

INTRODUCCIÓN A LA CRIPTOGRAFÍA

ANÁLISIS DE LOS PRINCIPALES
ALGORITMOS Y SUS APLICACIONES



Jose Ignacio Gomez Garcia
nagomez97@gmail.com

PRÓLOGO

Al lo largo de la historia, el envío de mensajes secretos ha sido una de las herramientas más importantes (y codiciadas) de la humanidad. La capacidad de poder compartir información de forma confidencial otorgaba una ventaja importante a quien la dominaba. Desde los antiguos espartanos hasta la actualidad, pasando por Julio César y la máquina alemana *Enigma*, la criptografía ha representado un papel fundamental en el campo militar y de las comunicaciones.

Con este propósito se idearon una gran cantidad de *algoritmos* o métodos capaces de ocultar el contenido de un mensaje, de modo que únicamente el receptor adecuado fuese capaz de entenderlo. A estos algoritmos se les denominó *cifrados*.

En este libro vamos a tratar numerosos algoritmos de cifrado, explicando sus peculiaridades, sus puntos fuertes y sus debilidades, así como las diferencias entre todos ellos. Tambiénaremos un viaje a las entrañas de algunos de los protocolos y tecnologías más utilizados, tratando de entender las implicaciones de la criptografía en todos ellos y en nuestra vida diaria.

Estructura del libro

Introducción a la Criptografía está formado por cuatro capítulos fuertemente conectados. En el primero de ellos, *Teoría Criptográfica*, comenzaremos con un breve glosario de términos que vamos a utilizar a menudo en este libro, para a continuación caracterizar un buen algoritmo de cifrado. El plato principal de este primer capítulo es la presentación de los principales algoritmos criptográficos: desde los históricos como *Caesar* hasta los más actuales, como *RSA*. Se trata de un capítulo denso, en el que he tratado de presentar una base matemática que permita al lector comprender el funcionamiento de cada algoritmo. Sin embargo, en muchos casos basta eludir las matemáticas y empaparse del algoritmo (para eso he incluido algunos *pseudocódigos*) para entender un algoritmo criptográfico.

En esta primera parte hablaremos de algoritmos de cifrado, funciones *hash*, conceptos como algoritmo *simétrico* o *asimétrico*, firma digital, PKIs... Por lo tanto, recomiendo al lector echar, aunque sea, un rápido vistazo a este capítulo para familiarizarse con los conceptos básicos necesarios para entender el resto del libro.

El segundo capítulo, *Criptosistemas Reales*, presenta algunos de los protocolos más utilizados en la actualidad: Bluetooth, 4G, WLAN, TLS, *end-to-end*... Aquí desarrollaremos el funcionamiento de cada uno, haciendo especial hincapié en su relación con la criptografía. ¿Cómo funciona *bluetooth*? ¿Cómo se asegura una conexión WiFi? ¿Qué es el cifrado extremo a extremo de *WhatsApp*? Estas son algunas de las preguntas cuyas respuestas podrás encontrar en este libro.

El tercer capítulo, *Implementaciones*, compone la parte más práctica. Trataremos diversas formas de implementar sistemas criptográficos en varios lenguajes, con consejos para mantener la robustez del algoritmo y evitar errores por parte del desarrollador.

Finalmente, en el capítulo cuarto presentaremos una serie de conclusiones acerca de todo lo mencionado en el libro. He tratado de presentar una especie de “árbol de decisión”

que permita al lector escoger un algoritmo de cifrado para cada situación, pero no deje de ser necesaria por su parte una fuerte labor de investigación, ya que el tipo de algoritmo a utilizar es exclusivamente dependiente de las condiciones del proyecto. También se presentarán algunos consejos para evitar problemas derivados de una mala implantación de los métodos explicados.

¿Qué aporta este libro?

A *priori*, puede parecer que esto no deja de ser otra fuente bibliográfica de curiosidades matemáticas. Sin embargo, hay que saber ver más allá de esto.

Es posible que hayas llegado hasta este libro buscando una metodología (esto es, una serie de pasos definidos y fijos) para la inclusión segura de contenido criptográfico en una aplicación. También es posible que hayas llegado por el lado contrario: eres un *pentester* o un analista que busca vulnerabilidades criptográficas en una aplicación.

En ambos casos, puede que echando un rápido vistazo al libro te lleves una decepción.

Es cierto que la aplicación práctica (inmediata) de este libro es poca. No contesta explícitamente a respuestas como *¿Qué algoritmo debo usar?* o *¿Con qué librería implemento RSA?* Esto tiene una explicación bastante sencilla: no existe ninguna metodología definida en criptografía.

Los algoritmos existentes han sido desarrollados por los mejores criptoanalistas del mundo y, desgraciadamente (para nosotros), todos son matemáticos. Por lo tanto, los algoritmos aquí tratados (salvo casos puntuales), son matemáticamente “perfectos”. Esto quiere decir que no se pueden romper, pero también implica que son muy rigurosos. Es necesario prestar muchísima atención a los parámetros, al sistema donde se ejecuta y, en general, a la implementación.

El famoso criptoanalista, Bruce Schneier, dijo en su día que “la criptografía no se rompe, se evita”, haciendo referencia al hecho de que resulta computacionalmente imposible efectuar ataques reales contra los algoritmos criptográficos más modernos. Es por eso que, cuando se lleva a cabo un ataque “contra la criptografía” de la aplicación, en realidad se está explotando una vulnerabilidad fruto de un error de diseño o desarrollo en la misma, no en el cifrado.

De hecho, se han dado situaciones en los que una entidad (digamos, la NSA) ha distribuido contenido criptográfico con *backdoors* ocultas que les permitían acceder a todo tipo de información privada. Por lo tanto, es imprescindible trabajar con fuentes de confianza, verificadas y, a ser posible, de código abierto.

Por lo tanto, pongámonos en el papel de una empresa. Todo lo comentado hasta ahora nos lleva a pensar que lo realmente valioso no es un técnico que, con un manual debajo del brazo, revise que se está usando AES256 en vez de DES; se necesitan analistas que se hayan empapado previamente de las entrañas de la criptografía, que comprendan los algoritmos en su totalidad y que sean capaces de descubrir el más mínimo error de desarrollo y solventarlo. Y de eso trata este libro.

Tabla de Contenido

1 Teoría Criptográfica	1
1.1 Definiciones útiles	1
1.2 ¿Qué caracteriza a un buen algoritmo de cifrado?	2
1.2.1 Confusión y difusión	3
1.2.2 Longitud de clave	3
1.2.3 Rendimiento	4
1.3 Objetivos de la criptografía	4
1.4 Clasificación de los algoritmos	4
1.4.1 Sustitución o transposición	4
1.4.2 Monoalfabético o polialfabético	4
1.4.3 Simétrico o asimétrico	5
1.4.4 Flujo o bloque	5
1.5 Algoritmos históricos	5
1.5.1 Atbash	5
1.5.2 Escítila	6
1.5.3 Caesar	6
1.5.4 ROT13	7
1.5.5 Vigenère	7
1.5.6 Enigma	8
1.6 Algoritmos de Clave Simétrica	11
1.6.1 One-Time Pad (Libreta de Un Solo Uso)	11
1.6.2 Modos de operación de algoritmos de cifrado en bloque	12
1.6.3 <i>Key Wrapping</i>	17
1.6.4 Data Encryption Standard (DES)	18
1.6.5 Triple-DES	24
1.6.6 Advanced Encryption Standard	25
1.6.7 IDEA (International Data Encryption Algorithm)	27
1.6.8 Blowfish	29
1.6.9 Twofish	33
1.6.10 Rivest Cipher (RC6)	40
1.6.11 Serpent	45
1.6.12 Salsa20	49
1.6.13 ChaCha20	53

1.6.14	Camellia.....	55
1.7	Algoritmos simétricos ligeros	62
1.7.1	Grain-128a	62
1.7.2	MICKEY 2.0.....	63
1.7.3	Trivium	63
1.8	Criptografía Asimétrica	64
1.8.1	Introducción	64
1.8.2	Usos de la criptografía asimétrica	64
1.9	Criptografía Híbrida	66
1.9.1	Introducción	66
1.9.2	Sobre digital	66
1.9.3	Clave de sesión	66
1.9.4	Claves asimétricas e implementación	66
1.10	Algoritmos de Clave Pública.....	67
1.10.1	Estándar PKCS	67
1.10.2	Intercambio de clave: Diffie-Hellman.....	68
1.10.3	RSA.....	71
1.10.4	EIGamal	75
1.10.5	ECDH (Elliptic-Curve Diffie-Hellman)	77
1.10.6	Cramer-Shoup	83
1.11	Criptografía Post-Cuántica.....	85
1.11.1	McEliece	85
1.11.2	NTRU	88
1.12	Funciones Hash.....	89
1.12.1	Definición.....	89
1.12.2	Seguridad.....	90
1.12.3	Aplicaciones.....	90
1.12.4	Rainbow Tables.....	90
1.12.5	Diseño.....	90
1.13	Algoritmos de Funciones Hash.....	91
1.13.1	MD5.....	91
1.13.2	MD6.....	93
1.13.3	SHA-1	94
1.13.4	SHA-2	95
1.13.5	SHA-3	95
1.13.6	BLAKE.....	96

1.13.7	Streebog	96
1.13.8	Whirlpool	¡Error! Marcador no definido.
1.14	Autenticación e Integridad	97
1.15	Funciones MAC	100
1.15.1	HMAC	100
1.15.2	CBC-MAC	100
1.15.3	OMAC/CMAC	101
1.15.4	GMAC	102
1.15.5	Poly1305	103
1.16	Firma Digital	103
1.16.1	Firma basada en RSA	104
1.16.2	Esquema de firma de ElGamal	105
1.16.3	Digital Signature Algorithm (DSA)	106
1.16.4	Elliptic Curve Digital Signature Algorithm (ECDSA)	107
1.16.5	EdDSA	108
1.17	Infraestructura de Clave Pública	110
1.17.1	Certificado Digital	110
1.17.2	Estándar para PKIs	110
1.17.3	Infraestructura	111
1.17.4	Web of Trusty PGP	112
1.17.5	GNU Privacy Guard	113
1.17.6	Blockchain PKI	113
1.17.7	Let's Encrypt	113
2	Criptosistemas Reales	114
2.1	Bluetooth	114
2.1.1	Funcionamiento	114
2.1.2	Versiones	115
2.1.3	Seguridad de BR/EDR	115
2.1.4	Seguridad de <i>Low Energy</i>	120
2.1.5	Conclusión	122
2.1.6	Ataques conocidos	122
2.2	Telefonía móvil: 4G/LTE	123
2.2.1	Conceptos	124
2.2.2	Evolved Packet Core	126
2.2.3	Protocolos de red	128
2.2.4	Seguridad en LTE	128

2.2.5	Conclusión	131
2.3	WLAN: WEPy WPA/WPA2.....	132
2.3.1	WEP.....	132
2.3.2	WPA/WPA2.....	133
2.3.3	Conclusiones	137
2.4	TLS	138
2.4.1	Detalles del protocolo	138
2.4.2	Versiones	140
2.4.3	Seguridad.....	141
2.4.4	ssl-pulse.....	142
2.4.5	HSTS.....	143
2.5	Cifrados End-to-End	144
2.5.1	Retos	145
2.5.2	Signal Protocol	146
2.5.3	WhatsApp	150
2.6	Bases de datos.....	154
3	Implementaciones.....	160
3.1	Introducción	160
3.2	Gestión de claves	161
3.2.1	Ciclo de vida de las claves	162
3.2.2	Seguridad del Gestor de Claves.....	163
3.2.3	Gestores de Claves	163
3.3	Java 11.....	164
3.3.1	javax.crypto (JCA)	164
3.3.2	<i>Bouncy Castle</i>	166
3.3.3	Comparación.....	167
3.3.4	AES usando SunJCE	167
3.3.5	RSA usando SunJCE	169
3.3.6	AES-GCM usando BouncyCastle.....	172
3.3.7	Otros algoritmos	173
3.4	EverCrypt.....	174
4	Conclusiones	175
4.1	¿Qué algoritmo debo elegir?	175
4.2	¿Cómo incluyo un algoritmo en mi proyecto?	178
4.3	¿Qué no debo hacer?	179
4.4	¿Qué sí debo hacer?	179

4.5	Trabajo futuro	180
4.6	Criptografía cuántica	181
5	Anexos	182
5.1	Anexo 1: Cajas-S para Redes Feistel	182
5.2	Anexo 2: Rijndael en profundidad.....	183
5.2.1	Conceptos matemáticos	183
5.2.2	Base de diseño.....	186
5.2.3	Especificaciones	186
5.2.4	Obteniendo las Round Keys.....	191
5.2.5	El cifrado	192
5.2.6	Aspectos de implementación	193
5.2.7	Descifrado.....	193
5.2.8	Ventajas e inconvenientes	195
5.2.9	Comentarios.....	195
5.3	Anexo 3: Comparación de algoritmos de cifrado	196
5.3.1	Comparación general de algoritmos.....	196
5.3.2	Comparación de seguridad en algoritmos de clave simétrica	198
5.4	Anexo 4: Matemáticas de RSA.....	200
5.4.1	Conceptos matemáticos	200
5.4.2	Demostración de la propiedad principal de RSA	200
5.5	Anexo 5: Pseudocódigo de MD5	202
5.6	Anexo 6: Comparación de librerías criptográficas	203
5.6.1	Principales librerías	203
5.6.2	Algoritmos de intercambio de clave.....	204
5.6.3	Algoritmos simétricos	205
5.6.4	Funciones hash	206
5.6.5	Algoritmos MAC	207

1 Teoría Criptográfica

1.1 Definiciones útiles

Cifrar

Comúnmente conocido como *encriptar* (derivado su homólogo inglés *encrypt*), es el proceso de someter un mensaje a un algoritmo capaz de ocultar su contenido al resto del mundo, de modo que únicamente el receptor adecuado será capaz de *descifrarlo*.

Codificar

La codificación es un método que permite transformar un carácter de un alfabeto a un símbolo de algún otro sistema de representación. En el caso de la criptografía, es muy común tratar los caracteres como números enteros (generalmente), lo que permite establecer álgebras y realizar operaciones entre ellos. Un ejemplo de codificación empleado es el sistema ASCII, que codifica 128 símbolos, entre mayúsculas, minúsculas, números, signos de puntuación...

Criptoanálisis

Disciplina dedicada a estudiar los métodos que permiten *romper* un cifrado para recuperar el contenido original, o la clave empleada. Lejos de tratarse de un procedimiento de fuerza bruta basado en, por ejemplo, probar claves a diestro y siniestro, el criptoanálisis se centra en comprender el algoritmo en sí, encontrar puntos débiles y usarlos para obtener información.

Criptología

Ciencia que estudia la criptografía y el criptoanálisis.

Alfabeto criptográfico

En un algoritmo por sustitución, es el conjunto de caracteres de entre los cuales se elige el elemento de cifrado. Por ejemplo, un cifrado basado en desplazar todas las letras una posición hacia delante en el abecedario tradicional, tendrá como alfabeto criptográfico al mismo conjunto de caracteres que el alfabeto del mensaje inicial.

Texto plano

Mensaje original previo a cualquier cifrado o codificación.

Clave criptográfica

Elemento “secreto” del algoritmo que únicamente el emisor (y, en ocasiones, el receptor) deben conocer. Puede ser una contraseña, un *token*... En general, cualquier elemento tal que únicamente quien lo posea sea capaz de descifrar un mensaje.

Federal Information Processing Standard (FIPS)

Estándares públicos que, obviamente han sido desarrollados por el gobierno de los Estados Unidos, o bien han sido aprobados para su uso en las agencias de dicho gobierno. Se trata, por tanto, de un referente mundial en lo que a seguridad respecta. En concreto, el FIPS 140-2 recoge los algoritmos públicamente aceptados para tratar datos confidenciales del Estado en los Estados Unidos.

Margen de seguridad

A la hora de “medir” la robustez de un algoritmo, nos podemos encontrar con numerosos problemas. El principal es que no podemos decir que un algoritmo está roto hasta que realmente lo hemos roto, y además no tenemos forma de saber que llevamos un 60% del camino recorrido hasta nuestro objetivo.

Es por eso que se tienen en cuenta ciertos factores a la hora de establecer dicha medida:

- **¿Cuánto esfuerzo se le ha dedicado al algoritmo?** Como el lector se podrá imaginar, no todos los algoritmos reciben la misma atención. Si nos fijamos en Rijndael, el actual estándar AES, nos daremos cuenta de que es el algoritmo más estudiado, debido a que es uno de los más utilizados. Lo mismo pasa con RSA. Es por eso que si pudiésemos decir que hemos “roto un 30% de Rijndael”, pero que todavía no hemos llegado al 10% en otro algoritmo menos conocido, estos valores no serían para nada comparables.
- **¿Cuántas rondas se han llegado a romper?** La mayoría de los algoritmos de cifrado contienen una parte iterativa que se repite un número arbitrario de veces. En cada ronda, aumenta considerablemente el esfuerzo necesario para romperlo, ya que se incrementa la entropía de los datos. Lo que se suele hacer es contemplar versiones simplificadas de los algoritmos, con menos rondas, y tratar de romperlos. No se trata de una medida del todo significativa, pero desde luego sí que se puede considerar a la hora de comparar algoritmos.

Con estos criterios, podremos otorgar un valor a la robustez del algoritmo, aunque no va a dejar de ser orientativo. Por lo tanto, cuanto mayor sea el margen de seguridad, más robusto se espera que sea el algoritmo.

PKCS

Son las siglas para *Public-Key Cryptography Standards*, y hace referencia a un conjunto de estándares concebidos por los laboratorios de RSA. Entre ellos, se recoge el estándar que define el modo de empleo de los algoritmos RSA y Diffie-Hellman, uso de certificados, curvas elípticas, generación de números pseudo-aleatorios...

1.2 ¿Qué caracteriza a un buen algoritmo de cifrado?

Esta pregunta ha sido el foco de numerosos estudios, especialmente a lo largo de los siglos XIX y XX, dando lugar a cuantiosas teorías al respecto. Es de esperar que un buen cifrado debe ocultar la mayor cantidad de información posible a toda persona ajena, haciendo que el contenido sea replicado exclusivamente a manos del receptor esperado. Sin embargo, no todo radica ahí: no tiene sentido un algoritmo demasiado lento, difícil de deshacer o que utilice claves muy difíciles de manejar.

Además, la robustez del algoritmo debe ser independiente del conocimiento de su uso: un atacante que conozca perfectamente el algoritmo debería ser incapaz de romperlo o de averiguar la clave. A esta característica se la conoce como *Principio de Kerckhoffs*.

1.2.1 Confusión y difusión

Estos dos conceptos están íntimamente ligados con la resistencia de estos algoritmos. Se entiende por **difusión** a la propiedad de que se disipe la redundancia de caracteres en el cifrado del mensaje. Se ha demostrado que, para mensajes largos, un análisis estadístico permite obtener información acerca de las veces que se emplea un cierto carácter, por lo que un atacante puede extraer datos de un mensaje con un nivel de difusión bajo. Otra forma de entender la propiedad de difusión es pensar que, cuando se modifica un único bit o carácter en el texto sin cifrar, se debe alterar la mayor cantidad posible de bits en el mensaje cifrado. De esta forma, el texto resultante del algoritmo estará estadísticamente muy poco relacionado con el original.

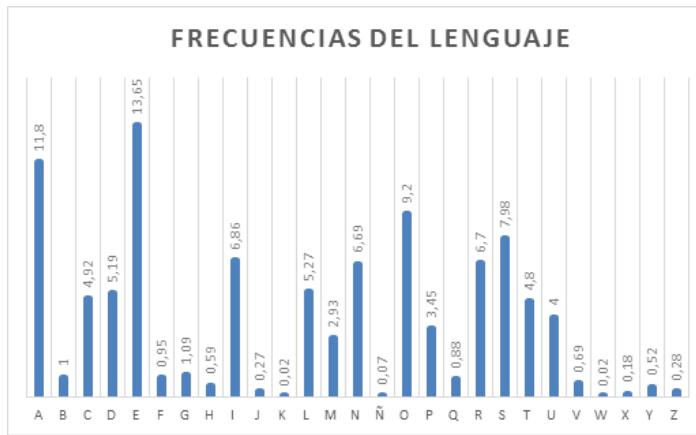


Figura 1.1 Frecuencias de los caracteres del alfabeto español.

Por otro lado, el término **confusión** hace referencia a la falta de relación entre el mensaje sin cifrar y la clave empleada. Uno de los procedimientos empleados para lograrlo es aplicar un método de cifrado por sustitución sobre el texto original (por ejemplo, Caesar), para posteriormente aplicar algún otro algoritmo, generalmente de transposición (es decir, que permuta los caracteres).

Es obvio que, cuanto mayores sean las propiedades de difusión y confusión de nuestro algoritmo, más robusto va a ser este.

1.2.2 Longitud de clave

Un buen algoritmo admite un amplio rango de claves y, preferiblemente, está ideado para poder ser ampliado en un futuro. Esto se debe a que, en función de las características de la implementación, puede que se prefiera sacrificar un poco de seguridad por una mayor velocidad (reduciendo el tamaño de clave) o que se busque el máximo grado de seguridad. Además, debemos tener en cuenta que lo que hoy en día se considera seguro (por ejemplo, claves de 256 bits en AES), puede que en unos años quede obsoleto debido a un aumento de la capacidad de computación o a la implantación de la computación cuántica.

Si bien es cierto que una clave lo suficientemente larga es prácticamente imposible de romper (ya que están pensadas para que, usando la fuerza bruta, hasta el ordenador más potente del mundo tarde muchos años en poder obtenerla), puede que una mala implementación del algoritmo (o el propio diseño) contenga una vulnerabilidad que permita al atacante obtener el mensaje sin necesidad de adivinar la contraseña.

1.2.3 Rendimiento

El rendimiento suele ser el principal factor a tener en cuenta (junto con la robustez, obviamente) a la hora de elegir un tipo de algoritmo de cifrado u otro. Normalmente, vamos a buscar la mejor relación velocidad/seguridad y, en algunas ocasiones, la velocidad es crítica (como, por ejemplo, en las comunicaciones en tiempo real o *streamings*).

Sin embargo, no todo es velocidad: la cantidad de memoria ocupada, la capacidad de paralelización de tareas, la capacidad de corregir errores (en el caso, por ejemplo, de un cifrado en flujo en el que el mensaje llega en varios *paquetes*)... Es necesario tener en cuenta todos estos factores y muchos a la hora de diseñar o elegir un algoritmo de cifrado.

1.3 Objetivos de la criptografía

La criptografía ha evolucionado a lo largo de los siglos con el objetivo de ofrecer una serie de garantías de gran importancia en seguridad:

- **Autenticación:** garantía de que la persona que envía o recibe un mensaje es quien dice ser.
- **Confidencialidad:** garantía de que únicamente alguien destinado a recibir un mensaje puede descifrarlo.
- **Integridad:** garantía de que cierta información no ha sido modificada por terceros.
- **No Repudio:** cuando se ha enviado un mensaje o realizado alguna acción, el actor no podrá negar su implicación.
- **Control de acceso:** garantiza que únicamente un grupo de personas autenticado y permitido tiene acceso a la información.

1.4 Clasificación de los algoritmos

Debido a la gran cantidad de algoritmos existentes, resulta bastante útil poder clasificarlos en función de una serie de propiedades que los caracterizan.

1.4.1 Sustitución o transposición

Los algoritmos por **sustitución** se basan, como su nombre indica, en sustituir un carácter por otro de forma arbitraria (por ejemplo, todas las *A* se convierten en *B*). Por su parte, los algoritmos por **transposición** se basan en la permutación de caracteres dentro del mismo mensaje (por ejemplo, invirtiendo el orden de los caracteres).

1.4.2 Monoalfabético o polialfabético

Cuando nos encontramos ante un algoritmo por sustitución, se entiende por algoritmos **monoalfabéticos** a aquellos que emplean un único *alfabeto criptográfico*, mientras que los algoritmos polialfabéticos cuentan con más de uno.

1.4.3 Simétrico o asimétrico

Cuando un algoritmo de cifrado requiere el uso de una misma clave criptográfica secreta tanto para el cifrado como para el descifrado, nos encontramos ante un algoritmo **simétrico**. Sin embargo, existen algunos algoritmos que emplean pares de claves pública-privada para cifrar y descifrar, denominados algoritmos **asimétricos** o de clave pública.

1.4.4 Flujo o bloque

Se entiende por cifrado de **flujo** al algoritmo que cifra el mensaje bit a bit, o carácter a carácter, consecutivamente. Un **cifrado en bloques**, sin embargo, dividirá el mensaje en bloques de k elementos o bits, aplicando el algoritmo al bloque completo. Este último presenta la principal ventaja de que es fácilmente escalable y permite su uso, por ejemplo, en comunicaciones a tiempo real, donde el flujo de datos puede variar o ser continuo. Sin embargo, son más difíciles y costosos de implementar, debido a su mayor lentitud y a la necesidad de generar flujos de cifrado resistentes ante un criptoanálisis (sin relación estadística con el mensaje ni aparición de patrones).

1.5 Algoritmos históricos

1.5.1 Atbash

Uno de los primeros algoritmos conocidos fue el usado por los hebreos, entre otras cosas, en el Libro de Jeremías. Categorizado dentro de los algoritmos por sustitución, se denomina método de espejo ya que consiste en identificar cada letra del abecedario con su simétrica en el alfabeto invertido. Es decir, la A se cifraría como Z, la B como Y...

a	b	c	d	e	f	g	h	i	j	k	l	m	n	ñ	o	p	q	r	s	t	u	v	w	x	y	z
Z	Y	X	W	V	U	T	S	R	Q	P	O	Ñ	N	M	L	K	J	I	H	G	F	E	D	C	B	A

Figura 1.2 Correspondencia con el alfabeto criptográfico de Atbash.

Este algoritmo no depende del uso de ninguna clave, por lo que depende exclusivamente de mantener su uso en secreto.

1.5.2 Escíptala

Entorno al año 400 A.C. en Esparta, surgió un nuevo algoritmo de cifrado, más complejo, que consistía en enrollar una tira de papel sobre un bastón (escíptala) de un grosor y una longitud determinados. Posteriormente, se rellenaba el resto de la tira con otros caracteres para hacer el mensaje ilegible a simple vista, de modo que únicamente un receptor con un bastón adecuado pudiese enrollar la tira y observar cómo se alineaban los caracteres adecuados.



Figura 1.3 Escíptala clásica espartana.

1.5.3 Caesar

Como el propio nombre indica, los primeros indicios del uso de este cifrado se remontan a los tiempos de Julio César quien, para enviar mensajes confidenciales, empleaba un mecanismo de transposición desplazando todos los caracteres del mensaje tres posiciones hacia delante.

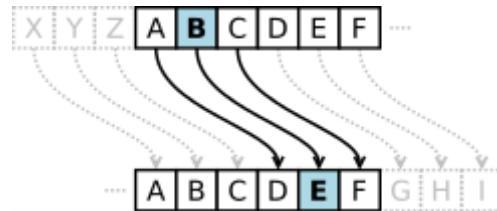


Figura 1.4 Ejemplo de cifrado Caesar.

Como se puede observar, se trata de un algoritmo muy simple y fácil de romper. Su robustez depende en gran medida de mantener secreto el algoritmo empleado, ya que el conjunto de claves es insignificante (la clave es el número de posiciones desplazadas que, en el caso del César, eran tres). Precisamente por esto último, un ataque de fuerza bruta requiere, como máximo, 26 intentos para tener éxito (en el caso del alfabeto tradicional español).

Una curiosidad de este algoritmo es que ya incluía aritmética modular básica, puesto que cuando una letra se tenía que desplazar más allá de los límites del alfabeto, continuaba de forma cíclica desde el inicio.

$$\text{Caesar}(M_i, k) = (M_i + k) \bmod L$$

En esta notación, M_i representa la letra i -ésima del mensaje, k representa la clave y L sería el número total de elementos del alfabeto.

1.5.4 ROT13

En los años 80 se popularizó una variación del cifrado Caesar, que consistía en emplear el número 13 como clave, desplazando todos los caracteres trece posiciones. Debido a lo fácil que resultaba romper este algoritmo, no se usaba para proteger información, sino simplemente como una forma de ocultar material inapropiado, como bromas de mal gusto, a aquellas personas que no querían leerlas.

1.5.5 Vigenère

En el siglo XVI, Blaise de Vigenère ideó un algoritmo por sustitución más complejo que los ya existentes. Inicialmente escogía una clave (por ejemplo, CIRUELA), en la que cada letra se codificaba con su posición en el alfabeto. A continuación, se repetía la clave tantas veces como hiciera falta para cubrir el mensaje completo.

M	E	N	S	A	J	E			S	E	C	R	E	T	O
C	I	R	U	E	L	A			C	I	R	U	E	L	A
2	8	18	21	4	9	1			2	8	18	21	4	9	1

Figura 1.5 Clave de Vigenère.

Por último, se le suma a cada letra del texto plano el valor correspondiente a cada letra de la clave (en el ejemplo no contemplamos la ñ).

O	M	E	M	E	U	E			A	V	W	V	P	T	Q
----------	----------	----------	----------	----------	----------	----------	--	--	----------	----------	----------	----------	----------	----------	----------

Figura 1.6 Ejemplo de mensaje cifrado usando Vigenère.

Hay formas más rápidas que el cálculo mental para aplicar este cifrado, como por ejemplo tablas de correspondencia:

×	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	Y	

Figura 1.7 Tabla de correspondencia para el algoritmo Vigenère.

1.5.6 Enigma

Durante la Segunda Guerra Mundial, las fuerzas alemanas idearon un mecanismo de rotores capaz de implementar uno de los algoritmos de cifrado más avanzados de la época: la máquina Enigma.

Este ingenioso aparato basaba su seguridad en el conocimiento previo de una configuración inicial por parte del operario, la cual determinaría la forma en la que el mensaje iba a ser cifrado.

Originalmente, la máquina Enigma contaba con tres hendiduras en las que se podían introducir los rotores, a elegir entre cinco distintos. Cada rotor disponía de 26 posiciones, una por cada letra del alfabeto tradicional, de modo que habría 26 configuraciones iniciales en cada rotor.

$$5 \times 4 \times 3 = 60 \text{ combinaciones de rotores}$$

$$26 \times 26 \times 26 = 17576 \text{ combinaciones de posiciones iniciales de los rotores}$$



Figura 1.8 Rotores de la máquina Enigma.

Sin embargo, los ingenieros alemanes idearon una última capa de seguridad, que consistía en un panel de conexiones similar al de las antiguas centrales telefónicas, que constaba de 26 agujeros (uno por cada letra), entre los cuales podían conectarse un total de 10 cables, formando parejas. Cada pareja de letras generaba intercambio entre ambas, de forma que si la E estaba conectada con la Q, se intercambiarían a la hora de cifrar.

$$\frac{26!}{6! \times 10! \times 2^{10}} = 150,738,274,937,250 \text{ posibles conexiones en el panel}$$

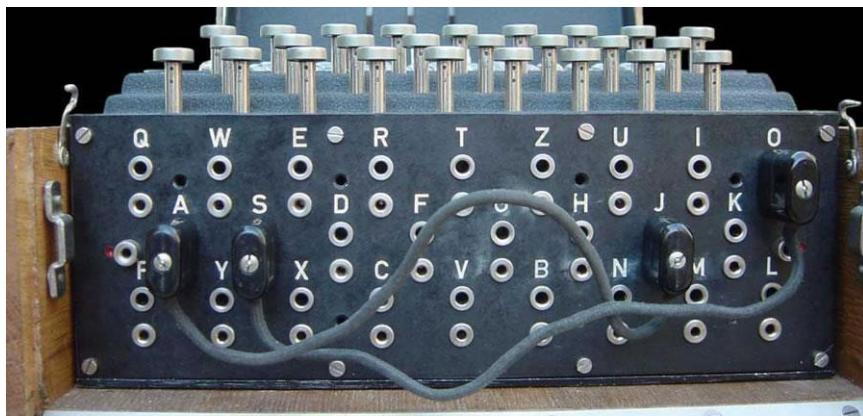


Figura 1.9 Panel de conexiones de la máquina Enigma.

Llegados a este punto, contamos con un total de 158,962,555,217,826,360,000 configuraciones iniciales de la máquina.

Como se puede imaginar, la robustez del cifrado Enigma dependía casi totalmente de mantener en secreto la configuración inicial elegida que, a su vez, debe ser conocida por el emisor y el receptor del mensaje para poder cifrar y descifrar. Es por eso que los operarios recibían mensualmente una hoja en la que se detallaba la configuración inicial de la máquina correspondiente a cada día. Estos detalles se escribían con tinta soluble para que, en caso de captura o hundimiento, los ejércitos enemigos no fueran capaces de recuperar su contenido.

Reihen- folge	Geheime Kommandosache! Jeder einzelne Tageschluessel ist genau: niemals im Zugzug voreben!										Nr. 0000082		
	Luftwaffen-Maschinen-Schlüssel Nr. 2744												
Achtung! Schlüsselmittel dürfen nicht unverfehlt in Feindeshand fallen. Bei Gefahr restlos und frühzeitig vernichten.													
	Wälzenlage	Ringstellung	Stellerverbindungen am Steckerkreis										
			an der Umkehrwelle	1	2	3	4	5	6	7	8	9	10
2744	31	III	V	IV	17	11	04						
2744	30	I	IV	V	08	17	21						
2744	29	V	II	III	11	14	05						
2744	28	II	IV	V	02	20	16						
2744	27	III	V	IV	18	13	22						
2744	26	I	III	II	24	10	01						
2744	25	IV	I	III	04	25	23						
2744	24	V	III	I	09	19	06						
2744	23	IV	I	V	15	03	19						
2744	22	I	V	III	12	26	07						
2744	21	III	IV	II	15	09	12						
2744	20	IV	II	I	02	22	05						
2744	19	V	I	II	08	19	17						
2744	18	III	IV	I	11	21	01						
2744	17	I	V	II	18	23	14						
2744	16	III	IV	V	16	01	07						
2744	15	V	III	IV	24	13	10						
2744	14	I	IV	II	06	20	25						
2744	13	III	II	I	03	26	18						
2744	12	II	IV	III	04	11	15						
2744	11	V	I	IV	16	07	02						
2744	10	IV	III	II	20	12	14						
2744	9	III	II	V	06	18	10						
2744	8	V	I	III	01	21	17						
2744	7	II	V	I	25	08	23						
2744	6	IV	II	V	13	26	03						
2744	5	III	I	II	24	19	22						
2744	4	II	IV	I	17	05	09						
2744	3	V	II	IV	20	16	11						
2744	2	II	III	V	14	03	19						
2744	1	III	I	IV	18	24	15						
					HL KN PW EL AG B9	DS OW FZ QX RU TV							

Figura 1.10 Hoja de configuraciones mensual de Enigma.

Las características que llevaron a Enigma a convertirse en uno de los mayores retos criptográficos de la historia fueron:

- La gran cantidad de alfabetos criptográficos que se podían emplear (dependiendo de la configuración de los rotores y el panel de conexiones).
- La gran *difusión* ocasionada por usar un algoritmo polialfabético cambiante. Era poco probable que una letra se codificase varias veces de la misma forma y, además, que se repitiera un carácter en el texto cifrado no implicaba que se tratase de un único carácter en el texto plano.
- La facilidad de uso y la portabilidad de Enigma, que la hacían perfecta para tiempos de guerra.
- La dificultad de análisis del código debido al desconocimiento de las configuraciones iniciales.

Afortunadamente, las fuerzas aliadas fueron capaces de romper el código Enigma, gracias a los esfuerzos de un equipo de criptógrafos polacos. Se cree que este descubrimiento pudo acortar la duración de la Segunda Guerra Mundial considerablemente.

Enlaces de interés:

[Numberphile – Enigma Machine \(youtube\)](#)

[Numberphile – Flaw in the Enigma Code \(youtube\)](#)

[Simulador de Enigma online](#)

La Segunda Guerra Mundial supuso un punto de inflexión en la historia de la criptografía, ya que desde entonces aumentó la preocupación por la confidencialidad y numerosos gobiernos y empresas dedicaron grandes esfuerzos y recursos para investigar este campo. Es por eso que en último siglo se han sucedido una gran cantidad de algoritmos, unos más exitosos que otros, generalmente cimentados con sólidos cálculos matemáticos y estadísticos

1.6 Algoritmos de Clave Simétrica

Como ya se ha avanzado en la introducción, un algoritmo de clave simétrica emplea la misma clave para cifrar que para descifrar, por lo que es totalmente necesario mantenerla en secreto. De hecho, un buen algoritmo simétrico hace recaer toda la seguridad en la clave y ninguna en el algoritmo en sí, por lo que un atacante debería ser incapaz de romper el cifrado aun conociendo cómo funciona el algoritmo al completo.

A continuación vamos a presentar los algoritmos de clave simétrica más representativos, algunos de los cuales siguen vigentes hoy en día.

1.6.1 One-Time Pad (Libreta de Un Solo Uso)

Se trata de un algoritmo de cifrado en flujo inventado en 1917 por Gilbert Vernam, que consistía en generar una clave aleatoria igual de larga que el mensaje a cifrar, de forma que al combinar la clave con el texto se conseguía un mensaje cifrado teóricamente imposible de descifrar.

En un comienzo, el Cifrado de Vernam implementaba este algoritmo, con el inconveniente de que reutilizaba las claves. Esto hacía que el cifrado fuese vulnerable. Sin embargo, más adelante se descubrió que, si se empleaba una clave totalmente aleatoria, la dificultad del criptoanálisis incrementaba considerablemente. De hecho, Claude Shannon acabaría descubriendo lo que llamaría secreto perfecto: el texto cifrado no proporcionaba absolutamente ninguna información sobre el mensaje.

El algoritmo Vernam-Mauborgne (nombre de la implementación totalmente aleatoria) presentaba serias desventajas pese a su supuesta robustez:

- Requiere el uso de claves *totalmente aleatorias*, lo que en esa época (e incluso hoy en día) era algo muy difícil de conseguir.
- No podía usarse la misma clave para cifrar dos textos distintos sin perder robustez.
- El intercambio de la clave debía ser seguro.
- Se requería una clave tan larga como el mensaje.

En definitiva, One-Time Pad demostró ser un algoritmo robusto, pero tremadamente ineficiente y poco útil a la hora de mantener una comunicación.

```
CIHJT UUHML FRUGC ZIBGD BQPNI PDNJG LPLLP YJYXM  
DCXAC JSJUK BIOYT MWQPX DLIRC BEXYK VKIMB TYIPE  
UOLYQ OKOXH PIJKY DRDBC GEFZG UACKD RARCD HBYRI  
DZJYO YKAIE LIUYW DFOHU IOHZV SRNDD KPSSO JMPQT  
MHQHL OHQQD SMHNP HHOHQ GXRPJ XBXIP LLZAA VCMOG  
AWSSZ YMFNI ATMON IXPBY FOZLE CVYSJ XZGPU CTFQY  
HOWHU OCJGU QMWQV OIGOR BFHIZ TYFDB VBRMN XNLZC
```

Figura 1.11 Ejemplo de clave usada en One-Time Pad.

1.6.2 Modos de operación de algoritmos de cifrado en bloque

Los algoritmos de cifrado en bloque cuentan con varios modos de operación que especifican cómo se operarán los bloques. Cada modo funciona mejor o peor en función de las características del problema:

1. **Electronic Code Book (ECB):** se trata del modo más rápido y sencillo. Divide el mensaje en bloques más pequeños aplicando directamente una clave para cifrar. El problema de este modo es que, para un mismo par mensaje-clave, siempre genera la misma salida. Es por eso que resulta muy fácil de romper.

