecuencias de radio).

En ASIC y FPGA modernos (2010) se puede realizar con eficiencia de 100 Gbits/s

Advanced Encryption Standard

El AES es el cifrado simétrico más usado actualmente (2010), e incluso si el "estándar" es nado más en EUA, se usa ampliamente en otras industrias y sistemas comerciales (Estándares de seguridad en Internet IPSec, TLS IEEE 802.11i, etc.).

Introducción

En 1999 el Instituto Nacional de Estándares y Tecnología de EUA (NIST) dijo que el DES ya nado más para lo viejito, y que hay que mejor usar el 3DES.

- ¡Problema! • El DES y el 3DES son MUY lentos en implementación por software.
- También, el tamaño de la llave es muy chico para ciertas aplicaciones (funciones hash).

Entonces el NIST hizo un llamado a nuevos estándares.

- 1997 (enero) NIST hace el llamado.
- 1997 (septiembre) Idem (pero formalmente)

Requerimientos:

- Cifrado de bloque con tamaño de bloques de 128 bits.
- Tamaño de llaves variable de 128, 192 y 256 bits.
- Seguridad relativa a otros algoritmos sometidos.
- Eficiencia en hardware y software

- 1998 (agosto) 15 candidatos

- 1999 (agosto) 5 finalistas:

- Mars (IBM)
- RC6 (RSA)
- Rijndael (Joan Daemen y Vincent Rijmen)
- Serpent (Anderson, Biham y Knudsen)
- Twofish (Schnaeier, Kelsey, Whiting, Wagner, Hall, Ferguson)

- 2000 (octubre) Rijndael es coronado AES.

- 2001 (noviembre) AES es aprobado como estándar federal estadounidense.

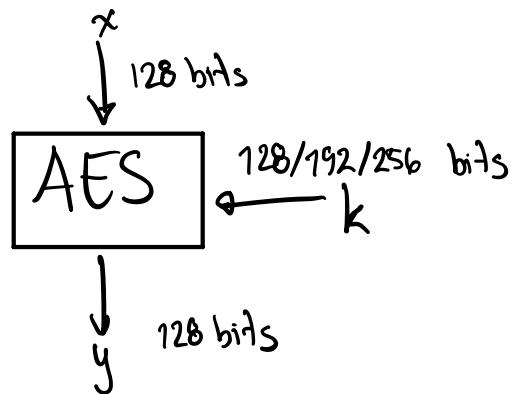
- 2003 La NSA lo declara apto para secretos de estado

para todo tipo de llaves para secreto de nivel normal y de llaves de 192 y 256 bits para secretos de alto nivel.

Vistazo rápido de AES

AES casi no cambió con respecto a Rijndael, aunque Rijndael acepta llaves de 128, 192 y 256 bits, mientras que AES oficialmente es de 128 bits. Esto último es lo que discutiremos.

Parámetros de AES:



El número de "rondas" por iteración depende de la longitud de llaves ($128 \mapsto 10, 192 \mapsto 12, 256 \mapsto 14$)

A diferencia de DES, AES cifra todos los 128 bits de un jalón por iteración

Como tal, AES consiste de 3 capas (por ronda... así) las cuales manipulan los 128 bits por capa:

(Capa de adición de llave: se suma XORicamente una llave/subllave de 128 bits.

(Capa de sustitución de bytes: cada elemento del estado (de los datos) es transformado no linealmente usando tablas. Esto agrega confusión (cambios individuales se propagan)

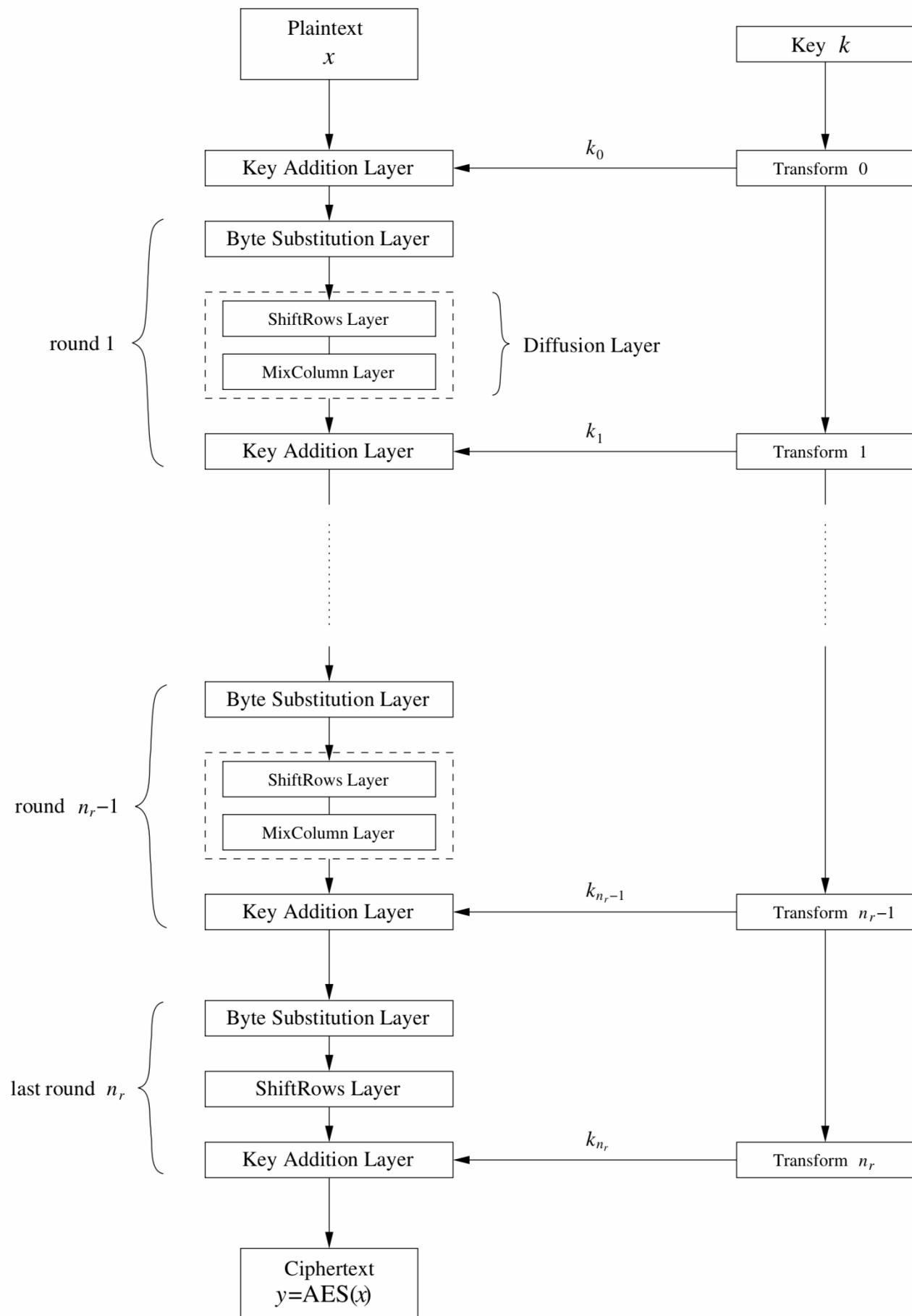
(Capa de difusión: consiste de dos subcapas:

—Subcapa de desplazamiento de columnas

(Shift Rows): permuta los datos a nivel bytes.

—Subcapa de mezclado de columnas

(Mix Column): es una operación matricial que combina/mezcla bloques de 4 bytes.

**Fig. 4.2** AES encryption block diagram

Conceptos Matemáticos

Grupos ✓

Campos ✓

Teorema 4.3.1 Un campo de orden m existe sólo si m es una potencia de un primo.

Denotamos por $GF(p)$ al campo de orden p . Como $GF(p)$ tiene tablas de operaciones explícitas y finitas, estas pueden usarse para facilitar/agilizar procesos.

La tabla de suma en $GF(2)$ es XOR.

La tabla de multiplicación en $GF(2)$ es AND.

En AES se usa $GF(2^8)$. Esto es especial porque cada elemento de $GF(2^8)$ se puede representar usando un solo byte.

Para la S-Box y las transformaciones MixColumn, AES usa cada byte de los datos internos como un elemento de

$GF(2^8)$ y usa la aritmética de $GF(2^8)$ para manipular los datos.

(Como 2^8 no es primo, $GF(2^8)$ es una extensión de campo, por lo que sus elementos los trataremos como polinomios en $GF(2)$ de grado a lo más 7 (\Rightarrow hay 8 coeficientes, no todos no cero, en cada elemento)):

$$\forall A \in GF(2^8) \quad A(x) = a_7x^7 + \dots + a_1x + a_0 \quad (\text{con } a_i \in GF(2))$$

$$\Rightarrow GF(2^8) \xrightarrow{\psi} GF(2)^8$$

ψ es un isomorfismo de grupos abelianos.

\Rightarrow la adición en $GF(2^8)$ es hacer XOR en cada entrada del byte.