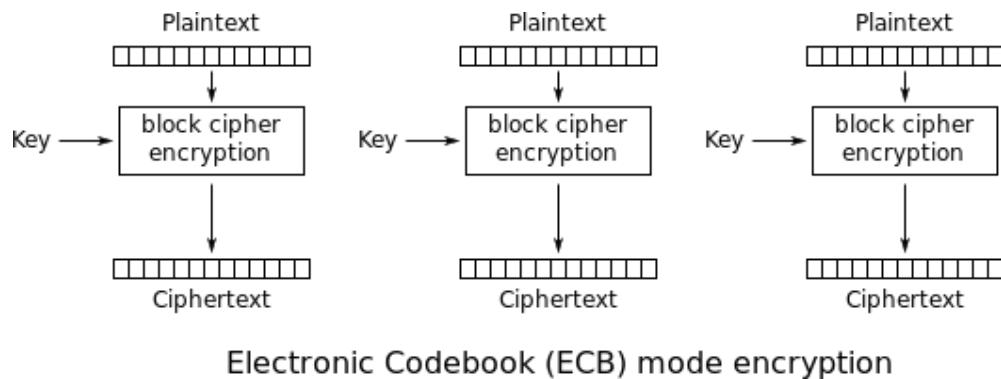


Figura 1.12 Esquema de funcionamiento del modo ECB.

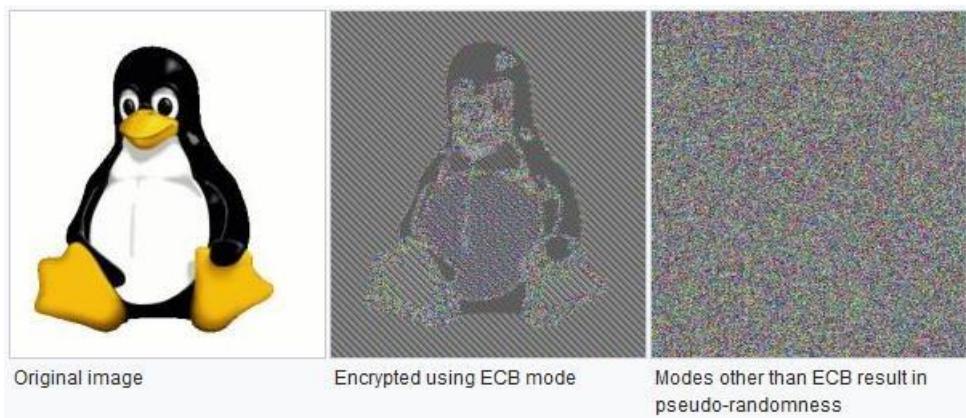


Figura 1.13 Ejemplo de cifrado usando ECB en una imagen bitmap.

2. **Cipher Block Chaining (CBC):** este modo de operación, inventado en 1976, se basa en hacer un XOR entre el bloque actual y el bloque anterior cifrado justo antes de aplicar el algoritmo de cifrado. En el caso del primer bloque, es necesario generar un *Vector de Inicialización (IV)*, ya que no existe un bloque previo con el que realizar un XOR. La aleatoriedad del IV es importante para la robustez del algoritmo, pero es importante usar el mismo vector tanto en el cifrado como en el descifrado. Se dice que la aleatoriedad del vector de inicialización aporta *seguridad semántica* al algoritmo.

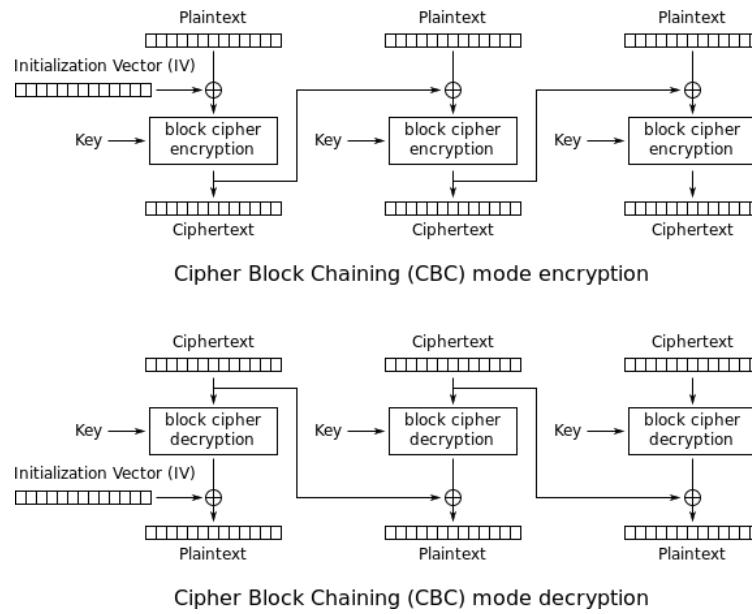


Figura 1.14 Esquema de funcionamiento del modo CBC.

3. **Cipher Feedback (CFB):** este modo es bastante similar al CBC, y también requiere el uso de un vector de inicialización. Como resulta bastante visual, el siguiente esquema resume su funcionamiento:

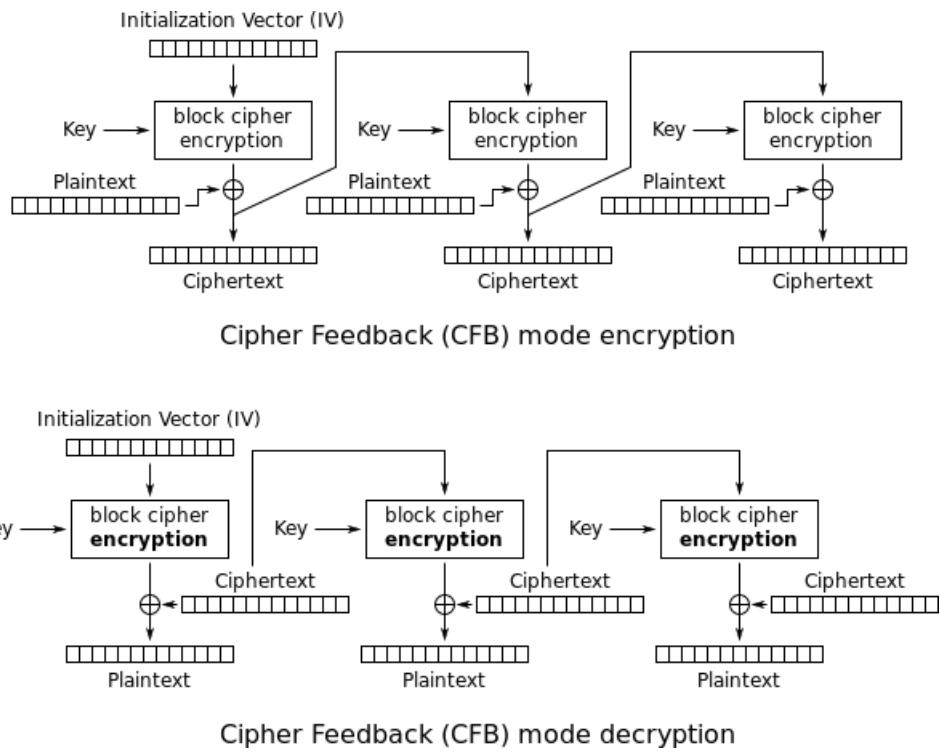


Figura 1.15 Esquema de funcionamiento del modo CFB.

Este modo presenta una ventaja clara, y es que puede ser usado para emular cifrados de flujo. Si suponemos que nuestra red admite únicamente mensajes de 8 bits, en vez de 64, no podremos emplear los modos anteriores. Por ese motivo, este algoritmo nos permite cifrar una secuencia de 8 bits y enviarla a través de la red, pero esa misma secuencia cifrada se empleará para obtener la siguiente, siguiendo el esquema 1.14. En general, podremos hacer esto con secuencias de cualquier longitud, incluso de 1 bit.

4. **Output Feedback (OFB):** este modo evita que errores en el envío de los bloques cifrados afecten al descifrado. Siguiendo el esquema OFB, se consigue que un error de transmisión en el bit n del texto cifrado se traduzca en un error en el mismo bit del texto descifrado. Esto es porque no se realiza el XOR entre el texto plano y el bloque cifrado antes de trasladarlo al siguiente bloque. De esta forma se pueden aplicar métodos de corrección y evitar la pérdida de información (por ejemplo, incluyendo bits de paridad en el mensaje).

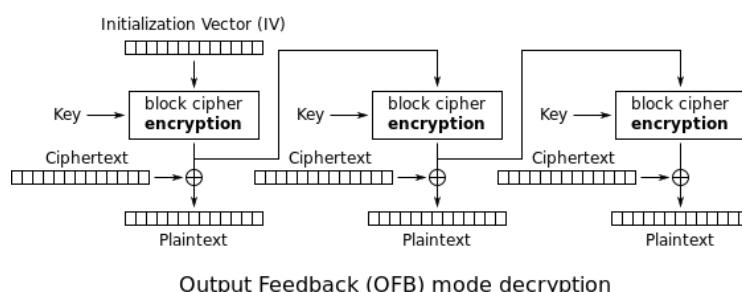
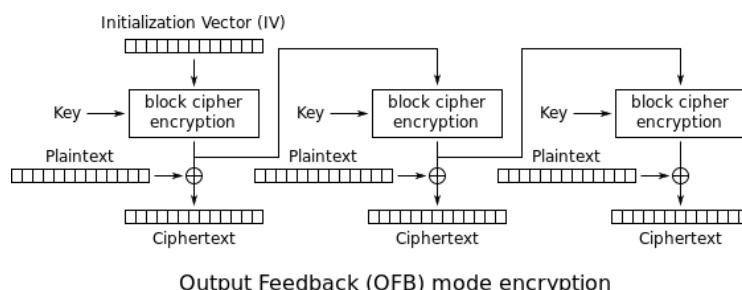


Figura 1.16 Esquema de funcionamiento del modo OFB.

5. **Counter (CTR):** denuevo, se trata de un modo que convierte el cifrado en bloques a un cifrado en flujo. En este caso, en vez de emplear el bloque anterior para la difusión, emplea un contador (que puede ser cualquier función que genere números que no se repitan, aunque la más usada es el incremento en una unidad) que será cifrado junto a un *nonce* que cumple la misión del vector de inicialización. La ventaja que tiene frente a OFB es que, con una filosofía muy similar, permite la paralelización de tareas ya que el cifrado de un bloque no depende del ningún otro bloque, sino únicamente del contador (que puede calcularse previamente).

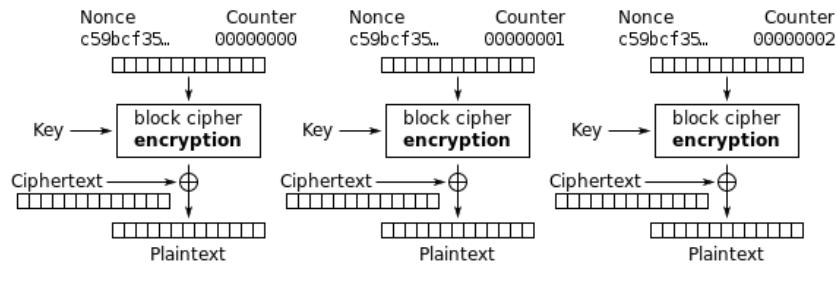
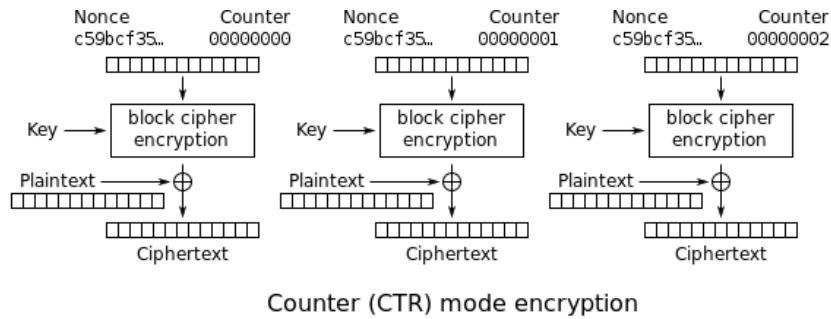


Figura 1.17 Esquema de funcionamiento del modo CTR.

6. **GCM (Galois Counter Mode):** este modo basa su funcionamiento en la misma idea que el CTR: cifra el Vector de Inicialización junto con un contador autoincremental para evitar ataques de texto repetido. Lo que lo diferencia del modo CTR es que incluye un método de autenticación de mensajes (MAC) para asegurar que el texto recibido no ha sido modificado por nadie ajeno al canal.

Para conseguir su objetivo, GCM trabaja en el cuerpo finito $GF(2^{128})$ (ver apéndice 2 para una introducción al álgebra de Galois en cuerpos finitos), lo que le permite definir una operación de multiplicación polinómica y adoptar una serie de propiedades que garantizan el funcionamiento del método MAC. Este tipo de álgebra empleada es el que otorga al modo el nombre de Galois (matemático que desarrolló gran parte de esta teoría).

El siguiente esquema detalla el funcionamiento del modo GCM, donde E_k indica la operación de cifrado usando la clave k (el método más usado es AES-Rijndael) y $mult_h$ hace referencia a la operación de multiplicación definida para el cuerpo finito.

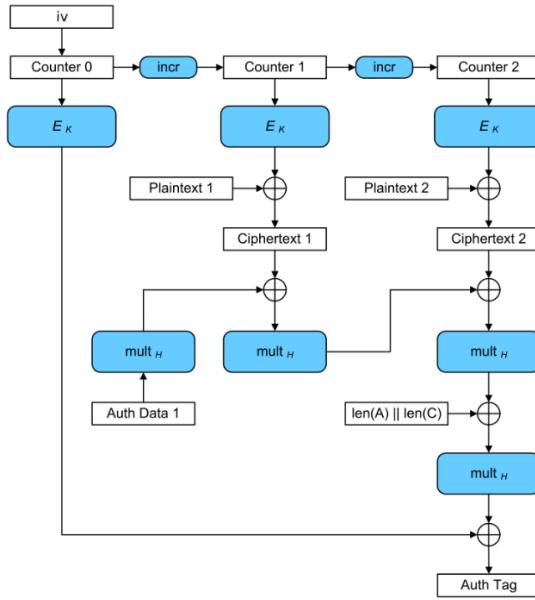


Figura 1.18 Esquema de funcionamiento del modo GCM.

Un detalle importante de los modos CTR y GCM es que presentan un importante fallo de seguridad: cuando se usa el mismo par clave/*nonce* para cifrar, se está permitiendo que un atacante pueda inferir una gran cantidad de información de la comunicación. Por eso, salvo que se vaya a cifrar un único bloque de datos, nunca se debe reutilizar la clave o el *nonce*. Además, por motivos de seguridad se recomienda el uso de *nonces* de al menos 12 bytes (para evitar ataques por falsificación de *nonces*).

7. **CCM (Counter with CBC-MAC mode):** este modo de operación es un derivado de CTR que incluye métodos de autenticación cifrada (ver el apartado **¡Error! No se encuentra el origen de la referencia.**) para cifrados en flujo, usando el MAC CBC-MAC. Se usa en protocolos como WPA2, *Bluetooth Low Energy*, TLS 1.2...

Enlaces de interés:

[Block Cipher modes of operation \(Wikipedia\)](#)

1.6.3 *Key Wrapping*

Los métodos de *Key Wrapping* son un conjunto de algoritmos simétricos empleados para *encapsular* (cifrar) material criptográfico en entornos no seguros. Se pueden considerar dentro del ámbito del *Key Management*. Entre otras cosas, este tipo de algoritmos permite proteger claves *at rest* almacenadas en un disco duro, por ejemplo.

No vamos a entrar en mayor detalle con este tipo de algoritmos, pero sí vamos a puntualizar que uno de los más empleados es un *operation mode* de AES, denominado *AES Key Wrap (KW)*, que se encuentra detallado en una especificación del NIST.

1.6.4 Data Encryption Standard(DES)

A principios de los años 70, las autoridades del entonces denominado National Bureau of Standards (NBS) determinaron la necesidad de adoptar un estándar a nivel gubernamental para garantizar la confidencialidad de la información. La NBS, en colaboración con la NSA, solicitó entonces que se presentaran a estudio algoritmos candidatos a ser adoptados como estándar. Tras una convocatoria fallida, en 1974 IBM presentó un candidato basado en el algoritmo Lucifer de Horst Feistel. Posteriormente, en 1975, se publicó la propuesta DES basada en dicho algoritmo y se presentó al público para discutir su implantación. Finalmente, pese al elevado número de críticas que acusaban a la NSA de debilitar voluntariamente al algoritmo para poder acceder a los mensajes cifrados, DES fue adoptado como algoritmo estándar federal en noviembre de 1976 y publicado en enero de 1977.

- **Cifrado**

Se trata de un algoritmo de cifrado en bloques de 64 bits de longitud, por lo que es necesario codificar previamente el mensaje para obtener una entrada hexadecimal. Recordamos que un carácter hexadecimal está compuesto por 8 bits, mientras que un carácter común puede codificarse empleando dos dígitos hexadecimales, esto es, 16 bits. Por tanto, DES recibirá entradas en bloques de 4 caracteres. Emplea claves de 64 bits y genera salidas cifradas de 64 bits, por lo que el texto cifrado tendrá la misma longitud.

Paso 1: creación de 16 sub-claves de 48 bits de longitud

La clave de 64 bits se permuta siguiendo la tabla PC-1 (Permutation Choice 1), de forma que únicamente 56 de los 64 bits se van a ver modificados; los 8 restantes serán empleados para paridad. En este caso, el 57º bit de la clave K se convertirá en el primer bit de la clave permutada K+ de 56bits.

PC-1							
57	49	41	33	25	17	9	
1	58	50	42	34	26	18	
10	2	59	51	43	35	27	
19	11	3	60	52	44	36	
63	55	47	39	31	23	15	
7	62	54	46	38	30	22	
14	6	61	53	45	37	29	
21	13	5	28	20	12	4	

Figura 1.19 Tabla PC-1 para DES.

$K = 00010011\ 00110100\ 01010111\ 01111001\ 10011011\ 10111100\ 11011111\ 11110001$

$K+ = 1111000\ 0110011\ 0010101\ 0101111\ 0101010\ 1011001\ 10011110001111$

A continuación, vamos a dividir la clave permutada en dos mitades de 28 bits cada una:

$C_0 = 1111000\ 0110011\ 0010101\ 0101111$ (izquierda)

$D_0 = 0101010\ 1011001\ 1001111\ 0001111$ (derecha)

Ahora generaremos 16 variaciones Cn Dn desplazando en cada caso m posiciones hacia la izquierda siguiendo la siguiente tabla:

Iteration Number	Number of Left Shifts
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

Figura 1.20 Tabla de desplazamientos hacia la izquierda.

$$C_1 = 1110000110011001010101011111$$

$$D_1 = 1010101011001100111100011110$$

$$C_2 = 110000110011001010101010111111$$

$$D_2 = 0101010110011001111000111101$$

$$C_3 = 00001100110010101010101111111$$

$$D_3 = 0101011001100111100011110101$$

El siguiente paso es formar 16 claves, aplicando la tabla PC-2 a cada concatenación **CnDn** .

PC-2					
14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Figura 1.21 Tabla PC-2 para DES.

Llegados a este punto, habremos obtenido las 16 sub-claves de 48 bits cada una.

Paso 2: codificar cada bloque de 64 bits de datos.

En primer lugar se producirá una permutación inicial (IP) de los 64 bits, siguiendo una tabla igual que en los casos anteriores.

IP							
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Figura 1.22 Tabla IP para DES.

M=0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

IP=1100 1100 0000 0000 1100 1100 1111 1111 1111 0000 1010 1010 1111 0000 1010 1010

Y se divide el bloque permutado en dos mitades de 32 bits:

$$L_0 = 1100 \ 1100 \ 0000 \ 0000 \ 1100 \ 1100 \ 1111 \ 1111$$

$$R_0 = 1111 \ 0000 \ 1010 \ 1010 \ 1111 \ 0000 \ 1010 \ 1010$$

Se define una función f (bloque-32b, clave-48b) de dos entradas y una salida. A partir de ahora usaremos el símbolo $+$ para referirnos a una suma XOR (suma bit a bit módulo 2). A continuación, para cada una de los 16 bloques permutados, realizamos la siguiente operación:

$$L_n = R_{n-1}$$

$$R_n = L_{n-1} + f(R_{n-1}, K_n)$$

A esta estructura, representada en la imagen 1.16, se le denomina Red de Feistel, y es uno de los métodos de cifrado en bloque más utilizados (no sólo se utiliza en DES).

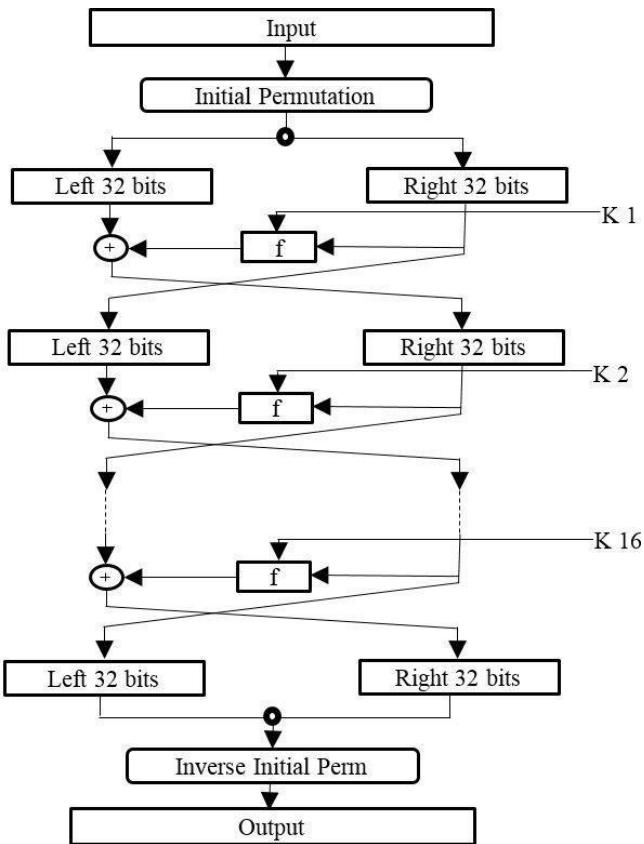


Figura 1.23 Red de Feistel para el algoritmo DES.

Faltaría explicar el funcionamiento de la función f . Inicialmente, se expande el contenido del bloque R_{n-1} de 32 a 48 bits empleando una nueva tabla, denominada E que, de la misma forma que las anteriores, permite formar bloques de 48 bits duplicando algunos bits ya existentes.

E BIT-SELECTION TABLE

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Figura 1.24 Tabla E para la extensión de un bloque de 32 bits.

De este modo, un bloque R se extendería de la siguiente forma:

$$R_0 = 1111\ 0000\ 1010\ 1010\ 1111\ 0000\ 1010\ 1010$$

$$E(R_0) = 011110\ 100001\ 010101\ 010101\ 011110\ 100001\ 010101\ 010101$$

Para seguir calculando la salida de la función f , realizamos la operación **XOR** entre $E(R_{n-1})$ y la clave K_n .

$$\begin{aligned}
 K_1 &= 0001101100000010111011111111000111000001110010 \\
 E(R_0) &= 01111010001010101010101110100001010101010101 \\
 K_1 + E(R_0) &= 011000 010001 011110 111010 100001 100110 010100 100111
 \end{aligned}$$

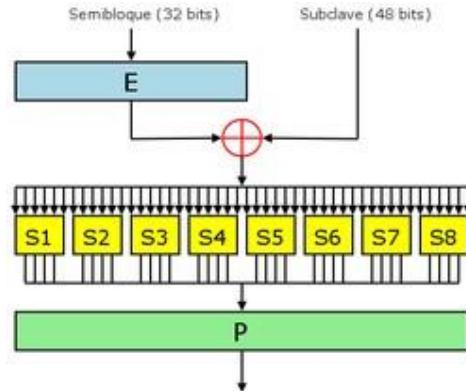


Figura 1.25 Esquema de la función f de Feistel.

Llegados a este punto, tenemos que introducir las **Cajas-S**. Se trata de una serie de tablas que contienen entradas con números de 4 bits. Habrá un total de 8 Cajas-S, y usaremos cada uno de los 8 grupos de bits obtenidos en $K_1 + E(R_0)$ como una dirección para cada una de las cajas. De esta forma, en la caja S_1 buscaremos el número con la dirección 011000. Como resultado obtendremos una salida de 32 bits.

		4 bit de entrada internos															
		S ₅		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101
Bits externos	00	0010	1100	0100	0001	0111	1010	1011	0110	1000	0101	0011	1111	1101	0000	1110	1001
	01	1110	1011	0010	1100	0100	0111	1101	0001	0101	0000	1111	1100	0011	1001	1000	0110
	10	0100	0010	0001	1011	1100	1101	0111	1000	1111	1001	1100	0101	0110	0011	0000	1110
	11	1011	1000	1100	0111	0001	1110	0010	1101	0110	1111	0000	1001	1100	0100	0101	0011

Figura 1.26 Ejemplo de Caja-S.

Las tablas correspondientes a las Cajas-S son siempre las mismas, y se encuentran recogidas en el **apéndice 1**.

En el ejemplo que hemos ido siguiendo, obtendríamos la siguiente salida:

$$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8) = 0101 1100 1000 0010 1011 0101 1001 0111$$

El paso final para obtener nuestra f es realizar una permutación del resultado anterior:

$$f = P(S_1(B_1)S_2(B_2) \dots S_8(B_8))$$

De nuevo nos referiremos a otra tabla, P , en la que se recoge el nuevo orden de los 32 bits:

P			
16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Figura 1.27 Tabla P para DES.

$$f = 0010\ 0011\ 0100\ 1010\ 1010\ 1001\ 1011\ 1011$$

$$R_1 = L_0 + f(R_0, K_1) =$$

$$\begin{aligned} &= 1100\ 1100\ 0000\ 0000\ 1100\ 1100\ 1111\ 1111 \\ &+ 0010\ 0011\ 0100\ 1010\ 1010\ 1001\ 1011\ 1011 \\ &= 1110\ 1111\ 0100\ 10100\ 1100\ 101\ 0100\ 0100 \end{aligned}$$

Si nos referimos de nuevo al esquema de Feistel de la figura 1.16, veremos que el siguiente paso es tomar $L_2 = R_1$, que sería lo que acabamos de calcular, y volver a realizar el paso anterior para $R_2 = L_1 + f(R_1, K_2)$. Esto continuaría así durante 16 rondas, hasta obtener L_{16} y R_{16} .

A continuación invertimos el orden de estos dos bloques y aplicamos IP^{-1} (la tabla “inversa” de la permutación inicial):

IP ⁻¹							
40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Figura 1.28 Tabla IP-1 para DES.

De la siguiente manera:

$$L_{16} = 0100\ 0011\ 0100\ 0010\ 0011\ 0010\ 0011\ 0100$$

$$R_{16} = 0000\ 1010\ 0100\ 1100\ 1101\ 1001\ 1001\ 0101$$

$$R_{16}L_{16} = 00001010\ 01001100\ 11011001\ 10010101\ 01000011\ 01000010\ 00110010\ 00110100$$

$$IP^{-1} = 10000101\ 11101000\ 00010011\ 01010100\ 00001111\ 00001010\ 10110100\ 00000101$$

Que en hexadecimal quedaría **85E813540F0AB405**.

Portanto, para el mensaje **M=0123456789ABCDE**, obtenemos el cifrado **C=85E813540F0AB405**.

- **Descifrado**

El proceso de descifrado es exactamente igual que el de cifrado, pero aplicando las 16 claves en orden inverso.

- **Enlaces de interés:**

[The DES Algorithm Illustrated \(J. Orlin Grabbe\)](#)

1.6.5 Triple-DES

A finales del siglo XX ya se habían descubierto varias vulnerabilidades para el algoritmo DES, por lo que era necesario adoptar un nuevo algoritmo como estándar. Una de las primeras ideas fue la de mantener el mismo algoritmo, pero triplicando las rondas de la Red de Feistel, llegando a un total de 48. A este método se le denominó **Triple-DES** y, pese a ser más seguro, también triplicaba los tiempos de cifrado y descifrado, por lo que resultó poco eficiente.

Para otorgar alguna capa de seguridad extra, se propuso un modo de implementación de Triple-DES que consistía en someter el mensaje a un proceso de cifrado-descifrado-cifrado. El truco estaba en que se usaba una clave distinta a la inicial para el descifrado intermedio por lo que, en vez de recuperar el mensaje inicial, lo complicaba todavía más.

1.6.6 Advanced Encryption Standard

Las conocidas vulnerabilidades del DES propiciaron la publicación de un nuevo concurso por parte del *National Bureau of Standards (NBS)* que, por entonces, había pasado a denominarse *National Institute of Standards and Technology (NIST)* en 1997. El objetivo era el de escoger un nuevo algoritmo capaz de sustituir al propuesto por *Feistel* e *IBM* anteriormente, para convertirse en el nuevo estándar del siglo XXI, denominado *Advanced Encryption Standard (AES)*.

Se admitieron numerosos algoritmos en la votación, pero únicamente cinco llegaron hasta la votación final, que tuvo lugar el 2 de octubre del año 2000:

- **MARS**
- **RC6**
- **RIJNDAEL**
- **SERPENT**
- **TWOFISH**

Finalmente, RIJNDAEL ganó el concurso y, en noviembre de 2001 fue publicado como estándar FIPS 197.

En cuanto al algoritmo empleado, hemos decidido resumir aquí su funcionamiento al más alto nivel debido a la complejidad de algunos términos matemáticos empleados, especialmente relacionados con el álgebra abstracta, como la Teoría de Grupos de Galois. Para mayor profundidad, en el *apéndice 2* se recoge el algoritmo completo.

A grandes rasgos, Rijndael basa toda su álgebra en la idea de contemplar los bytes como elementos de un anillo de polinomios de grado menor que 8. Esto quiere decir que el byte '00' corresponde al polinomio $p(x) = 0$, el '01' al polinomio $p(x) = 1$, el '11' al $p(x) = x + 1 \dots$

Esta aproximación aporta una serie de características relacionada con la aparición de polinomios irreducibles y operaciones modulares, entre otras, dando lugar a la base matemática necesaria para que Rijndael funcione correctamente.

- **Cifrado**

El algoritmo se detalla en el Anexo 2, apartados 3 al 7. En resumen, como se aprecia en la imagen 1.27, Rijndael consta de una fase de expansión de la clave, una adición de la *Round Key*, un número fijo de Rondas y una Ronda Final.

```
Rijndael(State,CipherKey)
{
    KeyExpansion(CipherKey,ExpandedKey) ;
    AddRoundKey(State,ExpandedKey);
    For( i=1 ; i<Nr ; i++ ) Round(State,ExpandedKey + Nb*i) ;
    FinalRound(State,ExpandedKey + Nb*Nr);
}
```

Figura 1.29 Pseudocódigo de Rijndael.

Cada Ronda realizará una serie de operaciones que, de nuevo, se desarrollan en el Anexo 2, apartado 3. Cada *Round Key* se obtendrá a partir de la Clave Expandida, cuya generación se explica en el Anexo 2, apartado 4.

```

Round (State, RoundKey)
{
    ByteSub (State) ;
    ShiftRow (State) ;
    MixColumn (State) ;
    AddRoundKey (State, RoundKey) ;
}

```

Figura 1.30 Pseudocódigo de una ronda en Rijndael.

```

FinalRound (State, RoundKey)
{
    ByteSub (State) ;
    ShiftRow (State) ;
    AddRoundKey (State, RoundKey) ;
}

```

Figura 1.31 Pseudocódigo de la ronda final en Rijndael.

- **Descifrado**

El proceso de descifrado es similar al de cifrado, pero requiere realizar las operaciones en orden inverso. Es importante que las operaciones de descifrado sean las inversas de las operaciones de cifrado, lo que añade cierto nivel de complejidad al algoritmo. Además, será necesario calcular la Clave Expandida inversa. De nuevo, en el Anexo 2, apartado 7, se recogen todos los detalles relativos a la implementación del proceso de descifrado.

```

I_Rijndael (State, CipherKey)
{
    I_KeyExpansion (CipherKey, I_ExpandedKey) ;
    AddRoundKey (State, I_ExpandedKey+ Nb*Nr) ;
    For( i=Nr-1 ; i>0 ; i-- ) Round (State, I_ExpandedKey+ Nb*i) ;
    FinalRound (State, I_ExpandedKey) ;
}

```

Figura 1.32 Inversa del cifrado Rijndael.

En conclusión, Rijndael supuso la aparición de un algoritmo muy complejo, que además fue específicamente diseñado para ser resistente a las técnicas por entonces conocidas de criptoanálisis lineal y diferencial, para lo que se basó en la Wide Trail Strategy: una estrategia creada con ese propósito. Además, el algoritmo resulta fácil de implementar, presentando características de paralelización que permiten una optimización de tiempo y recursos a la hora de cifrar (aunque el proceso de descifrado es algo más costoso).

Rijndael ha demostrado ser resistente a casi todos los ataques conocidos, en función de las longitudes de bloque y clave empleadas, y el número de rondas aplicadas. En concreto, para claves de 128 bits y un total de 10 rondas, el algoritmo resulta bastante robusto frente a técnicas shortcut (es decir, que no incluyen fuerza bruta).

1.6.7 IDEA (International Data Encryption Algorithm)

Se trata de un algoritmo simétrico de cifrado en bloques diseñado por James Massey y Xuejia Lai, ideado en 1991 como posible sucesor de DES. Aunque en su día fue un algoritmo privado y patentado, actualmente su uso es libre desde 2012, año en que expiró la patente. Fue usado en PGP v2.0 y se encuentra entre los estándares opcionales de OpenPGP.

IDEA admite claves de 128bits, operando bloques de 64bits de longitud. Emplea una estructura denominada **Lai-Massey scheme**, cuyo funcionamiento se recoge en la siguiente figura:

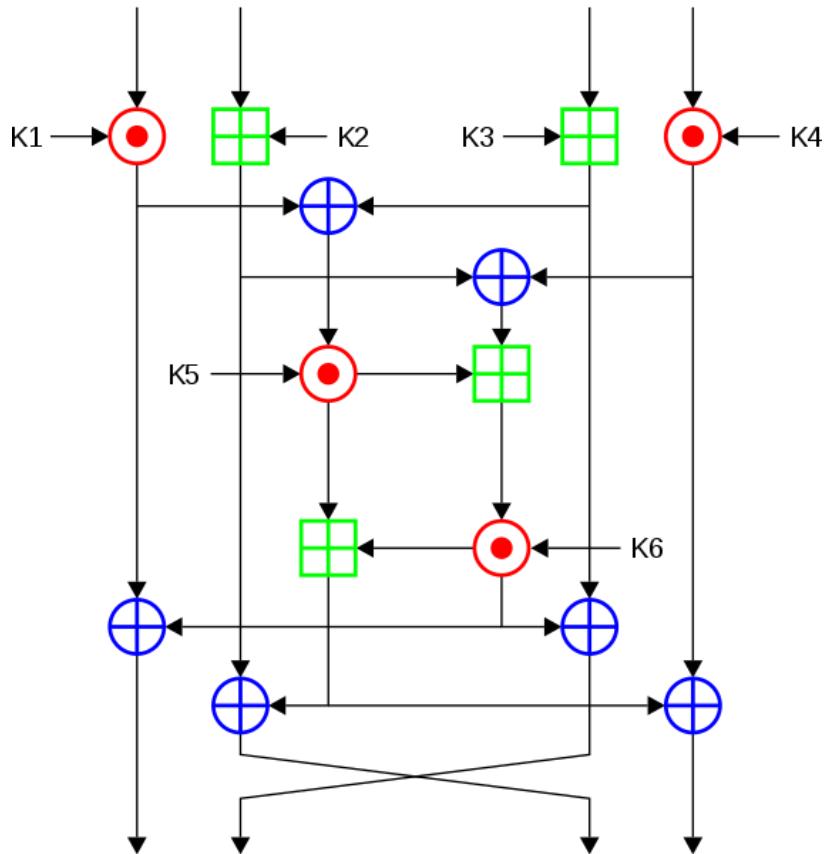


Figura 1.33 Esquema Lai-Massey para IDEA.

Aquí, los operadores verdes indican una suma módulo 2^{16} , los azules un bitwise XOR y los rojos una multiplicación módulo $2^{16} + 1$.

El algoritmo cuenta con 8 rondas del esquema anterior, seguidas por una “media ronda” que funciona de la siguiente forma:

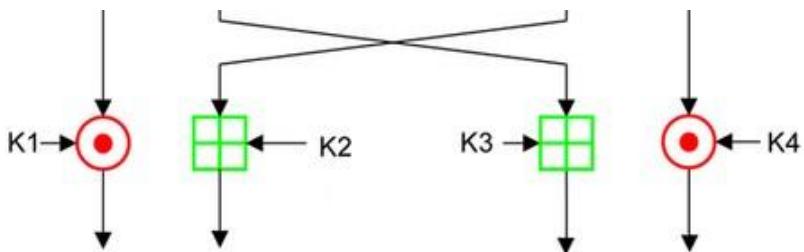


Figura 1.34 "Media Ronda" final de Lai-Massey

- **Obtención de subclaves**

Cada ronda emplea 6 sub-claves de 16 bits, mientras que la “media ronda” usa 4. En total, son necesarias 52 sub-claves. Las primeras 8 sub-claves se extraen directamente de la clave K , tomando los bits de la clave en grupos de 16. Los siguientes grupos de 8 claves se obtienen de la misma forma, rotando la clave 25 bits a la izquierda entre grupos.

Este proceso es bastante sencillo, y de hecho supone la principal vulnerabilidad del algoritmo: una clave que contenga muchos ceros producirá un cifrado bastante endeble.

- **Conclusión**

El algoritmo IDEA es bastante simple y, pese a la vulnerabilidad de claves débiles mencionada (que se soluciona evitando aquellas claves que contengan muchos ceros), se ha incluido en diversas herramientas (aunque en una versión posterior, denominada IDEA NXT). De hecho está incluido dentro de los algoritmos estándar de OpenSSL (en sus modos CBC, CFB y OFB).

1.6.8 Blowfish

Ya mencionamos que en el concurso de selección del Advanced Encryption Standard se presentaron varios algoritmos prometedores. Uno de ellos fue *Twofish*, ideado por el famoso criptólogo Bruce Schneier. Sin embargo, antes de entrar a explicar *Twofish* vamos a pasar por su predecesor, *Blowfish*.

Mantiene la estructura de una *Red de Feistel* para cifrado en bloques, iterando una función de cifrado simple 16 veces. Emplea bloques de 64 bits de longitud y una longitud de clave de hasta 448 bits.

- **Cifrado**

El cifrado en *Blowfish* consta de dos etapas: expansión de la clave y cifrado de datos. La expansión convierte una clave de, como mucho, 448 bits en varios arrays de sub-claves con un total de 4168 bytes.

El cifrado de datos, como ya se ha mencionado, se produce a través de una *Red de Feistel* de 16 etapas. Cada ronda consiste en una permutación dependiente de la clave, seguida de una sustitución. Todas las operaciones empleadas con XORs y sumas con palabras de 32 bits.

- **Generación de subclaves**

Un P-array está formado por 18 sub-claves de 32 bits cada una, y se cuenta con un total de 32 Cajas-S de 256 entradas cada una.

El primer paso es inicializar el primer P-array con valores fijos, arbitrarios para, a continuación, ejecutar XORs entre P_i y los 32 bits i-ésimos de la clave. De esta forma, los 32 primeros bits de la clave y P_1 pasan por el XOR; posteriormente, los 32 bits siguientes con P_2 ... Así hasta un posible P_{14} (para el caso de claves de 448 bits). Esta operación se ejecuta de forma cíclica hasta haber pasado por el P-array al completo (las 18 posiciones), de forma que, cuando se llegue al final de la clave, el siguiente XOR se hará con los 32 primeros bits de la clave.

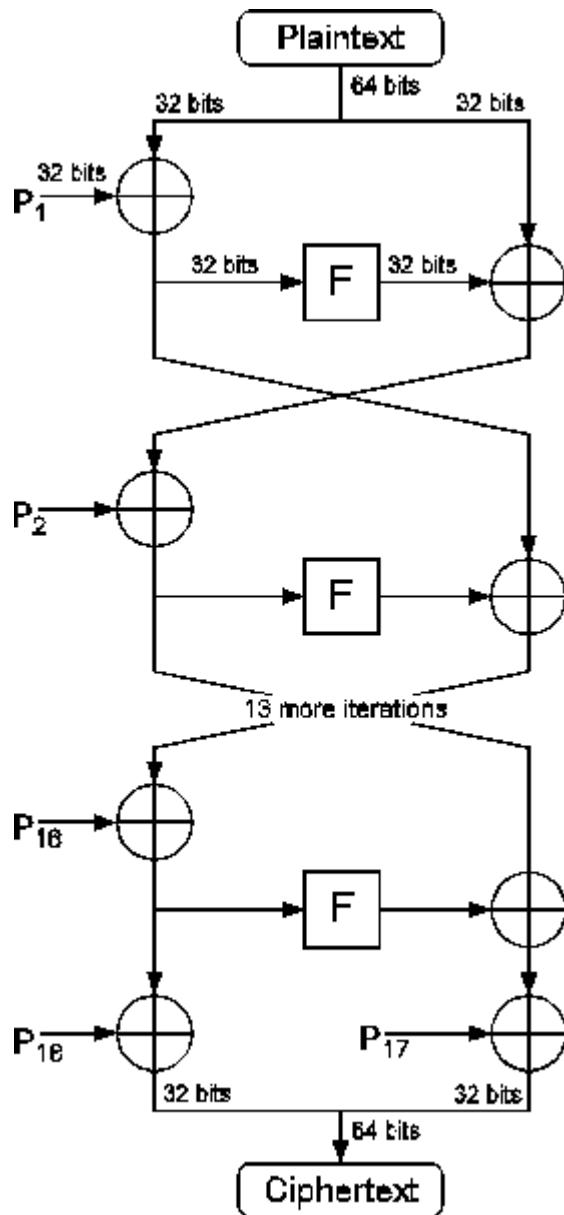


Figura 1.35 Red de Feistel para Blowfish.

El siguiente paso consiste en cifrar un string inicializado con todo ceros usando el algoritmo de la Red de Feistel de Blowfish representado en la imagen 1.35, donde las sub-claves K_1, \dots, K_{16} son los resultados obtenidos de los XOR anteriores.

```

Divide x into two 32-bit halves: xL, xR
For i = 1 to 16:
  xL = xL XOR Pi
  xR = F(xL) XOR xR
  Swap xL and xR
  Next i
  Swap xL and xR (Undo the last swap.)
  xR = xR XOR P17
  xL = xL XOR P18
  Recombine xL and xR

```

Figura 1.36 Pseudocódigo de la Red de Feistel en Blowfish.

La función f , en este caso difiere ligeramente de la empleada en el cifrado DES: en vez de emplear 8 Cajas-S, usará sólo 4. Para ello se divide los 32 bits de la parte “izquierda” del resultado de la iteración previa en 4 partes de 8 bits cada una (llamadas a , b , c y d), y aplica la siguiente función:

$$F(x_L) = ((S1, a + S2, b \bmod 2^{32}) \text{ XOR } S3, c) + S4, d \bmod 2^{32}$$

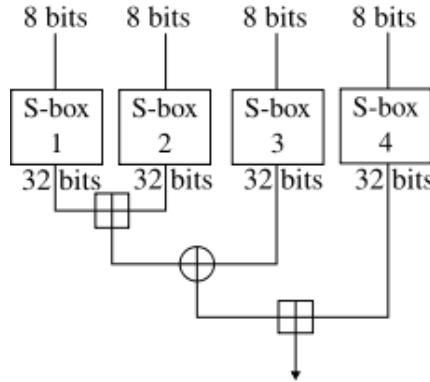


Figura 1.37 Representación gráfica de la función f de Blowfish.

A continuación, se reemplazarán P_1 y P_2 por la salida de la Red de Feistel. Dicha salida volverá a cifrarse empleando las claves modificadas, reemplazando esta vez a P_3 y P_4 . Este proceso se repetirá hasta modificar el P-array al completo y, posteriormente, las entradas de las cuatro Cajas-S. En total, serán necesarias 521 iteraciones para generar todas las sub-claves (es preferible que una aplicación las almacene para una misma clave, en vez de recalcularlas cada vez, asumiendo que para una conexión la clave no sea modificada).

- **Cifrado**

Una vez hemos generado todas las sub-claves, tendremos también unas Cajas-S “personalizadas” para nuestra clave. Llegados a este punto, dividiremos el texto plano en bloques de 64 bits, los cuales introduciremos en la Red de Feistel, usando el P-array generado en la fase anterior. A la salida de las 16 rondas, tendremos el mensaje cifrado.

- **Descifrado**

El proceso de descifrado es perfectamente análogo, introduciendo las sub-claves del P-array en orden contrario ($P_{16}, P_{15}, \dots, P_1$).

- **Conclusiones**

Blowfish es, entonces, un algoritmo de cifrado por bloques basado en Redes de Feistel, que cuenta con una fase de generación de sub-claves bastante costosa debido al número de iteraciones necesarias, lo que lo hace bastante resistente frente a ataques por fuerza bruta. Además, el hecho de que las Cajas-S dependan de la clave escogida, evita que las técnicas de criptoanálisis lineal y diferencial tengan éxito.

Por otro lado, el algoritmo de generación de sub-claves no necesita que la clave inicial sea aleatoria. De hecho, cadenas de bits que presenten una alta correlación producirán sub-claves aleatorias, con elevada entropía.

Finalmente, una de las características más llamativas de Blowfish fue que, en una época en la que empezaba a crecer la cantidad de algoritmos de cifrado privados, Schneier decidió crear una herramienta open source, que todo el mundo pudiera modificar e implementar libremente. Por ese motivo, a lo largo de los años ha sido fruto de numerosos estudios por parte de criptoanalistas, llegando a la conclusión de que es prácticamente irrompible salvo realizando búsquedas exhaustivas en el espacio de claves (lo que, para 256 bits, resulta inviable incluso hoy en día).

Sin embargo, hoy en día se considera débil: no está reconocido como estándar en el FIPS 140-2 y se han encontrado algunas vulnerabilidades.

- **Enlaces de interés**

[Original Schneier et al. paper](#)

1.6.9 Twofish

En 2007, el NIST organizó una nueva competición para elegir un nuevo estándar de hashing, con el objetivo de sustituir a los antiguos SHA-1 y SHA-2, que en ese momento habían presentado vulnerabilidades por colisiones encontradas. El candidato propuesto por Bruce Schneier y compañía fue *Twofish*: un algoritmo de cifrado en bloque basado en el antiguo *Blowfish* que, adicionalmente, podría usarse como una función hash robusta y eficiente.

Twofish admite claves de 128, 192 y 256 bits y, al igual que su predecesor, busca minimizar el uso de operaciones aritméticas que afecten a la eficiencia o fiabilidad del algoritmo. Además, fue diseñado buscando los siguientes objetivos:

- Una implementación de 16 rondas permanecerá segura frente a ataques de texto arbitrario que cuenten con menos de 2^{80} muestras y que consuman menos de 2^N tiempo, donde N es la longitud de clave.
- Una implementación de 12 rondas permanecerá segura frente a ataques a la clave que usen menos de 2^{64} muestras y consuman menos de $2^{N/2}$ tiempo.

Al igual que su predecesor, *Twofish* se basa en el empleo de una *Red de Feistel* de 16 rondas y de cuatro Cajas-S dependientes de la clave.

Matrices MDS

Una de sus características adicionales frente a *Blowfish* es el empleo de *Matrices MDS* (*Maximum Distance Separable*). Una matriz MDS entre dos campos a y b no es más que una aplicación lineal con la propiedad de que la mínima cantidad de elementos no-nulos en un vector no-nulo es estrictamente mayor que b . Se puede demostrar que no existe ninguna otra aplicación que tenga una distancia mayor entre dos vectores. Una condición necesaria y suficiente para que una matriz sea MDS es que todos sus menores sean no-nulos.

Transformadas de Pseud-Hadamard (PHT)

Una transformada Pseudo-Hadamard (PHT) es una función bastante simple que permite obtener difusión criptográfica con operaciones a nivel de bit. *Twofish* emplea PHTs de 32 bits que pueden ser ejecutadas en tan solo dos opcodes en la mayoría de procesadores modernos. La transformada se obtiene de la siguiente forma:

$$a' = a + b \bmod 2^n$$

$$b' = a + 2b \bmod 2^n$$

En ambos casos, n representa el número de bits a utilizar.

Whitening

La técnica de *whitening* consiste en ejecutar un XOR con el material clave del algoritmo antes de introducirlo en la primera ronda. Como hemos explicado previamente, esta técnica ya se usaba en algoritmos como DES, ya que se ha demostrado un incremento sustancial en la robustez del algoritmo al aplicarla. Esto es porque esconde los inputs específicos de la entrada y la salida del algoritmo de cara a un atacante.

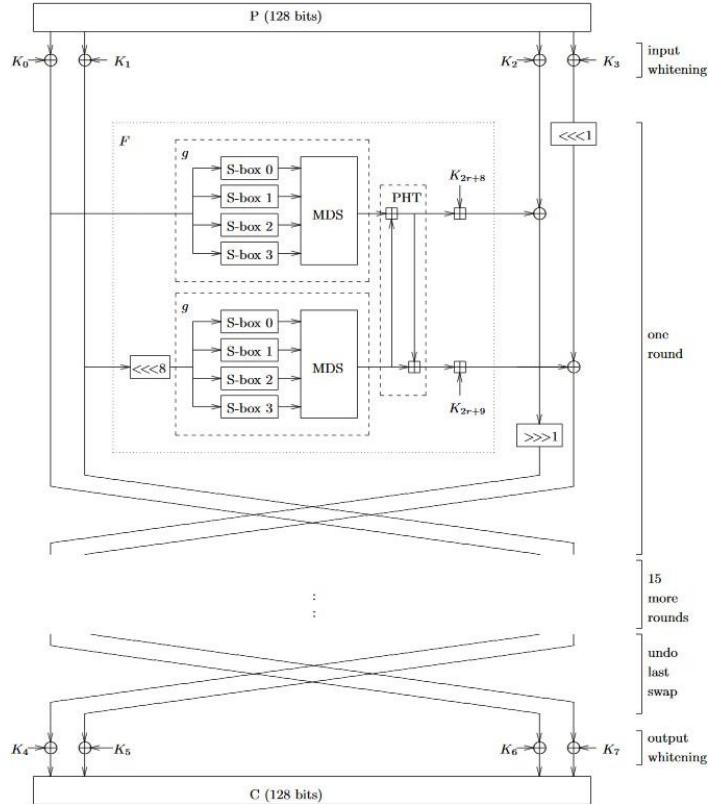


Figura 1.38 Esquema de funcionamiento de Twofish.

Twofish recibe bloques de 128 bits que divide en cuatro fragmentos de 128 bits cada uno.

- **Obtención de las Round-Keys**

Al igual que *Blowfish*, es necesario obtener una serie de *round-keys* a partir de la clave original. En este caso, se necesitan 40 claves ($K_0 \dots K_{39}$). Como ya hemos comentado, este algoritmo acepta claves de 128, 192 y 256 bits, pero cualquier longitud menor a este rango puede ser rellenada con ceros hasta el siguiente rango aceptado.

Definimos $k = N/64$, donde N es el tamaño de la clave. La clave M estará formada por $8k$ bytes m_0, \dots, m_{8k-1} . Estos bytes se convierten previamente a palabras de 32 bits cada una:

$$M_i = \sum_{j=0}^3 m_{(4i+j)} \cdot 2^{8j} \quad i = 0, \dots, 2k-1$$

Y obtenemos así dos vectores de longitud k :

$$\begin{aligned} M_e &= (M_0, M_2, \dots, M_{2k-2}) \\ M_o &= (M_1, M_3, \dots, M_{2k-1}) \end{aligned}$$

Se obtiene también un tercer vector de longitud k a partir de la clave. Para ello, se toman los bytes de la clave en grupos de 8 y se interpretan como vectores sobre $GF(2^8)$ (*¿Te suena de AES?*), y se multiplican por una matriz 4×8 derivada de un código de corrección de errores de *Reed-Solomon* (matriz con la característica MDS).

$$\begin{pmatrix} s_{i,0} \\ s_{i,1} \\ s_{i,2} \\ s_{i,3} \end{pmatrix} = \begin{pmatrix} \cdot & \cdots & \cdot \\ \vdots & \text{RS} & \vdots \\ \cdot & \cdots & \cdot \end{pmatrix} \cdot \begin{pmatrix} m_{8i} \\ m_{8i+1} \\ m_{8i+2} \\ m_{8i+3} \\ m_{8i+4} \\ m_{8i+5} \\ m_{8i+6} \\ m_{8i+7} \end{pmatrix}$$

$$S_i = \sum_{j=0}^3 s_{i,j} \cdot 2^{8j}$$

for $i = 0, \dots, k-1$, and

$$S = (S_{k-1}, S_{k-2}, \dots, S_0)$$

En este caso, para la multiplicación por la matriz RS trabajaremos sobre $\text{GF}(2^8)$ módulo $w(x) = x^8 + x^6 + x^3 + x^2 + 1$. La matriz RS viene entonces dada por:

$$\text{RS} = \begin{pmatrix} 01 & A4 & 55 & 87 & 5A & 58 & DB & 9E \\ A4 & 56 & 82 & F3 & 1E & C6 & 68 & E5 \\ 02 & A1 & FC & C1 & 47 & AE & 3D & 19 \\ A4 & 55 & 87 & 5A & 58 & DB & 9E & 03 \end{pmatrix}$$

- **Función F**

La función F es una permutación dependiente de la clave sobre valores de 64 bits. Toma tres argumentos: dos palabras R_0 y R_1 , y el número de rondas. R_0 pasará por la función g , dando lugar a T_0 , mientras que R_1 rotará a la izquierda 8 bits antes de pasar por la función g y generar T_1 .

A continuación, T_0 y T_1 se combinan en una PHT y se añaden dos palabras de la clave expandida (cuyo funcionamiento mostraremos más adelante):

$$\begin{aligned} T_0 &= g(R_0) \\ T_1 &= g(\text{ROL}(R_1, 8)) \\ F_0 &= (T_0 + T_1 + K_{2r+8}) \bmod 2^{32} \\ F_1 &= (T_0 + 2T_1 + K_{2r+9}) \bmod 2^{32} \end{aligned}$$

Figura 1.39 Función F de Twofish.

Análogamente, existirá una función F' idéntica a F , que no añadirá las palabras de la clave expandida al final.

- **Función g**

Se trata del núcleo de Twofish. La palabra de entrada se divide en 4 bytes, cada uno de los cuales pasará por su correspondiente Caja-S. Los cuatro resultados se interpretan como elementos del ya conocido grupo finito $\text{GF}(2^8)$ y se multiplican por la matriz MDS 4x4. El resultado obtenido se interpreta como una palabra de 32 bits, que será el resultado de g .

$$\begin{aligned}
x_i &= \lfloor X/2^{8i} \rfloor \bmod 2^8 \quad i = 0, \dots, 3 \\
y_i &= s_i[x_i] \quad i = 0, \dots, 3 \\
\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} &= \begin{pmatrix} \cdot & \cdots & \cdot \\ \vdots & \text{MDS} & \vdots \\ \cdot & \cdots & \cdot \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} \\
Z &= \sum_{i=0}^3 z_i \cdot 2^{8i}
\end{aligned}$$

Figura 1.40 Función g de Twofish.

En este caso, $s_i[x_i]$ representa el paso de cada byte por su Caja-S correspondiente. La correspondencia entre bytes y los elementos del anillo polinómico es similar a la empleada en la obtención de las *round-keys*.

$$\text{MDS} = \begin{pmatrix} 01 & \text{EF} & 5B & 5B \\ 5B & \text{EF} & \text{EF} & 01 \\ \text{EF} & 5B & 01 & \text{EF} \\ \text{EF} & 01 & \text{EF} & 5B \end{pmatrix}$$

Figura 1.41 Matriz MDS para la función g de Twofish.

- **Función h**

Llegados a este punto, es necesario introducir una nueva función que será utilizada para el cálculo de las Cajas-S modificadas y la obtención de la Clave Expandida. Dicha función toma dos entradas: una palabra de 32 bits X y una lista L de longitud k , con elementos de 32 bits.

La función h consta de k etapas, en cada cual los 4 bytes pasan por sus respectivas Cajas-S y se operan XOR con un byte derivado de la lista. Finalmente, los bytes pasan una vez más por una Caja-S fija y se multiplican por la matriz MDS al igual que en g .

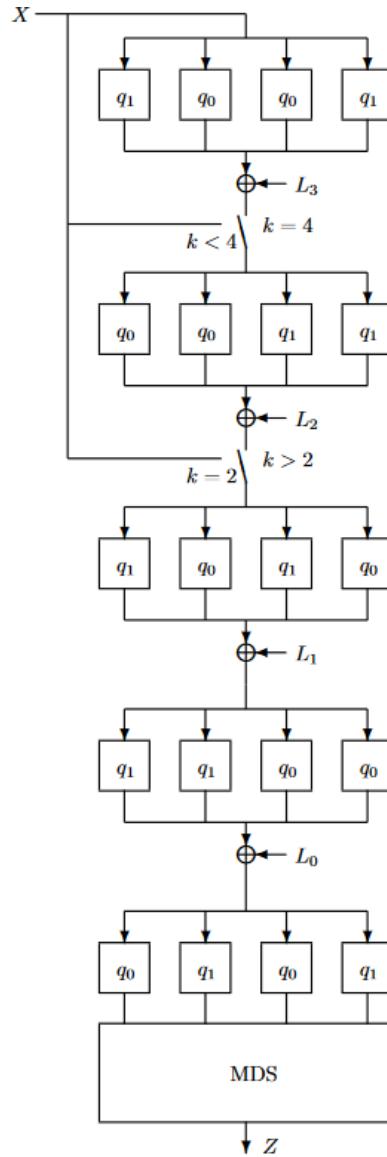


Figura 1.42 Esquema de funcionamiento de la función h de Twofish.

En el esquema de la figura 1.42 podemos apreciar que se emplean dos permutaciones, q_0 y q_1 , que explicaremos más adelante.

- **Cajas-S dependientes de clave**

Podemos definir las Cajas-S de la función g como:

$$g(X) = h(X, S)$$

En este caso, la lista L es igual al vector S derivado de la clave (formado por las *round-keys*).

- **Clave expandida**

Al igual que en *Blowfish*, este algoritmo necesita ampliar la clave. De hecho, ya hemos visto que alguno de los pasos explicados previamente necesita esta clave expandida para funcionar (por ejemplo, la función **F**). Para ello, se empleará la función *h* definida:

$$\begin{aligned}
 \rho &= 2^{24} + 2^{16} + 2^8 + 2^0 \\
 A_i &= h(2i\rho, M_e) \\
 B_i &= \text{ROL}(h((2i+1)\rho, M_o), 8) \\
 K_{2i} &= (A_i + B_i) \bmod 2^{32} \\
 K_{2i+1} &= \text{ROL}((A_i + 2B_i) \bmod 2^{32}, 9)
 \end{aligned}$$

Figura 1.43 Obtención de la clave expandida en Twofish.

Es importante comentar que las operaciones ROL y ROR indican rotaciones a la izquierda y a la derecha, respectivamente.

- **Permutaciones q_0 y q_1**

Se trata de dos permutaciones fijas sobre valores de 8 bits. Para un valor de entrada x , definimos su valor de salida y como:

$$\begin{aligned}
 a_0, b_0 &= \lfloor x/16 \rfloor, x \bmod 16 \\
 a_1 &= a_0 \oplus b_0 \\
 b_1 &= a_0 \oplus \text{ROR}_4(b_0, 1) \oplus 8a_0 \bmod 16 \\
 a_2, b_2 &= t_0[a_1], t_1[b_1] \\
 a_3 &= a_2 \oplus b_2 \\
 b_3 &= a_2 \oplus \text{ROR}_4(b_2, 1) \oplus 8a_2 \bmod 16 \\
 a_4, b_4 &= t_2[a_3], t_3[b_3] \\
 y &= 16b_4 + a_4
 \end{aligned}$$

Figura 1.44 Algoritmo de permutación q_0 / q_1 .

Aquí, ROR_4 es una función idéntica a ROR, pero con rotaciones de 4 bits. Como se puede observar, el primer paso es dividir el byte en dos nibbles que, tras combinarse con algunas operaciones, van a pasar por unas Cajas-S definidas en las figuras 1.45 y 1.46. A continuación, se procede a otra combinación y a otra sustitución a través de otra Caja-S para, finalmente, combinar ambos nibbles transformados en un único byte. La única diferencia entre q_0 y q_1 es la Caja-S empleada.

$$\begin{aligned}
 t_0 &= [8 \ 1 \ 7 \ D \ 6 \ F \ 3 \ 2 \ 0 \ B \ 5 \ 9 \ E \ C \ A \ 4] \\
 t_1 &= [E \ C \ B \ 8 \ 1 \ 2 \ 3 \ 5 \ F \ 4 \ A \ 6 \ 7 \ 0 \ 9 \ D] \\
 t_2 &= [B \ A \ 5 \ E \ 6 \ D \ 9 \ 0 \ C \ 8 \ F \ 3 \ 2 \ 4 \ 7 \ 1] \\
 t_3 &= [D \ 7 \ F \ 4 \ 1 \ 2 \ 6 \ E \ 9 \ B \ 3 \ 0 \ 8 \ 5 \ C \ A]
 \end{aligned}$$

Figura 1.45 Caja-S para q_0 .

$$\begin{aligned}
 t_0 &= [2 \ 8 \ B \ D \ F \ 7 \ 6 \ E \ 3 \ 1 \ 9 \ 4 \ 0 \ A \ C \ 5] \\
 t_1 &= [1 \ E \ 2 \ B \ 4 \ C \ 3 \ 7 \ 6 \ D \ A \ 5 \ F \ 9 \ 0 \ 8] \\
 t_2 &= [4 \ C \ 7 \ 5 \ 1 \ 6 \ 9 \ A \ 0 \ E \ D \ 8 \ 2 \ B \ 3 \ F] \\
 t_3 &= [B \ 9 \ 5 \ 1 \ C \ 3 \ D \ E \ 6 \ 4 \ 7 \ F \ 2 \ 0 \ 8 \ A]
 \end{aligned}$$

Figura 1.46 Caja-S para q_1 .

- **Conclusión**

Twofish supuso un avance importante frente a su predecesor y, pese a haber sido concebido con el objetivo de funcionar como función hash, su utilidad como algoritmo de cifrado no pasó desapercibida. De hecho, hoy en día se considera incluso más seguro que Rijndael gracias al uso de Cajas-S dependientes de clave: esto hace que un atacante no tenga forma de conocer el contenido de la caja (mientras que en Rijndael no se modifican y se usan siempre las mismas tablas). Sin embargo, Rijndael fue escogido como estándar debido, entre otras cosas, a su eficiencia y velocidad.

En cuanto qué mejoras aporta *Twofish* frente a su predecesor, *Blowfish*, las principales son que admite una longitud de bloque mayor y ofrece una eficiencia superior. En general, su propio autor, Bruce Schneier, insiste en el empleo de *Twofish* como algoritmo de cifrado frente a *Blowfish*.

Existe una tercera versión del algoritmo, llamada *Threefish*. Sin embargo, fue creada específicamente para labores de *hashing* por lo que, pese a que pueda ser empleado como algoritmo de cifrado, está demasiado especializado en la optimización del método para el que fue concebido. Por lo tanto no lo tendremos en cuenta en esta sección.

- **Enlaces de interés**

[Twofish: A 128-Bit Block Cipher \(Schneier et al. 1998\)](#)

1.6.10 Rivest Cipher (RC6)

El Cifrado de Rivest, más conocido por sus siglas RC4, RC5 y RC6, es una serie de algoritmos de cifrado ideados por el criptógrafo Ronald L. Rivest y compañía. En su versión inicial, se trataba de un algoritmo de cifrado en bloque incluido en el protocolo WEP 802.11. Sin embargo, tras descubrirse numerosas vulnerabilidades (como los ataques por modificación), dejó de utilizarse.

En sus versiones posteriores, se realizó un enfoque de cifrado en bloque, con 32, 64 o 128 bits, y claves de hasta 2048 bits y hasta 255 rondas de cifrado. Estructuralmente muy parecidos, RC5 y RC6 difieren en el rendimiento: RC6 fue concebido para incluirse en el estándar AES, por lo que surgió de la optimización de RC5 para incrementar su velocidad.

RC6 recibe bloques de 128 bits (como solicitaba el estándar AES), para lo que emplea cuatro registros de 32 bits. Esta decisión se tomó en base a necesidad de extender el estándar a procesadores que no traten de manera eficiente los registros de 64 bits. En el *paper* original, se propone una nomenclatura del algoritmo que incluye mayor detalle: RC6-*w/r/b*, donde *w* representa el tamaño de la palabra en bits, *r* el número de rondas y *b* denota la longitud de la clave, en bytes. En concreto, las versiones de RC6 que aplican al estándar AES son las de 16, 24 y 32 bytes de clave (128, 192 y 256 bits).

En cuanto a las operaciones empleadas, todas se aplican sobre unidades de cuatro palabras de *w* bits. Todas las operaciones aritméticas serán módulo 2^w , tal y como recoge la siguiente descripción (*log w* representa la operación logaritmo en base 2):

$a + b$	integer addition modulo 2^w
$a - b$	integer subtraction modulo 2^w
$a \oplus b$	bitwise exclusive-or of <i>w</i> -bit words
$a \times b$	integer multiplication modulo 2^w
$a \ll b$	rotate the <i>w</i> -bit word <i>a</i> to the left by the amount given by the least significant $\lg w$ bits of <i>b</i>
$a \gg b$	rotate the <i>w</i> -bit word <i>a</i> to the right by the amount given by the least significant $\lg w$ bits of <i>b</i>

Figura 1.47 Resumen de las operaciones empleadas en RC6.

- **Expansión de la clave**

Sea la clave *K*, será expandida hasta obtener un array *S* de $2^*(r+4)$ palabras binarias derivadas de *K*. Para el algoritmo de expansión, es necesario definir dos *Constantes Mágicas*:

$$P_w = \text{Odd}((e - 2)2^w) \quad (1)$$

$$Q_w = \text{Odd}((\phi - 1)2^w) \quad (2)$$

Figura 1.48 Constantes Mágicas de RC6.

Donde

$$e = 2.718281828459\dots \text{ (base del logaritmo neperiano)}$$

$$\phi = 1.618033988749\dots \text{ (número áureo)}$$

Y la función *Odd(x)* toma el número impar más cercano a *x*. Para *w* = 16, 32 y 64, las constantes aparecerán a continuación en hexadecimal:

```

P16 = 1011011111100001 = b7e1
Q16 = 1001111000110111 = 9e37

P32 = 10110111111000010101000101100011 = b7e15163
Q32 = 10011110001101110111100110111001 = 9e3779b9

P64 = 1011011111100001010100010110001010001010111011010010101001101011
      = b7e151628aed2a6b
Q64 = 100111100011011101111001101110010111111010010100111110000010101
      = 9e3779b97f4a7c15

```

Figura 1.49 Constantes Mágicas en hexadecimal de RC6.

El primer paso del algoritmo de expansión de clave consiste en copiar la clave K de longitud b (en bytes) en un array $L[0\dots c-1]$ de longitud $c = \text{ceil}(b/u)$ palabras, donde $u=w/8$ y representa el número de bytes por palabra. Todos los bytes vacíos de L se rellenan con ceros.

En procesadores *Little-Endian* dicha tarea se puede implementar inicializando L con ceros y, a continuación, rellenando con la clave K :

```

for  $i = b - 1$  downto 0 do
     $L[i/u] = (L[i/u] \lll 8) + K[i];$ 

```

Figura 1.50 Algoritmo perteneciente a la expansión de clave RC6.

A continuación es necesario inicializar el array S de la clave expandida, con un valor fijo pseud-aleatorio, dependiente de la clave. Para ello se empleará una progresión aritmética módulo 2^w determinada por las *Constantes Mágicas*:

```

 $S[0] = P_w;$ 
for  $i = 1$  to  $t - 1$  do
     $S[i] = S[i - 1] + Q_w;$ 

```

Figura 1.51 Progresión para inicializar S en RC6.

Por último, procederá a mezclar la clave de cifrado con nuestra clave expandida, pasando tres veces por el array de menor longitud (S o L), y tantas veces como haga falta por el otro. A continuación se presenta el pseudocódigo del proceso, donde t y c representan las longitudes de S y L respectivamente:

```

 $i = j = 0;$ 
 $A = B = 0;$ 
do  $3 * \max(t, c)$  times:
     $A = S[i] = (S[i] + A + B) \lll 3;$ 
     $B = L[j] = (L[j] + A + B) \lll (A + B);$ 
     $i = (i + 1) \bmod(t);$ 
     $j = (j + 1) \bmod(c);$ 

```

Figura 1.52 Pseudocódigo de la mezcla de claves en RC6.

Como observación, el proceso de expansión de clave es relativamente *unidireccional*, ya que resulta complicado obtener K a partir de S .

- **Cifrado**

RC6 emplea cuatro registros de w bits: A, B, C y D. Cada uno contiene un fragmento del texto plano al inicio del algoritmo, que acabará convirtiéndose en un fragmento de la salida cifrada, la cual se almacenará en los mismos registros. El primer byte del texto plano se almacenará en el byte menos significativo del registro A, mientras que el último byte del mensaje acabará en el byte más significativo del registro D.

Encryption with RC6- $w/r/b$

Input: Plaintext stored in four w -bit input registers A, B, C, D
 Number r of rounds
 w -bit round keys $S[0, \dots, 2r + 3]$

Output: Ciphertext stored in A, B, C, D

Procedure:

```

 $B = B + S[0]$ 
 $D = D + S[1]$ 
for  $i = 1$  to  $r$  do
    {
         $t = (B \times (2B + 1)) \ll\ll \lg w$ 
         $u = (D \times (2D + 1)) \ll\ll \lg w$ 
         $A = ((A \oplus t) \ll\ll u) + S[2i]$ 
         $C = ((C \oplus u) \ll\ll t) + S[2i + 1]$ 
         $(A, B, C, D) = (B, C, D, A)$ 
    }
 $A = A + S[2r + 2]$ 
 $C = C + S[2r + 3]$ 

```

Figura 1.53 Algoritmo RC6 para el cifrado.

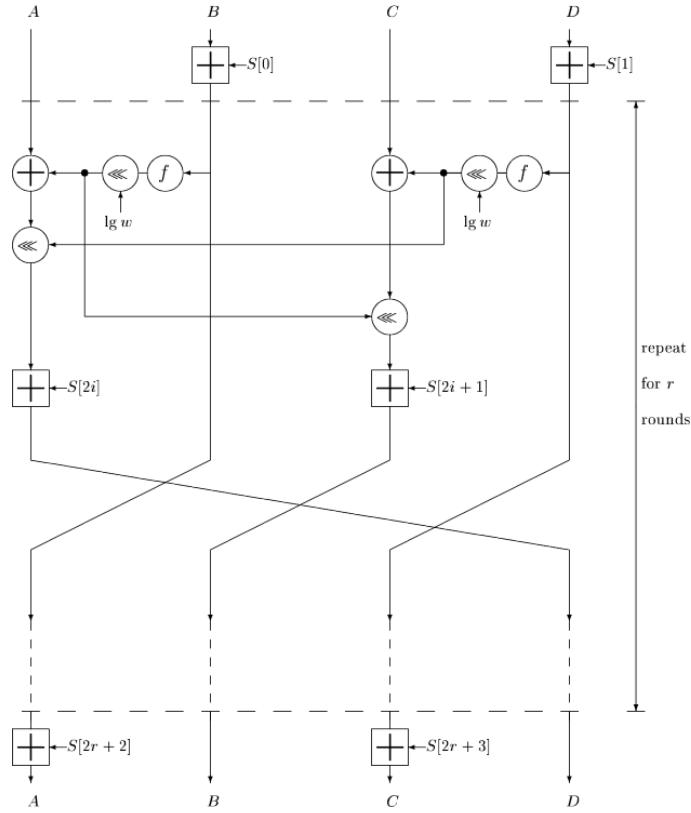


Figura 1.54 Esquema del algoritmo RC6 para cifrado $f(x) = x * (2x + 1)$.

• Descifrado

El descifrado es un proceso análogo, tal y como se recoge en el siguiente pseudocódigo:

Decryption with RC6-w/r/b

Input: Ciphertext stored in four w -bit input registers A, B, C, D
 Number r of rounds
 w -bit round keys $S[0, \dots, 2r + 3]$

Output: Plaintext stored in A, B, C, D

Procedure: $C = C - S[2r + 3]$
 $A = A - S[2r + 2]$
for $i = r$ **downto** 1 **do**
 {
 $(A, B, C, D) = (D, A, B, C)$
 $u = (D \times (2D + 1)) \ll\ll \lg w$
 $t = (B \times (2B + 1)) \ll\ll \lg w$
 $C = ((C - S[2i + 1]) \gg\gg t) \oplus u$
 $A = ((A - S[2i]) \gg\gg u) \oplus t$
 }
 $D = D - S[1]$
 $B = B - S[0]$

Figura 1.55 Pseudocódigo del proceso de descifrado de RC6.

- **Conclusión**

RC6 es un algoritmo de cifrado por bloques bastante simple y fácil de implementar. Sin embargo, presenta un alto índice de robustez, compitiendo seriamente con Rijndael. En el concurso para AES quedó finalista debido a que presentaba algunas dificultades de implementación en ciertas plataformas.

Pese a todo, se trata de un algoritmo relativamente habitual, que ha llegado a ser empleado en algunas comunicaciones confidenciales de gran importancia, aunque no es muy habitual encontrarlo en servicios comunes.

- **Enlaces de interés**

[The RC6 Block Cipher \(Ronald L. Rivest et al.\)](#)

1.6.11 Serpent

Otro de los finalistas del concurso del NIST para la elección del nuevo estándar AES fue *Serpent*. Se trata de un algoritmo de cifrado en flujo basado en el uso de una red de sustitución-permutación de 32 rondas. Fue propuesto por Ross Anderson, Eli Biham y Lars Knudsen.

Este algoritmo permite el uso de bloques de 128 bits y claves de 128, 192 y 256 bits (como solicitaba el NIST). Sin embargo, también acepta otras longitudes de clave.

Inicialmente divide el texto plano P en cuatro fragmentos, y obtiene un total de 33 sub-claves K_1, K_2, \dots, K_{33} a partir de la clave original. El proceso consta de tres fases:

- Una permutación inicial IP.
 - 32 rondas consistentes en una operación de mezcla de claves, una consulta a Cajas-S y una transformación lineal (en la última ronda, la transformación lineal es sustituida por una operación de mezcla de claves adicional).
 - Una permutación final FP.
- **Mezcla de claves**

Para la ronda i -ésima, la sub-clave correspondiente K_i pasa por la operación XOR junto con el dato intermedio B_i . Nótese que en la ronda inicial, el dato intermedio $B_0 = P$.

- **Permutación inicial**

Cada bloque pasa por una permutación inicial, de acuerdo a la siguiente tabla:

0	32	64	96	1	33	65	97	2	34	66	98	3	35	67	99
4	36	68	100	5	37	69	101	6	38	70	102	7	39	71	103
8	40	72	104	9	41	73	105	10	42	74	106	11	43	75	107
12	44	76	108	13	45	77	109	14	46	78	110	15	47	79	111
16	48	80	112	17	49	81	113	18	50	82	114	19	51	83	115
20	52	84	116	21	53	85	117	22	54	86	118	23	55	87	119
24	56	88	120	25	57	89	121	26	58	90	122	27	59	91	123
28	60	92	124	29	61	93	125	30	62	94	126	31	63	95	127

Figura 1.56 Tabla para la IP.

De esta forma, la cadena resultante tendrá en la posición 0 el valor correspondiente a la posición 0 de la entrada, en la posición 1 el valor correspondiente a la posición 32 de la entrada y así sucesivamente.

- **Permutación final**

De forma análoga a la Permutación Inicial se emplea la siguiente tabla:

0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	124
1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61
65	69	73	77	81	85	89	93	97	101	105	109	113	117	121	125
2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62
66	70	74	78	82	86	90	94	98	102	106	110	114	118	122	126
3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63
67	71	75	79	83	87	91	95	99	103	107	111	115	119	123	127

Figura 1.57 Tabla para la FP.

- **Transformación lineal**

Los 32 bits de cada palabra se mezclan de forma lineal según el siguiente conjunto de operaciones:

$$\begin{aligned}
 X_0, X_1, X_2, X_3 &:= \mathcal{S}_i(B_i \oplus K_i) \\
 X_0 &:= X_0 \lll 13 \\
 X_2 &:= X_2 \lll 3 \\
 X_1 &:= X_1 \oplus X_0 \oplus X_2 \\
 X_3 &:= X_3 \oplus X_2 \oplus (X_0 \lll 3) \\
 X_1 &:= X_1 \lll 1 \\
 X_3 &:= X_3 \lll 7 \\
 X_0 &:= X_0 \oplus X_1 \oplus X_3 \\
 X_2 &:= X_2 \oplus X_3 \oplus (X_1 \lll 7) \\
 X_0 &:= X_0 \lll 5 \\
 X_2 &:= X_2 \lll 22 \\
 B_{i+1} &:= X_0, X_1, X_2, X_3
 \end{aligned}$$

Figura 1.58 Transformación lineal.

Aquí \lll denota rotación y \ll denota desplazamiento, \mathcal{S}_i denota consulta en la Caja-Si-ésima y \oplus denota la operación XOR.

Esta aplicación lineal, por las propiedades del espacio algebraico en el que se trabaja, tiene una inversa asegurada, lo que permite un descifrado análogo.

- **Obtención de subclaves**

El primer paso para obtener las 33 claves de 128 bits es ampliar la clave inicial hasta obtener 256 bits (en caso de ser necesario). Para ello se añade un 1 al final del mensaje y se rellena con ceros. A continuación escribimos la clave K como ocho palabras de 32 bits cada una, $w_8 \dots w_1$ y las expandimos a una clave intermedia (*prekey*) $w_0 \dots w_{131}$ como sigue:

$$w_i := (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \phi \oplus i) \lll 11$$

Aquí ϕ es la parte fraccional del número áureo $(\sqrt{5} + 1)/2$ o **0x9e3779b9** en hexadecimal. El polinomio subyacente, relacionado con el ya tratado grupo finito $GF(2^8)$, asegura una distribución adecuada de los bits de la clave a través de las rondas.

A continuación, las *round keys* se calculan a partir de las *prekeys* usando las Cajas-S, de la siguiente forma:

$$\begin{aligned}
 \{k_0, k_1, k_2, k_3\} &:= S_3(w_0, w_1, w_2, w_3) \\
 \{k_4, k_5, k_6, k_7\} &:= S_2(w_4, w_5, w_6, w_7) \\
 \{k_8, k_9, k_{10}, k_{11}\} &:= S_1(w_8, w_9, w_{10}, w_{11}) \\
 \{k_{12}, k_{13}, k_{14}, k_{15}\} &:= S_0(w_{12}, w_{13}, w_{14}, w_{15}) \\
 \{k_{16}, k_{17}, k_{18}, k_{19}\} &:= S_7(w_{16}, w_{17}, w_{18}, w_{19}) \\
 &\dots \\
 \{k_{124}, k_{125}, k_{126}, k_{127}\} &:= S_4(w_{124}, w_{125}, w_{126}, w_{127}) \\
 \{k_{128}, k_{129}, k_{130}, k_{131}\} &:= S_3(w_{128}, w_{129}, w_{130}, w_{131})
 \end{aligned}$$

Figura 1.59 Cálculo de round keys usando Cajas-S.