Para la multiplicación, lo que hacemos es multiplicar los polinomios y reducir módulo el polinomio irreducible

$$P(x) = x^8 + x^4 + x^3 + x + 1$$

Por ejemplo: $x^8 \equiv 1 \cdot P(x) + (x^4 + x^3 + x + 1) \pmod{P(x)}$

$$\Rightarrow x^8 \equiv x^4 + x^3 + x + 1 \pmod{P(x)}$$

$$\text{Sean } f_1(x) = x^7 + x^5 + x^3 + 1 \quad , \quad f_2(x) = x^4 + x^2 + 1$$

$$\Rightarrow f_1(x) \cdot f_2(x) = (x^7 + x^5 + x^3 + 1)(x^4 + x^2 + 1)$$

$$= x^{11} + x^9 + x^7 + x^4 \\ + x^9 + x^7 + x^5 + x^2 \\ + x^7 + x^5 + x^3 + 1$$

$$= x^{11} + x^7 + x^4 + x^3 + x^2 + 1$$

$$= x^3 x^8 + \dots + 1$$

$$= x^3 (x^4 + x^3 + x + 1) + x^7 + \dots + 1$$

$$= x^7 + x^6 + x^4 + x^3 \\ + x^7 + x^4 + x^3 + x^2 + 1$$

$$= x^6 + x^2 + 1$$

$$\Rightarrow (1, 0, 1, 0, 1, 0, 0, 1) \cdot (0, 0, 0, 1, 0, 1, 0, 1)$$

$$\begin{matrix} & & & & & & 11 \\ & & & & & & \\ (0, 1, 0, 0, 0, 1, 0, 1) \end{matrix}$$

Para la inversión de elementos, se usan tablas prefabricadas. En particular, para $\text{GF}(2^8)$ se usa notación hexadecimal y la tabla siguiente

Byte Substitution

Table 4.2 Multiplicative inverse table in $GF(2^8)$ for bytes xy used within the AES S-Box

	Y																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	00	01	8D	F6	CB	52	7B	D1	E8	4F	29	C0	B0	E1	E5	C7	
1	74	B4	AA	4B	99	2B	60	5F	58	3F	FD	CC	FF	40	EE	B2	
2	3A	6E	5A	F1	55	4D	A8	C9	C1	0A	98	15	30	44	A2	C2	
3	2C	45	92	6C	F3	39	66	42	F2	35	20	6F	77	BB	59	19	
4	1D	FE	37	67	2D	31	F5	69	A7	64	AB	13	54	25	E9	09	
5	ED	5C	05	CA	4C	24	87	BF	18	3E	22	F0	51	EC	61	17	
6	16	5E	AF	D3	49	A6	36	43	F4	47	91	DF	33	93	21	3B	
7	79	B7	97	85	10	B5	BA	3C	B6	70	D0	06	A1	FA	81	82	
X	8	83	7E	7F	80	96	73	BE	56	9B	9E	95	D9	F7	02	B9	A4
	9	DE	6A	32	6D	D8	8A	84	72	2A	14	9F	88	F9	DC	89	9A
	A	FB	7C	2E	C3	8F	B8	65	48	26	C8	12	4A	CE	E7	D2	62
	B	0C	E0	1F	EF	11	75	78	71	A5	8E	76	3D	BD	BC	86	57
	C	0B	28	2F	A3	DA	D4	E4	0F	A9	27	53	04	1B	FC	AC	E6
	D	7A	07	AE	63	C5	DB	E2	EA	94	8B	C4	D5	9D	F8	90	6B
	E	B1	0D	D6	EB	C6	0E	CF	AD	08	4E	D7	E3	5D	50	1E	B3
	F	5B	23	38	34	68	46	03	8C	DD	9C	7D	A0	CD	1A	41	1C

Example 4.7. From Table 4.2 the inverse of

Por ejemplo:

$$x^7 + x^6 + x \rightsquigarrow (1, 1, 0, 0, 0, 0, 1, 0)$$

$$\rightsquigarrow 11000010 \text{ base 2}$$

↓

194

base 10

(128 + 64 + 2)

↓

C2

base 16

$$\Rightarrow (C2)^{-1} = 2F \rightsquigarrow 47 \text{ base 10} \rightsquigarrow 101111 \rightsquigarrow (0, 0, 1, 0, 1, 1, 1, 1)$$

$$\Rightarrow (x^7 + x^6 + x)^{-1} = x^5 + x^3 + x^2 + x + 1$$

Y lo podemos verificar:

$$(x^7 + x^6 + x) (x^5 + x^3 + x^2 + x + 1) =$$

||

$$\begin{aligned} x^{12} &+ x^{10} + x^9 + x^8 + x^7 \\ &+ x^9 + x^8 + x^7 + x^6 + x^6 \\ &+ x^6 + x^4 + x^3 + x^2 + x \end{aligned}$$

||

$$\begin{aligned} x^{12} &+ x^{11} + x^{10} + x^9 + x^8 + x^7 + x \\ &+ x^6 + x^4 + x^3 + x^2 + x \end{aligned}$$

||

$$x^4(x^4 + x^3 + x + 1) + x^3(x^4 + x^3 + x + 1) + x^2(x^4 + x^3 + x + 1) + x^4 + x^3 + x^2 + x$$

||

$$\begin{aligned} x^8 &+ x^7 + x^5 + x^4 \\ &+ x^7 + x^4 + x^6 + x^3 \\ &+ x^5 + x^6 + x^3 + x^2 \\ &+ x^4 + x^3 + x^2 + x \end{aligned}$$

||

$$x^8 + x^7 + x^5 + x$$

$$(x^4 + x^3 + x + 1) + x^4 + x^3 + x = 1$$

Dicho esto, hay ocasiones en las que se codifica el proceso de buscar inversas, en vez de usar tablas.

Estructura interna de AES

Dada una cadena de datos de 128 bits, lo primero que se hace es dividir esto en 16 bytes, obteniendo así una cadena $(A_0, A_1, \dots, A_{15}) \in GF(2^8)^{16}$.

El funcionamiento de una ronda es entonces así:

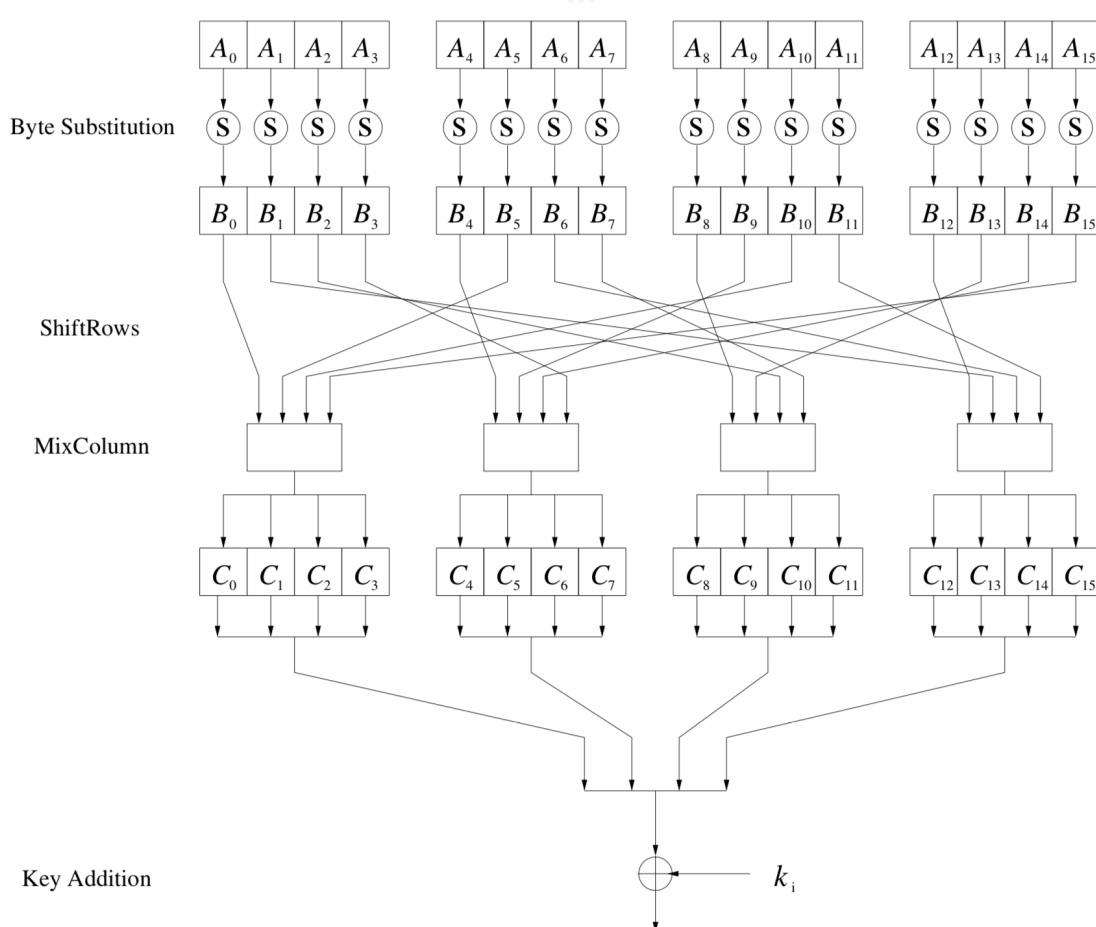


Fig. 4.3 AES round function for rounds $1, 2, \dots, n_r - 1$

Algo que ha sido útil en AES es arcomodar el estado de datos en matrices:

$$(A_0, A_1, \dots, A_{15}) \rightarrow \begin{pmatrix} A_0 & A_4 & A_8 & A_{12} \\ A_1 & A_5 & A_9 & A_{13} \\ A_2 & A_6 & A_{10} & A_{14} \\ A_3 & A_7 & A_{11} & A_{15} \end{pmatrix}$$

La llave de igual forma la ponemos en bytes y luego en matrices de tamaño 4×4 (128 bits), 4×6 (192 bits) o 4×8 (256 bits).

Capa de sustitución

Para la capa de sustitución, en implementaciones de software se usa una función $S(A_i) = B_i$ donde el resultado se busca en una tabla como esta:

AES S-Box: Substitution values in hexadecimal notation for input byte (xy)

		y																
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
x		0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
x		1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
x		2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
x		3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
x		4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
x		5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
x		6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
x		7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
x		8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
x		9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
x		A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
x		B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
x		C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
x		D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
x		E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
x		F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Matemáticamente S es $S(A_i) = F(A_i')$ donde F es una transformación afín

Ejemplo:

$$11000010 \rightsquigarrow 00101111$$

$$\begin{matrix} \\ \\ C_2 \end{matrix}$$

$$\begin{matrix} \\ \\ 2F = (C_2)^{-1} \end{matrix}$$

$$T \in GL(8, \mathbb{Z}/2\mathbb{Z}), \quad v \in GF(2^8)$$

$$S(C_2) = T \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} + v$$

Observación: Teniendo el byte con notación binaria, para pasarlo a vector se lee de derecha a izquierda.

Como se usa inversión del campo se rompe la linealidad, haciendo no efectivas muchas estrategias de criptoanálisis.

Como se usa una transformación afín tampoco se puede sacar ventaja de la inversión del campo.

Capa de difusión

Esta capa se divide en dos, una subcapa es una permutación de las filas, mientras que la otra subcapa es la aplicación de una transformación lineal a las columnas.

Shift Rows

Tras la capa de sustitución la matriz quedó

$$\begin{pmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_1 & B_5 & B_9 & B_{13} \\ B_2 & B_6 & B_{10} & B_{14} \\ B_3 & B_7 & B_{11} & B_{15} \end{pmatrix}$$

En Shift Rows se aplican permutaciones a cada fila.