Por último, renumeramos los valores k_i de 32 bits a sub-claves K_i de 128 bits como:

$$K_i = \{k_{4i}, k_{4i+1}, k_{4i+2}, k_{4i+3}\}$$

Y aplicamos de nuevo la permutación IP para colocar los bits en la columna correspondiente.

- **Descifrado**

El proceso de descifrado es completamente análogo, salvo por la necesidad de usar la inversa de la transformación lineal y de las Cajas-S, así como invertir el orden de empleo de las sub-claves.

- **Conclusión**

Serpent es, actualmente, uno de los algoritmos más seguros que existen. De hecho, como se puede observar en la tabla comparativa del anexo 3, hay estudios que lo sitúan por encima de AES-Rijndael en términos de márgenes de seguridad. Sin embargo, por criterios de implementación y eficiencia se optó por Rijndael como algoritmo AES en vez de Serpent.

- **Enlaces de interés**

[Serpent: A Proposal for de Advanced Encryption Standard](#)

En el año 2004 se crea la **ECRYPT** (*European Network of Excellence in Cryptology*): una iniciativa lanzada por la Unión Europea para promocionar la colaboración entre investigadores europeos en el área de la seguridad de la información. Uno de los proyectos más destacables que fomentaron fue **eSTREAM**, cuyo objetivo era el de “*identificar nuevos cifradores de flujo aptos para una adopción generalizada*”. Ya entre los años 2000 y 2003 se había llevado a cabo un proceso similar, llamado **NESSIE** (*New European Schemes for Signatures, Integrity and Encryption*), comparable al proyecto del NIST para el AES del que tanto hemos hablado, o de su análogo japonés, el proyecto **CRYPTREC**. Sin embargo, los algoritmos recibidos en esta fase no se amoldaban a las necesidades planteadas en el momento, por lo que *ECRYPT* nació como una continuación de *NESSIE*.

Uno de los algoritmos más destacables enviados como candidatos fue el conocido como **Salsa20**, cuyo parente próximo **ChaCha** es considerado hoy en día como uno de los mejores algoritmos de cifrado en flujo existentes, llegando a ser empleado en gran cantidad de canales de comunicación, como por ejemplo, en el protocolo TLS (adoptado por Google).

1.6.12 Salsa20

Nos encontramos ante un algoritmo de cifrado en flujo ideado por Daniel J. Bernstein en el año 2005 con el objetivo de participar en la convocatoria del *ECRYPT*. Admite claves de 128 o 256 bits, con bloques de 64 bits, y ofrece un rendimiento bastante bueno, además de contar con una gran cantidad de implementaciones de dominio público gracias a que no existe ninguna patente del algoritmo.

El autor defiende en el *paper* original el uso de operaciones simples y cortas, frente a operaciones más complicadas, ya que las primeras son *consistentemente* rápidas: su velocidad no depende de las circunstancias. Por ese mismo argumento rechaza el empleo de multiplicaciones, añadiendo la posibilidad de que esta operación aumente el riesgo de *leaks* de tiempo (recordamos que Bernstein es también el autor de una serie de estudios de referencia en lo que a *Cache Timing Attacks* se refiere, tratándose de los pocos ataques conocidos que afectan a AES-Rijndael). De la misma forma, descarta el empleo de Cajas-S por aumentar también el riesgo frente a *Cache Timing Attacks*.

- **Detalles de implementación**

Aclararemos una serie de conceptos tratados en la implementación de Salsa20:

Una **palabra** es un elemento del conjunto $\{0, 1, \dots, 2^{32} - 1\}$, aunque generalmente se hará referencia a ella usando su notación hexadecimal, indicada por el prefijo **0x**.

La **suma** de dos palabras u , v es $u + v \bmod 2^{32}$. Se denotará por $u + v$.

La operación XOR entre dos palabras se denotará por el símbolo \oplus .

Dado un entero c , la **rotación c bits a la izquierda** de una palabra u se denotará por $u \lll c$.

- **Función quarterround**

Recibe una secuencia y de cuatro palabras $y = (y_0, y_1, y_2, y_3)$. De este modo, $\text{quarterround}(y) = (z_0, z_1, z_2, z_3)$ donde:

$$\begin{aligned} z_1 &= y_1 \oplus ((y_0 + y_3) \lll 7) \\ 7) \quad z_2 &= y_2 \oplus ((z_1 + y_0) \lll 9) \\ <<< 9) \quad z_3 &= y_3 \oplus ((z_2 + z_1) \lll 13) \\ <<< 13) \quad z_0 &= y_0 \oplus ((z_3 + z_2) \lll 18) \end{aligned}$$

Es fácil observar que cada operación es invertible, por lo que $\text{quarterround}()$ es una función que tiene inversa.

- **Función rowround**

Recibe una secuencia de 16 palabras y y devuelve una salida similar. Si $y = (y_0, \dots, y_{15})$, entonces $\text{rowround}(y) = (z_0, z_1, \dots, z_{15})$ donde:

$$\begin{aligned} (z_0, z_1, z_2, z_3) &= \text{quarterround}(y_0, y_1, y_2, y_3), \\ (z_5, z_6, z_7, z_4) &= \text{quarterround}(y_5, y_6, y_7, y_5), \\ (z_{10}, z_{11}, z_8, z_9) &= \text{quarterround}(y_{10}, y_{11}, y_8, y_9), \end{aligned}$$

$$(z_{15}, z_{12}, z_{13}, z_{14}) = \text{quarterround}(y_{15}, y_{12}, y_{13}, y_{14}).$$

- **Función columnround**

Recibe una secuencia de 16 palabras y devuelve una salida similar. Si $x = (x_0, \dots, x_{15})$ entonces $\text{columnround}(x) = (y_0, y_1, \dots, y_{15})$ donde:

$$(y_0, y_4, y_8, y_{12}) = \text{quarterround}(x_0, x_4, x_8, x_{12}),$$

$$(y_5, y_9, y_{13}, y_1) = \text{quarterround}(x_5, x_9, x_{13}, x_1),$$

$$(y_{10}, y_{14}, y_2, y_6) = \text{quarterround}(x_{10}, x_{14}, x_2, x_6),$$

$$(y_{15}, y_3, y_7, y_{11}) = \text{quarterround}(x_{15}, x_3, x_7, x_{11}),$$

- **Función doubleround**

Recibe secuencias de 16 palabras, devolviendo una salida similar. Se define como sigue:

$$\text{rowround}(\text{columnround}(x))$$

- **Función littleendian**

Recibe secuencias de 4 bytes y devuelve una palabra. Sea $b = (b_0; b_1; b_2; b_3)$ entonces $\text{littleendian}(b) = b_0 + 2^8 b_1 + 2^{16} b_2 + 2^{24} b_3$.

- **Función hash**

Recibe secuencias de 64 bytes y devuelve secuencias similares.

$$\text{Salsa20}(x) = x + \text{doubleround}^{10}(x)$$

Cada secuencia de 4 bytes será vista como una palabra en *Little-Endian*, por lo que habrá un total de 16 palabras en las secuencias.

Sea ahora $(z_0, z_1, \dots, z_{15}) = \text{doubleround}^{10}(x_0, x_1, \dots, x_{15})$, entonces $\text{Salsa20}(x)$ es la concatenación de:

$$\text{Littleendian}^{-1}(z_0 + x_0),$$

...

$$\text{Littleendian}^{-1}(z_{15} + x_{15}).$$

- **Función de expansión**

Sea k una secuencia de 16 o 32 bytes, y n una secuencia de 16 bytes, entonces $\text{Salsa20}_k(n)$ es una secuencia de 64 bytes.

Sean ahora:

$$\sigma_0 = (101,120,112,97),$$

$$\sigma_1 = (110,100,32,51),$$

$$\sigma_2 = (50,45,98,121),$$

$$\sigma_3 = (116,101,32,107).$$

Si k_0, k_1, n son secuencias de 16 bytes, entonces:

$$\text{Salsa20}_{k_0, k_1}(n) = \text{Salsa20}(\sigma_0, k_0, \sigma_1, n, \sigma_2, k_1, \sigma_3).$$

Si k, n son secuencias de 16 bytes:

$$\text{Salsa20}_k(n) = \text{Salsa20}(\sigma_0, k, \sigma_1, n, \sigma_2, k, \sigma_3).$$

- **Cifrado**

Sea ahora k una secuencia de 16 o 32 bytes (128 o 256 bits), sea v una secuencia de 8 bytes, m una secuencia de l bytes entre 0 y 2^{70} . En este caso diremos que k es la clave, v es el nonce y m el mensaje. El cifrado devuelve una secuencia de l bytes, denotada por $\text{Salsa20}_k(v) \oplus m$.

$\text{Salsa20}_k(v)$ es la secuencia de 2^{70} bytes:

$$\text{Salsa20}_k(v;0), \text{Salsa20}_k(v;1), \text{Salsa20}_k(v;2), \dots, \text{Salsa20}_k(v;2^{64} - 1).$$

Aquí, i es la única secuencia (i_0, \dots, i_7) tal que $i = i_0 + 2^8i_1 + 2^{16}i_2 + \dots + 2^{56}i_7$.

En general,

$$\text{Salsa20}_k(v) \oplus (m[0], \dots, m[l-1]) = (c[0], \dots, c[l-1])$$

Donde

$$c[i] = m[i] \oplus \text{Salsa20}_k(v, \text{floor}(i/64)) [i \bmod 64].$$

- **Algoritmo**

Inicialmente se genera un Estado Inicial en forma de matriz 4x4 de la siguiente forma (asumiendo claves de 256 bits):

Initial state of Salsa20			
Cons	Key	Key	Key
Key	Cons	Nonce	Nonce
Pos	Pos	Cons	Key
Key	Key	Key	Cons

Figura 1.60 Estado Inicial de Salsa20.

En el dibujo apreciamos el uso de 16 palabras: en azul las 8 palabras que conforman la clave, en naranja las cuatro palabras que conforman la frase “expand 32-byte k” (“expa”, “nnd 3”, “2-by”, “te k”), en verde las dos palabras que conforman el *nonce* y en rojo dos palabras para indicar la posición en el flujo.

El algoritmo es de naturaleza iterativa y cuenta con un total de 20 rondas. Durante el proceso, en las rondas pares se procesarán columnas enteras, mientras que en las impares se procesarán filas enteras. De esta forma, el pseudocódigo queda como sigue (suponiendo que *in*[16] es la entrada de la función):

```

for (i = 0; i < 16; ++i)
    x[i] = in[i];

for (i = 0; i < ROUNDS; i += 2) {
    //Ronda impar
    quarterround(x[0], x[4], x[8], x[12]); //columna 1
    quarterround(x[5], x[9], x[13], x[1]); //columna 2
    quarterround(x[10], x[14], x[2], x[6]); //columna 3
    quarterround(x[15], x[3], x[7], x[11]); //columna 4

    //Ronda par
    quarterround(x[0], x[1], x[2], x[3]); //fila 1
    quarterround(x[5], x[6], x[7], x[4]); //fila 2
    quarterround(x[10], x[11], x[8], x[9]); //fila 3
    quarterround(x[15], x[12], x[13], x[14]); //fila 4
}

for (i = 0; i < 16; ++i)
    out[i] = x[i] + in[i];

```

- **Conclusión**

Salsa20 se convirtió en uno de los algoritmos de cifrado en flujo más avanzados de su época. Su autor tomó todas las decisiones de diseño con el objetivo de conseguir un algoritmo robusto frente al *Cache Timing Attack* que él mismo había descubierto, por lo que su nivel de seguridad es bastante alto. Además, desde la perspectiva de los algoritmos de *hashing*, Salsa20 también ofrecía una serie de prestaciones interesantes.

Sin embargo, pese al éxito del sistema, Bernstein desarrolló en 2008 una variante del mismo denominada *ChaCha* que buscaba incrementar la difusión de cada ronda mientras conseguía una eficiencia similar o incluso mayor.

Aunque en general se desconocen ataques factibles contra Salsa20 con 8 y 12 rondas, el autor defiende su uso con 20 rondas por “estar más allá de lo que se puede romper” y, además, estar dentro del tiempo adecuado para un algoritmo de cifrado en flujo.

1.6.13 ChaCha20

Esta variante del algoritmo Salsa20 publicada en 2008 por el mismo autor se concibió con el objetivo de aumentar la difusión de cada ronda del algoritmo, tratando de mejorar la eficiencia del mismo. Esencialmente es muy parecido a su predecesor, empleando las mismas funciones ligeramente modificadas.

- **Estado inicial**

Al igual que Salsa20, el Estado Inicial está compuesto por una matriz 4x4 (en el caso de claves de 256bits). La única diferencia en esta versión es la disposición de sus celdas.

Initial state of ChaCha			
Cons	Cons	Cons	Cons
Key	Key	Key	Key
Key	Key	Key	Key
Pos	Pos	Nonce	Nonce

Figura 1.61 Estado Inicial de ChaCha.

Como se puede apreciar, las componentes de la matriz son las mismas, pero en distinta posición.

- **Función quarterround**

De nuevo, el corazón del algoritmo vuelve a ser la función *quarterround()*. Sin embargo, esta vez se verá ligeramente modificada: el autor decidió incluir dos actualizaciones de cada valor y cambiar las distancias de rotación.

```
a += b; d ^= a; d <<<= 16;  
c += d; b ^= c; b <<<= 12;  
a += b; d ^= a; d <<<= 8;  
c += d; b ^= c; b <<<= 7;
```

Figura 1.62 Función *quarterround()* de ChaCha.

Esta modificación es la que más contribuye a la difusión de cambios en el algoritmo y a su eficiencia, ya que está pensada de forma que, en ausencia de acarreos en las operaciones, la velocidad promedio aumenta. El cambio en las distancias de rotación está justificado como que, en ciertas plataformas, parece aumentar la eficiencia (aunque no es apreciable en otras muchas situaciones).

Por lo demás, el algoritmo es idéntico al de Salsa20.

- **Conclusión**

Pese a que los dos algoritmos son prácticamente iguales, a día de hoy se usa siempre la variante ChaCha, ya que ha demostrado una mayor eficiencia con el mismo grado de seguridad.

Por otro lado, se desconocen ataques viables frente a ambas versiones, por lo que son muy utilizadas a día de hoy en numerosos canales de comunicación. Sin ir más lejos, la propia Google y OpenSSL han incluido a ChaCha como método de cifrado junto a Poly-1305 (como MAC) dentro de sus estándares de comunicación por protocolo TLS. Además, el protocolo VPN WireGuard emplea exclusivamente ChaCha20 en sus cifrados.

- **Enlaces de interés**

[*ChaCha, a variant of Salsa20 \(Daniel J. Bernstein\)*](#)

[*Detalle del protocolo WireGuard*](#)

1.6.14 Camellia

En el año 2000, dos grandes empresas japonesas, Mitsubishi y NTT, publicaron el algoritmo de cifrado *Camellia*. Se trata de un algoritmo simétrico de bloques, basado como tantos otros en las ya conocidas *Redes de Feistel*. Emplea claves de 128, 192 o 256 bits, con bloques de 128 bits (mismas especificaciones que AES).

En sus orígenes fue concebido para aportar el mayor grado de seguridad y robustez posible y una eficiencia considerable en múltiples plataformas.

- **Estructura**

Camellia emplea una *Red de Feistel* de 18 rondas para claves de 128 bits, y de 24 rondas para claves de 192 y 256 bits. También aplica *blanqueamientos* o *whitenings* similares a los que vimos en *Twofish* para las entradas y las salidas, y dos *funciones lógicas* denominadas Función-FL y Función- FL^{-1} que se aplicarán cada 6 rondas.

El proceso de expansión de la clave genera sub-claves de 64 bits a partir de la clave K : kw_t ($t = 1, 2, 3, 4$) para los blanqueamientos, k_u ($u = 1, 2, \dots, r$) para las funciones de cada ronda y kl_v ($v = 1, 2, \dots, r/3 - 2$) para las funciones lógicas. Nótese que r es el número de rondas.

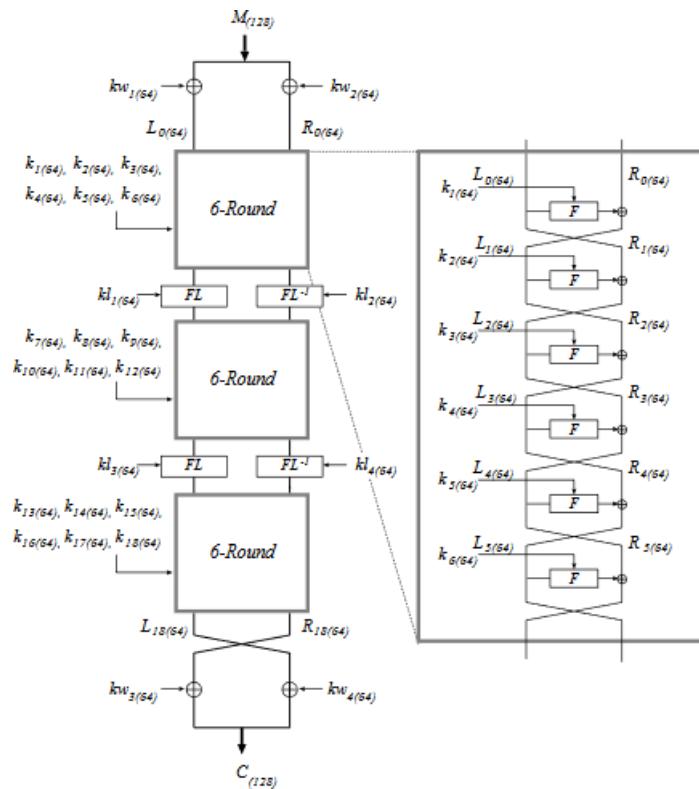


Figura 1.63 Estructura de Camellia para claves de 128 bits.

- **Notación**

X_L, X_R : mitades izquierda y derecha (respectivamente) del dato X .

\oplus, \cap, \cup : operaciones XOR, AND y OR *bitwise* (a nivel de bit).

\parallel : concatenación de dos operandos.

\gg_n, \ll_n : rotación a la derecha/izquierda por n bits.

$0x$: representación hexadecimal.

- **Obtención de subclaves**

Se definen dos variables de 128 bits como sigue: para claves de 128 bits, K_L es igual que la clave y K_R vale 0; para claves de 192 bits, K_R es la concatenación de los 64 bits restantes y su complemento; para claves de 256 bits, los 128 bits de la derecha compondrán K_R .

Las variables de 128 bits K_A y K_B se obtienen a partir de K_L y K_R tal y como muestra la figura 1.64. Nótese que K_B sólo se usará en caso de que la clave usada sea de 192 o 256 bits.

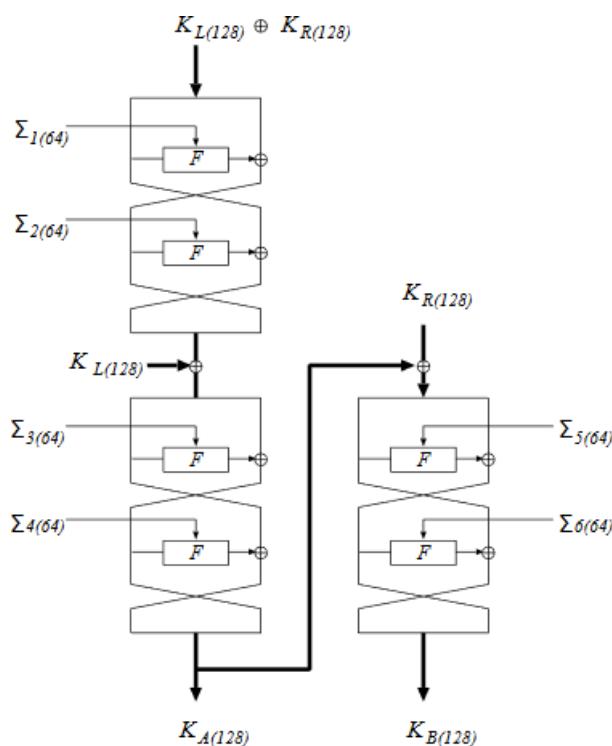


Figura 1.64 Obtención de sub-claves.

Las constantes $\Sigma_i (i=1,2,\dots,6)$ de 64 bits serán usadas como claves en Feistel y sus valores se recogen en la figura 1.65.

Σ_1	0xA09E667F3BCC908B	Σ_4	0x54FF53A5F1D36F1C
Σ_2	0xB67AE8584CAA73B2	Σ_5	0x10E527FADE682D1D
Σ_3	0xC6EF372FE94F82BE	Σ_6	0xB05688C2B3E6C1FD

Figura 1.65 Constantes de 64 bits para Camellia.

Finalmente, las sub-claves que buscamos kw_t, k_u , y kl_v serán generadas a partir de K_A, K_B, K_L y K_R usando rotaciones y tomando fragmentos como se muestra en la siguiente tabla:

128-bit keys	subkey	value	192/256-bit keys	subkey	value
Prewhitening	kw_1	$(K_L \lll 0)_L$	Prewhitening	kw_1	$(K_L \lll 0)_L$
	kw_2	$(K_L \lll 0)_R$		kw_2	$(K_L \lll 0)_R$
F (Round1)	k_1	$(K_A \lll 0)_L$	F (Round1)	k_1	$(K_B \lll 0)_L$
F (Round2)	k_2	$(K_A \lll 0)_R$	F (Round2)	k_2	$(K_B \lll 0)_R$
F (Round3)	k_3	$(K_L \lll 15)_L$	F (Round3)	k_3	$(K_R \lll 15)_L$
F (Round4)	k_4	$(K_L \lll 15)_R$	F (Round4)	k_4	$(K_R \lll 15)_R$
F (Round5)	k_5	$(K_A \lll 15)_L$	F (Round5)	k_5	$(K_A \lll 15)_L$
F (Round6)	k_6	$(K_A \lll 15)_R$	F (Round6)	k_6	$(K_A \lll 15)_R$
FL	kl_1	$(K_A \lll 30)_L$	FL	kl_1	$(K_R \lll 30)_L$
FL^{-1}	kl_2	$(K_A \lll 30)_R$	FL^{-1}	kl_2	$(K_R \lll 30)_R$
F (Round7)	k_7	$(K_L \lll 45)_L$	F (Round7)	k_7	$(K_B \lll 30)_L$
F (Round8)	k_8	$(K_L \lll 45)_R$	F (Round8)	k_8	$(K_B \lll 30)_R$
F (Round9)	k_9	$(K_A \lll 45)_L$	F (Round9)	k_9	$(K_L \lll 45)_L$
F (Round10)	k_{10}	$(K_L \lll 60)_R$	F (Round10)	k_{10}	$(K_L \lll 45)_R$
F (Round11)	k_{11}	$(K_A \lll 60)_L$	F (Round11)	k_{11}	$(K_A \lll 45)_L$
F (Round12)	k_{12}	$(K_A \lll 60)_R$	F (Round12)	k_{12}	$(K_A \lll 45)_R$
FL	kl_3	$(K_L \lll 77)_L$	FL	kl_3	$(K_L \lll 60)_L$
FL^{-1}	kl_4	$(K_L \lll 77)_R$	FL^{-1}	kl_4	$(K_L \lll 60)_R$
F (Round13)	k_{13}	$(K_L \lll 94)_L$	F (Round13)	k_{13}	$(K_R \lll 60)_L$
F (Round14)	k_{14}	$(K_L \lll 94)_R$	F (Round14)	k_{14}	$(K_R \lll 60)_R$
F (Round15)	k_{15}	$(K_A \lll 94)_L$	F (Round15)	k_{15}	$(K_B \lll 60)_L$
F (Round16)	k_{16}	$(K_A \lll 94)_R$	F (Round16)	k_{16}	$(K_B \lll 60)_R$
F (Round17)	k_{17}	$(K_L \lll 111)_L$	F (Round17)	k_{17}	$(K_L \lll 77)_L$
F (Round18)	k_{18}	$(K_L \lll 111)_R$	F (Round18)	k_{18}	$(K_L \lll 77)_R$
Postwhitening	kw_3	$(K_A \lll 111)_L$	FL	kl_5	$(K_A \lll 77)_L$
	kw_4	$(K_A \lll 111)_R$	FL^{-1}	kl_6	$(K_A \lll 77)_R$
			F (Round19)	k_{19}	$(K_R \lll 94)_L$
			F (Round20)	k_{20}	$(K_R \lll 94)_R$
			F (Round21)	k_{21}	$(K_A \lll 94)_L$
			F (Round22)	k_{22}	$(K_A \lll 94)_R$
			F (Round23)	k_{23}	$(K_L \lll 111)_L$
			F (Round24)	k_{24}	$(K_L \lll 111)_R$
			Postwhitening	kw_3	$(K_B \lll 111)_L$
				kw_4	$(K_B \lll 111)_R$

Figura 1.66 Obtención de sub-claves en Camellia.

En general, los criterios de diseño de esta fase aseguran que las operaciones empleadas son similares a las del cifrado, facilitando su implementación; la generación de sub-claves usa el mismo circuito que el resto del algoritmo, y además consume un tiempo inferior al de cifrado; es resistente a ataques debidos a la relación entre claves.

- **Función F**

La Función F usa una estructura SPN (*Substitution-Permutation Network*), lo que significa que únicamente emplea funciones de sustitución (Función S, no lineal) y funciones de permutación (Función P, lineal). La Función F queda explicada por el siguiente esquema.

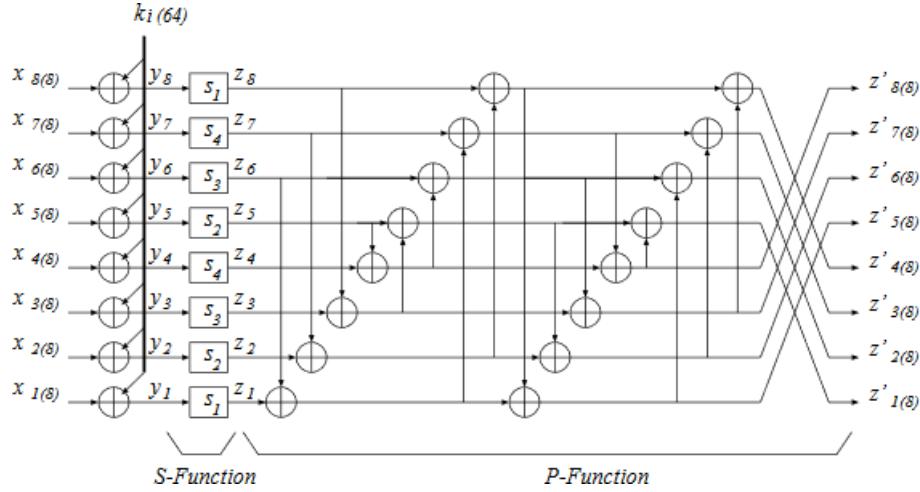


Figura 1.67 Función F para Camellia.

- **Función S, Cajas-S**

La Función S consiste en consultar ocho Cajas-S, para lo que usará cuatro Cajas-S distintas s_1 , s_2 , s_3 y s_4 . En este algoritmo se entienden las Cajas-S como endomorfismos de $GF(2^8)$, tal y como se recoge a continuación:

$$\begin{aligned}
 s_1 &: GF(2)^8 \rightarrow GF(2)^8, x \mapsto h(g(f(0xc5 \oplus x))) \oplus 0x6e \\
 s_2 &: GF(2)^8 \rightarrow GF(2)^8, x \mapsto s_1(x) \lll_1 \\
 s_3 &: GF(2)^8 \rightarrow GF(2)^8, x \mapsto s_1(x) \ggg_1 \\
 s_4 &: GF(2)^8 \rightarrow GF(2)^8, x \mapsto s_1(x \lll_1)
 \end{aligned}$$

Figura 1.68 Endomorfismos de las Cajas-S.

Aquí las funciones f y h son aplicaciones afines y g es la inversa en $GF(2^8)$.

$$\mathbf{f} : \text{GF}(2)^8 \rightarrow \text{GF}(2)^8, (a_1, a_2, \dots, a_8) \mapsto (b_1, b_2, \dots, b_8),$$

where

$$\begin{aligned} b_1 &= a_6 \oplus a_2, & b_2 &= a_7 \oplus a_1, & b_3 &= a_8 \oplus a_5 \oplus a_3, & b_4 &= a_8 \oplus a_3, \\ b_5 &= a_7 \oplus a_4, & b_6 &= a_5 \oplus a_2, & b_7 &= a_8 \oplus a_1, & b_8 &= a_6 \oplus a_4. \end{aligned}$$

$$\mathbf{h} : \text{GF}(2)^8 \rightarrow \text{GF}(2)^8, (a_1, a_2, \dots, a_8) \mapsto (b_1, b_2, \dots, b_8),$$

where

$$\begin{aligned} b_1 &= a_5 \oplus a_6 \oplus a_2, & b_2 &= a_6 \oplus a_2, & b_3 &= a_7 \oplus a_4, & b_4 &= a_8 \oplus a_2, \\ b_5 &= a_7 \oplus a_3, & b_6 &= a_8 \oplus a_1, & b_7 &= a_5 \oplus a_1, & b_8 &= a_6 \oplus a_3. \end{aligned}$$

$$\mathbf{g} : \text{GF}(2)^8 \rightarrow \text{GF}(2)^8, (a_1, a_2, \dots, a_8) \mapsto (b_1, b_2, \dots, b_8),$$

where

$$\begin{aligned} &(b_8 + b_7\alpha + b_6\alpha^2 + b_5\alpha^3) + (b_4 + b_3\alpha + b_2\alpha^2 + b_1\alpha^3)\beta \\ &= ((a_8 + a_7\alpha + a_6\alpha^2 + a_5\alpha^3) + (a_4 + a_3\alpha + a_2\alpha^2 + a_1\alpha^3)\beta)^{-1}. \end{aligned}$$

Figura 1.69 Funciones f , g y h .

En este caso la inversión en $\text{GF}(2^8)$ se lleva a cabo asumiendo que el 0 es su propio inverso, que β es un elemento en el grupo finito que satisface $\beta^8 + \beta^6 + \beta^5 + \beta^3 + 1 = 0$ y que $\alpha = \beta^{238} = \beta^6 + \beta^5 + \beta^3 + \beta^2$ es un elemento de $\text{GF}(2^8)$ que satisface $\alpha^4 + \alpha + 1 = 0$.

• **Función P**

Se define en la figura siguiente:

$$P : (\text{GF}(2)^8)^8 \rightarrow (\text{GF}(2)^8)^8, (z_1, z_2, \dots, z_8) \mapsto (z'_1, z'_2, \dots, z'_8),$$

where

$$\begin{aligned} z'_1 &= z_1 \oplus z_3 \oplus z_4 \oplus z_6 \oplus z_7 \oplus z_8, & z'_2 &= z_1 \oplus z_2 \oplus z_4 \oplus z_5 \oplus z_7 \oplus z_8, \\ z'_3 &= z_1 \oplus z_2 \oplus z_3 \oplus z_5 \oplus z_6 \oplus z_8, & z'_4 &= z_2 \oplus z_3 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7, \\ z'_5 &= z_1 \oplus z_2 \oplus z_6 \oplus z_7 \oplus z_8, & z'_6 &= z_2 \oplus z_3 \oplus z_5 \oplus z_7 \oplus z_8, \\ z'_7 &= z_3 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_8, & z'_8 &= z_1 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7. \end{aligned}$$

Figura 1.70 Función P de Camellia.

- **Función-FL y Función-FL⁻¹**

La Función-FL se define como:

$$FL : GF(2)^{64} \times GF(2)^{64} \rightarrow GF(2)^{64}, (X_L||X_R, kl_L||kl_R) \mapsto Y_L||Y_R,$$

where

$$Y_R = ((X_L \cap kl_L) \lll_1) \oplus X_R, \quad Y_L = (Y_R \cup kl_R) \oplus X_L.$$

Figura 1.71 Definición de la Función FL.

Como se puede apreciar, usa las sub-claves obtenidas previamente para dar lugar a una nueva cadena como salida.

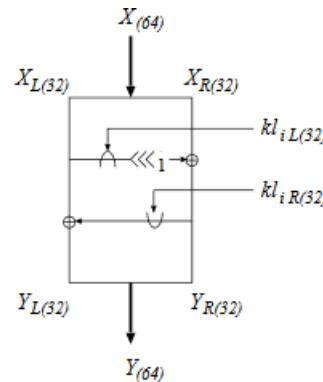


Figura 1.72 Esquema de la función FL.

Por su parte, la Función-FL⁻¹ queda definida por el siguiente esquema:

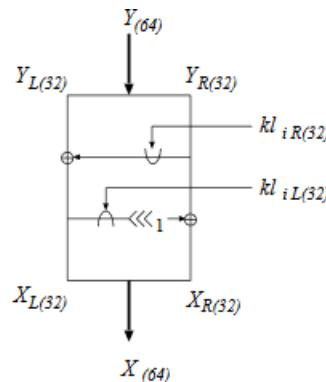


Figura 1.73 Esquema de la Función-FL⁻¹.

Y cumple la propiedad de la inversa:

$$FL^{-1}(FL(x, k), k) = x$$

Figura 1.74 Propiedad inversa de la Función-FL⁻¹.

- **Descifrado**

Como en todos los algoritmos basados en Feistel, el proceso es análogo al de cifrado, introduciendo las sub-claves en orden inverso al de cifrado.

- **Conclusión**

Camellia es un algoritmo bastante robusto, aunque no deja de resultar muy simple. Además, los criterios en los que se basaron sus diseñadores desde el inicio aseguran la eficiencia del algoritmo: usa operaciones a nivel de bit (rotaciones, XORs, ANDs...) que son muy poco costosas, evita operaciones aritméticas complejas, las operaciones se mantienen simples independientemente de la arquitectura utilizada... Así como su robustez: fue diseñado específicamente para resistir todo tipo de ataque conocido hasta la fecha.

En comparación con AES, sus características son muy similares. *Camellia* fue diseñado de acuerdo a las características del NIST, por lo que admiten longitudes de clave y bloque similares. En general, actualmente AES es el algoritmo más utilizado puesto que fue elegido como estándar hace muchos años y la mayoría de procesadores incluyen hardware específico para optimizar este algoritmo. Sin embargo, *Camellia* presenta la ventaja de que puede aprovechar parte de este hardware para trabajar con un rendimiento similar (aunque algo inferior) al de AES.

Pese a todo, *Camellia* es un algoritmo que ha calado bastante en diversas organizaciones. Cuenta con certificaciones como estándar otorgadas por CRYPTREC, NESSIE, IETF... y que además se usa en gran cantidad de herramientas y protocolos: TLS, IPsec, Kerberos, OpenPGP, S/MIME...

- **Enlaces de interés**

[Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms \(Kuzamaro Aoki et al.\)](#)

1.7 Algoritmos simétricos ligeros

El auge de la tecnología móvil ha generado la necesidad de controlar en todo momento la eficiencia de los programas que corren en un dispositivo. Uno de los principales motivos es el del consumo, teniendo en mente dispositivos móviles como *smartphones*, en los que se debe cuidar la batería y los recursos de los que se dispone (generalmente reducidos).

Es por eso que en el portfolio del eCRYPT sobre criptografía simétrica de flujo (eSTREAM portfolio) se incluye un apartado de algoritmos ligeros que cuidan estos detalles. Los más destacados son Trivium, Grain-128a y MICKEY 2.0.

1.7.1 Grain-128a

Este algoritmo de flujo fue el sucesor de Grain-128 (propuesto en sus inicios para el eSTREAM), aportando mejoras en la seguridad. Grain 128a fue propuesto inicialmente en 2011 en el *Symmetric Key Encryption Workshop* organizado por el eCRYPT.

Sus principales ventajas incluyen un diseño bastante simple, un consumo de memoria muy reducido y un *throughput* bastante grande. Además, incluye un método de autenticación.

Su diseño consiste en dos registros (NLFSR y LFSR) que se deben inicializar previamente con valores dependientes de la clave. Estos registros modificarán su contenido en cada iteración del algoritmo. Aparte del mecanismo de modificación de estos registros (usando XORs, ANDs y una función h que aquí trataremos como una *caja negra*), Grain 128a cuenta con un mecanismo de autenticación MAC que, además, contribuye en el cifrado del mensaje. Este mecanismo está formado por un registro *shift* y un acumulador que generarán el *tag* de autenticación. El siguiente esquema recoge el funcionamiento del algoritmo.

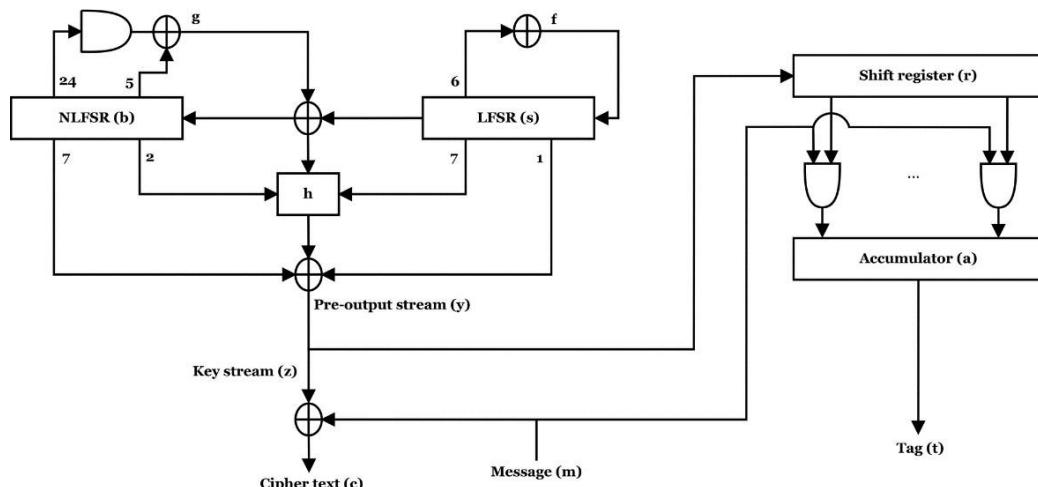


Figura 1.75 Esquema de funcionamiento de Grain-128a.

1.7.2 MICKEY 2.0

El algoritmo ***Mutual Irregular Clocking KEYstrem generator*** (MICKEY) fue desarrollado por Steve Babbage y Mattherw Dodd como participante en el concurso para el portfolio eSTREAM en su versión para algoritmos ligeros. Fue diseñado con el objetivo de implementarse en plataformas hardware con recursos muy limitados de una manera eficiente.

Debido al poco uso de este algoritmo en los sistemas actuales, no vamos a entrar en detalle en su implementación, aunque sí queremos mencionar que trabaja con claves de 80 bits, generando un *keystream* bastante extenso.

Aunque se han reportado numerosos ataques contra MICKEY 2.0, ninguno de ellos es practicable en la realidad.

1.7.3 Trivium

Con Trivium completamos el apartado del portfolio del eCRYPT para algoritmos de flujo ligeros. Con un diseño elegante y un *throughput* muy elevado, junto con un consumo de recursos mínimo, Trivium se situó como el más llamativo de los algoritmos presentados. Además, la posibilidad de paralelizar su ejecución lo convertía en la opción principal para sistemas con recursos limitados.

En concreto, Trivium presentó una velocidad de cifrado de 4 ciclos/byte en algunas plataformas x86, mientras que AES requiere 19 ciclos/byte en las mismas plataformas.

Por otro lado, no se conocen ataques prácticos contra este algoritmo, lo que lo convierte en una de las principales opciones de cifrado en flujo para plataformas con recursos limitados, ya que además no está patentado y su uso es libre.

1.8 Criptografía Asimétrica

1.8.1 Introducción

Todos los algoritmos vistos hasta ahora basan su funcionamiento en una clave *secreta*, necesaria tanto para cifrar como para descifrar. De hecho, hasta la década de 1970 no se conocían alternativas a este tipo de cifrado. Todas las comunicaciones basaban su confidencialidad en mantener la clave en secreto.

Como el lector se puede imaginar, esto suponía ciertas complicaciones, ya que era bastante probable que un atacante insistente fuese capaz de hacerse con esa clave en algún momento, pudiendo acceder al contenido del mensaje. Además, este tipo de cifrados no garantizaba de modo alguno la autenticidad o el no repudio.

La solución a estos problemas vino de la mano de prestigiosos criptógrafos como Whitfield Diffie o Martin Hellman, y pasaba por que cada nodo generaba dos claves: una clave pública (que será distribuida libremente) y una clave privada. Este nuevo paradigma basaba su seguridad en proteger la clave privada, con la ventaja de que no sería necesario compartir la para establecer una comunicación efectiva. Dependiendo del algoritmo empleado, es posible que el nodo B cifre un mensaje usando la clave pública de A, de modo que sólo A pueda descifrarlo, usando para ello su clave privada.

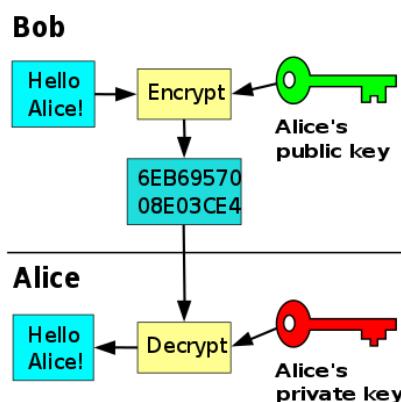


Figura 1.76 Ejemplo de uso de un cifrado asimétrico.

1.8.2 Usos de la criptografía asimétrica

Este nuevo paradigma abrió las puertas a la aparición de una infinidad de aplicaciones para la criptografía.

CIFRADO CON CLAVE PÚBLICA

Como ya hemos mencionado, hay algoritmos que pueden emplear dos claves relacionadas (aunque distintas) para cifrar y descifrar, tal y como se recoge en la figura 1.75. Sin embargo, estos métodos son mucho más lentos que los de clave simétrica, que han sido mucho más estudiados.

Como solución se han desarrollado los denominados *criptosistemas híbridos*, que generalmente emplean los algoritmos de clave pública para garantizar un canal seguro a través del cual se intercambia una clave que se usará en un algoritmo simétrico (más rápido) para cifrar la comunicación.

FIRMA DIGITAL

La criptografía asimétrica permite *firmar* un mensaje para garantizar la autenticidad de la fuente y la integridad del mensaje. Para ello, el emisor A cifra un texto corto con su clave privada, de modo que todo el mundo podrá descifrarlo a través de la clave pública de A, obteniendo la garantía de que el mensaje procede, efectivamente, de A. En caso de querer garantizar la integridad del mensaje, se puede aplicar una función resumen (o hash) al mensaje, y *firmar* cifrando dicho resumen. En este caso, cuando un receptor quiera comprobar la integridad del mensaje, primero descifra el mensaje junto con el resumen *firmado*, posteriormente aplica la función resumen al mensaje recibido. En caso de haber alguna modificación en el mensaje recibido, los resúmenes no coincidirán.

NO REPUDIO

Otro de los usos más frecuentes es el de garantizar el no repudio, mediante el uso de firmas digitales. De esta forma se asegura que una parte no pueda negar la autoría de un documento o comunicación.

OTRAS APLICACIONES

A parte de todo lo ya mencionado, los avances en la criptografía asimétrica han permitido desarrollar tecnologías basadas en este paradigma: divisas digitales, certificados, contratos, *timestamping* confiable...

1.9 Criptografía Híbrida

1.9.1 Introducción

Como ya hemos explicado, los métodos criptográficos híbridos emplean algoritmos asimétricos para crear un canal seguro, a través del cual comparten una clave de cifrado que será usada en un algoritmo simétrico para asegurar la confidencialidad de la comunicación. Esto es así porque los algoritmos de clave pública son computacionalmente mucho más costosos que los simétricos.

A este procedimiento de enviar una clave simétrica usando un método de protección se le denomina *Key Encapsulation*.

1.9.2 Sobre digital

El proceso de creación de un canal seguro para compartir la clave simétrica se denomina *sobre digital*. Consiste en:

1. Generar una clave simétrica única.
2. Cifrar la clave simétrica con la clave privada del emisor para garantizar la autenticidad del contenido.
3. Cifrar el resultado anterior con la clave pública del receptor para garantizar la confidencialidad del contenido, de modo que únicamente el receptor podrá descifrarlo.
4. Se cifra el mensaje con la clave simétrica generada, y se envía junto con el “envoltorio” que contiene la clave simétrica cifrada.

Para poder leer el mensaje, el receptor podrá aplicar primero su clave privada al “envoltorio” y, posteriormente, la clave pública del emisor para obtener la clave simétrica con la que poder descifrar.

1.9.3 Clave de sesión

Se denomina *clave de sesión* a una clave simétrica única que se va a utilizar durante una sesión de comunicación entre dos nodos y que no podrá reutilizarse. El uso de claves de sesión supone una buena práctica, ya que, en caso de que se capture la clave de una comunicación, el atacante sólo podrá acceder a los datos de esa sesión y nada más.

1.9.4 Claves asimétricas e implementación

Es importante aclarar que todos los algoritmos de clave asimétrica que vamos a ver a continuación están planteados de manera teórica. En la práctica, presentan ligeras variaciones de cara a una correcta implantación del algoritmo. Dichas variaciones se recogen en estándares que, aunque los mencionamos en el siguiente apartado, aportan demasiada información como para incluirla en este documento. Por lo tanto, animamos al lector a consultar el estándar asociado con el algoritmo que planea utilizar.

Una de las variaciones más comunes y que sí merece la pena explicar, es que las claves públicas y privadas generadas suelen ir acompañadas de más información (dependiendo del algoritmo o el *framework* empleado). Por ejemplo, las claves generadas para el protocolo SSH generalmente cuentan con una cabecera que indica el tipo de algoritmo empleado y otra información de interés, como longitud de la clave.

1.10 Algoritmos de Clave Pública

1.10.1 Estándar PKCS

Las siglas PKCS hacen referencia a *Public-Key Cryptography Standards*. Se trata de un conjunto de familias de estándares publicados por los Laboratorios RSA que recogen las definiciones y recomendaciones básicas necesarias para implementar algoritmos de clave pública de forma segura. A continuación resumimos las principales características de cada estándar (hay quince en total, pero únicamente mencionaremos los no obsoletos):

- **PKCS#1 v2.2** (RSA Cryptography Standard): define las propiedades matemáticas y el formato de las claves públicas y privadas de RSA, así como los algoritmos y esquemas de relleno para cifrar, descifrar y generar firmas digitales. Está recogido, a su vez, en el [RFC 807](#).
- **PKCS#3 v1.4** (Diffie-Hellman Key Agreement Standard): establece las bases para la implementación del algoritmo de intercambio de claves Diffie-Hellman.
- **PKCS#5 v2.1** (Password-based Encryption Standard): recoge las recomendaciones de implementación para los algoritmos criptográficos basados en contraseñas, incluyendo las funciones de derivación de claves, esquemas de cifrado, autenticación de mensajes... Se encuentra a su vez en el [RFC 8018](#).
- **PKCS#7 v1.5** (Cryptographic Message Syntax Standard): cubre la firma y el cifrado de mensajes bajo una Infraestructura de Clave Pública (PKI), así como la difusión de certificados. Aparece en el [RFC 2315](#).
- **PKCS#8 v1.2** (Private-key Information Syntax Standard): recoge la sintaxis y la información que incluye una clave privada. Aparece en el [RFC 5958](#).
- **PKCS#10 v1.7** (Certification Request Standard): especifica el formato de los mensajes que se envían a una Autoridad de Certificación para solicitar una clave pública (o certificado). Incluido en el [RFC 2986](#).
- **PKCS#11 v2.4** (Cryptographic Token Interface): API que define una interfaz genérica para el uso de *tokens* criptográficos (dispositivos *hardware* o físicos que contienen claves criptográficas).
- **PKCS#12 v1.1** (Personal Information Exchange Syntax Standard): define el formato de los archivos usados para almacenar claves privadas junto con los certificados de clave pública correspondientes. Recogido en el estándar [RFC 7292](#).
- **PKCS#15 v1.1** (Cryptographic Token Information Format Standard): define un estándar para permitir que los usuarios con *tokens* criptográficos puedan identificarse en una aplicación.

1.10.2 Intercambio de clave:Diffie-Hellman

El algoritmo Diffie-Hellman no es un algoritmo cuyo objetivo sea el de cifrar un mensaje, sino que busca establecer un canal seguro para intercambiar una clave en una comunicación. El esquema en el que se basa el algoritmo fue publicado por primera vez en el año 1976, por los criptólogos Whitfield Diffie y Martin Hellman, convirtiéndose en uno de los primeros ejemplos prácticos del campo de la criptografía asimétrica.

Los detalles de su implementación se encuentran en el estándar **PKCS#3 v1.4**.

CONCEPTO

El objetivo final es que Alice y Bob acaben conociendo una clave común, con la asunción de que toda comunicación que hagan será a través de un canal inseguro. Para ello, supongamos el siguiente proceso:

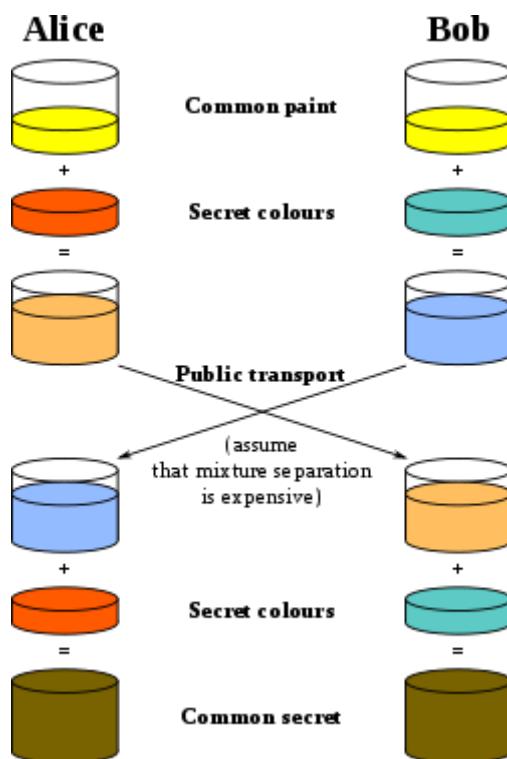


Figura 1.77 Ilustración conceptual del algoritmo Diffie-Hellman.

Supongamos que Alice y Bob se ponen de acuerdo (de forma pública) en un color, digamos, amarillo. Ambas partes cuentan con un color secreto que nadie más conoce, así que mezclan el amarillo con su color secreto, intercambiando el resultado. Llegados a este punto, es importante asumir que es computacionalmente muy complejo separar los colores que han dado lugar a la mezcla, aun sabiendo que uno de ellos es amarillo.

El último paso consiste en que tanto Alice como Bob añaden de nuevo su color secreto obteniendo así el mismo color final.

En esta analogía, el color final sería una posible clave simétrica común con la que establecer una comunicación, que nadie más aparte de ellos podrá obtener (debido, principalmente, a la imposibilidad ya mencionada de separar los colores de las mezclas que se intercambian en público).

CONCEPTOS PREVIOS

Definición: sea $G = (C, *)$ un grupo con una operación $*$, se dice que $a \in C$ es un generador del grupo G si todo elemento del grupo puede escribirse como un producto finito de a y su inverso.

Definición: dado un número natural n , decimos que a es una raíz primitiva módulo n si a es generador de \mathbb{Z}_n^* como grupo (siendo \mathbb{Z}_n^* el conjunto de elementos invertibles módulo n).

IMPLEMENTACIÓN DEL ALGORITMO

Originalmente, *Diffie-Hellman* se implementaba usando el grupo multiplicativo de enteros módulo p , con p primo, siendo g una raíz primitiva módulo p . Estos valores se toman así para asegurar que la clave resultante pueda ser cualquier valor entre 1 y $p - 1$. Un ejemplo del esquema sería el siguiente:

1. Alice y Bob acuerdan utilizar $p = 23$ y $g = 5$ (que es raíz primitiva módulo 23).
2. Alice elige un entero secreto, $a = 4$ y le envía a Bob $A = g^a \bmod p$.
 - $A = 5^4 \bmod 23 = 4$
3. Bob elige un entero secreto $b = 3$ y envía a Alice $B = g^b \bmod p$.
 - $B = 5^3 \bmod 23 = 10$
4. Alice opera $S = B^a \bmod p$
 - $S = 10^4 \bmod 23 = 18$
5. Bob opera $S = A^b \bmod p$
 - $S = 4^3 \bmod 23 = 18$
6. Alice y Bob comparten ahora una clave secreta.

Obviamente, los números empleados en casos reales son mucho mayores, pero los cálculos siguen siendo los mismos. De hecho, si tomamos primos de al menos 600 dígitos, ni siquiera el ordenador más potente del mundo sería capaz de encontrar los datos necesarios para romper el esquema. Al problema de encontrar a y b dados únicamente g y S se le denomina **problema del logaritmo discreto** y, en el caso de grupos cíclicos como \mathbb{Z}_p^* con p muy grande son prácticamente irresolubles.

CONCLUSIONES

Diffie-Hellman supuso un gran avance en el mundo de la criptografía. Permitía establecer, de una vez por todas, un método de intercambio de claves de manera segura. Otra de las ventajas que aportaba es la capacidad de albergar a más de dos personas en un mismo acuerdo de clave, permitiendo que un grupo de participantes comparten la misma clave de manera eficiente.

En concreto, si se eligen bien los secretos y el grupo sobre el que trabajar, el protocolo es perfectamente hermético. Sin embargo, si se llegara a descubrir un algoritmo para resolver eficientemente el problema del logaritmo discreto, este criptosistema, como tantos otros, se volvería completamente inseguro.

Sin embargo, para acelerar el proceso, muchas implementaciones de Diffie-Hellman trabajan sobre grupos de orden 1024 bits o menos. En este caso, existe un ataque práctico basado en el Algoritmo de Criba General del Cuerpo de Números. Este algoritmo consiste en cuatro pasos computacionales. Si un atacante *precomputa* los tres primeros pasos para los grupos más comunes y órdenes pequeños (no lleva demasiado tiempo), tan sólo tendría que interceptar los datos de una comunicación y aplicar el cuarto paso, que es muchísimo menos costoso computacionalmente que los demás. Por ese motivo se recomienda usar grupos con primos de, al menos, 2048 bits, u otros algoritmos como el de **curvas elípticas**.

Este algoritmo está recogido en el estándar FIPS 140-2 como método de intercambio de claves, para claves de entre 2048 a 5012 bits.

Eso sí, Diffie-Hellman cuenta con un gran fallo de seguridad: es vulnerable a un *Man-In-The-Middle*. ¿Cómo sabemos que la clave que recibe Alice realmente la ha mandado Bob? Es necesario establecer alguna medida de autenticación para evitar este problema, tal y como se plantea en el siguiente algoritmo: RSA.

1.10.3 RSA

El algoritmo RSA, llamado así por sus creadores Ron Rivest, Adi Shamir y Leonard Adleman, emplea mecanismos de clave pública (criptografía híbrida) para conseguir una forma segura y muy popular de realizar firmas digitales, intercambios de clave y cifrados. Como ya hemos mencionado, su principal ventaja frente a otros métodos como Diffie-Hellman es que aporta un mecanismo de autenticación. Este algoritmo fue descrito inicialmente en 1977 y patentado en 1983 en Estados Unidos, aunque la patente expiró en el año 2000 y hoy en día es de uso público.

RSA basa su seguridad en la complejidad del problema de factorización de números enteros, que es un problema para el cual no se ha encontrado una solución en tiempo polinómico. Es por eso que necesita del uso de números compuestos por una gran cantidad de dígitos, para que el tiempo de ejecución de los algoritmos actuales haga imposible la resolución del problema. Sin embargo, los ejemplos tratados en este apartado incluirán números más pequeños, manejables por un ser humano

IDEA DEL ALGORITMO

Supongamos que podemos generar dos claves relacionadas entre sí con ciertas propiedades algebraicas que veremos más adelante, e y d . Diremos que e es la clave pública de Alice, y que d es su clave privada. Ahora, si esas claves tienen forma de números enteros, Bob podrá cifrar un mensaje m de la forma, donde n es un número que Alice y Bob conocen, y $m < n$:

$$c = m^e \pmod{n}$$

A continuación, Alice podrá descifrar el mensaje usando la operación inversa, que vienen dada por la propiedad algebraica que relaciona a e y a d :

$$m = c^d \pmod{n}$$

GENERACIÓN DE CLAVES

El corazón de RSA es el proceso de generación de sus claves pública y privada tales que cumplan la relación apreciada en el apartado anterior. El proceso de generación es el siguiente:

1. Se eligen dos números primos muy grandes, p y q , de forma aleatoria (y de longitud similar en bits). Existen numerosos *tests de primalidad* para acelerar este proceso.
2. Se toma $n = p * q$. Este nuevo número se usará como el módulo.
3. Se toma la función $\varphi(n) = (q - 1)(p - 1)$ llamada *phi de Euler*, por las propiedades explicadas anteriormente.
4. Se toma un número e como clave pública, que cumpla las condiciones de ser mayor que 1 pero menor que $\varphi(n)$. Es necesario que e y $\varphi(n)$ sean coprimos. Por motivos de optimización, se suele usar $e = 65537$. El hecho de escoger un e determinado no afecta a la seguridad del algoritmo, ya que cada n va a ser diferente y, aunque e sea siempre el mismo, la clave nunca será igual.

5. Se calcula la clave privada d , tal que $e \cdot d \equiv 1 \pmod{\varphi(n)}$. Es decir, d es el inverso multiplicativo de e . De esta forma, tenemos que $\varphi(n) \mid d \cdot e - 1$ (el símbolo \mid se lee como “divide a”). Para este proceso suele emplearse el *Algoritmo de Euclides*.
6. Ahora, los pares (n, e) y (n, d) conforman las claves pública y privada, respectivamente.
7. Los primos p y q se descartan de forma segura (ya no son necesarios).

Llegados a este punto, el proceso de cifrado y descifrado es tal y como hemos explicado en el apartado anterior:

$$c = m^e \pmod{n}$$

$$m = c^d \pmod{n}$$

Observando atentamente, nos damos cuenta de que se propone la igualdad siguiente:

$$m = m^{ed} \pmod{n}$$

Esto es así gracias a la teoría aportada por el *Teorema de Euler* y el *Teorema chino del resto*. En el Anexo 4 incluimos una demostración en profundidad de este hecho.

Para una mayor eficiencia en el algoritmo, los valores p , q , $d \bmod(p-1)$, $d \bmod(q-1)$ y $q^{-1} \bmod(p)$ se calculan de antemano y se almacenan.

COMENTARIO

En los estándares **PKCS#1 v2.0 y v2.1** se emplea la función de Carmichael $\lambda(n) = \text{lcm}(p - 1, q - 1)$ en lugar de la *phi* de Euler.

CIFRADO

Para que Bob pueda mandarle un mensaje M a Alice, esta debe compartir su clave pública con Bob. A continuación, Bob debe convertir M en un número m menor que n usando un protocolo reversible que debe acordarse de antemano (generalmente se incluye en el estándar). Por último se calcula el texto cifrado c usando el método de *exponenciación binaria* para optimizar el proceso.

PROBLEMAS DE RSA

RSA es un algoritmo determinista. Esto quiere decir que, para una misma entrada, siempre genera una misma salida: no tiene componentes aleatorios. Es por eso que para un atacante podría resultar bastante fácil obtener información acerca del mensaje.

Por otro lado, una mala elección de las claves podría permitir a un atacante vulnerar la comunicación. Esto sucede cuando elegimos números muy pequeños como clave, ya que existen algoritmos de factorización poco eficientes pero eficaces para valores pequeños. Es por eso que se recomienda usar claves de más de 2048 bits de longitud.

ESQUEMAS DE RELLENO

Para contrarrestar la naturaleza determinista del algoritmo, se puede aplicar un esquema de relleno, que no es más que un método reversible para incorporar información irrelevante (generalmente aleatoria) dentro de un mensaje. Es necesario que este esquema sea lo menos predecible posible para evitar ataques. Algunos ejemplos son:

- RSA-OAEP (Optimal Asymmetric Encryption Padding): se basa en una *Red de Feistel* con la estructura de la figura 1.77. En la imagen, las funciones G y H son *oráculos aleatorios* (cajas

negras que responden con salidas únicas irrepetibles, como por ejemplo una función hash). Este tipo de esquema se usa en el estándar PKCS#1.

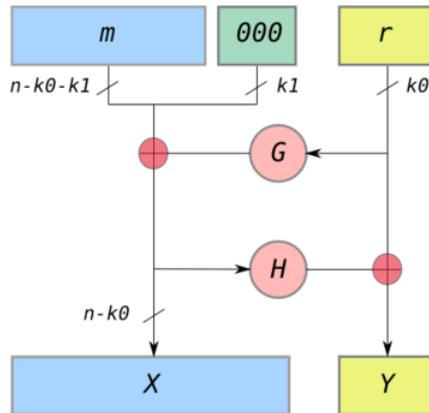


Figura 1.78 Red de Feistel para RSA-OAEP.

- RSA-SAEP+ (Simplified Asymmetric Encryption Padding)
- RSA-PSS (Probabilistic Signature Scheme): actualmente incluido en el estándar PKCS#1 v2.1.

USOS DE RSA

Como algoritmo de clave pública, RSA puede usarse para cifrar mensajes, tal y como hemos visto; puede usarse como forma de intercambiar una clave simétrica de manera segura, e incluso para generar sobres o envoltorios digitales firmados electrónicamente, para garantizar la autenticidad del mensaje. En este último caso, la idea consiste en generar un hash o resumen del mensaje a enviar y cifrarlo con la clave privada del emisor, de modo que un receptor puede comprobar que los *hashes* coinciden (integridad) y que el emisor es quien dice ser (porque conozco su clave pública, autenticación).

CONCLUSIÓN

RSA resuelve los principales problemas que presentaba Diffie-Hellman. Sin embargo, el algoritmo es mucho más lento que el de los algoritmos simétricos existentes, por lo que normalmente se emplea únicamente como intercambio de claves. En este caso, es necesario contar con un buen generador aleatorio de claves simétricas para evitar ataques.

Para la generación de claves es necesario utilizar algoritmos avanzados que garanticen la efectividad del test de primalidad en el menor tiempo posible. Además, es muy importante que los primos generados sean aleatorios y muy grandes, de forma que, además, $q-1$ y $p-1$ se descompongan en muchos factores primos.

Por otro lado, en 1993 Peter Shor publicó un algoritmo cuántico capaz de mejorar la factorización de números enteros hasta un tiempo polinomial, lo que dejaría RSA obsoleto. Sin embargo, se cree que todavía falta bastante tiempo para conseguir un avance en esta tecnología capaz de tumbar RSA.

En conclusión, RSA es el algoritmo de clave asimétrica más utilizado hoy en día. La empresa con su mismo nombre (en honor a sus fundadores) es la encargada de publicar el conjunto de estándares PKCS entre los que se encuentra el uso de este algoritmo. Además, cuenta con

definiciones en el estándar FIPS 140-2, en el que se recoge su uso junto con claves de 2048 bits como mínimo, y los algoritmos SHA1 y SHA2 con claves de 256 y 512 bits de longitud.

ENLACES DE INTERÉS

[A Method for obtaining Digital Signatures and Public-Key Cryptosystems \(R. L. Rivest et al.\)](#)

1.10.4 ElGamal

Este algoritmo diseñado por Taher Elgamal en 1985 basa su funcionamiento en el intercambio de clave Diffie-Hellman. Es conocido por ser usado en el GNU Privacy Guard, versiones recientes de PGP y porque DSA está basado en su funcionamiento (en la versión de esquema de firmadigital).

CONCEPTOS MATEMÁTICOS

Def: Se dice que un grupo G es cíclico si puede ser generado por un solo elemento, denominado generador.

Def: Se denomina *orden* de un grupo G a su cardinalidad (el número de elementos del grupo). Por otro lado, se entiende como *orden* de un elemento e a su período (es decir, el número m más pequeño tal que $e^m = 1$). En caso de no existir dicho m , se dice que el elemento tiene *orden infinito*.

GENERACIÓN DE CLAVES

Al tratarse de un algoritmo de clave asimétrica, uno de los procesos más importantes que lo conforman es el de generación del par de claves pública-privada. En ElGamal se procede de la siguiente manera:

1. Alice toma un grupo cíclico G de orden q con generador g . Digamos que e representa el elemento unidad de G .
2. Ahora toma aleatoriamente un entero $x \in \{1, \dots, q-1\}$. 3. Calcula $h := g^x$.
4. La clave pública es la tupla (G, q, g, h) que será publicada mientras Alice retiene el número x , que actúa como clave privada.

CIFRADO

Ahora Bob quiere cifrar un mensaje dirigido a Alice. Para ello:

1. Usa una aplicación reversible para identificar el mensaje M con un elemento de G, m .
2. Elige un entero $y \in \{1, \dots, q-1\}$ de forma aleatoria (clave efímera). 3. Calcula $s := h^y$.
4. Calcula $c_1 := g^y$.
5. Calcula $c_2 := m \cdot s$.
6. Envía el mensaje cifrado (c_1, c_2) a Alice.

Destacamos que, si alguien obtiene el mensaje cifrado y el texto plano m , es bastante fácil encontrar la clave secreta usando m^1 . Por eso, tras cada mensaje se recomienda calcular una nueva clave efímera.

DESCIFRADO

A continuación Alice quiere recuperar el texto del mensaje cifrado (c_1, c_2) , procediendo como sigue:

1. Calcula $s := c^x$. Como $c_1 = g^y$, $c^x = g^{xy} = h^y$, Alice y Bob pueden así compartir una clave secreta.
2. Calcula s^{-1} , la inversa de s en el grupo G (usando el Algoritmo de Euclides Extendido, por ejemplo).
3. Calcula $m := c_2 \cdot s^{-1}$. Este cálculo produce el texto original m puesto que $c_2 := m \cdot s$.
4. Emplea la aplicación de identificación inversa para obtener M a partir de m .

CONCLUSIÓN

La seguridad de ElGamal depende de las propiedades del grupo cíclico subyacente, así como del esquema de relleno utilizado: si el grupo G sostiene la **asunción computacional de Diffie-Hellman**, entonces el cifrado es unidireccional. Si, además, G sostiene la **asunción decisional de Diffie-Hellman**, entonces el algoritmo logra el estado de seguridad semántica (no se puede extraer información semántica del mensaje plano a partir del mensaje cifrado, sin pasar por un proceso de descifrado). Además, con el objetivo de evitar ataques como el de *chosen-ciphertext*, es necesario emplear esquemas de relleno eficaces e impredecibles.

Este algoritmo se usa en GNUPrivacyGuard, una implementación del OpenPGP definida en el RFC4880.

Sin embargo, su poca eficiencia como algoritmo de cifrado hace que la forma más habitual de encontrarlo sea en esquemas de criptografía híbrida, como método de intercambio de claves, por ejemplo, junto con AES.

1.10.5 ECDH (Elliptic-Curve Diffie-Hellman)

Los algoritmos de curvas elípticas fueron ideados en torno a 1980 en lo que su aplicación a criptografía se refiere. Las siglas ECC hacen referencia a *Elliptic-Curve Cryptography*, que es un conjunto de métodos criptográficos que emplean el álgebra y la aritmética de las Curvas Elípticas como estructuras algebraicas sobre campos finitos para conseguir modelos de acuerdo de claves, firma digital o generación de números pseudo-aleatorios, entre otras cosas. Una de sus principales ventajas es que aportan una función unidireccional para el cálculo de claves que es especialmente difícil de invertir, por lo que permiten el uso de claves de menor longitud que, por ejemplo, RSA.

Algunos de los algoritmos englobados en ECC están recogidos en estándares como el FIPS 140-2, siempre y cuando trabajen con ciertas curvas en concreto. Además, el algoritmo ECDH (*Elliptic-Curve Diffie-Hellman*) es uno de los métodos de intercambio de claves más utilizados hoy en día.

CONCEPTOS MATEMÁTICOS

Def 1: Se dice que un punto en una curva es **singular** si no queda expresado por una función continuamente diferenciable. En el caso de curvas algebraicas en el plano, el hecho de que un punto sea **no-singular** se traduce en que no es un vértice ni un punto de autointersección de la curva.

Def 2: Una curva plana es **no-singular** si todos sus puntos son *no-singulares*.

Prop 1: Una condición necesaria y suficiente para que una curva plana sea *no-singulares* que el discriminante $\Delta = -16(4a^3 + 27b^2)$ sea distinto de cero. Es decir, sí y sólo si $4a^3 + 27b^2 \neq 0$.

Def 3: Se dice que una *curva elíptica sobre el cuerpo K* es un conjunto de puntos $E = \{(x, y) \mid y^2 = x^3 + ax + b\} \cup \vartheta$ con $a, b \in K$, siendo ϑ un *punto en el infinito* en el *plano proyectivo* (que servirá de elemento identidad en la variedad abeliana que genera la curva). Adicionalmente, la curva tiene que cumplir la propiedad de *no-singularidad*, por lo que debe cumplir la condición $4a^3 + 27b^2 \neq 0$.

Proposición 2: toda curva elíptica es simétrica sobre el eje X. Además, para dos puntos cualesquiera de la curva, la recta que los une va a cortar la curva en un tercer punto (en caso de que los dos puntos iniciales tengan la misma coordenada X, el punto resultante será el *punto en el infinito* ϑ).

Anotación: para simplificar los cálculos, y debido a los problemas que supone el caso contrario, vamos a asumir que los cuerpos sobre los que se definen nuestras curvas tienen característica mayor que 3. De hecho, en todos nuestros ejemplos vamos a trabajar con curvas definidas sobre cuerpos finitos $\mathbb{Z}/p\mathbb{Z}$ con p primo.

Proposición 3: trabajando sobre un *plano proyectivo* en el que poder tomar un *punto en el infinito*, se puede definir una estructura de grupo sobre cualquier curva elíptica con las siguientes características:

- El elemento identidad será el *punto en el infinito* ϑ .

- La operación suma de dos puntos P y Q queda definida de forma única de la siguiente forma: se traza la línea que pasa por P y por Q , tomando el punto $-R$ de intersección entre dicha recta y la curva. El punto R será el opuesto de $-R$ sobre el eje X , teniendo que $P + Q = R$.
- En caso de que $x_P = x_Q$, para $P \neq Q$, el resultado de la suma es $P + Q = \vartheta$.
- Se define el *producto por un escalar* de un punto P como $kP = \underbrace{P + \dots + P}_k$.

Def 4: dado un elemento $G \in E(\mathbb{Z}/p\mathbb{Z})$, se entiende por **orden** de G $ord(G)$ al mínimo número n tal que $nG = \vartheta$.

Def 5: dado el grupo K , se denomina **generador** del grupo a un elemento $G \in K$ tal que $ord(G) = |K|$. Es decir, que todos los elementos del grupo se pueden escribir como múltiplos de G .

Def 6: se entiende por el **cofactor** de una curva $E(\mathbb{Z}/p\mathbb{Z})$, dado un generador G de un subgrupo de la curva, al siguiente resultado: $h = \frac{|E(\mathbb{Z}/p\mathbb{Z})|}{n}$.

OBTENCIÓN DE LA CLAVE ACORDADA

Cuando Bob y Alice quieran intercambiar un mensaje usando ECDH, tendrán que acordar previamente una serie de parámetros para establecer la comunicación:

p : define el cuerpo sobre el que trabajar $E(\mathbb{Z}/p\mathbb{Z})$.

a, b : parámetros que definen la curva empleada.

G : punto generador del subgrupo.

n : orden de G .

h : cofactor de la curva.

Para generar las claves se siguen los siguientes pasos:

1. Bob elige una clave privada $1 \leq \beta \leq n - 1$.
2. Alice elige una clave privada $1 \leq \alpha \leq n - 1$.
3. Bob calcula $B = \beta G$ y lo comparte con Alice.
4. Alice calcula $A = \alpha G$ y lo comparte con Bob.
5. Bob calcula ahora $P = \beta A = \beta \alpha G$.
- Alice calcula $P = \alpha B = \alpha \beta G$.
7. Tanto Bob como Alice comparten ahora una misma clave P , gracias a la propiedad comutativa en el grupo empleado.

CÁLCULOS NECESARIOS

Para poder realizar las operaciones de suma definidas en nuestro grupo sobre la curva elíptica, es necesario conocer un par de fórmulas geométricas básicas:

- **Pendiente de la recta que une dos puntos:** si los puntos son distintos, P y Q , la pendiente se calcula como $s = \frac{y_P - y_Q}{x_P - x_Q}$. En caso de querer hallar la pendiente para $2P$, se calcula como $s = \frac{3x_P^2 + a}{2y_P}$.
- **Coordenada X:** en el caso de la suma de puntos distintos, P y Q , la fórmula es $x_R = s^2 - (x_P + x_Q)$. Si se quiere hallar $2P$, entonces sería $x_R = s^2 - 2x_P$.
- **Coordenada Y:** en ambos casos sería $y_R = s(x_P - x_R) - y_P$.

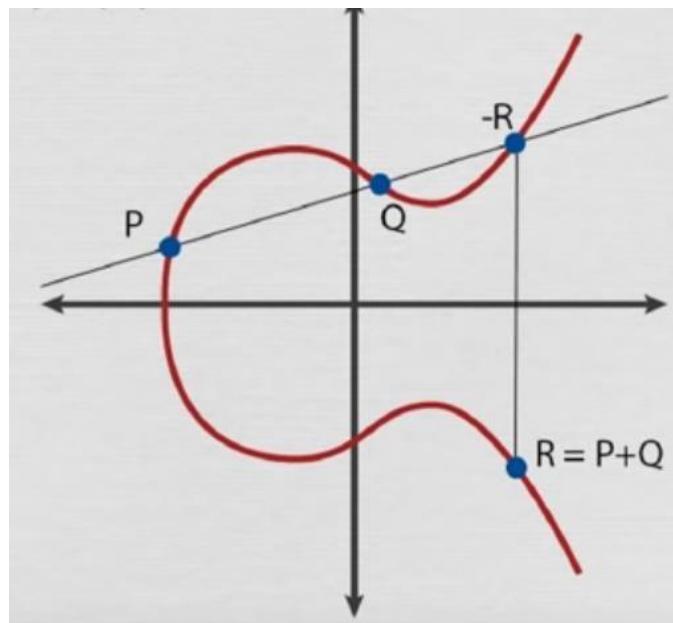


Figura 1.79 Operación suma para ECDH.

COMPUTACIÓN

Para optimizar la ejecución de los cálculos planteados, existen ciertos algoritmos con un uso bastante extendido. Uno de ellos es la llamada *escalera de Montgomery* que propone un enfoque en tiempo fijo, lo que podría evitar *side-channel attacks* basados en analizar el tiempo de computación del algoritmo.

SEGURIDAD

Situándonos ahora en el papel de un atacante llamado Eve, estos serían los datos que se han compartido en la comunicación (y que podríamos interceptar):

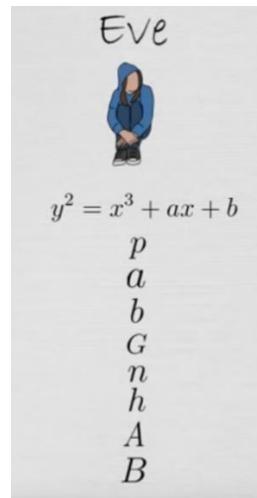


Figura 1.80 Información interceptada por un atacante en ECDH.

En esta situación, Eve tiene muy complicado hallar P ya que, aunque conoce A y B , no tiene forma de obtener β o α . Este problema se conoce como *Problema del Logaritmo Discreto en Curvas Elípticas*. La operación que nos aporta este nivel de seguridad es el producto por un escalar, que es una función unidireccional (es muy difícil calcular su inversa sin tener la información necesaria).

Además, como ya hemos comentado, el hecho de que sea más difícil resolver el problema del logaritmo discreto sobre curvas elípticas nos permite emplear claves más cortas (y, por lo tanto, obtener un algoritmo computacionalmente más rápido) que en otros algoritmos asimétricos vistos. En la siguiente tabla aparecen en la misma fila longitudes de clave que aportan el mismo nivel de seguridad para cada tipo de algoritmo.

Symmetric Encryption (Key Size in bits)	RSA and Diffie-Hellman (modulus size in bits)	ECC Key Size in bits
56	512	112
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

Figura 1.81 Comparación de longitudes de clave en varios algoritmos.

Se puede ver que para una clave de 384 bits en ECC obtenemos la seguridad equivalente a la de un RSA con claves de 7860 bits, que es mucho más de lo que realmente se usa hoy en día.

Un ejemplo de parámetros reales usados por Microsoft para implementar un algoritmo de este tipo sería:

$$\begin{aligned}p &= 785963102379428822376694789446897396207498568951 \\a &= 317689081251325503476317476413827693272746955927 \\b &= 79052896607878758718120572025718535432100651934 \\G_x &= 771507216262649826170648268565579889907769254176 \\G_y &= 390157510246556628525279459266514995562533196655\end{aligned}$$

Figura 1.82 Ejemplo de parámetros para ECDH usado por Microsoft.

CURVA25519

La **Curva25519** es una definición de curva elíptica propuesta por Daniel J. Bernstein, que ofrece una seguridad de 128 bits y fue específicamente diseñada para su uso en ECDH, siendo una de las más rápidas en ejecutar este algoritmo. Sus propiedades son:

$$y^2 = x^3 + 486662x^2 + x$$

$$p = 2^{255} - 19$$

$$G = 9$$

$$n = 2^{252} + 27742317777372353535851937790883648493$$

Este tipo de curva usa puntos elípticos comprimidos (sólo usa sus coordenadas x), lo que le permite un uso eficiente el algoritmo de la *escalera de Montgomery*, lo que la vuelve segura frente a ataques centrados en el análisis del tiempo de ejecución (el algoritmo tiene tiempo fijo).

CONCLUSIONES

ECDH es un método de intercambio de claves bastante seguro y eficaz (más que el algoritmo Diffie-Hellman original). De hecho, está recogido en el estándar FIPS 104-2 para longitudes por encima de los 112 bits y actualmente es usado por una gran cantidad de servicios y aplicaciones, como el “letter sealing” de la aplicación **LINE Messenger** o implementaciones del *Signal Protocol*, que es un protocolo para cifrados *end-to-end* que está incluido en **WhatsApp**, **Facebook Messenger** y **Skype**.

Una de sus implementaciones más famosas, la **Curva25519**, tuvo gran repercusión cuando en 2013 Edward Snowden descubrió que la NSA habría implementado una *backdoor* en una implementación basada en curvas elípticas para la generación de números pseudo-aleatorios. Este hecho llevó a sospechar de todas las constantes definidas hasta el momento para curvas elípticas, lo que llevó a adoptar la **Curva25519** como el nuevo estándar de implementación. De hecho, librerías como OpenSSL o Crypto++, protocolos como SSH o Signal Protocol y aplicaciones como WhatsApp y Vibes usan esta curva en sus implementaciones de ECDH.

Sin embargo, se estima que se necesitarían casi el doble de qubits en un computador cuántico para romper RSA que para romper algoritmos de curvas elípticas, lo que hace que ECDH sea un blanco más fácil de la criptografía cuántica (aunque hoy en día es prácticamente irrompible).

ENLACES DE INTERÉS

[Elliptic Curve Diffie-Hellman: A Key Exchange Protocol \(Robert Pierce, youtube\)](#)

1.10.6 Cramer-Shoup

Este algoritmo simétrico de cifrado, desarrollado por R. Cramer y V. Shoup en el año 1998, fue el primer esquema eficiente que demostró ser seguro frente a ataques *adaptive chosen ciphertext* (CCA2), que se basaban en la inyección de textos cifrados arbitrarios para su descifrado con el objetivo de conseguir la información necesaria para descifrar un mensaje en concreto. Actualmente, este nivel de seguridad es el más alto conocido para algoritmos de clave asimétrica, ya que asume que un atacante puede descifrar cualquier texto (salvo el objetivo) usando la clave de la comunicación (aunque no tiene por qué conocerla).