Fila 1 ~ id Fila 2 ~ (1 4 3 2)

Fila 3 ~ (1 3)(4 2) Fila 4 ~ (1 2 3 4)

Esto nos deja la matriz estando como:

$$\begin{pmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_5 & B_9 & B_{13} & B_1 \\ B_{10} & B_{14} & B_2 & B_6 \\ B_{15} & B_3 & B_7 & B_{11} \end{pmatrix}$$

Mix Column

Aquí se toma una matriz $M \in GL(4, GF(2^8))$ y se fija. Luego se toma cada columna de la última matriz es- tado y se le aplica M

$$M \cdot \begin{pmatrix} B_0 \\ B_5 \\ B_{10} \\ B_{15} \end{pmatrix} = \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix}, \quad M \cdot \begin{pmatrix} B_4 \\ B_9 \\ B_{14} \\ B_3 \end{pmatrix} = \begin{pmatrix} C_4 \\ C_5 \\ C_6 \\ C_7 \end{pmatrix},$$

$$M \cdot \begin{pmatrix} B_8 \\ B_{13} \\ B_2 \\ B_7 \end{pmatrix} = \begin{pmatrix} C_8 \\ C_9 \\ C_{10} \\ C_{11} \end{pmatrix}, \quad M \cdot \begin{pmatrix} B_{12} \\ B_1 \\ B_6 \\ B_{11} \end{pmatrix} = \begin{pmatrix} C_{12} \\ C_{13} \\ C_{14} \\ C_{15} \end{pmatrix}.$$

$$M = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}$$

La nueva matriz de estado es:

$$\begin{pmatrix} C_0 & C_4 & C_8 & C_{12} \\ C_1 & C_5 & C_9 & C_{13} \\ C_2 & C_6 & C_{10} & C_{14} \\ C_3 & C_7 & C_{11} & C_{15} \end{pmatrix}$$

Capa de adición de llave

La matriz estado se regresa a sucesión de bits y se le suma (XORea) la subllave de 128 bits.

La subllave se obtiene de la llave original en el proceso de key schedule.

Key schedule

Para la generación de subllaves el proceso varía un poco dependiendo de la longitud de la llave. De igual forma, el número de rondas también depende de esto.

Recordando la tabla del funcionamiento del AES, siempre se empieza con una adición de subllave y ya después se empiezan las rondas (cada una con una subllave).

Por lo tanto, si se usan n_r rondas, se necesitan $n_r + 1$ subllaves.

Por cierto, la adición de subllave antes de las rondas

se llama key whitening / blanqueado de llave.

Como tal, el proceso de obtención de llave se realiza usando palabras (32 bits = 4 bytes). Las palabras se van guardando porque se obtienen de forma iterativa.

$$l(K) = 128 \text{ bits}$$

$n_r = 10 \Rightarrow$ hay 11 subllaves.

k_0 en este caso no es otra cosa mas que K , pero se divide en bytes y luego esos bytes se agrupan en palabras:

$$K \rightsquigarrow (k_0, k_1, \dots, k_{15})$$

$$\rightsquigarrow W[0] = (k_0, k_1, k_2, k_3)$$

$$W[1] = (k_4, k_5, k_6, k_7)$$

$$W[2] = (k_8, k_9, k_{10}, k_{11})$$

$$W[3] = (k_{12}, k_{13}, k_{14}, k_{15})$$

$$\Rightarrow k_0 = (W[0], W[1], W[2], W[3])$$

Ahora, para $i = 1, \dots, 10$, obtenemos $W[4:i]$ como sigue:

$$W[4i] = W[4i-4] \oplus g(W[4i-1])$$

dónde g es una función no lineal que definiremos más abajo.

Posteriormente tenemos para $j=1, 2, 3$:

$$W[4i+j] = W[4i+j-4] \oplus W[4i+j-1]$$

Dejando así $k_i := (W[4i], W[4i+1], W[4i+2], W[4i+3])$.

Finalmente, para definir g primero definimos $RC(i)$, llamada la constante de i -ésima ronda, como el elemento de $GF(2^8)$ correspondiente al polinomio x^i

$$RC(1) = 00000010, \quad RC(2) = 00000100,$$

$$RC(3) = 00001000, \dots, \quad RC(7) = 10000000,$$

$$RC(8) = 00011011, \quad RC(9) = 00110110$$

Luego, g de una palabra se obtiene primero agrupando en bytes, luego permutando cíclicamente (1 a la izquierda), de ahí aplicando S (de la capa de permutación), luego al byte de la extremidad izquierda se le suma $RC(i)$ y finalmente se agrupan los bytes en una palabra.

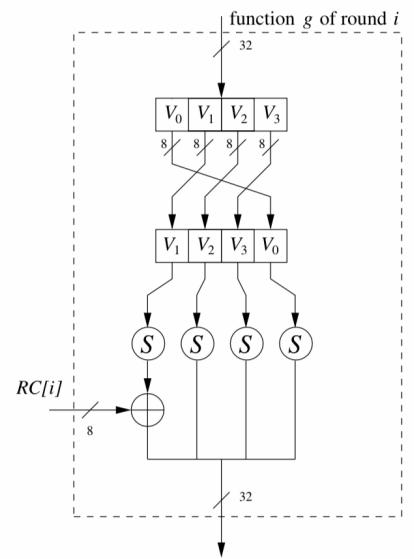
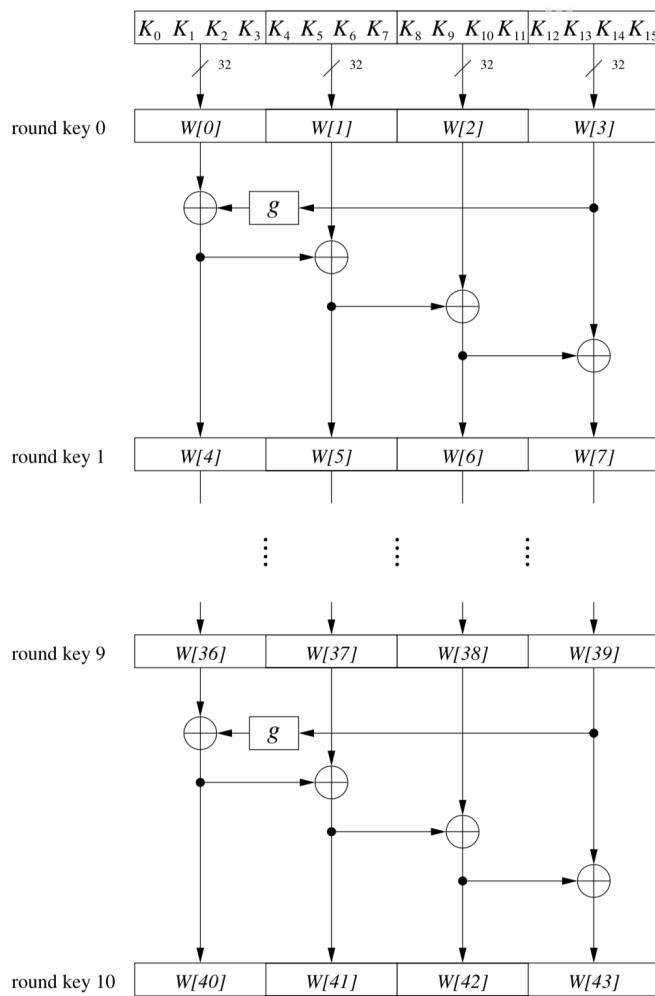


Fig. 4.5 AES key schedule for 128-bit key size

La función g sirve para remover linealidad y simetría.

$$l(K) = 192 \text{ bits}$$

Para este caso $n_r = 12 \Rightarrow$ se necesitan 13 llaves \Rightarrow
 \Rightarrow se necesitan 52 palabras.

El proceso es casi el mismo que para 128 bits pero se hace módulo 6, en vez de módulo 4.

Una vez generadas todas las palabras, se acomodan en grupos de 4 para obtener las llaves.

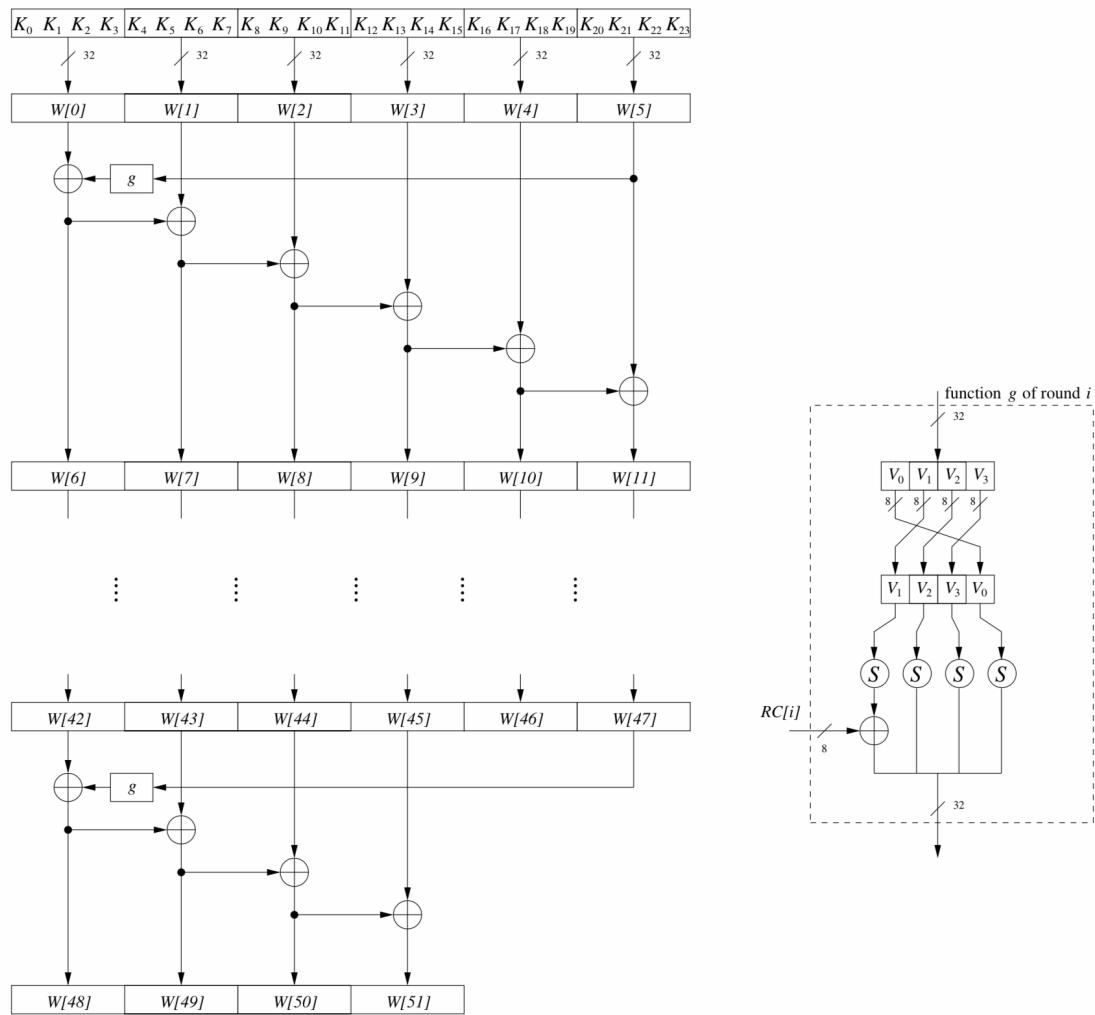


Fig. 4.6 AES key schedule for 1024 bit key sizes

$$l(k) = 256 \text{ bits}$$

Aquí tenemos $n_r = 14 \Rightarrow$ Hay 15 subllaves $\Rightarrow 60$ palabras.