Al igual que ElGamal (Cramer-Shoup está basado en este algoritmo) consta de tres fases: generación de la clave, cifrado y descifrado. Toda la teoría se basa en los mismos conceptos de *grupos cíclicos* que ElGamal.

GENERACIÓN DE CLAVES

Para generar las claves de cifrado, Cramer-Shoup requiere de los siguientes pasos:

1. Alice toma una descripción de un grupo \mathbf{G} de orden q usando dos generados distintos aleatorios \mathbf{g}_1 y \mathbf{g}_2 .
2. Alice toma cinco valores aleatorios (x_1, x_2, y_1, y_2, z) del conjunto $\{0, \dots, q-1\}$.
3. Calcula $c = g_1^{x_1} g_2^{x_2}$, $d = g_1^{y_1} g_2^{y_2}$, $h = g_1^z$
4. La clave pública de Alice será la descripción del grupo \mathbf{G} , q , \mathbf{g}_1 , \mathbf{g}_2 junto con (c, d, h) .
5. La clave privada de Alice será (x_1, x_2, y_1, y_2, z) .

CIFRADO

Para cifrar un mensaje m para Alice usando su clave pública $(\mathbf{G}, q, \mathbf{g}_1, \mathbf{g}_2, c, d, h)$, Bob seguirá los siguientes pasos:

1. Bob identifica m con un elemento de \mathbf{G} .
2. Elige un número aleatorio k del conjunto $\{0, \dots, q-1\}$.
3. Calcula:
 - $u_1 = g_1^k$, $u_2 = g_2^k$
 - $e = h^k m$
 - $\alpha = H(u_1, u_2, e)$ donde $H()$ es una función *hash* unidireccional (preferiblemente resistente a colisiones).
 - $v = c^k d^{\alpha k}$.
4. Bob envía el mensaje cifrado (u_1, u_2, e, v) a Alice.

DESCIFRADO

Teniendo ahora el mensaje cifrado (u_1, u_2, e, v) y la clave privada de Alice (x_1, x_2, y_1, y_2, z) se sigue el siguiente proceso:

1. Alice calcula $\alpha = H(u_1, u_2, e)$ y verifica que $u_1^{x_1} u_2^{x_2} (u_1^{y_1} u_2^{y_2})^\alpha = v$. Si el test falla, se cancela el descifrado.
2. En caso de que se pase el test, el mensaje sería $m = e / u_1^{x_1}$.

CONCLUSIONES

Este algoritmo surgió como una mejora de ElGamal para paliar su *maleabilidad*, lo que lo hacía vulnerable a ataques *chosen-ciphertext*. La principal mejora es la inclusión de una función hash unidireccional, generando un texto cifrado el doble de largo que el de ElGamal.

En cuanto a seguridad, está considerado un algoritmo CCA2, es decir, que el texto cifrado es indistinguible ante ataques *chose-ciphertext* (incluso los adaptativos). Esto convierte a Cramer-Shoup en uno de los algoritmos asimétricos de cifrado más robustos conocidos, preferible ante otros como ElGamal.

1.11 Criptografía Post-Cuántica

Uno de los principales retos científicos actuales consiste en conseguir desarrollar los denominados “ordenadores cuánticos”: unos computadores capaces de conseguir una capacidad y velocidad de computación que hasta ahora eran impensables. La principal característica de estos ordenadores es que sustituyen los clásicos bits por *qubits*, que basan su funcionamiento en la teoría cuántica. De una forma muy superficial, se podría decir que un *qubit* tiene infinitos estados, aunque a la hora de leer su contenido siempre va a decantarse por un 1 o un 0.

El hecho de trabajar con *qubits* en vez de bits ha abierto las puertas a una serie de operaciones lógicas que hasta ahora no eran posibles, y se han diseñado algoritmos muy elegantes que resuelven problemas que hasta hoy en día eran computacionalmente imposibles de resolver. Un ejemplo de estos algoritmos es el *Algoritmo de Shor*, que permite resolver el problema de la descomposición en factores de enteros de manera eficiente (en notación O , tardaría un tiempo $O((\log N)^3)$, siendo mucho más rápido que cualquier otro algoritmo). En concreto, la eficiencia del Algoritmo de Shor deja obsoleto el criptosistema RSA, así como otros algoritmos de clave asimétrica basados en problemas relacionados con la descomposición de enteros.

Sin embargo, hoy en día no se ha alcanzado la madurez necesaria en el campo de la computación cuántica como para que esta clase de algoritmos sean posibles (ni se va a alcanzar en unos cuantos años). Por poner un ejemplo, para poder implementar un Algoritmo de Shor capaz de romper RSA en unas pocas semanas, haría falta tener un computador cuántico con miles (e incluso cientos de miles) de *qubits*. Hoy en día, el ordenador cuántico de IBM apenas cuenta con cien *qubits*.

Pero nunca está de más ser precavido, así que la organización europea ECRYPT lanzó en 2016 una iniciativa para encontrar el primer algoritmo post-cuántico con las características necesarias para sustituir a los criptosistemas actuales. Aunque actualmente se encuentra abierto el proceso de selección (se espera que finalice para 2024), hay un par de algoritmos bastante prometedores: *McEliece* y *NTRU*.

1.11.1 McEliece

En 1978, Robert McEliece desarrolló un algoritmo capaz de incluir verdadera *aleatoriedad* en el proceso de cifrado. Pese a que nunca ha sido muy aceptado por la comunidad criptográfica, hoy en día es uno de los pocos candidatos de la *criptografía post-cuántica* ya que es inmune al algoritmo de Shor y otros ataques basados en algoritmos cuánticos.

La base de este algoritmo es la dificultad de decodificar los denominados *códigos lineales*, tratándose de un conocido problema NP (de dificultad no polinómica).

CÓDIGOS LINEALES

Se trata de un tipo concreto de código para corrección de errores en señales, con la propiedad de que cualquier combinación lineal de palabras del código es también una palabra del código, lo que los hace más eficientes a la hora de codificar/decodificar.

Def: En álgebra, un **espacio vectorial sobre un cuerpo K** es una estructura formada por un conjunto no vacío V , una operación interna (suma) y una operación externa (producto) con las propiedades:

- *Cerrado para las dos operaciones.*
- *Suma conmutativa yasociativa.*
- *Elemento neutro para la suma.*
- *Elemento opuesto para la suma.*
- *Producto conmutativo yasociativo.*
- *Propiedad distributiva para elementos y escalares.*
- *Existencia de un neutro para el producto.*

A los elementos de un espacio vectorial se les denomina **vectores**. A todo subconjunto de un espacio vectorial que satisface por sí mismo todas las propiedades anteriores se le denomina **subespacio vectorial**.

Def: un conjunto vectores $v_1 \dots v_n$ son **linealmente independientes** si dada la ecuación $a_1v_1 + \dots + a_nv_n = 0$, esta sólo se satisface cuando $a_1 = \dots = a_n = 0$.

Def: se entiende por **base** de un espacio vectorial a un conjunto de vectores *linealmente independientes* que generan el espacio vectorial completo (operándose entre sí mismos).

Def: dado un espacio vectorial V y una base de dicho espacio B , se dice que V **tiene una dimensión k** si hay k elementos linealmente independientes en la base B .

Def: un **código lineal** de longitud n y rango k es un subespacio lineal C con dimensión k del espacio vectorial \mathbb{F}_q^n donde \mathbb{F}_q es el cuerpo finito con q elementos. A cada vector en C se le denomina *palabra de código*.

ANOTACIÓN

El uso de códigos lineales abre las puertas de una teoría matemática completa al respecto. Con el objetivo de ilustrar un poco esta área, hemos incluido las definiciones anteriores, que ofrecen unas pinceladas del uso de códigos lineales como espacios vectoriales sobre los que definir una aritmética sobre la que trabajar.

Sin embargo, como no es objetivo de este documento profundizar en la base matemática más allá de lo necesario para comprender un algoritmo, no vamos a profundizar más en este nuevo campo. Sin embargo, animamos al lector a indagar acerca de la elección de códigos binarios y sus algoritmos de codificación/decodificación.

En los siguientes puntos vamos a asumir que el usuario es capaz de escoger un código lineal con las características solicitadas para cada caso.

GENERACIÓN DE LA CLAVE

Inicialmente se acuerda un conjunto de valores de seguridad: n, k, t .

1. Alice toma un (n, k) -Código Lineal \mathbf{C} capaz de corregir t errores. Dicho código tiene que contar con un algoritmo eficiente de decodificación y generar una matriz generadora $k \times n$, denominada \mathbf{G} para el código \mathbf{C} .
2. Alice toma una matriz *no-singular*, binaria, aleatoria de orden $k \times k$, denominada \mathbf{S} .
3. Toma a continuación una matriz \mathbf{P} de permutación aleatoria, $n \times n$.
4. Calcula la nueva matriz de orden $k \times n$, $\hat{\mathbf{G}} = \mathbf{S}\mathbf{G}\mathbf{P}$.
5. La clave pública de Alice es $(\hat{\mathbf{G}})$ y la privada es $(\mathbf{S}, \mathbf{G}, \mathbf{P})$.

CIFRADO

Si Bob ahora quiere mandar un mensaje a Alice conociendo su clave pública $(\hat{\mathbf{G}})$, se sigue el siguiente proceso:

1. Bob codifica el mensaje m como una cadena binaria de longitud k .
2. Calcula el vector $\mathbf{c}' = \mathbf{m}\hat{\mathbf{G}}$
3. Genera un vector de n bits aleatorio, denominado \mathbf{z} , que contenga exactamente t unos (se dice que tiene *peso* t).
4. Se computa el texto cifrado como $\mathbf{c} = \mathbf{c}' + \mathbf{z}$.

DESCIFRADO

Al recibir \mathbf{c} , Alice podrá descifrar el mensaje:

1. Calcula la inversa de \mathbf{P} .
2. Calcula $\hat{\mathbf{c}} = \mathbf{c}\mathbf{P}^{-1}$.
3. Alice usa el algoritmo de decodificación del código \mathbf{C} para decodificar $\hat{\mathbf{c}}$ y obtener así $\hat{\mathbf{m}}$.
4. Finalmente, obtiene $m = \hat{\mathbf{m}}\hat{\mathbf{G}}^{-1}$.

COMENTARIOS

Originalmente se sugirió el uso de parámetros de seguridad de longitud $n = 1024, k = 524$ y $t = 50$. Sin embargo, para ofrecer *resiliencia cuántica* se recomiendan tamaños $n = 6960, k = 5413$ y $t = 119$, con unas longitudes de clave pública de 8.373.911 bits.

Aunque es raro su uso hoy en día, algunas implementaciones del criptosistema se han visto, por ejemplo, en el cliente de Mensajería Instantánea y E-Mail gloobug.sf a partir de la versión 3.1.

El algoritmo McEliece presenta, no obstante, algunas ventajas frente a otros como RSA, como que el proceso de cifrado/descifrado es más rápido. Además, recientemente se propuso un esquema similar, denominado Niederreiter, que permite producir firmas digitales. Sin embargo, el principal problema de McEliece es que produce claves en forma de matrices enormes, lo que lo vuelve poco útil en la práctica.

1.11.2 NTRU

Se trata de un criptosistema basado en retículos (*lattices*) teóricamente resistente al algoritmo de Shor, por lo que es uno de los principales representantes de la *criptografía post-cuántica*. Se basa en la dificultad del problema de factorizar ciertos polinomios en un anillo polinómico arbitrario. Se cree que la dificultad está íntimamente relacionada con la del problema de la reducción de retículos, que es uno de los más resistentes a la computación cuántica conocidos.

Este criptosistema está formado por dos algoritmos: NTRUEncrypt, como algoritmo de cifrado, y NTRUSign, como algoritmo de firma digital. Sin embargo, el esquema de firma digital presenta algunos problemas que lo excluyen del estándar IEEE P1363 (de momento).

En este apartado vamos a estudiar exclusivamente el proceso de cifrado en intercambio de claves, siguiendo el algoritmo NTRUEncrypt.

El anillo de polinomios sobre el que se trabaja es $R = \mathbb{Z}[X]/(X^n - 1)$, con polinomios de coeficientes enteros y grado menor o igual a $N - 1$: $a = a_0 + a_1x + \dots + a_{N-1}x^{N-1}$.

El sistema necesario para el proceso viene definido por un entero N presumiblemente primo y dos números q y p ($q > p$) coprimos entre sí, así como cuatro conjuntos de polinomios $\mathcal{L}_f, \mathcal{L}_g, \mathcal{L}_m, \mathcal{L}_r$ para la clave privada, pública, el mensaje y un *blinding value* respectivamente.

GENERACIÓN DE CLAVES

Se necesitan dos polinomios f y g de grado menor o igual a $N - 1$ con coeficientes en $\{-1, 0, 1\}$, aleatorios. Se pueden considerar como las representaciones de las clases residuales de polinomios módulo $X^N - 1$ en R . El polinomio $f \in \mathcal{L}_f$ debe cumplir, además, que existan sus inversos módulo q y módulo p (usando el algoritmo de Euclides), siendo éstos f_p y f_q . De este modo, si f no es invertible, es necesario reiniciar el algoritmo.

La terna (f, f_p, f_q) conforman la clave privada de Alice. La clave pública h se genera como:

$$h = pf_q \cdot g \pmod{q}$$

CIFRADO

Ahora Bob quiere mandarle un mensaje a Alice conociendo la clave pública h . Para ello calcula $c = r \cdot h + m \pmod{q}$, que será el texto cifrado.

DESCIFRADO

Alice recibe el mensaje cifrado c y quiere descifrarlo, conociendo su clave privada (f, f_p, f_q) . Para ello:

1. $a = f \cdot e \pmod{q} = f \cdot (r \cdot h + m) \pmod{q} = f \cdot (r \cdot pf_q \cdot g + m) \pmod{q} = pr \cdot g + f \cdot m \pmod{q}$
2. $b = a \pmod{p} = f \cdot m \pmod{p}$. Esto se debe a que $pr \cdot g \pmod{p} = 0$.
3. Por último, conociendo b , Alice puede recuperar el mensaje original usando f_p : $m = f_p \cdot b \pmod{p} = f_p \cdot f \cdot m \pmod{p}$ por la propiedad de que f_p es la inversa de f .

CONCLUSIONES

La criptografía basada en retículos conforma una de las principales líneas de investigación de la criptografía *post-cuántica*, y NTRU es uno de sus mayores representantes (pese a que no contiene el problema de reducción de retículos como tal). Inicialmente fue publicado bajo la patente a nombre de NTRU Cryptosystems, Inc. de la mano de Daniel Lieman, Joseph H. Silverman y otros colaboradores, aunque en 2017 caducó la patente y fue publicado para uso general bajo la licencia GPL.

Presenta ciertas ventajas frente a otros criptosistemas asimétricos, como RSA o los basados en curvas elípticas: es resistente a los algoritmos cuánticos conocidos y, además, la simplicidad de sus operaciones lo vuelven más rápido en comparación. Sin embargo, se encuentra en proceso de estudio y su uso no está muy extendido.

1.12 Funciones Hash

1.12.1 Definición

Una función *hashes* un algoritmo matemático que convierte un mensaje de una longitud arbitraria en una cadena de bits de longitud fija de forma unidireccional (es decir, es muy difícil obtener el texto original a partir de la cadena de bits resultante). Idealmente, la única forma de encontrar un mensaje a partir de su *hash digest* (así se llama a la salida de la función) es mediante la fuerza bruta (probando mensajes hasta que los *hashes* coincidan).

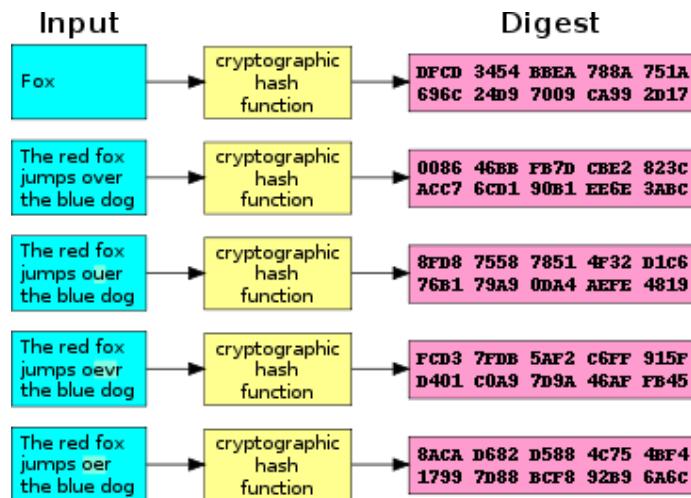


Figura 1.83 Ejemplo de función hash.

Una función *hash* tiene que cumplir ciertas características:

- Es una función determinista: una misma entrada siempre da la misma salida.
- Es rápida.
- Es “imposible” obtener un mensaje a partir de su *hash*.
- Es “imposible” encontrar *colisiones* (dos mensajes distintos que produzcan el mismo *hash*).
- Un pequeño cambio en el mensaje se traduce en un *hash* totalmente distinto (efecto *avalancha*) y aparentemente sin correlación.

1.12.2 Seguridad

Las funciones *hash* se diseñan teniendo en mente una serie de objetivos que garantizan su seguridad. El principal, ya mencionado, es que sea una función unidireccional. Además, es muy importante que no se produzcan colisiones, ya que esto podría dar lugar a *ataques de cumpleaños* (un tipo de ataque basado en teoría probabilística que depende de que exista una probabilidad elevada de encontrar colisiones). De hecho, no sólo se busca que no se produzcan colisiones, sino que, idealmente, dos mensajes distintos no deberían generar un *hash* parecido.

En definitiva, una función *hash* debe comportarse de forma ideal como una función aleatoria.

1.12.3 Aplicaciones

En criptografía, las funciones *hash* tienen una gran cantidad de aplicaciones:

- **Asegurar la integridad del mensaje:** si un emisor calcula el *hash digest* de un mensaje y luego lo envía, el receptor puede calcular el *hash* del mensaje recibido para confirmar que nadie ha modificado su contenido.
- **Firma digital:** uno de los procedimientos de firma digital más empleados consiste en calcular el *hash* de un mensaje y enviarlo cifrado usando la clave privada del emisor. De ese modo, cualquiera puede descifrar el mensaje y calcular el *hash* para comprobar que no se ha modificado el contenido, y que el emisor es quien dice ser.
- **Verificación de contraseñas:** muchos sistemas operativos, como Windows, no almacenan las contraseñas en texto plano, ya que esto supondría una importante brecha de seguridad. En su lugar, almacenan el *hash*, de modo que para comprobar la validez de una contraseña introducida, basta con comprobar que su *hash* coincide con el almacenado.
- **Proof-of-work:** una prueba de trabajo o proof-of-work consiste en solicitar a un usuario un trabajo que requiera cierto esfuerzo computacional. Se usa, por ejemplo, para evitar denegaciones de servicio, y su uso está muy extendido por las redes de criptomonedas.

1.12.4 Rainbow Tables

Como ya hemos mencionado, el único ataque (teóricamente) factible contra una función *hash* es por un proceso de fuerza bruta. Sin embargo, existen unas tablas que contienen pares *hash*-mensaje que han sido precomputados y que permiten acelerar el proceso de fuerza bruta. A este tipo de tablas se las denomina *Rainbow Tables*.

1.12.5 Diseño

Muchas funciones *hash* se diseñan basándose en algoritmos simétricos de cifrado en bloque y en sus modos de operación. Sin embargo, los algoritmos tomados como base se modifican para garantizar el principio de **construcción Merkle-Damgard**: el algoritmo debe transformar cadenas de texto de longitud variable a *hashes* de la misma longitud, garantizando que la función *hash* sea tan resistente a colisiones como lo es su función de compresión. SHA-1 y MD5 usan este tipo de construcción.

1.13 Algoritmos de Funciones Hash

1.13.1 MD5

Ideado por Ronald Rivest y publicado por primera vez en 1992 como [RFC 1321](#), MD5 supuso el primer algoritmo de la serie MD (MD4 y MD2 le precedieron, ambos desarrollados por el mismo autor) en ser ampliamente utilizado. Denominado inicialmente como **message-digest algorithm**, genera *hashes* de 128 bits independientemente del tamaño de la entrada. Sin embargo, a lo largo de los años se le han encontrado numerosas vulnerabilidades y, actualmente, no se recomienda su uso (salvo para el cálculo de *checksums* y otras actividades no criptográficas).

SEGURIDAD

Parte de la seguridad de una función *hash* criptográfica se basa en la propiedad de que dos vectores de inicialización diferentes generan dos *hashes* diferentes. En caso de encontrar dos entradas distintas que generen la misma salida, se dice que se ha producido una *colisión*. Este suceso vulnera considerablemente a la función *hash*, ya que si es posible encontrar colisiones de manera sistemática y en un período de tiempo pequeño, no tiene sentido usar esa función para, por ejemplo, almacenar contraseñas (dos contraseñas distintas pero que colisionen permitirían el acceso al sistema) o firmar digitalmente.

En diciembre de 2010, un grupo de investigadores publicaron la primera colisión de un bloque de 512 bits en MD5. Y ya desde 2006 se conocía un algoritmo capaz de encontrar colisiones individuales en un tiempo muy reducido. De hecho, se han publicado métodos para vulnerar PKI's al completo, falsificando certificados. Es por eso que, en 2011, se publicó el [RFC 6151](#) en el que se declara obsoleto a MD5 en términos criptográficos.

ALGORITMO

MD5 divide el mensaje de entrada en bloques de 512 bits, extendiendo el mensaje para que su longitud sea divisible por 512. Para extenderlo, se añade inicialmente un único bit '1' al final; a continuación se añaden tantos ceros como haga falta hasta conseguir que el mensaje tenga 64 bits menos que un múltiplo de 512; por último, se rellenan los 64 bits restantes con la longitud del mensaje inicial, módulo 2^{64} .

El algoritmo opera sobre un estado de 128 bits dividido en cuatro fragmentos de 32 bits cada uno, denominados A, B, C y D, que serán inicializados con unas constantes fijas. Los bloques de 512 bits en los que se ha dividido el mensaje se usarán para modificar el estado, en un procedimiento denominado *ronda*. Para cada bloque, se someterá al estado a cuatro de estas rondas, compuestas cada una por 16 operaciones similares basadas en una función *F* no lineal, sumas modulares y rotaciones a la izquierda.

Hay cuatro posibles funciones *F* y cada una de ellas se aplicará en una ronda diferente:

$$\begin{aligned} F(B, C, D) &= (B \wedge C) \vee (\neg B \wedge D) \\ G(B, C, D) &= (B \wedge D) \vee (C \wedge \neg D) \\ H(B, C, D) &= B \oplus C \oplus D \\ I(B, C, D) &= C \oplus (B \vee \neg D) \end{aligned}$$

Los símbolos \wedge , \vee , \oplus , \neg denotan las operaciones AND, OR, XOR y NOT, respectivamente.

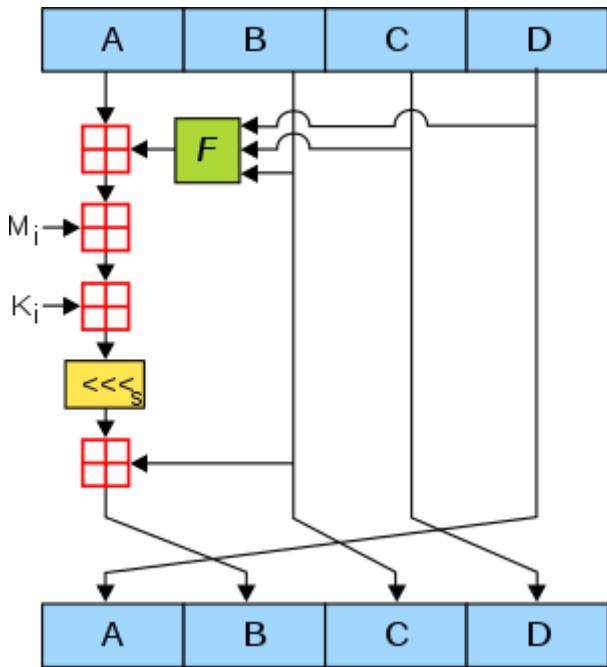


Figura 1.84 Esquema de una ronda MD5.

En la figura anterior se muestra el esquema de una ronda de MD5. M_i representa el fragmento i -ésimo de 32 bits perteneciente al mensaje inicial, de 512 bits. De esta forma, el bloque de 512 bits se verá sometido a 16 operaciones en cada ronda. K_i representa una constante de 32 bits definida al comienzo del algoritmo.

Tras pasar por todas las rondas, la salida de la función *hash* será la concatenación de las cuatro variables A, B, C y D, de 32 bits cada una, sumando 128 bits en total.

En el Anexo 5 se incluye el pseudocódigo de este algoritmo. Es importante indicar que todos los datos trabajados se encuentran en formato *Little-Endian*.

CONCLUSIÓN

MD5 es una de las funciones *hash* más empleadas de la historia. De hecho, pese a que hoy en día se conocen sus vulnerabilidades, todavía se sigue usando en gran cantidad de circunstancias y sigue teniendo soporte en muchas librerías, como OpenSSL.

Por todo lo comentado, su uso no es nada recomendable en general y existen otras muchas alternativas para usar como función *hash* criptográfica.

1.13.2 MD6

Tras reconocer públicamente que MD5 quedó obsoleto al encontrarse un algoritmo para generar colisiones, el equipo de Rivest desarrolló su sucesor: MD6. El objetivo de este nuevo algoritmo era convertirse en el nuevo estándar NIST SHA-3. Sin embargo, los propios ingenieros comentaron que el algoritmo no estaba preparado para convertirse en estándar ya que, pese a no haberse encontrado ataques que vulneraran MD6, tampoco se habían encontrado evidencias de lo contrario.

MD6 acepta mensajes de hasta $2^{64} - 1$ bits, generando *digests* de la longitud deseada, entre 1 y 512 bits. Emplea una única función de compresión, independientemente del tamaño de la salida deseada, para convertir bloques de 4096 bits en bloques de 1024 bits. Además, únicamente se usan las operaciones XOR, AND y SHIFT (rotaciones a la izquierda y a la derecha) sobre palabras de 64 bits.

El algoritmo está basado en árboles de Merkle, que son estructuras en forma de árbol en las que las hojas contienen el hash de un bloque de datos, mientras que los nodos no-hojas contienen el hash de los contenidos de sus hijos.

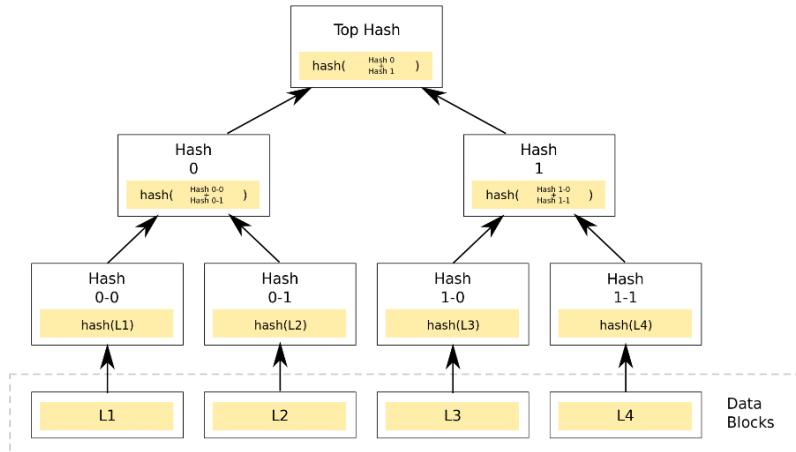


Figura 1.85 Ejemplo de árbol de Merkle

Actualmente MD6 es un algoritmo más bien experimental, que muy poca gente usa. No se encuentra implementado en las principales librerías ni hay evidencias de su seguridad. Por estos motivos, no vamos a profundizar en el algoritmo. Sin embargo, si el lector lo desea, en el *paper* original se detalla el funcionamiento de MD6 y, además, existen diversas implementaciones en la web.

ENLACES DE INTERÉS

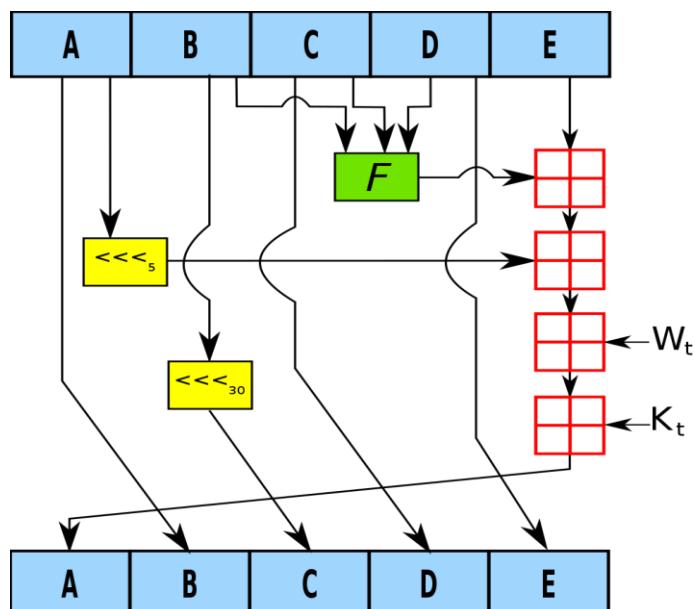
[The MD6 hash function: A proposal to NIST for SHA-3 \(R. L. Rivest et al. 2009\)](https://www.cse.lehigh.edu/~rivest/pubs/paper154.pdf)

1.13.3 SHA-1

Secure Hash Algorithm 1 (SHA-1) es una función *hash* desarrollada por la NSA en 1995, que se convirtió en estándar nacional FIPS 180-4. Su diseño formó parte del proyecto Capstone del gobierno estadounidense, cuyo objetivo era desarrollar estándares criptográficos para uso público y gubernamental, aunque cesó sus actividades debido en parte a una gran resistencia de la comunidad criptográfica, ya que la NSA y el NIST mantenían diseños clasificados y, por lo tanto, sospechosos.

Una de sus principales aplicaciones en criptografía tiene lugar en el proceso de firma digital. De hecho, el estándar *Digital Signature Standard* (DSA) incorporaba este algoritmo (aunque en sus versiones). Además, como función hash criptográfica permite asegurar integridad de datos, por lo que sistemas como *Git* lo incorporan para identificar revisiones y asegurar que no se hayan alterado datos.

SHA-1 produce hashes de 160 bits (20 bytes), aunque está basado en principios muy similares a los de MD5 (Merkle-Damgård), pero con 80 rondas en total.



1.13.4 SHA-2

En 2001 la NSA publicó una nueva familia de funciones hash, recogidas bajo el nombre SHA-2. Se trata de un conjunto de seis funciones que generan hashes de 224, 256, 384 y 512 bits: **SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 y SHA-512/256**.

Actualmente, todos los algoritmos de esta familia se encuentran recogidos en el estándar FIPS 180-4, así como en el estándar japonés CRYPTREC y en el europeo NESSIE.

Estos algoritmos, al igual que SHA-1, se basan en la construcción de Merkle-Damgard, usando la función de compresión Davies-Meyer, que consiste en usar un algoritmo de cifrado en bloque para cifrar el *hash* de la ronda anterior (usando como clave un fragmento del mensaje), haciendo un XOR con el resultado de este cifrado.

En la actualidad no se ha encontrado ninguna colisión para SHA 256 (ni para ninguno de los demás algoritmos de esta familia). Por lo tanto, cuando se usen funciones hash, se recomienda que sean usadas las de las familias SHA-2 o SHA-3, ya que son mucho más seguras que las anteriores. De hecho, gran cantidad de aplicaciones y protocolos actuales utilizan funciones de esta familia; en concreto, las más utilizadas son SHA256 y SHA512.

1.13.5 SHA-3

Se trata del último miembro de la familia *Secure Hash Algorithm*. Este estándar fue publicado por el NIST en 2015, y tiene como objetivo el sustituir a SHA-2 en aplicaciones.

Los algoritmos de esta familia se basan en el enfoque de la “construcción de esponja”, introducido por primera vez por el algoritmo Keccak, en el que se basa SHA-3. El término procede de que el algoritmo utilizado puede “absorber” cualquier cantidad de datos y “exprimir” cualquier cantidad, lo que lo convierte en un algoritmo muy flexible.

En cuanto al funcionamiento interno, únicamente vamos a decir que se basa en la aplicación de permutaciones y de una transformación en bloque *f*, que combina XOR, AND y NOT y está diseñada para una fácil implementación.

1.13.6 BLAKE

El algoritmo BLAKE está basado en el cifrado ChaCha de Daniel Bernstein, con la diferencia de añadir una copia permutada del bloque de entrada en cada ronda, operada XOR con una serie de constantes. BLAKE ofrece salidas de 224, 256, 384 y 512 bits.

Debido a la velocidad y robustez del algoritmo, BLAKE llegó a la final de la competición para formar parte del estándar SHA-3, que ganó Keccak. Sin embargo, sigue siendo un algoritmo válido que se puede encontrar en gran cantidad de librerías criptográficas.

En 2012 se publicó una segunda versión de este algoritmo, denominada BLAKE2, que pretendía sustituir a MD5 y SHA-1 en aplicaciones que requieran un alto rendimiento.

1.13.7 Streebog

Esta función hash se incluyó en 2012 en el estándar ruso GOST como sustituto de la función hash previa (ya obsoleta). Además, fue propuesto como candidato a la competición SHA-3.

Streebog está basado en la construcción *Merkle-Damgard*, al igual que MD5, SHA-1 o SHA-2, pero incluye algunas características que lo convierten en una opción más segura que los dos primeros. Sin embargo, la eficiencia y seguridad de los algoritmos SHA-2, SHA-3 y BLAKE elevaron a Streebog a un segundo plano, por lo que en este libro carece de mayor interés.

1.14 Autenticación e Integridad

Como ya mencionamos al comienzo del documento, hay una serie de condiciones que un sistema criptográfico debe garantizar. Dos de las más importantes son la de **integridad de los datos** (el receptor debe ser capaz de verificar que el mensaje recibido no ha sufrido una modificación no permitida) y la de **autenticación** (el receptor debe poder verificar la identidad del emisor), aunque muchas veces vienen de la mano. Existen diversas formas de garantizar estas propiedades, ya sea empleando MACs (*message authentication codes*), *cifrado autenticado* (AE) o *firmas digitales*.

MAC

Un código de autenticación de mensajes o MAC (a veces conocido como *tag*) es un fragmento de información que permite garantizar la autenticidad (y la integridad) de un mensaje.

En general, el proceso para generar y verificar MACs consta de tres algoritmos:

- Un generador de claves en un espacio aleatorio.
- Un algoritmo de firma que genere un *tag* único para cada par clave-mensaje.
- Un algoritmo de verificación que garantice la seguridad dadas la clave y el *tag*.

Aunque conceptualmente una función MAC es muy parecida a una función *hash*, en la realidad se les exige una serie de características diferentes. La más importante es que las funciones MAC deben ser resistentes a ataques de falsificación existencial, lo que significa que incluso un atacante que tenga acceso a un *oráculo* con acceso a la clave y pueda solicitarle que genere MACs para cualquier mensaje arbitrario, el atacante no debe ser capaz de inferir información acerca de otros MACs.

Por otro lado, los MACs difieren de las firmas digitales en que estos emplean la misma clave para firmar que para verificar, mientras que las firmas digitales emplean mecanismos asimétricos.

CIFRADO AUTENTICADO

Este tipo de cifrado se caracteriza por garantizar la autenticidad del mensaje, aparte de la confidencialidad del mismo. Para ello, a partir del mensaje y la clave de cifrado se genera un texto cifrado y, por otro lado, un *tag* de autenticación.

Al igual que las funciones MAC, un algoritmo de estas características debe ser resistente a atacantes a los que se les presupone el acceso a un *oráculo*. Es por eso que, en muchas ocasiones, los algoritmos de cifrado autenticado combinan un cifrador en bloque y una función MAC, de forma que el primero se emplea para cifrar el mensaje, y la segunda para generar el código MAC. Si, por ejemplo, recordamos el algoritmo simétrico Grain-128a, nos daremos cuenta de que incluye en su esquema un método para garantizar la autenticación, añadiendo una función MAC que genera un *tag*. El modo de operación GCM para algoritmos de bloque es otro conocido método de cifrado autenticado.

Un caso concreto de cifrado autenticado es el AEAD (*Authenticated Encryption with Associated Data*). Consiste en añadir datos asociados al texto cifrado que necesitan verificar su autenticidad e integridad. Por ejemplo, en los paquetes de red, las cabeceras requieren integridad, pero sin estar cifradas.

FIRMA DIGITAL

Este esquema se diseñó con el objetivo de garantizar la integridad y la autenticidad de mensajes y documentos. Se trata del esquema más utilizado en la actualidad y cuenta con numerosas implementaciones bastante elegantes (basadas, por ejemplo, en Diffie-Hellman).

La idea subyacente bajo el concepto de firma digital es la de emplear criptografía asimétrica para generar firmas. De este modo, emisor y receptor no necesitan compartir una clave para poder emitir y verificar una firma.

El esquema consta de tres algoritmos:

- Un generador de claves dentro de un espacio aleatorio (pública y privada).
- Un algoritmo de firma que, dado el mensaje y una clave privada, genere el contenido de la firma.
- Un algoritmo de verificación que, dada la firma y la clave pública, garantice la autenticidad del contenido.

De este modo, cuando Alice quiere mandar un mensaje a Bob garantizando su autenticidad y su integridad, firmará un fragmento de texto acordado (generalmente, un resumen *hash* del mensaje) usando su clave privada. Bob podrá descifrar el texto de la firma mediante la clave pública de Alice y comprobar que el *hash* resumen del mensaje recibido se corresponde con el que acaba de descifrar.

Sin embargo, este planteamiento presenta un problema importante: cualquier atacante tiene acceso al contenido de la firma digital. Por lo tanto, sería posible engañar a Bob y hacerle creer que una nueva firma “modificada” por el atacante es la que realmente se corresponde con el mensaje recibido. Es cierto que este ataque presenta una dificultad extra, que es la de hacer creer a Bob que la nueva firma procede de Alice, por lo que es prácticamente imposible llevarlo a cabo.