El proceso es casi el mismo que los anteriores pero módulo 8, aunque se agrega una función h (que es aplicación de S a cada byte) cada palabra congruente con 4 módulo 8

Una vez generadas todas las palabras, se acomodan en grupos de 4 para obtener las llaves.

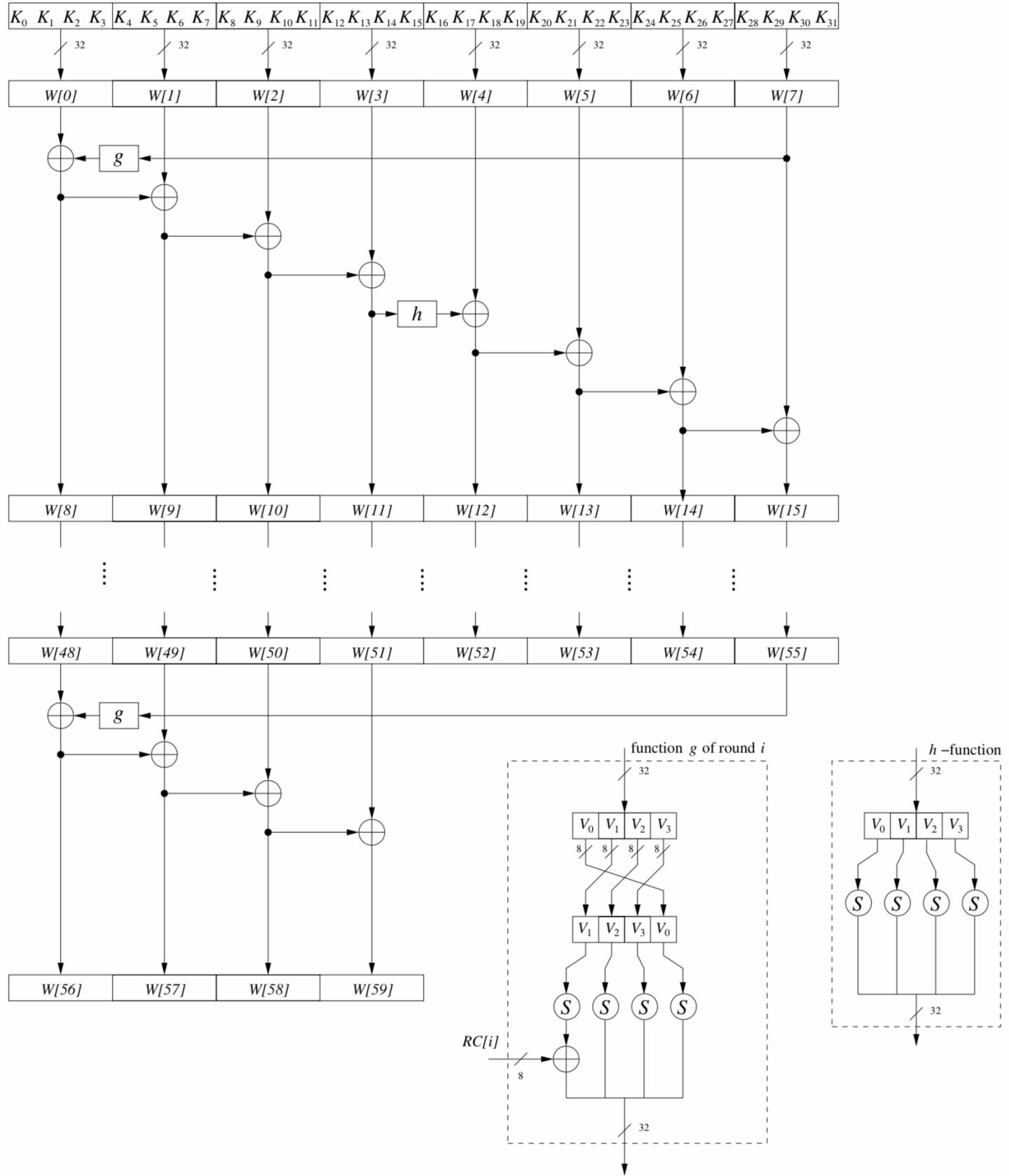


Fig. 4.7 AES key schedule for 256-bit key size

Producción de palabras

Hay en general 2 estrategias para la producción de las palabras:

1) Computación previa: Antes de cifrar o descifrar se computan todas las palabras, se crean las subllaves y se guardan. Esto hace que se necesite una memoria dedicada para esto de al menos $(n_r+1) \cdot 16$ bytes. Esto hace que esta estrategia no sea popular en implementaciones de hardware con limitaciones en memoria.

Esta estrategia se usa cuando una PC o un servidor va a estar cifrando y descifrando con la misma llave varias veces.

2) Al momento: Como lo dice el nombre, las palabras se calculan al momento y conforme se van necesitando para formar las llaves.

Como para el descifrado se necesita la última subllave para empezar, esto hace que para descifrar se calculen todas primero y luego se empieza $\Rightarrow t(\text{Descifrado}) > t(\text{Cifrado})$.

Técnicas de aceleración de RSA

Utilizar exponentiación (incluso usando el método de multiplicación y elevar al cuadrado) es muy costoso computacionalmente. Para esto se desarrollaron técnicas para acelerar este proceso.

Cifrado rápido con llave pública chica

El "truco" aquí es utilizar como llave pública un número "chico" con el que sea necesario hacer pocos cálculos. En particular, si e es la llave pública, se mencionan los valores

	valor base 2	Operaciones
3	11_2	2
17	10001_2	5
$2^{16} + 1$	1000000000000001_2	17

Notemos que estos valores son más fáciles de calcular, pues tienen distancia de Hamming chima.

Ahora, nos podemos preguntar si elegir estos valores tiene el efecto de agilizar el cifrado al costo de sacrificar seguridad. El primero se obtiene, pero el segundo no. En este tipo de protocolos, la seguridad recae en realidad en la llave privada.

Así, el cifrado puede hacerse increíblemente rápido (en comparación a otros protocolos de llave pública).

Decifrado rápido con CRT

Para la llave privada no se puede hacer el mismo truco, pues entonces un atacante podría aplicar fuerza bruta para tener la llave.

De hecho, se recomienda que la longitud en bits de la llave privada sea al menos $0.3t$, donde t es la longitud en bits del módulo n que se utiliza en RSA.

Por lo tanto, en la práctica se usa una llave pública corta y una llave privada larga. Para hacer esto práctico se usa el Teorema Chino de Residuos

(CRT).

El objetivo es: teniendo el texto cifrado y , calcular $y^d \bmod n$, con d la llave privada y n el módulo.

→ Recorremos $n = p \cdot q$

→ Reajustamos $y_p := y \bmod p$

$$y_q := y \bmod q$$

→ Cambiamos exponentes $d_p := d \bmod (p-1)$

$$d_q := d \bmod (q-1)$$

→ Deciframos parcialmente $x_p := y_p^{d_p} \bmod p$

$$x_q := y_q^{d_q} \bmod q$$

Observación: $d_p \leq p$ & $d_q \leq q$

→ Calculamos coeficientes $c_p := q^{-1} \bmod p$

$$c_q := p^{-1} \bmod q$$

→ Decifrado completo

$$x = [q c_p] x_p + [p c_q] x_q \bmod n$$

Ejemplo: $p=11, q=13 \rightsquigarrow n=143; e=7 \rightsquigarrow d=e^{-1}=7^{-1} \bmod 120$

$$\text{Sea } y=15 \rightsquigarrow y_p=15 \bmod 11 \Rightarrow y_p=4$$

$$y_q=15 \bmod 13 \Rightarrow y_q=2$$

103

$$\text{Luego } d_p = 103 \bmod 10 = 3 \bmod 10$$

$$d_q = 103 \bmod 12 = 7 \bmod 12$$

$$\Rightarrow x_p = 4^3 \bmod 11 = 64 \bmod 11 = 9 \bmod 11$$

$$x_q = 2^7 \bmod 13 = 128 \bmod 13 = 11 \bmod 13$$

$$\rightsquigarrow c_p = 13^{-1} \bmod 11 \equiv 2^{-1} \bmod 11 = 6 \bmod 11$$

$$c_q = 11^{-1} \bmod 13 \equiv 6 \bmod 13$$

$$\begin{aligned}\Rightarrow x &= [13 \cdot 6] \cdot 9 + [11 \cdot 6] \cdot 11 \bmod 143 \\ &= 702 + 726 \bmod 143 \\ &\equiv 141 \bmod 143.\end{aligned}$$

Complejidad de RSA con CRT

Analizando las operaciones en RSA con CRT, la complejidad de RSA con CRT es "fragada" por la complejidad de $x_p = y_p^{d_p} \bmod p$ & $x_q = y_q^{d_q} \bmod q$

Si n tiene longitud en bits de $t+1$, ent

$$- l(p), l(q) \sim t/2$$

$$- l(x_p), l(x_q), l(d_p), l(d_q) \sim t/2$$

(Como el proceso de multiplicación y elevar al cuadrado)

usa aproximadamente $\frac{3}{2}t$ operaciones, para números de longitud t

$$\Rightarrow \# \text{Mult.} + \# \text{Squared} \sim 2 \cdot \left[\frac{3}{2} \cdot \frac{t}{2} \right] = \frac{3}{2}t$$

Con esto podríamos pensar que con o sin CRT hay la misma complejidad. Sin embargo, la multiplicación y elevar al cuadrado estarán usando factores de longitud $\sim t/2$ y su complejidad es cuadrática con respecto a t