Para solucionar el problema anterior, se propone un nuevo esquema de comunicación:

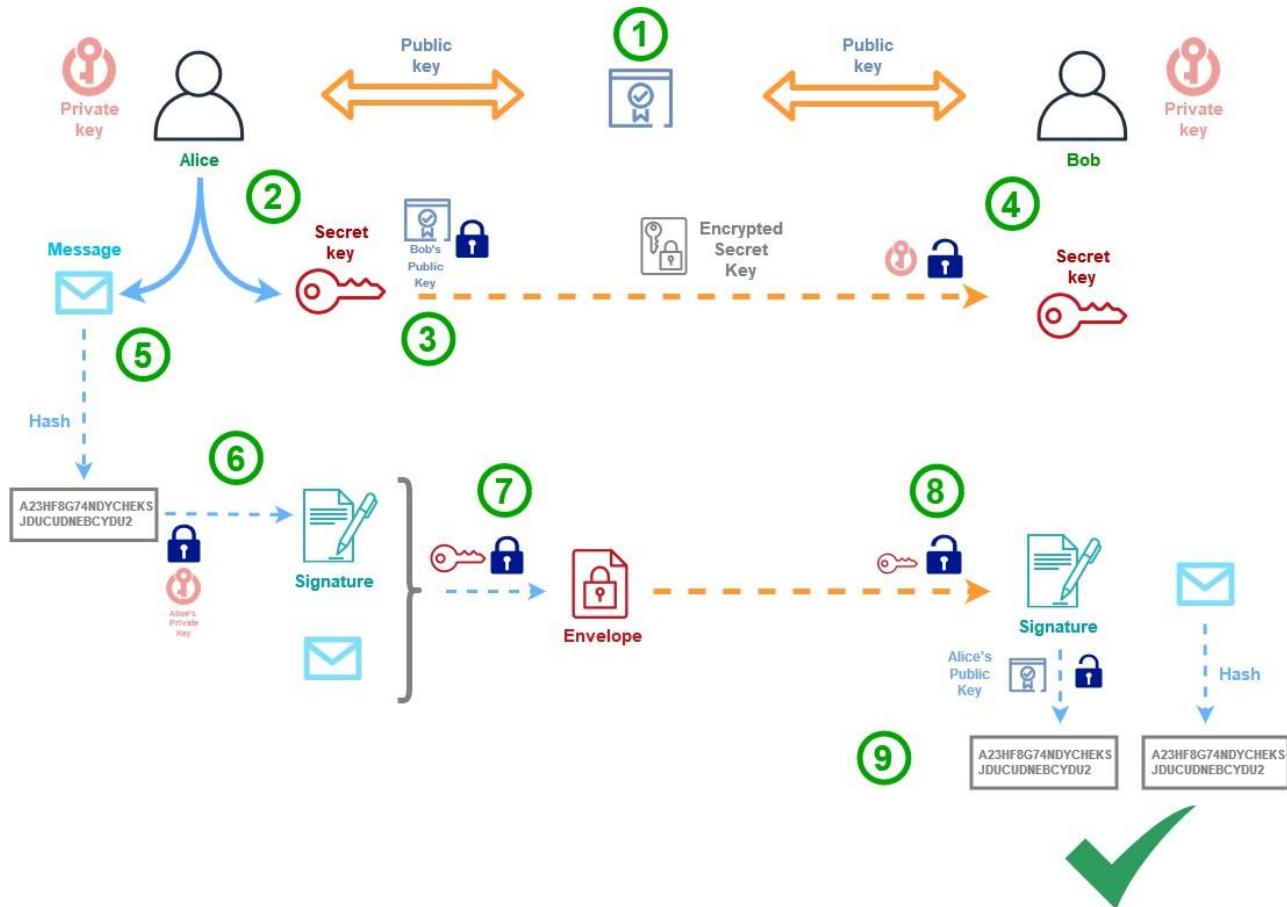


Figura 1.87 Esquema de comunicación con cifrado y firma digital.

En primer lugar, se intercambian las claves públicas o certificados (1). A continuación, Alice genera una clave se sesión (2) y la cifra usando la clave pública de Bob para intercambiarla (3). De esta forma, sólo Bob tendrá acceso a la clave de sesión (4).

Ahora, cuando Alice le quiera mandar un mensaje a Bob podrá calcular el *hash* del mensaje (5) y firmarlo usando su propia clave privada (6). Usando la clave de sesión, Alice cifrará el conjunto firma + mensaje (7) y le enviará el resultado a Bob, quien podrá descifrarlo usando la clave previamente intercambiada (8).

Por último, Bob podrá calcular el *hash* del mensaje recibido y, tras descifrar la firma usando la clave pública de Alice, comparará ambos resúmenes para asegurar la integridad del mensaje.

De esta forma se estará garantizando **integridad, autenticidad y no repudio** del mensaje, de una sola vez.

Existen numerosos algoritmos de firma, aunque el más extendido está basado en el esquema RSA que emplea métodos de factorización de enteros para el intercambio de claves.

1.15 Funciones MAC

Con el objetivo de garantizar la integridad y la autenticidad del mensaje, existen hoy en día numerosas funciones para generar MACs. Muchas de ellas están basadas en el uso de funciones *hash* (como HMAC), mientras que otras emplean algoritmos de cifrado en bloque (como OMAC, CBC-MAC y PMAC). Existe otro tipo de funciones MAC basadas en el concepto de *universal hashing* (como UMAC y VMAC), que básicamente consiste en emplear funciones *hash* de forma aleatoria dentro de un conjunto de funciones predefinido. De esta forma se garantiza la disminución de la probabilidad de colisión.

1.15.1 HMAC

Se trata de un tipo de función MAC que involucra una función *hash*, generalmente SHA-256 o SHA-3. Dependiendo de la función *hash* empleada, HMAC recibe el nombre de HMAC-X, donde X es el nombre del *hash* empleado (por ejemplo, HMAC-SHA256).

En el estándar [RFC 2104](#) se define HMAC como:

$$\text{HMAC}(K, m) = \text{H} \left((K' \oplus \text{opad}) \parallel \text{H} \left((K' \oplus \text{ipad}) \parallel m \right) \right)$$
$$K' = \begin{cases} \text{H}(K) & K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$$

Figura 1.88 Definición de la función HMAC.

Donde *H* es la función *hash*, *m* es el mensaje que queremos autenticar, *K* es la clave secreta, *K'* es una clave de bloque derivada de *K*, \parallel denota concatenación, \oplus denota XOR y *opad* e *ipad* se calculan como:

$$\text{opad} = \text{key} \oplus [0x5c * \text{blockSize}]$$

$$\text{ipad} = \text{key} \oplus [0x36 * \text{blockSize}]$$

SEGURIDAD

Las funciones HMAC son tan seguras como la clave elegida, puesto que el ataque más habitual es aplicar fuerza bruta para obtener la clave secreta, puesto que es más difícil encontrar colisiones en HMAC que en las funciones *hash* por sí solas.

En general, usando una función *hash* segura junto a claves de una longitud aceptable obtendremos un alto grado de seguridad, por lo que HMAC es uno de los métodos MAC más empleados. De hecho, está incluido en el estándar FIPS 140-2 siempre y cuando se use de la mano de SHA1 o SHA2.

1.15.2 CBC-MAC

Cipher block chaining message authentication code (CBC-MAC) es un método de generación de códigos MAC basado en el empleo de algoritmos de cifrado en bloque usando el modo de operación CBC para bloques interdependientes, lo que garantiza que cualquier modificación en un bloque intermedio provocará un cambio impredecible en la salida.

Para generar el *tag*, se cifra en modo CBC usando un vector de inicialización nulo (con valor cero). El *tag* final será la concatenación de los valores obtenidos en cada bloque. En la siguiente imagen se presenta el esquema de funcionamiento, donde m_i representa un fragmento del mensaje, k es la clave secreta y E representa el algoritmo de cifrado simétrico.

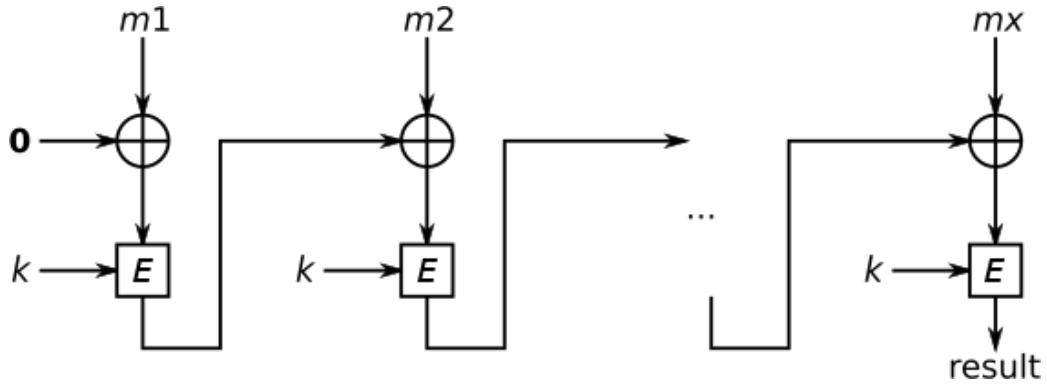


Figura 1.89 Esquema de funcionamiento de CBC-MAC.

SEGURIDAD

Este tipo de función MAC sólo es segura cuando se emplean mensajes de una longitud fija. Esto es porque si un atacante conoce dos pares mensaje-*tag* (m, t) y (m', t') , entonces será capaz de generar un tercer mensaje m'' tal que el *tag* correspondiente a este mensaje sea t' (el mismo que en el segundo par). Para ello sólo tendrá que generar el mensaje como $m'' = m \parallel [(m' \text{ XOR } t) \parallel m' \parallel \dots \parallel m']$.

Sin embargo, si se fija una longitud para el mensaje, un atacante no podrá emplear el mensaje extendido para generar el mismo *tag*.

Por otro lado, existen ciertas modificaciones que garantizan la seguridad de CBC-MAC para mensajes de longitud variable, aunque siempre es más recomendable usar algún otro algoritmo de generación de MACs.

1.15.3 OMAC/CMAC

One-Key MAC es un tipo de esquema MAC que emplea algoritmos de cifrado en bloque. Aunque oficialmente hay dos algoritmos OMAC, ambos son muy similares y, de hecho, OMAC1 es también conocido como **CMAC**, que se convirtió en un estándar recomendado por el NIST en 2005.

En concreto, OMAC soluciona el problema de concatenación de mensajes que generaba CBC-MAC para longitudes variables.

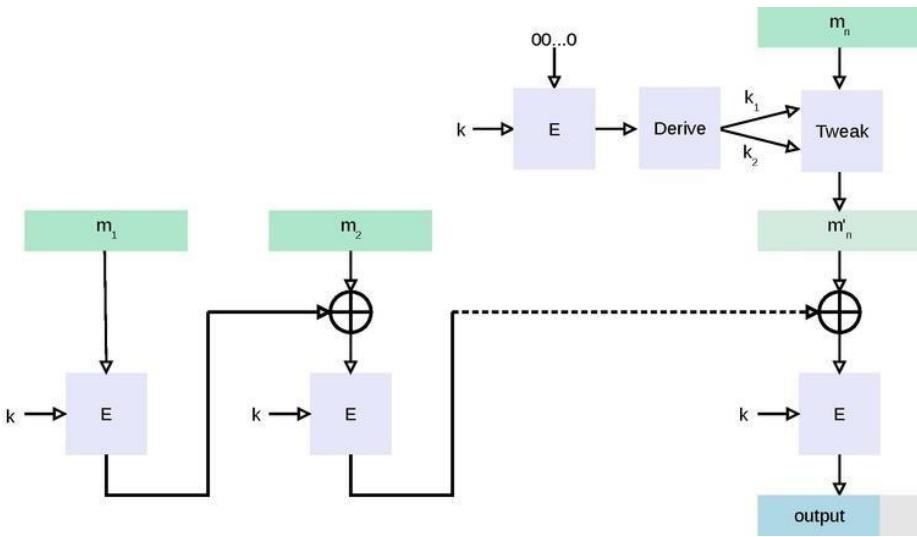


Figura 1.90 Esquema de funcionamiento de la función CMAC.

1.15.4 GMAC

Ya hemos hablado del modo de operación GCM (Galois/Counter Mode) para algoritmos de cifrado en bloque, que actúa como AEAD. GMAC (Galois Message Authentication Code) nace como una variante de GCM en la que únicamente se tiene en cuenta la función de autenticación (no se cifra el contenido), tal y como muestra el siguiente esquema.

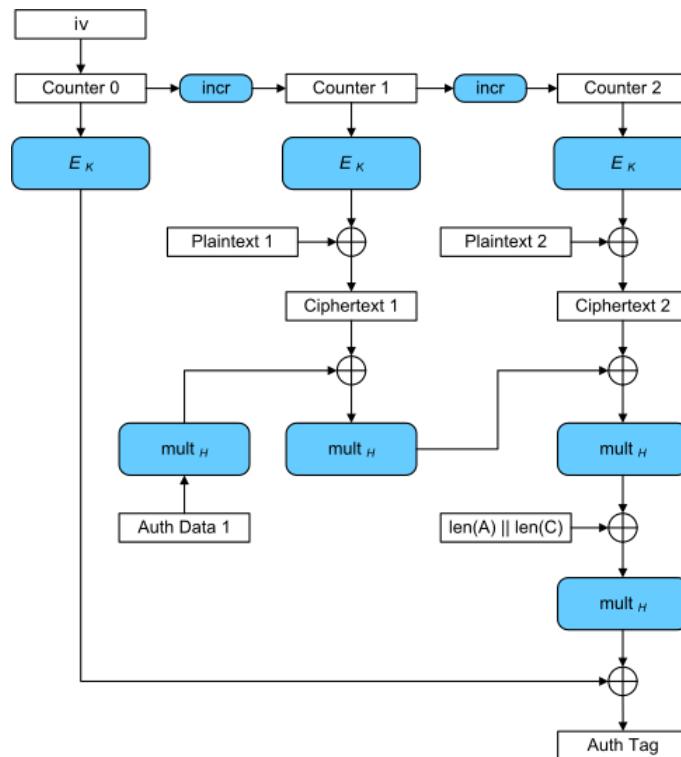


Figura 1.91 Esquema de funcionamiento de GMAC.

GMAC y GCM han sido dos de los métodos de autenticación más empleados, llegando incluso a formar parte del estándar SSH y TLS 1.2 y de herramientas como OpenVPN y SoftEtherVPN. Hoy en día se consideran seguros, aunque se prefiere utilizar otros métodos de autenticación, como Poly1305, ya que GCM y GMAC presentan serias dificultades a la hora de ser implementados de forma eficiente evitando vulnerabilidades ante ataques *side channel*.

1.15.5 Poly1305

Este método de generación de MACs fue ideado por Daniel J. Bernstein y actualmente es uno de los más utilizados. Aunque en sus inicios fue diseñado como una extensión de AES, actualmente la forma más común de encontrarlo es de la mano de ChaCha. De hecho, el estándar [RFC 8446](#) reconoce esta última combinación como el estándar a implementar en la versión 1.3 de TLS/SSL (junto con AES-256_GCM). Además, su reconocimiento como estándar por parte de Google se tradujo en un aumento de las librerías que lo incorporan. Actualmente Poly1305 tiene soporte en Libgcrypt, OpenSSL, OpenSSH, Crypto++...

ALGORITMO

Sea $m = m[0], \dots, m[l-1]$ un mensaje, con l entero no negativo y $q = \lceil \frac{l}{16} \rceil$, definimos los enteros $c_1, \dots, c_q \in \{1, 2, \dots, 2^{129}\}$ como:

$$c_q = m[16q-16] + 2^8 m[16q-15] + \dots + 2^{8(l \bmod 16)-8} m[l-1] + 2^{8(l \bmod 16)}$$

El MAC se genera entonces como:

$$\text{Poly1305}_r(m, \text{AES}_k(n)) = ((c_1 r^q + c_2 r^{q-1} + \dots + c_q r^1) \bmod 2^{130} - 5) + \text{AES}_k(n) \bmod 2^{128}$$

VENTAJAS

La aritmética que emplea Poly1305 (en el campo $\mathbb{Z}/(2^{130} - 5)\mathbb{Z}$) lo hace fácil de implementar de una forma eficiente y rápida evitando *side channels leaks* (como por ejemplo la filtración de información por tiempo de ejecución). Esto lo convierte en la principal opción a la hora de elegir un método de autenticación, frente a otros métodos como GMAC.

1.16 Firma Digital

Ya hemos explicado que las firmas digitales permiten garantizar la integridad y autenticidad del mensaje, aunque por su diseño también garantizan la propiedad de **no repudio**. Actualmente es el método de autenticación más usado, ya que se incluye en la mayoría de protocolos criptográficos (en concreto, el protocolo RSA se basa en un modelo de firma digital). Entre sus usos más habituales se encuentran la firma de documentos digitales, transacciones financieras, distribución de software...

Sin embargo, muchas veces el nombre lleva a confusión: el término “firma digital” hace referencia al algoritmo, mientras que “firma electrónica” se emplea para designar el concepto jurídico de firma, en su equivalente electrónico. Es decir, la firma digital se puede emplear para implementar una firma electrónica, pero no todas las firmas electrónicas cuentan con una implementación mediante este algoritmo.

El objetivo de este apartado es ofrecer al lector una visión más amplia de los algoritmos empleados para elaborar una firma electrónica. Muchos de estos métodos derivan directamente de los algoritmos asimétricos de cifrado que ya hemos visto, especialmente de sus intercambios de clave.

Es bastante común encontrar el caso en el que un algoritmo de firma calcula un *hash* previo. Esto se debe a que, por lo general, las firmas digitales se basan en el cifrado de cierta información accesible por ambas partes. Sin embargo, no se quiere garantizar la confidencialidad del mensaje (para eso está el algoritmo de cifrado), ni se pretende incluir todo el mensaje en una firma. Es por eso que se prefiere calcular un resumen (por ejemplo, usando una función *hash*) y cifrarlo, ya que ambas partes van a tener acceso al mensaje y, poniéndose de acuerdo en qué función resumen emplear, serán capaces de contrastar esta firma.

1.16.1 Firma basada en RSA

Recordamos que, en RSA, Alice y Bob generaban pares de claves pública/privada conforme a un algoritmo que basa su seguridad en la dificultad de resolver el problema de la factorización de enteros de forma eficiente.

Digamos que, dado un mensaje m , las claves de Alice son $((n, e), d)$ (pública y privada, respectivamente). Si ahora Alice quiere mandarle un mensaje privado a Bob, podrá hacerlo de la siguiente manera:

1. Genera un *hash* resumen del mensaje m : $h = \text{hash}(m)$.
2. Cifra h con su clave privada: $h^d \pmod n$.
3. Bob recibe el mensaje y descifra la firma usando la clave pública de Alice: $(h^e)^d \pmod n = h \pmod n$.
4. Por último, Bob calcula su propio *hash* del mensaje m y lo compara con el obtenido de la firma. Si coinciden, se habrá asegurado la integridad y la autenticidad del mensaje y delemisor.

Algunos ejemplos importantes del uso de este esquema de firma los encontramos en los certificados emitidos a Wikipedia o en la versión 1.3 de TLS usada por Google, que firma los certificados con RSA y SHA-256.

1.16.2 Esquema de firma de ElGamal

Este esquema de firma digital está basado en la misma aritmética que el algoritmo de cifrado ElGamal. Consta de cuatro operaciones: generación de la clave, distribución de la misma, firma y verificación.

Generación de clave

Esta fase consta de dos partes: generación de parámetros para el algoritmo y computación de la clave.

Para generar los parámetros, se toma un número primo p de longitud arbitraria (N bits) y el grupo cíclico \mathbb{Z}_p^* . Se elige una función hash H que genere resúmenes de longitud L . Si $L > N$, únicamente se usan los N primeros bits del hash. Por último, tomamos un generador g del grupo, y se comparten los parámetros (p, g) .

Para generar las claves, cada usuario toma un entero x aleatorio entre $\{1 \dots p - 2\}$ y calcula $y := g^x \bmod p$. Llegados a este punto, x sería la clave privada mientras que y sería la clave pública.

Distribución de la clave

El usuario que quiere firmar envía la clave pública y al receptor, manteniendo la clave x .

Firma

Dado un mensaje m :

- Se toma un entero k aleatoriamente entre $\{2 \dots p - 2\}$ con k coprimo a $p - 1$.
- Se calcula $r := g^k \bmod p$.
- Se calcula $s := (H(m) - xr)k^{-1} \bmod (p - 1)$.
- Si $s = 0$, aunque improbable, se empezaría de nuevo con un nuevo k .

La firma es (r, s) .

Verificación

Dada una firma (r, s) y un mensaje m , el receptor puede validar la firma de la siguiente forma:

- Verifica que $0 < r < p$ y $0 < s < p - 1$.
- La firma es válida si y sólo si $g^{H(m)} \equiv y^r r^s$.

1.16.3 Digital Signature Algorithm (DSA)

DSA es un esquema de firma digital basado en el esquema de ElGamal, por lo que también emplea el concepto de exponentiación modular. En 1991, el NIST propuso DSA para ser usado en el estándar DSS (*Digital Signature Standard*) y, en 1994, fue aceptado como tal y adoptado como FIPS 186. Pese a que se trata de un algoritmo patentado, el NIST ha permitido su uso sin cargos en todo el mundo. Además, se encuentra en las principales librerías criptográficas (como Botan, BouncyCastle, OpenSSL...).

Al igual que ElGamal, consta de cuatro fases: generación de claves, distribución, firma y verificación.

Generación de clave

Consta de dos partes: generación de parámetros y generación de claves de usuario.

Para la generación de parámetros, se siguen los siguientes pasos:

- Se elige una función *hash* aprobada, H , con longitud de salida $|H|$. En el estándar original, H era SHA-1, pero actualmente se usa SHA-2. Si $|H| > N$, solo se usan los N primeros bits del *hash*.
- Se elige una longitud de clave L . Actualmente el NIST recomienda longitudes de 2048 bits.
- Se elige un N tal que $N < L$ y $N \leq |H|$. El FIPS 186-4 especifica que L y N tienen que encontrarse en uno de estos pares de valores: (1024, 160), (2048, 224), (2048, 256), o (3072, 256).
- Se toma un primo q de N bits.
- Se toma un primo p de L bits, tal que $p - 1$ es múltiplo de q .
- Se toma un entero h aleatoriamente entre $\{2 \dots p - 2\}$
- Se calcula $g := h^{(p-1)/q} \bmod p$.

Finalmente, los parámetros del algoritmo son (p, q, g) .

Para generar las claves de cada usuario se hace exactamente igual que en ElGamal: se toma un entero x aleatorio entre $\{1 \dots q - 1\}$ y se calcula $y := g^x \bmod p$. Entonces x es la clave privada y y la pública.

Distribución de la clave

Al igual que en ElGamal, se comparte la clave pública y a través de un canal confiable.

Firma

Dado un mensaje m , se firma siguiendo los pasos:

- Elige un entero k aleatorio entre $\{1 \dots q - 1\}$.
- Calcula $r := (g^k \bmod p) \bmod q$.
- Calcula $s := (k^{-1}(H(m) + xr)) \bmod q$.

La firma es, finalmente, (r, s) .

Verificación

Dada la firma (r, s) y el mensaje m :

- Comprobar que $0 < r < q$ y $0 < s < q$.
- Calcular $w := s^{-1} \bmod q$.
- Calcular $u_1 := H(m) \cdot w \bmod q$.

- Calcular $u_2 := r \cdot w \bmod q$.
- Calcular $v := (g^{u_1} y^{u_2} \bmod p) \bmod q$.
- La firma es válida si y sólo si $v = r$.

1.16.4 Elliptic Curve Digital Signature Algorithm (ECDSA)

Se trata de una variante del algoritmo DSA que usa criptografía basada en curvas elípticas. La ventaja principal de este esquema es que, como todos los algoritmos de curvas elípticas, puede trabajar al mismo nivel de seguridad con claves mucho menores. Por ejemplo, donde DSA usaría claves de 1024 bits, ECDSA ofrece la misma seguridad con 160 bits.

Actualmente es uno de los esquemas digital más usados, aunque se tiende a preferir otros esquemas como EdDSA. Se encuentra incluido en una gran cantidad de librerías y protocolos, y su uso está muy extendido en el mundo de las *smartcards*. Uno de sus usos más sonados (y negligentes) fue en el algoritmo de firma digital de la *PlayStation3*. Sin embargo, al utilizar un generador de *nonces* predecible, fue posible obtener información sobre las claves y falsificar firmas para este dispositivo.

Elección de parámetros

Antes de comenzar la firma, ambas partes deberán ponerse de acuerdo en los parámetros a usar. Se debe especificar una *curva elíptica*, definida por un cuerpo y una ecuación, un punto base G que viva en la curva elegida y un entero n que representa el orden del punto G (que debe ser primo).

Algoritmo de firma

En primer lugar, Alice, la emisora, crea un par de claves pública-privada, Q_A y d_A (aleatorio entre $\{1, n-1\}$), respectivamente, donde $Q_A := d_A \cdot G$. Aquí el operador \cdot representa el *producto escalar de un punto en curvas elípticas*, tal y como lo definimos en su momento para ECDH. Todo punto de la curva se representará usando b bits (siendo b la longitud necesaria en bits para poder representar cualquier punto).

A continuación, para firmar el mensaje m :

- Se calcula $e = \text{Hash}(m)$ donde la función *hash* empleada suele ser SHA-2.
- Sea z los b primeros bits de e .
- Se toma un entero k de forma criptográficamente segura, entre $\{1 \dots n-1\}$. Este número cumplirá la función del *nonce*.
- Se calcula el punto de la curva $(x_1, y_1) = k \cdot G$.
- Se calcula $r = x_1 \bmod n$.
- Se calcula $s = k^{-1}(z + r d_A) \bmod n$.
- La firma es el par (r, s) .

Nótese que es necesario, no sólo mantener k en secreto, sino utilizar uno distinto para cada firma, con el objetivo de evitar que un atacante pueda deducir información al respecto.

Verificación de la firma

Inicialmente, el receptor debe verificar que la clave pública recibida Q_A es válida:

- El punto Q_A no debe coincidir con el elemento identidad del grupo.
- El punto debe residir en la curva y cumplir su ecuación.
- $n \cdot Q_A = 0$

A continuación, se siguen los pasos:

- Se verifica que r y s son enteros entre 1 y $n - 1$.
- Se calcula $e = \text{Hash}(m)$ manteniendo la misma función *hash* que antes.
- Sea el mismo z que antes:
- Se calcula $u_1 = z s^{-1} \bmod n$ y $u_2 = r s^{-1} \bmod n$.
- Se calcula el punto de la curva $(x_1, y_1) = u_1 \cdot G + u_2 \cdot Q_A$.
- La firma es válida si y sólo si $r \equiv x \bmod n$.

1.16.5 EdDSA

Se trata de uno de los esquemas de firma digital con más adeptos debido a que está basado en la aritmética de curvas elípticas, pero usa una familia de curvas con características óptimas para convertirlo en un algoritmo muy rápido sin sacrificar seguridad. Esta familia se denomina Curvas de Edwards (de ahí el nombre, Edwards-curve DSA). En concreto, el esquema Ed25519 usa SHA-512 y la Curva25519, siendo este el más conocido.

Parámetros

Para establecer un esquema EdDSA, es necesario fijar un primo q que caracterice a un grupo finito \mathbb{F}_q , una curva elíptica E sobre el campo definido por q , donde el grupo de puntos que viven en la curva tiene orden 2^l , donde l es un primo grande.

Además es necesario tomar un punto base B de orden l (es decir, que $lB \equiv 0$) y una función *hash* H con una salida de $2b$ bits donde $2^{b-1} > q$, de modo que todo punto de la curva se pueda representar con b bits.

Generación de claves

La clave privada usada en EdDSA es una cadena de b bits, denominada k , que debe ser elegida de forma uniformemente aleatoria. Una vez elegida k , la clave pública se calcula como $A = sB$ donde $s = H_{0 \dots 2b-1}(k)$, es decir, s está formado por los b bits menos significativos del *hash*, interpretado en Little-Endian. Por lo tanto, la clave pública A es un punto de la curva, codificado usando b bits.

Firma y verificación

Dado un mensaje m , se puede firmar tomando un punto de la curva $R = rB$, siendo $r = H(H_{b \dots 2b-1}(k), m)$. Por otro lado, definimos $S \equiv r + H(R, A, M)s \pmod l$. La firma es, por tanto, el par formado por (R, S) , codificado con $2b$ bits.

Para validar la firma, tiene que verificarse la ecuación $2^b SB = 2^b R + 2^b H(R, A, M)A$.

Ed25519

Este esquema, como ya mencionamos, utiliza la famosa Curva25519 de Bernstein, y SHA-512 como función *hash*. Los parámetros que caracterizan a este esquema son:

- $q = 2^{255} - 19$
- La curva tomada se denomina *Twisted Edwards Curve* y viene dada por la ecuación $-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$
- El punto B es el único punto en la curva con $y = 4/5$ y x positivo.
- La función *hash* H es SHA-512 con $b = 256$.

Entre las principales ventajas del uso de este esquema se encuentra la resistencia frente a *side channel attacks*. Además, al contrario que ECDSA, el *nonce* se elige de forma determinista, usando el *hash* de un fragmento de la clave privada, evitando así problemas como el ocurrido con la *PlayStation3*.

1.17 Infraestructura de Clave Pública

Una Infraestructura de Clave Pública (más conocida por sus siglas en inglés, *PKI*), es un conjunto de políticas, roles, software y procedimientos necesarios para crear, manejar, intercambiar, almacenar... certificados de clave pública. Dicha infraestructura asegura el intercambio de mensajes cifrados de forma confidencial, la autenticación de usuarios, el no repudio y la integridad de los mensajes mediante el uso de criptografía asimétrica (generalmente, híbrida).

1.17.1 Certificado Digital

Un Certificado Digital es un tipo de documento que asocia una clave pública a una identidad (por ejemplo, una persona física o una empresa) que posee la clave privada correspondiente. La fiabilidad de este documento se basa en la confianza que depositamos en la entidad que lo otorga, denominada **Autoridad de Certificación** (o CA por sus siglas en inglés). Para garantizar que un certificado ha sido expedido por una CA en concreto, el documento irá firmado digitalmente.

1.17.2 Estándar para PKIs

Actualmente el principal estándar para las Infraestructuras de Clave Pública es el **X.509**. Especifica tanto formatos para la certificación de claves públicas, como el algoritmo de validación de la ruta de certificación.

ESTRUCTURA

La estructura de un certificado digital estipulada por el X.509 es:

- Número de serie del certificado.
- Versión.
- ID del algoritmo usado para firmar.
- Emisor del certificado(CA).
- Fecha de expedición del certificado (o, en su defecto, de inicio de su validez).
- Fecha de expiración del certificado.
- Sujeto a quien la entidad certifica. Puede ser una persona, un servidor o un servicio.
- Clave pública del sujeto certificado.
- Algoritmo de clave pública empleado.
- Algoritmo usado para firmar el certificado.
- Firma de la CA para garantizar la autenticidad del documento.

Los archivos que contienen el certificado pueden tener diversas extensiones. Las más habituales son **.CER**, **.DER**, **.PEM**, **.P7C**(si usa la estructura definida en PKCS#7), **.P12**(si usa la estructura definida en PKCS#12).

```

Certificate:
Data:
    Version: 1 (0x0)
    Serial Number: 7829 (0x1e95)
    Signature Algorithm: md5WithRSAEncryption
    Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte Consulting cc,
             OU=Certification Services Division,
             CN=Thawte Server CA/Email=server-certs@thawte.com
    Validity
        Not Before: Jul  9 16:04:02 1998 GMT
        Not After : Jul  9 16:04:02 1999 GMT
    Subject: C=US, ST=Maryland, L=Pasadena, O=Brent Baccala,
             OU=FreeSoft, CN=www.freesoft.org/Email=baccala@freesoft.org
    Subject Public Key Info:
        Public Key Algorithm: rsaEncryption
        RSA Public Key: (1024 bit)
            Modulus (1024 bit):
                00:b4:31:98:0a:c4:bc:62:c1:88:aa:dc:b0:c8:bb:
                33:35:19:d5:0c:64:b9:3d:41:b2:96:fc:f3:31:e1:
                66:36:d0:8e:56:12:44:ba:75:eb:e8:1c:9c:5b:66:
                70:33:52:14:c9:ec:4f:91:51:70:39:de:53:85:17:
                16:94:6e:ee:f4:d5:6f:d5:ca:b3:47:5e:1b:0c:7b:
                c5:cc:2b:6b:c1:90:c3:16:31:0d:bf:7a:c7:47:77:
                8f:a0:21:c7:4c:d0:16:65:00:c1:0f:d7:b8:80:e3:
                d2:75:6b:c1:ea:9e:5c:5c:ea:7d:c1:a1:10:bc:b8:
                e8:35:1c:9e:27:52:7e:41:8f
            Exponent: 65537 (0x10001)
    Signature Algorithm: md5WithRSAEncryption
                93:5f:8f:5f:c5:af:bf:0a:ab:a5:6d:fb:24:5f:b6:59:5d:9d:
                92:2e:4a:1b:8b:ac:7d:99:17:5d:cd:19:f6:ad:ef:63:2f:92:
                ab:2f:4b:cf:0a:13:90:ee:2c:0e:43:03:be:f6:ea:8e:9c:67:
                d0:a2:40:03:f7:ef:6a:15:09:79:a9:46:ed:b7:16:1b:41:72:
                0d:19:aa:ad:dd:9a:df:ab:97:50:65:f5:5e:85:a6:ef:19:d1:
                5a:de:9d:ea:63:cd:cb:cc:6d:5d:01:85:b5:6d:c8:f3:d9:f7:
                8f:0e:fc:ba:1f:34:e9:96:6e:6c:cf:f2:ef:9b:bf:de:b5:22:
                68:9f

```

Figura 1.92 Ejemplo de certificado X.509.

1.17.3 Infraestructura

Generalmente, una PKI consta de los siguientes elementos:

- **Autoridad de Certificación (CA):** entidad encargada de almacenar, generar y expedir certificados digitales.
- **Autoridad de Registro (RA):** entidad encargada de verificar la identidad de las entidades que solicitan certificados.
- **Directorio Central:** localización segura donde almacenar e indexar los certificados.
- **Política de Certificados:** establece los procedimientos a seguir en la PKI.
- **Autoridad de Validación (VA):** entidad que verifica la validez de un certificado digital por los mecanismos descritos en el estándar X.509.

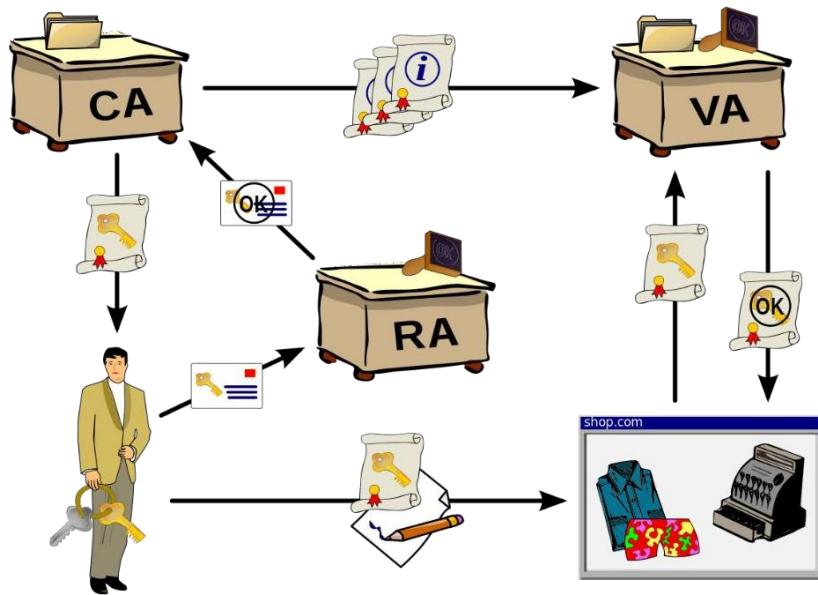


Figura 1.93 Esquema de una PKI.

En la imagen anterior aparece un esquema del funcionamiento de una PKI. En un primer momento, el usuario genera un par de claves pública-privada y envía la pública a una Autoridad de Registro, que verifica la identidad del usuario y valida la solicitud de registro de su clave pública. A continuación, la RA informa a la Autoridad de Certificación de la validez de la clave y esta expedirá un certificado a nombre del usuario. Este certificado, a su vez, se enviará a la Autoridad de Validación.

A partir de ese momento, cuando el usuario quiera realizar una transacción que requiera del uso del certificado, bastará con consultar a la VA para confirmar que el certificado del usuario es válido.

1.17.4 Web of Trust y PGP

Un enfoque alternativo para el problema de la autenticación de la clave pública es el esquema Web of Trust. Se basa en el empleo de certificados auto-firmados que se apoyan en la testificación de un tercero. La ventaja de este enfoque es que no es necesario confiar en una autoridad de certificación suprema (a quien, por cierto, nadie certifica salvo ella misma). La idea es que un usuario puede *firmar* el certificado auto-firmado de otro usuario para dar testimonio de que se puede confiar en él, basándose en un conocimiento externo.

Un criptosistema basado en este concepto es PGP (*Pretty Good Privacy*), que incluye un mecanismo de criptografía híbrido basado en sobres digitales (con una clave simétrica usada para cifrar, que se comparte a través de un canal seguro generado usando criptografía asimétrica). Fue desarrollado por Phil Zimmermann y cuenta con un estándar libre, llamado OpenPGP, que se describe en su versión actual en el [RFC 4880](#). En este estándar se admiten los siguientes algoritmos:

- **Firma digital:** RSA PKCS#1, DSA (con MD5, SHA-1, SHA224, SHA256, SHA384, SHA512), ECDSA (en la versión [RFC 6637](#)).
- **Cifrado simétrico:** IDEA, TripleDES, Blowfish (clave de 128 bits, 16 rondas), AES (con claves de 128, 192 y 256 bits), Twofish (con claves de 256 bits).
- **Cifrado asimétrico:** RSA, ElGamal, Camellia (en la versión [RFC 5581](#)), ECDH (en la versión RFC 6637).

1.17.5 GNU Privacy Guard

PGP se creó como un sistema privado, pese a que cuenta con un estándar público. Es por eso que en 1997 se lanzó GNU Privacy Guard (GPG), que básicamente reemplaza a PGP pero bajo la licencia GPL (se basa en el estándar OpenPGP). De hecho, sólo usa algoritmos no patentados, como ElGamal, TripleDES, AES, Blowfish...

Pese a que se le encontró una vulnerabilidad crítica en 2004, fue corregida y actualmente se encuentra integrado en numerosos servicios y herramientas como *Kmail*, *Evolution* y cuenta con un plugin para *Mozilla* y *Thunderbird* denominado *Enigmail*.

1.17.6 Blockchain PKI

Actualmente está emergiendo una corriente que busca incorporar la tecnología blockchain en las PKIs. El hecho de que blockchain cuente con un método distribuido e inalterable de compartir información, lo convierte en la tecnología perfecta para almacenar e intercambiar claves públicas. Algunos ejemplos de PKIs basadas en blockchain son *CertCoin* o *FlyClient*.

1.17.7 Let's Encrypt

Un problema que presentaba el esquema PKI es que, en su uso más extendido, se basa en la existencia de autoridades de certificación. Esto implica que hay empresas encargadas de gestionar los certificados, cobrando por sus servicios. Una de las principales empresas que actúan como CA es *Symantec*.

Sin embargo, a partir de 2016 se incorporó una nueva empresa, *Let's Encrypt*, que actúa como una autoridad de certificación sin ánimo de lucro, emitiendo certificados confiables para su uso en TLS sin necesidad de pagar nada. Los certificados emitidos por esta empresa tienen una duración de 90 días, pero pueden renovarse en cualquier momento. Se estima que en 2018 expedieron alrededor de 380 millones de certificados.

Otro ejemplo de empresa sin ánimo de lucro que emite certificados al público es *CACert* la cual, además, admite (y promueve) el uso de redes de confianza (web of trust).

2 Criptosistemas Reales

Una vez presentados los principios que conforman la teoría criptográfica, nos gustaría ofrecer al lector un punto de vista más práctico y aplicable al día a día. Debido a la inmensa cantidad de algoritmos, protocolos y dispositivos que existen, no resulta nada sencillo ofrecer una aproximación completa a la realidad criptográfica del mundo. Sin embargo, vamos a intentar hacerlo lo mejor posible y pasar por encima de algunos de los sistemas más conocidos y utilizados.

En esta sección vamos a explicar los fundamentos de protocolos como WiFi, *Bluetooth* o 3G para poder entender los criptosistemas que alberga cada uno.

2.1 Bluetooth

La tecnología *Bluetooth*, desarrollada por *Ericsson Mobile* en los años noventa, supuso en su día un avance considerable en las comunicaciones. Se caracteriza por permitir la creación de redes de pequeño tamaño, denominadas *piconets*, que requieren de la cercanía física de sus participantes.

Como en todo canal de comunicación que se considere seguro, el protocolo *Bluetooth* emplea criptosistemas para establecer el mayor nivel de seguridad posible. Estos criptosistemas, en un comienzo, contaban con algoritmos propios como E1 o SAFER+, aunque en la actualidad incluyen otros más extendidos, como AES o ECDH. Sin embargo, gran parte de su seguridad procede del diseño mismo del protocolo, y no de algoritmos criptográficos.