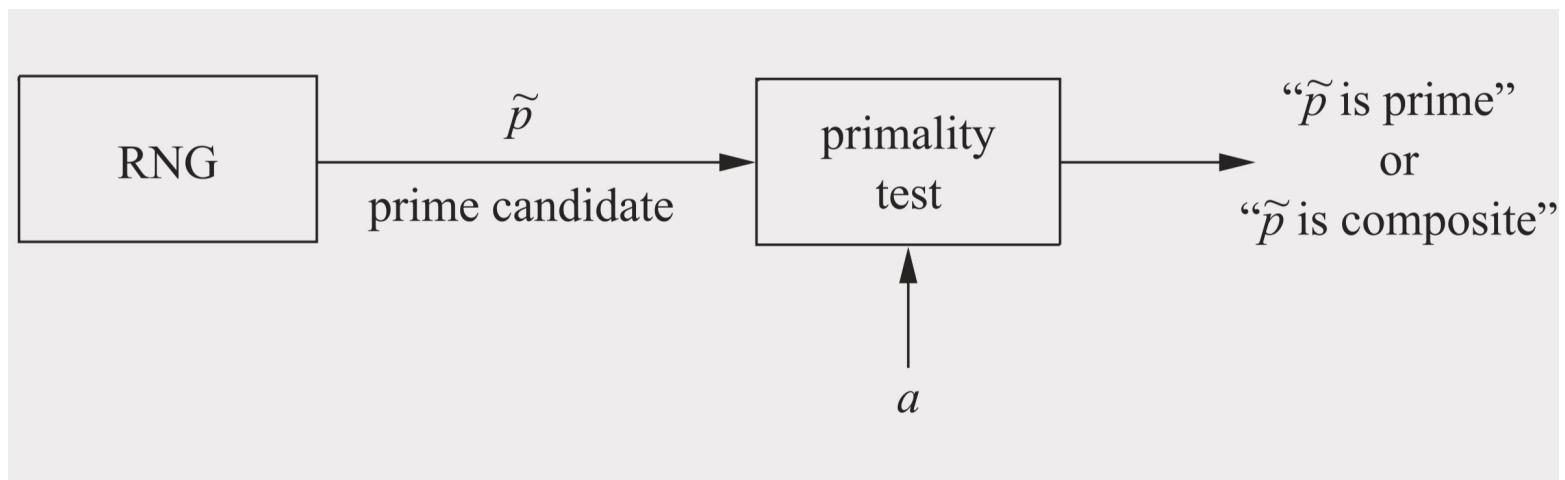
\Rightarrow Con CRT el RSA es al menos 4 veces más rápido

Encontrando primos grandes

Como hemos visto, es de suma importancia el poder encontrar primos p y q de la longitud apropiada. Por ejemplo, si queremos que n sea de longitud 2048, queremos encontrar primos p y q de longitudes (aproximadas) de 1024.

¿Qué tan grande es esto? Si $l(p)=1024$, ent. $p \geq 2^{1024}$

Lo que hacemos entonces es crear números aleatorios y revisarlos para ver si son primos.



El generador de números aleatorios no debe ser predecible.

Detalle: Los tests de primalidad son probabilísticos.

Esto nos lleva a dos preguntas:

- 1) ¿Cuántos números hay que probar antes de llegar a un primo?
- 2) ¿Qué tan rápidos son los tests?

¿Qué tan comunes son los primos?

Dado \tilde{p} un número aleatorio impar, se tiene que:

$$P(\tilde{p} \text{ es primo}) \approx \frac{2}{\ln(\tilde{p})}$$

\Rightarrow Si $\ell(\tilde{p}) \approx 1024$, ent:

$$\begin{aligned}
 P(\tilde{p} \text{ es primo}) &\approx \frac{2}{\ln(\tilde{p})} \\
 &\approx \frac{2}{\ln(2^{1024})} \\
 &= \frac{2}{1024 \ln(2)} \\
 &\approx \frac{1}{354}
 \end{aligned}$$

Con 354 tests se esperaría al menos un primo.

Tests de primalidad

Vamos a ver en esencia los tests (Fermat & Miller-Rabin). Dado que son tests probabilísticos, existe la posibilidad de un falso positivo. La forma de "solucionarlo" es: si sale positivo bajo cierto parámetro (el a de la figura de arriba), volver a pasarlo por el test con un parámetro diferente. Esto disminuye la probabilidad de un falso positivo. Nos gustaría llegar a menos de 2^{-80} .

Test de Fermat

Fermat Primality Test

Input: prime candidate \tilde{p} and security parameter s

Output: statement " \tilde{p} is composite" or " \tilde{p} is likely prime"

Algorithm:

- ```

1 FOR $i = 1$ TO s
1.1 choose random $a \in \{2, 3, \dots, \tilde{p}-2\}$
1.2 IF $a^{\tilde{p}-1} \not\equiv 1 \pmod{\tilde{p}}$
1.3 RETURN (" \tilde{p} is composite")
2 RETURN (" \tilde{p} is likely prime")

```

Observación: No hay falsos negativos.

Problema: los números de Carmichael<sup>C</sup> pasan el test para todo  $a \neq 1$  tq  $\gcd(a, C) = 1$

Aún con dicho problema, los números problemáticos son raros. Dicho eso, con los números de Carmichael pueden haber complicaciones (puede que sea primo relativo con muuuuchos números).

## Test de Miller-Rabin

Este test se basa en el siguiente teorema

**Teorema:** Sea  $\tilde{p}$  un candidato a primo y tómese la descomposición en primos de  $\tilde{p}-1$  siguiente:  $\tilde{p}-1 = 2^u r$ , con  $r$  impar.

Si  $\exists a \in \mathbb{Z} \forall j \in \{0, 1, \dots, u-1\}$  tq  $a^r \not\equiv 1 \pmod{\tilde{p}}$  &  $a^{r^{2^j}} \not\equiv \tilde{p}-1 \pmod{\tilde{p}}$ , ent.  $\tilde{p}$  no es primo. En caso contrario, es probable que sea primo.

Con este teorema se tiene el algoritmo siguiente.

## Miller–Rabin Primality Test

**Input:** prime candidate  $\tilde{p}$  with  $\tilde{p} - 1 = 2^u r$  and security parameter  $s$

**Output:** statement “ $\tilde{p}$  is composite” or “ $\tilde{p}$  is likely prime”

**Algorithm:**

```

1 FOR $i = 1$ TO s
 choose random $a \in \{2, 3, \dots, \tilde{p} - 2\}$
1.2 $z \equiv a^r \pmod{\tilde{p}} \rightsquigarrow S_q + \text{Mult.}$
1.3 IF $z \not\equiv 1$ AND $z \not\equiv \tilde{p} - 1$ ← Teorema con $j = 0$
 $j = 1$
1.4 WHILE $j \leq u - 1$ AND $z \not\equiv \tilde{p} - 1$ ←
 $z \equiv z^2 \pmod{\tilde{p}}$
 IF $z \equiv 1$ RETURN (“ \tilde{p} is composite”) ← Teorema, $j = 1, \dots, u-1$, con
 ELSE $j = j + 1$
1.6 IF $z \not\equiv \tilde{p} - 1$ RETURN (“ \tilde{p} is composite”) ← Teorema, $j = u$
2 RETURN (“ \tilde{p} is likely prime”)

```

La probabilidad de falsos positivos es muy baja conforme  $\ell(\tilde{p})$  crece.

**Table 7.2** Number of runs within the Miller–Rabin primality test for an error probability of less than  $2^{-80}$

| Bit length of $\tilde{p}$ | Security parameter $s$ |
|---------------------------|------------------------|
| 250                       | 11                     |
| 300                       | 9                      |
| 400                       | 6                      |
| 500                       | 5                      |
| 600                       | 3                      |

Ejemplo:  $\tilde{p} = 91 \rightsquigarrow \tilde{p}-1 = 2 \cdot 45$ . Tomamos  $s = 4$ .

Números a generar:

$$1) a = 12 \Rightarrow z = 12^{45} \equiv 90 \pmod{91} \rightsquigarrow \text{probablemente s\'i}$$

$$2) a = 17 \Rightarrow z = 17^{45} \equiv 90 \pmod{91} \rightsquigarrow \text{probablemente s\'i}$$

$$3) a = 38 \Rightarrow z = 38^{45} \equiv 90 \pmod{91} \rightsquigarrow \text{probablemente s\'i}$$

$$4) a = 39 \Rightarrow z = 39^{45} \equiv 78 \pmod{91} \rightsquigarrow \text{es compuesto.}$$

Si a da un falso positivo se le llama a ese valor un mentiroso para  $\tilde{p}$ .

# Esquema de Firma digital Elgamal

El esquema de firma digital de Elgamal fue publicado en 1985 y se basa en el logaritmo discreto. Sin embargo, este esquema es diferente del protocolo de cifrado.

## Versión de libro de texto

Hay dos fases: generación de llaves, y generación + verificación de firmas.

## Generación de llaves

- 1.- Se escoge un primo  $p$  grande.
- 2.- Se escoge un  $\alpha \in \mathbb{Z}_p^*$  primitivo ó un subgrupo de  $\mathbb{Z}_p^*$ .
- 3.- Se escoge un  $d \in \{2, 3, \dots, p-2\}$  aleatorio.
- 4.- Se calcula  $\beta = \alpha^d \bmod p$ .

Así, la llave pública es  $k_{\text{pub}} = (p, \alpha, \beta)$ , mientras que la llave privada es  $k_{\text{priv}} = d$ .

## Generación de firma

1. Se escoge  $k_E \in \{2, \dots, p-2\}$  aleatorio tq  $\text{gcd}(k_E, p-1) = 1$ ,  
a  $k_E$  se le llama llave efímera.
2. Se calculan  $r \equiv \alpha^{k_E} \pmod{p}$   
 $s \equiv (x - dr) k_E^{-1} \pmod{p}$

Con esto, la firma digital es  $\text{sig}_{k_{\text{priv}}}(x, k_E) = (r, s)$

## Verificación de firma

1. Se calcula  $t \equiv \beta^r r^s \pmod{p}$
2. Se valida la firma si y sólo si  $t \equiv \alpha^x \pmod{p}$ .

Ejemplo:

llave pública

$$\xleftarrow{(p, \alpha, \beta) = (29, 2, 7)}$$

### Bob

1. choose  $p = 29$
2. choose  $\alpha = 2$
3. choose  $d = 12$
4.  $\beta = \alpha^d \equiv 7 \pmod{29}$

compute signature for message  
 $x = 26$ :

choose  $k_E = 5$

(note that  $\text{gcd}(5, 28) = 1$ )

$$r = \alpha^{k_E} \equiv 2^5 \equiv 3 \pmod{29}$$

$$s = (x - dr) k_E^{-1} \\ \equiv (-10) \cdot 17 \equiv 26 \pmod{28}$$

mensaje y firma

$$\xleftarrow{(x, (r, s)) = (26, (3, 26))}$$

verify:

$$t = \beta^r \cdot r^s \equiv 7^3 \cdot 3^{26} \equiv 22 \pmod{29}$$

$$\alpha^x \equiv 2^{26} \equiv 22 \pmod{29}$$

$$t \equiv \alpha^x \pmod{29} \implies \text{valid signature}$$

◇

## Aspectos computacionales

Como la generación de la llave es idéntica al set-up del cifrado Elgamal, ent. ya conocemos restricciones.

En particular,  $p$  debe tener longitud de bits al menos de 2048.

También,  $d$  debe obtenerse con un TRNG

La  $\beta$  se puede obtener con el método de multiplicación y cuadrado.

La firma  $(x, (r, s))$  tiene que  $l(r) \sim l(p) \sim l(s)$

$$\Rightarrow l((x, (r, s))) \sim 3l(x)$$

Para calcular  $r$  se usa el método de multiplicación y cuadrado.

Para calcular  $s$  lo "difícil" es calcular  $k_t^{-1}$ . Esto se puede hacer ya sea con el algoritmo de Euclides extendido o con una tabla precalculada. De hecho, la pareja  $(k_t, r)$  puede precalcularse y guardarse para su uso (incluyendo el cálculo de  $s$ )

La verificación se realiza con el método de multiplicación y cuadrado.

## Seguridad

El primer punto concerniente a la seguridad es análogo al del esquema RSA de firmas: asegurarse que "Alice" tiene la llave pública correcta.

Fuera de esto, se tienen 3 ataques principales

- Cálculo de logaritmo discreto  $l(p) \geq 2048$ ;  $p$  primo
- Reutilización de llave efímera

Si se reutiliza  $k_E$  para dos firmas  $(x_1, (r_1, s_1))$  y  $(x_2, (r_2, s_2))$ , es "obvio" pues  $r_1 \equiv \alpha^{k_E} \equiv r_2 \pmod{p}$

$$\Rightarrow s_1 - s_2 \equiv (x_1 - x_2) k_E^{-1} \pmod{p-1}$$

$$\Rightarrow k_E \equiv \frac{x_1 - x_2}{s_1 - s_2} \pmod{p-1}$$

$\Rightarrow$  el atacante puede calcular (eventualmente)  $k_E$

$$\Rightarrow d = \frac{x_1 - s_1 k_E}{r_1} \pmod{p-1}$$

$\Rightarrow$  el atacante puede calcular (eventualmente)  $d$

- Ataque de imitación existencial

...

### Existential Forgery Attack Against Elgamal Digital Signature

Alice

Oscar

Bob

$$k_{pr} = d$$

$$k_{pub} = (p, \alpha, \beta)$$

$\xleftarrow{(p, \alpha, \beta)}$

$\xleftarrow{(p, \alpha, \beta)}$

1. select integers  $i, j$   
where  $\gcd(j, p-1) = 1$
2. compute signature:  
 $r \equiv \alpha^i \beta^j \pmod{p}$   
 $s \equiv -r j^{-1} \pmod{p-1}$
3. compute message:  
 $x \equiv s i \pmod{p-1}$

$\xleftarrow{(x, (r, s))}$

verification:

$$t \equiv \beta^r \cdot r^s \pmod{p}$$

since  $t \equiv \alpha^x \pmod{p}$ :

valid signature!

El atacante puede crear firmas válidas para mensajes pseudocaleatorios.

Para prevenir esto, se "hashea" el mensaje y se utiliza el mensaje hasheadó para la firma.

## Algoritmo de firma digital

DSA es una variante del algoritmo de firmas Elgamal. Es también conocido como Estándar de firma digital de EUA (US DSS).

El NIST dejó de recomendar el DSA en 2020 y recomienda más el RSA o ECDSA

Las ventajas de DSA sobre Elgamal es que las firmas son cortas en comparación al módulo (448 vs 2048), y que hay ataques de Elgamal que no funcionan en DSA.

### Algoritmo DSA (2048 bits, pero hay de 3072 bits)

## Generación de llaves

- 1.- Se genera un primo  $tq$   $2^{2047} < p < 2^{2048}$
- 2.- Encuentra  $q$  un divisor primo de  $p-1$   $tq$   
 $2^{223} < q < 2^{224}$
- 3.- Tomamos  $\alpha \in \mathbb{Z}_p^*$   $tq$   $\langle \alpha \rangle \cong \mathbb{Z}_q^*$
- 4.- Tomamos  $d$  aleatorio  $tq$   $0 < d < q$
- 5.- Calculamos  $\beta = \alpha^d \pmod{p}$
- 6.- La llave es  $k_{\text{pub}} = (p, q, \alpha, \beta)$ ,  $k_{\text{priv}} = d$

Como tenemos dos grupos cíclicos  $(\mathbb{Z}_p^* \& \mathbb{Z}_q^*)$ , ent. la firma va a resultar pequeña.

Las combinaciones de primos pueden ser más grandes:

| $p$  | $q$ | firma |
|------|-----|-------|
| 1024 | 160 | 320   |
| 2048 | 224 | 448   |
| 3072 | 256 | 512   |

## Generación de firmas

1. Se escoge una llave efímera  $t_q$   $0 < k_t < q$ .
2. Se calcula  $r \equiv (\alpha^{k_t} \bmod p) \bmod q$
3. Se calcula  $s \equiv (SHA(x) + dr) k_t^{-1} \bmod q$ , donde SHA es la función hash SHA-2

Lo que necesitamos ahorita es saber que SHA-2 comprime  $x$  y calcula una "huella digital" de al menos 224 bits.

## Verificación de firmas

1. Calcular los valores auxiliares  $w \equiv s^{-1} \bmod q$
2. Calcular los valores auxiliares  $u_1 \equiv w \cdot SHA(x) \bmod q$
3. Calcular los valores auxiliares  $u_2 \equiv wr \bmod q$
4. Calcular  $v \equiv (\alpha^{u_1} \beta^{u_2} \bmod p) \bmod q$
5. Se valida la firma si y sólo si  $v \equiv r \bmod q$

¿Qué significa que la firma no se validó? O bien el mensaje o la firma fueron modificados, o el verificador no tiene la llave pública correcta.

¿Por qué funciona la verificación?

$$s \equiv (\text{SHA}(x) + dr) k_E^{-1} \pmod{q}$$

$$\Rightarrow k_E \equiv s \cdot \text{SHA}(x) + s' dr \pmod{q}$$

$$\Rightarrow k_E \equiv u_1 + du_2 \pmod{q}$$

$$\rightsquigarrow \alpha^{k_E} \equiv \alpha^{u_1 + du_2} \pmod{p}$$

$$\equiv \alpha^{u_1} (\alpha^d)^{u_2} \pmod{p}$$

$$\equiv \alpha^{u_1} \beta^{u_2} \pmod{p}$$

$$\Rightarrow (\underbrace{\alpha^{k_E} \pmod{p}}_r) \equiv (\underbrace{\alpha^{u_1} \beta^{u_2} \pmod{p}}_v) \pmod{q}.$$

Ejemplo:

Bob

1. choose  $p = 59$
2. choose  $q = 29$
3. choose  $\alpha = 3$
4. choose private key  $d = 7$
5.  $\beta = \alpha^d \equiv 4 \pmod{59}$

$$\xleftarrow{(p,q,\alpha,\beta)=(59,29,3,4)}$$

sign:

compute hash of message  $h(x) = 26$

1. choose ephemeral key  $k_E = 10$
2.  $r = (3^{10} \pmod{59}) \equiv 20 \pmod{29}$
3.  $s = (26 + 7 \cdot 20) \cdot 3 \equiv 5 \pmod{29}$

$$\xleftarrow{(x,(r,s))=(x,(20,5))}$$

verify:

1.  $w = 5^{-1} \equiv 6 \pmod{29}$
  2.  $u_1 = 6 \cdot 26 \equiv 11 \pmod{29}$
  3.  $u_2 = 6 \cdot 20 \equiv 4 \pmod{29}$
  4.  $v = (3^{11} \cdot 4^4 \pmod{59}) \pmod{29} = 20$
  5.  $v \equiv r \pmod{29} \implies \text{valid signature}$
- 

### Aspectos computacionales

Para este algoritmo, la parte más pesada en realidad es la generación de llave.

El reto es encontrar un  $p$  primo tq  $\ell(p) = 2048$  & que tenga un divisor primo  $q$  de  $p-1$  tq  $\ell(q) \sim 224$ .

Normalmente se procede empezando por  $q$ , calculando de ahí la  $p$ .

## Prime Generation for DSA

**Output:** two primes  $(p, q)$ , where  $2^{2047} < p < 2^{2048}$  and  $2^{223} < q < 2^{224}$ , such that  $p - 1$  is a multiple of  $q$ .

**Initialization:**  $i = 1$

**Algorithm:**

- 1 find prime  $q$  with  $2^{223} < q < 2^{224}$  using the Miller–Rabin algorithm
- 2 FOR  $i = 1$  TO 4096
  - 2.1 generate random integer  $M$  with  $2^{2047} < M < 2^{2048}$
  - 2.2  $M_r \equiv M \pmod{2q}$
  - 2.3  $p - 1 \equiv M - M_r$  (note that  $p - 1$  is a multiple of  $2q$ .)  
IF  $p$  is prime (use Miller–Rabin primality test)
  - 2.4 RETURN  $(p, q)$
  - 2.5  $i = i + 1$
- 3 GOTO Step 1

En comparación, el resto es "sencillo".

$r = \alpha^{k_E} \rightsquigarrow$  multiplicación y cuadrado  $\sim 336$  requeridas

Se obtiene un resultado de 2048 bits

$\rightsquigarrow$  al hacer  $\pmod{q}$  se tiene un resultado de 224 bits

Para  $s$  se opera solo con 224 bits y lo más "raro" es  $k_E^{-1}$ .

Notemos  $r$  y  $k_E$  se pueden precalcular.

Ya la verificación se hace en 224 bits, y lo más "caro" es la exponentiación.

## Seguridad

La primera forma de atacar es resolver

$$d \equiv \log_{\alpha} \beta \pmod{p}$$

$$\Rightarrow l(p) \geq 2048$$

La segunda forma es usar que  $\langle \alpha \rangle = \mathbb{Z}_q^*$  el cual es un subgrupo chico en comparación. Sin embargo, el ataque clásico para esto no es realmente aplicable. Los ataques clásicos aquí nos dejan una seguridad de  $2^{112}$

**Table 10.2** Bit lengths and security levels for DSA

| $p$  | $q$ | Hash output (min) | Security levels |
|------|-----|-------------------|-----------------|
| 1024 | 160 | 160               | 80              |
| 2048 | 224 | 224               | 112             |
| 3072 | 256 | 256               | 128             |

Cabe mencionar que, al igual que en Elgamal, la reutilización de las llaves efímeras rompe la seguridad del algoritmo.

# Funciones Hash

Las funciones hash son partes importantes de la criptografía. En esencia son una "huella digital" de longitud fija de un mensaje.

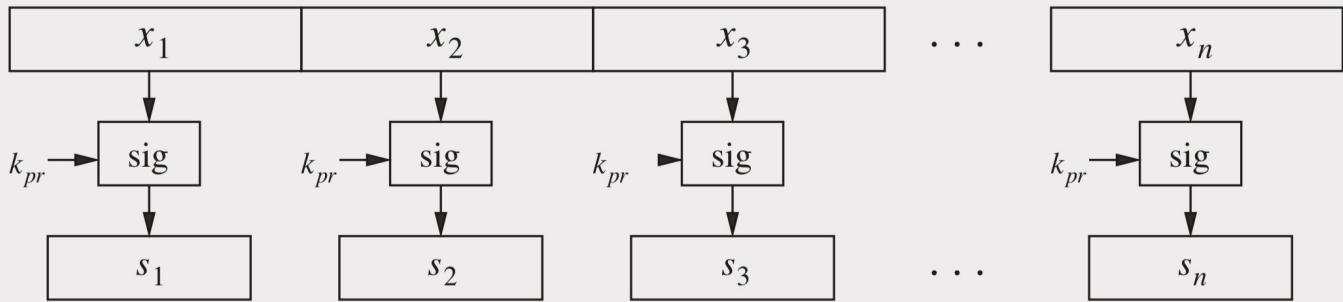
Estas funciones son parte esencial de los esquemas de firmado digital, códigos de autenticación de mensajes, etc.

## Motivación

Realmente lo que nos va a interesar es la motivación nacida por los esquemas de firmado digital.

Para los esquemas de firmado, algo que realmente "nos salta la vista" es el hecho de que estos esquemas tienen un espacio muy limitado para sus mensajes.

¿Cómo entonces se pueden firmar mensajes largos?  
Una respuesta es por bloques:



**Fig. 11.1** Insecure approach to signing of long messages

pero esto conlleva 3 grandes problemas:

### 1.- Carga computacional alta

Dependiendo de la longitud del mensaje, la  $n$  podría ser muy grande, lo cual hace que se necesiten bastantes recursos computacionales y energéticos tanto para quien firma como para quien revisa la firma.

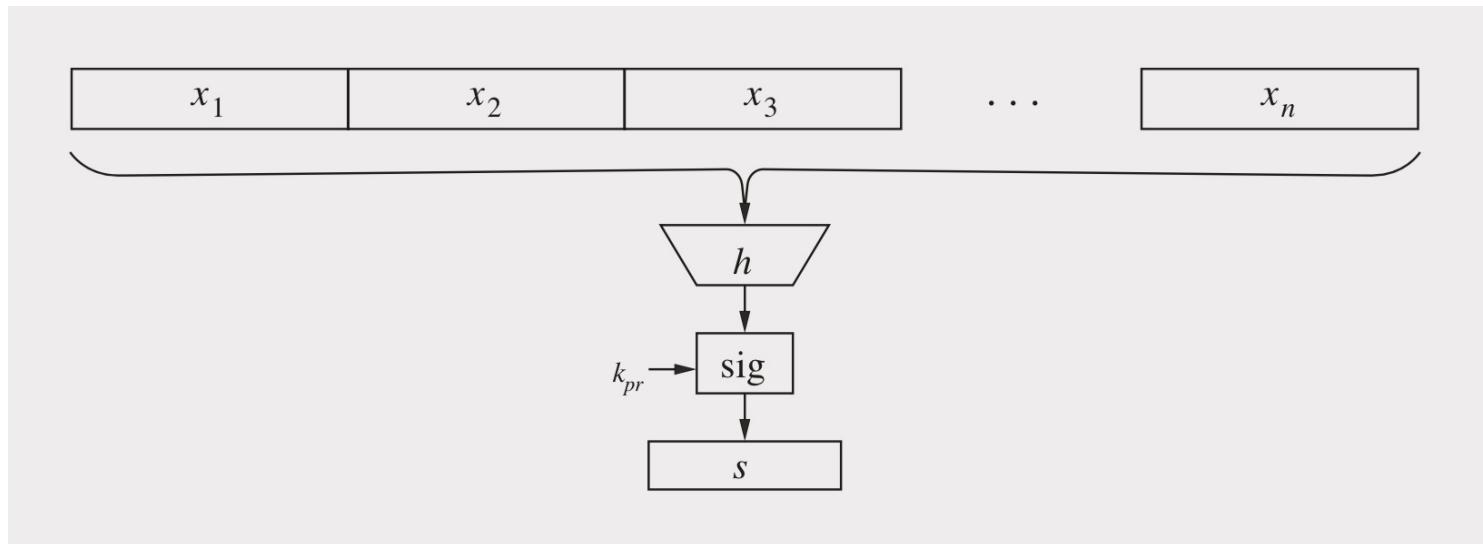
### 2.- Ancho de banda sobresaturado

No solo se debe mandar el mensaje, sino también las  $n$  firmas. Esto puede ser excesivo si  $n$  es muy grande.

### 3.- Limitaciones de seguridad

Mandar los  $n$  mensajes permiten nuevas vías de ataque (eliminación de bloques, reordenamiento de bloques, ensamble de nuevos mensajes, etc.).

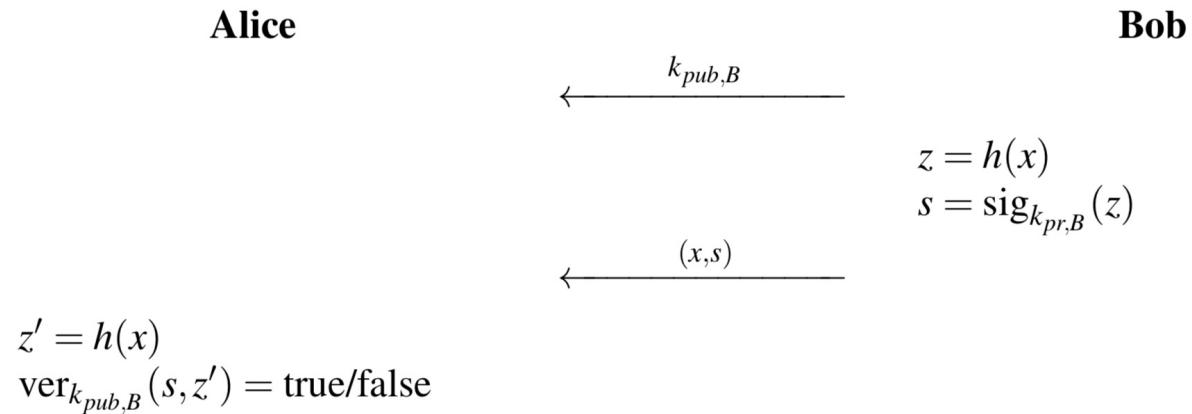
Con esto en mente, nos gustaría una única firma de corta longitud, para un mensaje de longitud arbitraria. La forma en que se soluciona esto es con las funciones hash:



Con esta función hash se puede tener el siguiente protocolo para un esquema de firma digital:

Aquí  $x$  es el mensaje,  $z$  su hasheadó y  $s$  su firma del hasheadó.

## Basic Protocol for Digital Signatures with Hash Function:

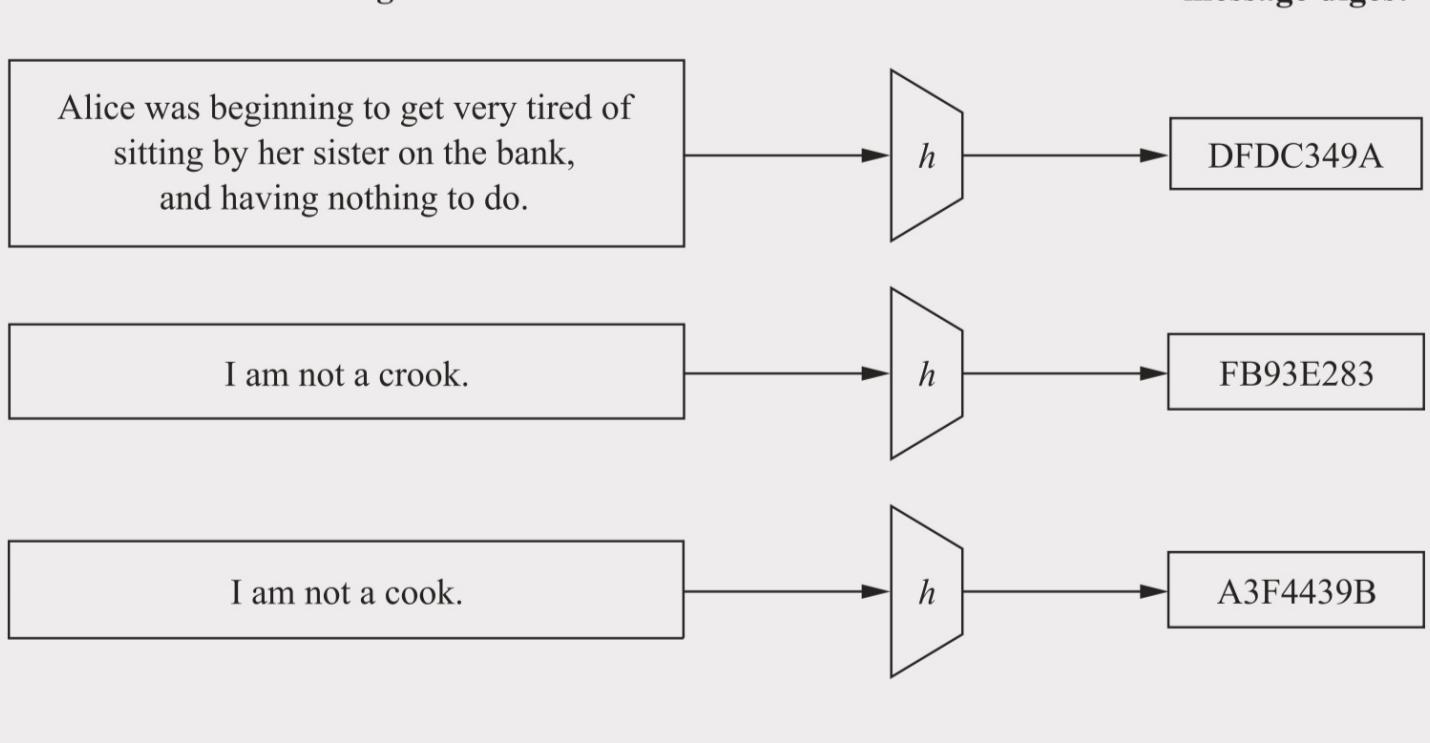


Nótese que el firmado y la verificación se realizan usando el hasheado, el cual es referido como **message digest** o **huella digital** del mensaje.

En particular, queremos una función matemáticamente eficiente, que admita mensajes arbitrariamente largos y que saque una huella de longitud fija  $n$ .

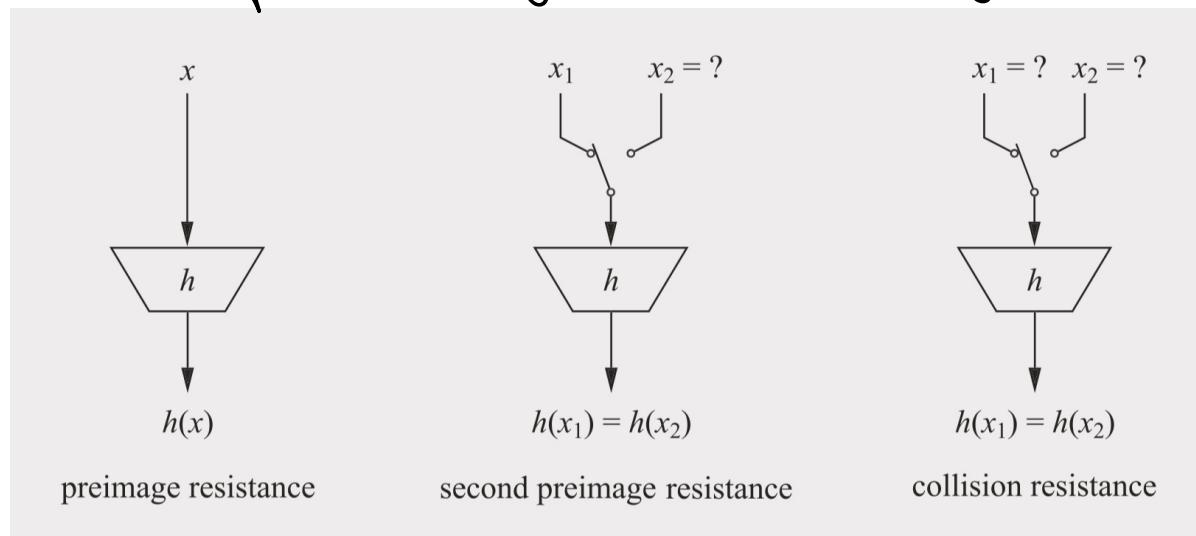
$$h: \{0,1\}^* \rightarrow \{0,1\}^n \quad n \approx 256, 512$$

También queremos que sea altamente sensible a cambios en el mensaje.



## Requisitos de seguridad

Dado que las funciones hash son bastante generales (claramente no inyectivas) y no tienen llaves, lo que vamos a requerir para su seguridad es lo siguiente.



## Resistencia a preimagen

Dada una función hash  $h$  y un  $z$  en la imagen de  $h$ , debe ser computacionalmente imposible encontrar un  $x \neq z$  tal que  $h(x) = z$ , es decir, cualquier elemento de  $h^{-1}(z)$  debe ser computacionalmente imposible de calcular.

La resistencia a preimagen además de lo obvio, también es importante pues podría dársele la vuelta a la seguridad del cifrado a través de la función hash.

Esto es aún más importante en cosas como derivación de llaves.

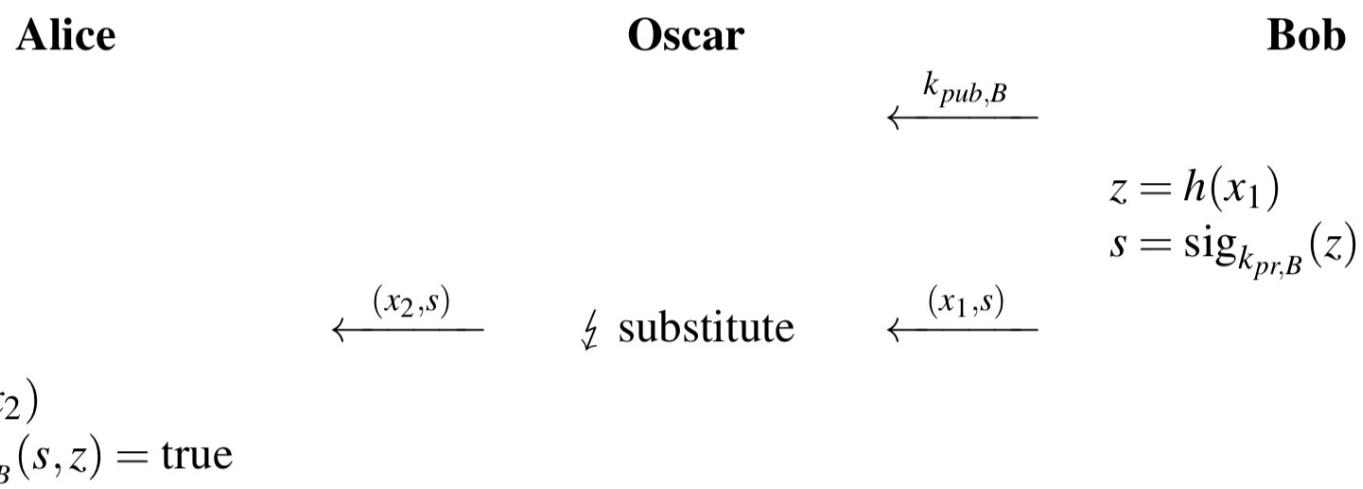
## Segunda resistencia a preimagen

En general, para un atacante debe ser imposible obtener dos mensajes  $x_1 \neq x_2$  tales que  $h(x_1) = h(x_2)$  (colisiones).

Si  $x_1$  es dado y  $x_2$  no se conoce se le llama colisión inversa.

sión débil.

Incluso en un esquema básico de firmas digitales, es clara la importancia de la resistencia a colisión débil (o segunda resistencia a preimagen).



Por lo tanto, queremos funciones hash  $h$  tq dabs  $x_i$  y  $h(x_i)$  no existen ataques analíticos con los q se pueda construir  $x_j$  con  $h(x_i)$ .

Para computadoras actuales  $n=128$  basta... por ahora.

### Resistencia a colisiones

Una función hash es resistente a colisiones fuertes si no es computacionalmente factible construir  $x_1 \neq x_2$  tales que  $h(x_1) = h(x_2)$ .

tg  $h(x_1) = h(x_2)$ .

De no ser así, se podría implementar ataques como el siguiente: Se empieza con  $x'_1$  = "Transfiera \$10 a la cuenta de Oscar" y  $x'_2$  = "Transfiera \$10,000 a la cuenta de Oscar", y se alteran los mensajes de forma "no visible" para obtener los dos mensajes con la misma huella digital.

A esto se le llama el ataque del cumpleaños de Yuval (Gideon Yuval, 1979), y depende de que Oscar pueda engañar a Bob para que firme  $x_1$ .

Siendo  $n$  la longitud de salida de la función hash  $h$ ,  $t$  el número de mensajes, y  $p$  la probabilidad de que los mensajes  $(x_1, \dots, x_t)$  no produzcan  $h$ -colisiones, siguiendo la misma estrategia de la paradoja de los cumpleaños, ent.

$$p \approx e^{-\frac{t(t-1)}{2^{n+1}}}.$$

Luego, haciendo  $\lambda = 1-p$  y haciendo varias aproximaciones, tenemos que

$$t \approx 2^{\frac{(n+1)}{2}} \sqrt{\ln \frac{1}{1-\lambda}}$$

⇒ Para encontrar colisiones bastan  $t \approx \sqrt{2^n}$  mensajes.

⇒ Para tener seguridad de  $m$  bits, es necesario tener  $n = (\geq) 2m$ .

Una tabla para la longitud de  $n$  de una función hash.

**Table 11.1** Number of required hash computations for a collision for different hash function output lengths and for two different collision likelihoods

| $\lambda$ | Hash output length [bits] |          |           |           |           |
|-----------|---------------------------|----------|-----------|-----------|-----------|
|           | 128                       | 160      | 256       | 384       | 512       |
| 0.5       | $2^{64}$                  | $2^{81}$ | $2^{129}$ | $2^{193}$ | $2^{257}$ |
| 0.9       | $2^{65}$                  | $2^{82}$ | $2^{130}$ | $2^{194}$ | $2^{258}$ |

Notese que el ataque de cumpleaños es "genérico", es decir, no depende de la función hash específica.

Al final de cuentas, aunque estos tres puntos de seguridad son los más deseados, hay aplicaciones en

las que en realidad basta incluso la resistencia a preimagen como en hasheado de contraseñas.

En resumen:

### Properties of Hash Functions

1. **Arbitrary message size**  $h(x)$  can be applied to messages  $x$  of any size.
2. **Fixed output length**  $h(x)$  produces a hash value  $z$  of fixed length.
3. **Efficiency**  $h(x)$  is relatively easy to compute.
4. **Preimage resistance** For a given output  $z$ , it is computationally infeasible to find any input  $x$  such that  $h(x) = z$ , i.e.,  $h(x)$  is one-way.
5. **Second preimage resistance** Given  $x_1$ , and thus  $h(x_1)$ , it is computationally infeasible to find any  $x_2 \neq x_1$  such that  $h(x_1) = h(x_2)$ .
6. **Collision resistance** It is computationally infeasible to find any pair  $x_1 \neq x_2$  such that  $h(x_1) = h(x_2)$ .