2.1.1 Funcionamiento

La transmisión de datos se realiza a través de ondas de radiofrecuencia (RF) a través de canales establecidos dentro de un rango de frecuencias. Con el objetivo de evitar interferencias, *Bluetooth* fue diseñado para que se utilizase un canal durante un período de tiempo muy pequeño. Tras ese intervalo, la comunicación continúa en otro canal, produciéndose lo que se conoce como *salto*. En función de la versión de *Bluetooth* utilizada, pueden llegar a producirse hasta 3200 saltos por segundo. Es necesario, entonces, que todos los miembros de una *piconet* establezcan previamente un algoritmo de *salto*, para estar enviando y escuchando todos a la vez en el mismo canal.

Lo que en su día fue una idea para evitar interferencias, se convirtió en un verdadero quebradero de cabeza para los atacantes ya que, además de tener que descifrar la comunicación, tendrían que adivinar la frecuencia en la que iba a transmitirse en todo momento. Sin embargo, en 2017 se publicó una vulnerabilidad denominada *BlueBorne* que permitía a un atacante predecir los *saltos* que se producen en la comunicación.

Para facilitar que los dispositivos establezcan una comunicación, existen dos modos especificados en el protocolo: *discoverable* y *connectable*. Cuando el primero está activado, el dispositivo escucha en un cierto canal (que suele ser fijo y estar determinado por la versión del protocolo) a la espera de llamadas *broadcast*, a las que responde con información necesaria para la conexión. Cuando el segundo modo está activado, el dispositivo se encuentra a la espera de que lleguen solicitudes de conexión a través de un canal específico (que ha compartido previamente en modo *discoverable*). Una vez llega una solicitud, se establecen los parámetros de la conexión (algoritmo de *salto*, entre otras cosas) y se inicia la misma.

2.1.2 Versiones

A lo largo de los años se han sucedido numerosas versiones de este protocolo, llegando incluso a cambiar algunos aspectos fundamentales del mismo, como la velocidad de transmisión o el consumo de energía.

2.1.2.1 Basic Rate/Enhanced Data Rate (BR/EDR)

Las versiones 1.1 y 1.2 únicamente soportaban velocidades de transmisión de hasta 1Mbps, lo que se denomina **Basic Rate**. La **Enhanced Data Rate**, introducida en la versión 2.0, permite velocidades de hasta 3 Mbps. Ambas versiones se diferencian también en el algoritmo de modulación que utilizan para establecer la frecuencia de los saltos.

2.1.2.2 Low Energy

A partir de la versión 4.0 se introdujo el concepto de “*Low-Energy*” *Bluetooth*, ideado para ofrecer una serie de características (como un bajo consumo energético, menor uso de memoria, paquetes más cortos...) con el objetivo de trabajar de forma óptima en dispositivos pequeños y provistos de baterías (como dispositivos médicos o *smartphones*, por ejemplo).

2.1.2.3 Alternative MAC/PHY (AMP)

En la versión 3.0 se introduce el concepto de “*High Speed*” *Bluetooth*, utilizando enlaces 802.11 (un protocolo LAN) para aumentar la velocidad de la conexión.

2.1.3 Seguridad de BR/EDR

Los mecanismos de seguridad incorporados en las primeras versiones de *Bluetooth* eran bastante toscos. Sin embargo, conforme evolucionaba el protocolo, también lo hizo su seguridad. En general, debemos considerar las siguientes fases de la comunicación:

2.1.3.1 Emparejamiento

Antes de iniciar una comunicación *Bluetooth*, los dispositivos deben establecer un canal a través del cual compartir la comunicación. A este proceso se le denomina *emparejamiento*.

Como cabe suponer, el emparejamiento debe ser el primer foco de estudio para asegurar la comunicación, ya que se debe impedir que un dispositivo no deseado o no autenticado se empareje.

a) PIN/Legacy Mode

Las primeras versiones *Bluetooth* empleaban el método conocido como *PIN/Legacy Pairing*, que consistía en hacer que los usuarios de dos dispositivos que querían emparejarse introdujesen el mismo PIN. A continuación, los dispositivos derivarían una clave a partir del PIN con la que cifrar la comunicación de forma simétrica.

En el proceso de derivación de clave se empleaba dos algoritmos de cifrado en bloque, denominados E21 y E22, que están basados en SAFER+ (uno de los candidatos a AES). Sin embargo, como estos algoritmos se encuentran desactualizados y apenas se usan hoy en día, no encontramos relevante detallar su funcionamiento.

Como apreciará el lector, se trata de un método en el que la clave depende del PIN, por lo que un atacante que, por ejemplo, esté mirando la pantalla en el momento de introducir el PIN, será capaz de derivar la clave simétrica y vulnerar la comunicación.

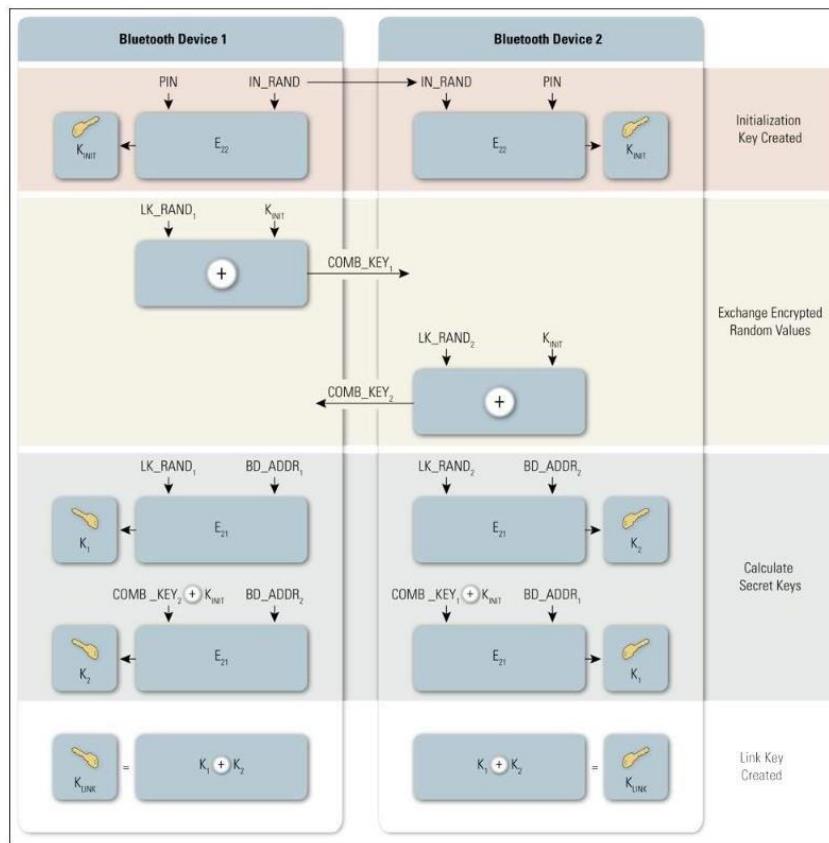


Figura 2.1 Emparejamiento usando PIN/Legacy Mode.

b) Secure Simple Pairing

A partir de la versión 2.1 se introdujo el concepto de *Secure Simple Pairing*, que fue mejorado en la versión 4.1 y, hoy en día, es el método de emparejamiento empleado. Una de sus principales ventajas es que ofrece una serie de modelos de asociación que se amoldan a las capacidades de cada dispositivo. Además, introduce algoritmos criptográficos, generalmente de clave pública, más avanzados, como ECDH, que protegen la comunicación frente a técnicas de *Man-In-The-Middle*.

- **Comparación Numérica:** para situaciones en las que ambos dispositivos pueden mostrar números de seis dígitos y permiten que el usuario responda con un “sí” en caso de que ambos números coincidan y “no” en caso contrario. Este método se diferencia con el PIN en que el número se usa únicamente para verificar el emparejamiento y nunca para derivar la clave. Por lo tanto, es irrelevante si un atacante lo conoce o no.
- **Introducción de código:** cuando un dispositivo permite que un usuario introduzca datos, pero el otro únicamente puede mostrarlos, el segundo muestra por pantalla un código que el usuario debe introducir. Es un caso muy similar al de Comparación Numérica.
- **Just Works:** fue diseñado para el caso en el que al menos uno de los dispositivos carece de todo medio de entrada/salida (por ejemplo, *headsets*). En este caso, el usuario debe aceptar la conexión, pero sin la necesidad de comparar un valor numérico. Este modelo no ofrece seguridad ante MITM.
- **Out Of Band (OOB):** fue diseñado para apoyarse en tecnología *Wireless* como NFC, siendo esta la que realiza el intercambio criptográfico. La seguridad de este método recae en el tipo de tecnología empleado.

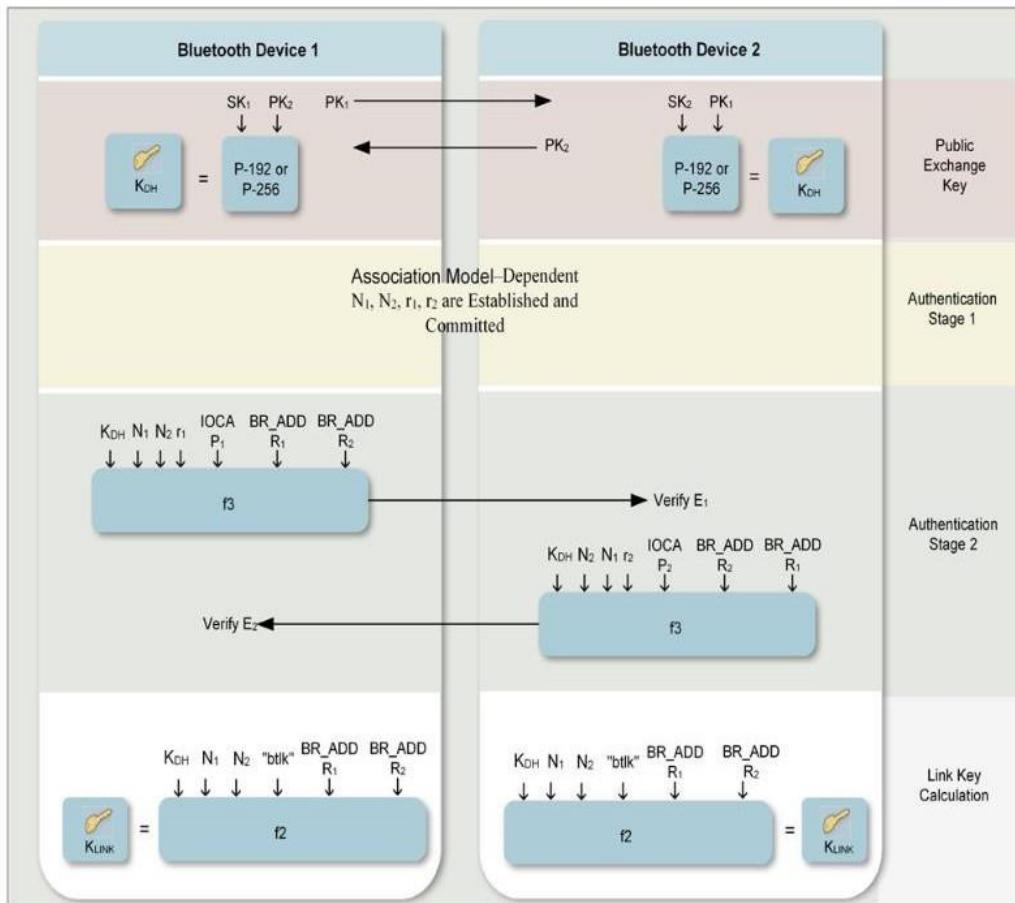


Figura 2.2 Emparejamiento usando SSP.

Este modo de emparejamiento consta de cuatro fases: intercambio de clave pública, dos fases de autenticación y cálculo de la clave de enlace.

En la primera fase, los dispositivos calculan e intercambian sus claves públicas usando ECDH y la curva P-256 (si ambos soportan SSP) o P-291 (si uno de ellos no lo soporta). El avance frente al modo PIN es que ambos dispositivos utilizan criptografía asimétrica en vez de derivar una clave simétrica a través del PIN.

A continuación se produce la primera fase de autenticación, que depende exclusivamente del modelo de asociación empleado. En segunda fase de autenticación, un usuario calcula un valor de confirmación E1 y lo envía para que la otra parte lo valide, tras lo que calculará su propio valor de confirmación E2, que esta vez tendrá que ser validado por el primer usuario.

Finalmente, usando los valores calculados a lo largo de las distintas fases y habiéndose autenticado ambas partes, se genera una clave de enlace simétrica.

c) AMP

El caso de AMP es ligeramente distinto. Esta versión del protocolo calcula la clave de enlace de la misma forma que SSP, pero la termina derivando usando la transformación HMAC-SHA-256.

2.1.3.2 Autenticación

El procedimiento de autenticación es necesario para que, una vez emparejados dos dispositivos, puedan verificar sus identidades sin necesidad de repetir el procedimiento anterior. Para ello deberán demostrar el conocimiento de la clave de enlace. Hay dos tipos de autenticación: *Legacy* y *Segura*. La *Legacy* únicamente se usará cuando al menos uno de los implicados no soporte la *Segura*.

a) Legacy

En este modo de autenticación, los dispositivos adoptan cada uno un rol: verificador y solicitante. El verificador generará un número aleatorio de 128 bits y se lo enviará al otro dispositivo, mientras que el solicitante intercambiará su dirección física. A continuación, usando la clave de enlace obtenida en el emparejamiento, ambos dispositivos generarán un MAC usando la información conjunta mediante el algoritmo E1. El solicitante le envía el resultado cifrado al verificador, quien comprobará que coincide con el que ha calculado.

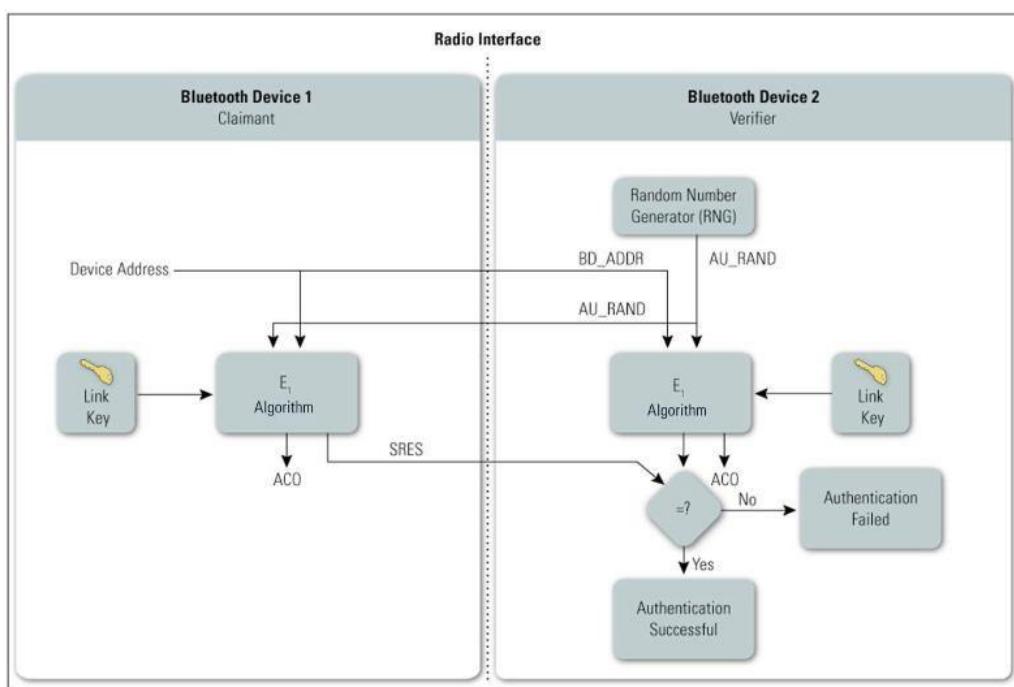


Figura 2.3 Autenticación Legacy.

b) Segura

Este otro modo, estándar actualmente, requiere que los dispositivos adopten ambos los dos roles, y que el emparejamiento se haya producido usando SSP con la curva P-256. Llamaremos maestro al dispositivo que mantiene la red, y esclavo al que se conecta a ella.

Inicialmente, maestro y esclavo intercambian números aleatorios de 128 bits y sus direcciones físicas. Ambos usarán los algoritmos *h4* y *h5* (variaciones de HMAC-SHA-256) para calcular un valor MAC que deberán intercambiar y validar.

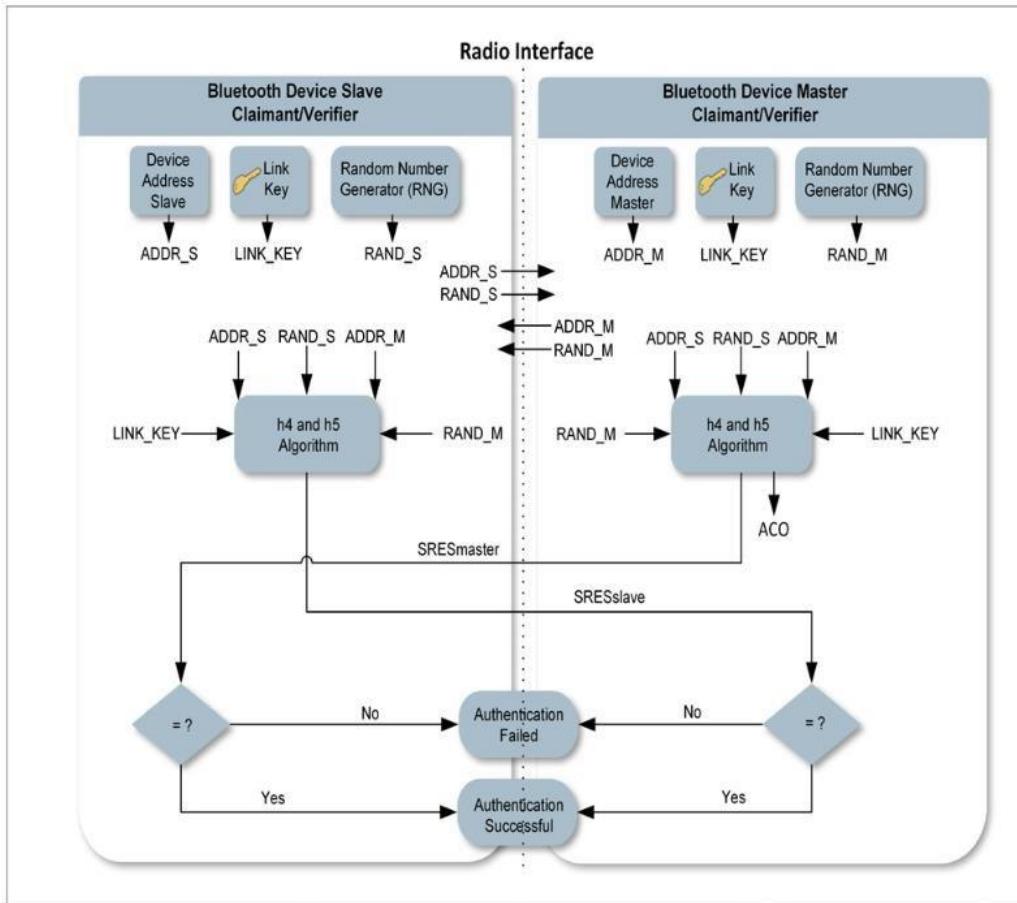


Figura 2.4 Autenticación SSP.

2.1.3.3 Confidencialidad

Para garantizar la confidencialidad, una vez se han autenticado ambas partes y se inicia la comunicación, esta deberá ser cifrada usando un algoritmo seguro. En las primeras versiones de *Bluetooth* se usaba el algoritmo E0, que consistía en generar una secuencia pseudoaleatoria y combinarla con los datos usando XORs, usando longitudes de clave, generalmente, de 128 bits.

Sin embargo, a partir de la versión 4.1, se regulariza el uso de AES-CCM como algoritmo de cifrado.

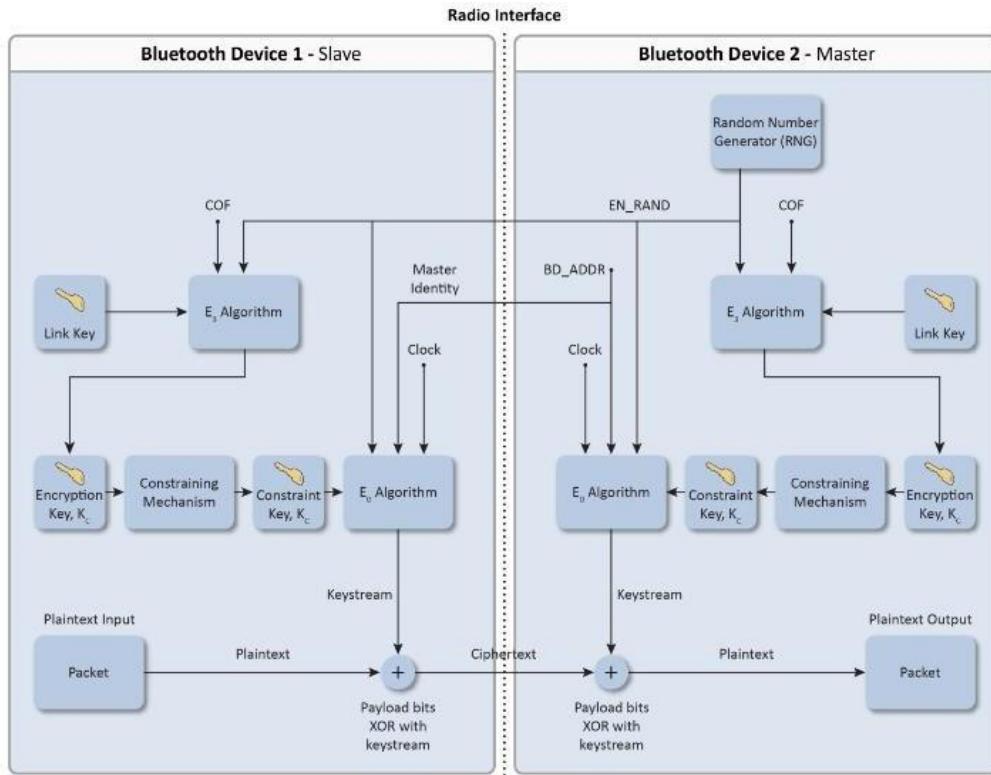


Figura 2.5 Algoritmo E0.

2.1.4 Seguridad de *Low Energy*

Una de las principales diferencias con BR/EDR es que, en vez de emplear una *clave de enlace*, en las versiones *Low Energy* se genera una *Long-Term Key* (LTK), que además se establece de forma distinta: en *Legacy Pairing*, la LTK es generada por una de las partes y posteriormente enviada a través de un canal seguro; en *Secure Connections* la clave se genera en cada dispositivo como resultado de un acuerdo de clave asimétrico.

A partir de la versión 4.0 se introduce, por primera vez en *Bluetooth* el algoritmo AES-CCM, dando lugar a la inicio de la validación FIPS-140 en dispositivos *Bluetooth*. En la versión 4.2 se incluirían las *Low Energy Secure Connections* con una modificación para utilizar algoritmos aprobados por FIPS-140 (AES-CMAC y curvas P-256).

Otra de las características incluidas en estas versiones de *Bluetooth* es el uso de las claves criptográficas conocidas como *Identity Resolving Key* (IRK) y *Connection Signature Resolving Key* (CSRK).

Cuando un dispositivo permanece visible, se le asigna una dirección *pública* de identidad, que podrán usar los demás para conectarse a él. Sin embargo, si esta dirección de identidad fuese constante, sería muy fácil para un atacante localizar al dispositivo incluso cuando este permaneciera invisible. La IRK permite que los dispositivos autenticados establezcan relaciones entre una *Resolvable Private Address* (RPA) y la identidad del dispositivo. Por lo tanto, basta con que se comparta la RPA para establecer conexiones con los dispositivos autenticados, evitando que los atacantes (que no tienen acceso a la IRK) conozcan la dirección de identidad.

La CSRK se usa para verificar los *data frames* del *Attribute Protocol* (ATT) que envía un dispositivo firmados criptográficamente. Esto permite garantizar la integridad y autenticación en una comunicación no cifrada.

2.1.4.1 Emparejamiento

A partir de la versión 4.2, como ya hemos mencionado, se introdujeron algoritmos FIPS-140, por lo que al método de emparejamiento de esta versión y su sucesoras se le denomina *Secure Connection pairing*. Las versiones 4.0 y 4.1 fueron renombradas como *Legacy Pairing*.

Es importante destacar que, aunque las versiones 4.0 y 4.1 de *Low Energy* usen métodos de emparejamiento con nombres similares a los BR/EDR SSP, no emplean criptografía basada en curvas elípticas ni proveen protección contra *eavesdropping*. Por lo tanto, los métodos *Legacy Pairing* de *Low energy* deberán considerarse rotos y obsoletos.

a) Legacy Pairing

En las versiones 4.0 y 4.1, el protocolo de emparejamiento se basa en el intercambio de una clave usando un canal criptográfico. Para ello, los dos dispositivos acuerdan una clave efímera, llamada *Short Term Key* (*STK*) que utilizarán para intercambiar la *Long Term Key* (*LTK*) y las ya mencionadas *IRK* y *CSRK* de forma segura.

b) Secure Connection Pairing

A partir de la 4.2, el protocolo emplea criptografía asimétrica para que ambas partes generen una clave que les permita establecer un canal seguro sobre el que intercambiar información (en concreto, la *LTK* para cifrar el resto de la comunicación).

- **Out of Band:** Si se usa *Out of Band* como método de emparejamiento, será la tecnología OOB empleada la responsable de intercambiar la clave *LTK*. Por lo tanto, es un método tan seguro como lo sea la tecnología.
- **Numeric Comparison:** Se trata de una adaptación del método BR/EDR/HS que tiene el mismo nombre. Se emplea en el caso en que ambos dispositivos pueden mostrar un código de seis dígitos por pantalla y permitir que el usuario introduzca un “SI” o un “NO” como respuesta. De nuevo, el código mostrado por pantalla no se usa para la generación de clave.
- **Passkey Entry:** Si, al menos, un dispositivo permite la entrada de datos por parte del usuario y el otro cuenta con una pantalla en la que mostrar un código, se puede usar este método. En *Legacy Pairing*, la *TK* se genera a partir del código introducido, por lo que si un atacante lo descubre, podrá deducir la clave. Sin embargo, en *Secure Connections*, los dispositivos emplean mecanismos asimétricos para acordar una clave que no derive directamente de este código.
- **Just Works:** Si ninguno de los otros métodos se pueden usar, entonces nos queda Just Works. Como ya explicamos, se trata del método menos seguro de todos, ya que la *TK* es 0x00 y no se provee ningún tipo de autenticación ni protección frente a MITM.

2.1.4.2 Secure Connection Key Generation

Como ya hemos dicho, *Secure Connection* no intercambia la *STK*, sino que genera directamente la *LTK* usando criptografía asimétrica. En concreto, un algoritmo basado ECDH y AES-CMAC.

Tiene sentido que *Secure Connection* no haga uso de la *STK*, ya que esta únicamente se usa para intercambiar la *LTK* de forma segura y, usando ECDH, ambas partes pueden acordar la *LTK* sin necesidad de intercambiarla.

Para generar la clave se usa una función llamada *f5*, que se define como:

$$f5(DHKey, N1, N2, A1, A2) = \text{AES-CMAC}_{\text{AES-CMAC}, \text{hash}(DDHKey)}(0 \parallel \text{ble} \parallel N1 \parallel N2 \parallel A1 \parallel A2 \parallel 256) \\ \parallel \text{AES-CMAC}_{\text{AES-CMAC}, \text{hash}(DDHKey)}(0 \parallel \text{ble} \parallel N1 \parallel N2 \parallel A1 \parallel A2 \parallel 256)$$

Donde AESCMAC_{hash} es la función AES-CMAC usando $hash$ como hash; N1 y N2 son dos números aleatorios generados por cada uno de los dispositivos; A1 y A2 son las direcciones MAC de cada dispositivo.

Llegados a este punto, otras claves como IRK o CSRK pueden intercambiarse usando la LTK acordada. Además, el diseño de este generador de claves garantiza, de forma implícita, la autenticación y la integridad de los datos.

2.1.5 Conclusión

Bluetooth es una tecnología que surgió hace muchos años, pero que hoy en día sigue siendo una de las más empleadas. Por lo tanto, aunque los dispositivos modernos incorporan versiones actualizadas, sigue existiendo una gran cantidad de aparatos que utilizan versiones antiguas y, en muchos casos, vulnerables.

La siguiente tabla muestra los algoritmos utilizados en las diversas fases del protocolo teniendo en cuenta las distintas versiones del mismo.

Characteristic	Bluetooth BR/EDR		Bluetooth Low Energy		
	Prior to 4.1	4.1 onwards	Prior to 4.2	4.2 onwards	
RF Physical Channels	79 channels with 1 MHz channel spacing		40 channels with 2 MHz channel spacing		
Discovery/Connect	Inquiry/Paging		Advertising		
Number of Piconet Slaves	7 (active)/255 (total)		Unlimited		
Device Address Privacy	None		Private device addressing available		
Max Data Rate	1–3 Mbps		1 Mbps via GFSK modulation		
Pairing Algorithm	Prior to 2.1: E21/E22/SAFER+	P-256 Elliptic Curve, HMAC-SHA-256	AES-128	P-256 Elliptic Curve, AES-CMAC	
	2.1-4.0: P-192 Elliptic Curve, HMAC-SHA-256				
Device Authentication Algorithm	E1/SAFER	HMAC-SHA-256	AES-CCM ⁹		
Encryption Algorithm	E0/SAFER+	AES-CCM	AES-CCM		
Typical Range	30 meters		50 meters		
Max Output Power	100 mW (20 dBm)		10 mW (10 dBm) ¹⁰		

Figura 2.6 Algoritmos de cifrado usados por *Bluetooth*.

En general, como todo, *Bluetooth* es teóricamente seguro si se siguen las recomendaciones de seguridad. Sin embargo, abundan las implementaciones que no tienen en cuenta dichas recomendaciones y, por lo tanto, contienen errores que pueden ser explotados de diversas formas.

El NIST publicó un estándar SP 800-121 que recoge los principales errores de seguridad que se comenten a la hora de desarrollar un dispositivo *Bluetooth*, y además incluye información sobre cómo maximizar la seguridad de estos dispositivos.

2.1.6 Ataques conocidos

Existe, por lo tanto, una variedad de ataques que, a lo largo de los años, han vulnerado diversas versiones del protocolo.

- **BlueJacking:** consiste en el envío de mensajes, imágenes o sonidos no autorizados a dispositivos *Bluetooth* cercanos.

No se trata de un ataque muy severo, sino más bien una molestia para los usuarios que se encuentran al alcance del atacante, ya que no se produce ningún acceso ni inyección. Se aprovecha de un protocolo de intercambio de objetos binarios denominado OBEX para enviar un *broadcast* a toda la piconet.

Los dispositivos que se encuentran en modo oculto o invisible no se ven afectados por este ataque, por lo que los fabricantes modernos implementan una solución bastante sencilla, que consiste en mantener al dispositivo siempre oculto y, cuando se quiera establecer una conexión, que cambie a modo visible durante muy poco tiempo.

- **Bluesnarfing:** permite el acceso no autorizado a cierta información de un dispositivo a través de una conexión *Bluetooth*.

De nuevo, una vulnerabilidad en el protocolo OBEX, usando un perfil de intercambio de objetos entre dispositivos denominado *Object Push Profile*, permite al atacante obtener información acerca de los ficheros a los que puede tener acceso usando *Bluetooth*.

Una versión más moderna, *Bluesnarf++*, permite, además, obtener permisos de lectura y escritura sobre dichos ficheros, siendo esta la más peligrosa.

En ambas versiones del ataque, un dispositivo en modo invisible sigue siendo vulnerable, pero obliga al atacante a descubrir su dirección MAC (por ejemplo, usando fuerza bruta). En este caso, el ataque se vuelve poco práctico.

- **Bluesmack:** se trata de un ataque de denegación de servicio (DOS) similar al *Ping-of-Death* que se puede llevar a cabo a través de un *buffer-overflow* mediante mensajes echo *L2CAP*. Como *Proof-of-Concept* se puede usar la herramienta *l2ping* de Linux (similar al *ping*, pero usando el protocolo *L2CAP* de *Bluetooth*).
- **Bluebugging:** este ataque se aprovecha de una vulnerabilidad en el stack del protocolo, específicamente en los comandos AT (usados para la configuración de dispositivos), para obtener control sobre la víctima. Permite al atacante realizar llamadas, acceder a todo tipo de archivos, enviar SMS... Se trata de uno de los ataques más peligrosos que existen para *Bluetooth*.
- **KNOB (Key Negotiation of Bluetooth):** por último, uno de los ataques *Bluetooth* más modernos que conocemos, aunque afecta únicamente a las versiones BR/EDR 5.1 y anteriores. Se basa en interceptar el paquete inicial de emparejamiento (donde se definen los bytes de entropía de la clave) y manipularlo para establecer claves con un único byte de entropía. De esta forma, el atacante puede llevar a cabo un ataque de fuerza bruta sobre la clave de cifrado, siendo esta mucho más fácil de romper.

2.2 Telefonía móvil: 4G/LTE

Las siglas 4G hacen referencia a la cuarta generación de tecnologías para la telefonía móvil. En concreto, cuando hablamos de 4G hablamos de un estándar que, al igual que DES o AES, se abrió a concurso para elegir la tecnología que mejor se adaptara a las necesidades existentes. Actualmente, la arquitectura empleada en este estándar es la conocida como LTE (*Long Term Evolution*).

En general, una red de telefonía móvil consiste en un conjunto de antenas de radio emitiendo en un rango concreto de frecuencias. Cada una de estas antenas pertenece a una entidad (compañía, gobierno...) a la que denominamos *operador*, que es la encargada de proveer servicios. Una de las principales ventajas de esta tecnología es su alcance, que se extiende a una gran parte del territorio mundial.

El estándar LTE permite establecer conexiones IP de alta velocidad en toda la red manteniendo interoperabilidad con tecnologías más antiguas (como 3G o GSM). Se trata, por tanto, de la forma más extendida de acceso a Internet en la actualidad.

2.2.1 Conceptos

Para comprender correctamente el funcionamiento de 4G, debemos introducir una serie de términos que usaremos a lo largo de este capítulo:

Dispositivos móviles

También referenciados como UEs, hacen referencia a todo tipo de terminal que se conecte a la red como cliente (*Smartphones*, dispositivos *IoT*...). Están compuestos por un sistema operativo para la interacción con el usuario, y un subsistema hardware dedicado a telefonía usado para interactuar con la red móvil.

Tarjeta SIM / UICC

El hardware contiene un dispositivo extraíble, denominado *tarjeta SIM*, que contiene información de seguridad crítica para poder acceder a la red, como las claves criptográficas necesarias para autenticar al dispositivo y cifrar la comunicación. Los estándares hacen referencia a esta tarjeta como **Universal Integrated Circuit Card (UICC)**. Esta tarjeta corre una aplicación Java conocida como *Universal Subscriber Identity Module (USIM)*, que actúa como interfaz entre la radio del dispositivo y la red móvil.

IMEI e IMSI

Se trata de dos identificadores usados en las redes de telefonía móvil. El *International Mobile Equipment Identifier (IMEI)* se usa para identificar un dispositivo en la red, y se almacena en la memoria *flash* (aunque también puede encontrarse en la UICC). Por otro lado, el *International Mobile Subscriber Identity (IMSI)* es un identificador a largo plazo para un suscriptor de la red (una forma que tiene el operador de identificar a un cliente).

La diferencia entre estos códigos y una dirección IP es que, mientras que la IP se emplea como identificador para establecer comunicaciones dentro de una red, el IMEI y el IMSI se usan para autenticar a un usuario y comprobar la legitimidad de su acceso a la misma, respectivamente.

GUTI

El *Globally Unique Temporary Identity (GUTI)* es un identificador único temporal que puede usarse para identificar un terminal en la red sin necesidad de mandar su IMSI. El GUTI es un derivado del IMSI, y se prefiere su uso por motivos de seguridad, para que un atacante no tenga acceso al código IMSI y pueda replicar una identidad.

E-UTRAN

Siglas para *Evolved Universal Terrestrial Radio Access Network*. Se trata de una red formada por estaciones base (o nodo) que gestionan las señales de radio para comunicarse con UEs y crear los paquetes IP que mandan a la red central. Los nodos se usan unos a otros como *repetidores* para poder alcanzar la red central. Esto permite que los UEs cuenten con una conexión *móvil*, ya que pueden conectarse a cualquier nodo a su alcance para intercambiar información.

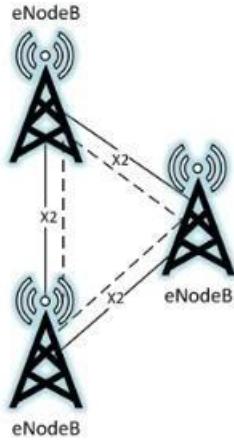


Figura 2.7 Conexión entre nodos en la E-UTRAN.

2.2.2 Evolved Packet Core

El *Evolved Packet Core (EPC)* es el cerebro de la red LTE, encargado de enrutar y procesar la información en la red. Está compuesto por:

- **Mobility Management Entity (MME):** nodo primario que no interactúa con el tráfico del usuario. Es el encargado, entre otras cosas, de gestionar la autenticación y la autorización de dispositivos, de seleccionar la *gateway* adecuada para el registro...
- **Serving Gateway (S-GW):** enruta y envía los paquetes de datos del usuario. Además, actúa como enlace entre LTE y otras tecnologías *Legacy*.
- **Packet Data Network Gateway (P-GW):** actúa como punto de entrada y salida del tráfico para los UEs en el EPC.
- **Backhaul:** conexión entre la red de radio y la red central (fibra, satélite, Ethernet...).
- **IP Multimedia Subsystem (IMS):** puerta de enlace hacia la red pública de teléfono, servicios multimedia (como mensajería instantánea, video...).
- **Packet Data Network (PDN):** cualquier red externa IP (por ejemplo, Internet).
- **Access Point Name (APN):** identificador del PDN.
- **Home Subscriber Server (HSS):** base de datos maestro que almacena datos de los suscriptores y claves criptográficas.

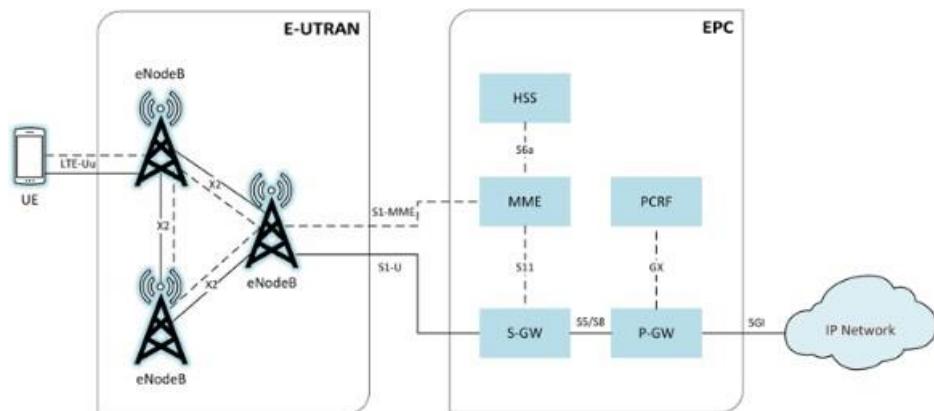


Figura 2.8 Arquitectura de red LTE.

Antes de que un UE pueda unirse a la red LTE y acceder a sus servicios, es necesario que se identifique a la red siguiendo el procedimiento conocido como *Initial Attach Procedure*, que se especifica en la siguiente figura:

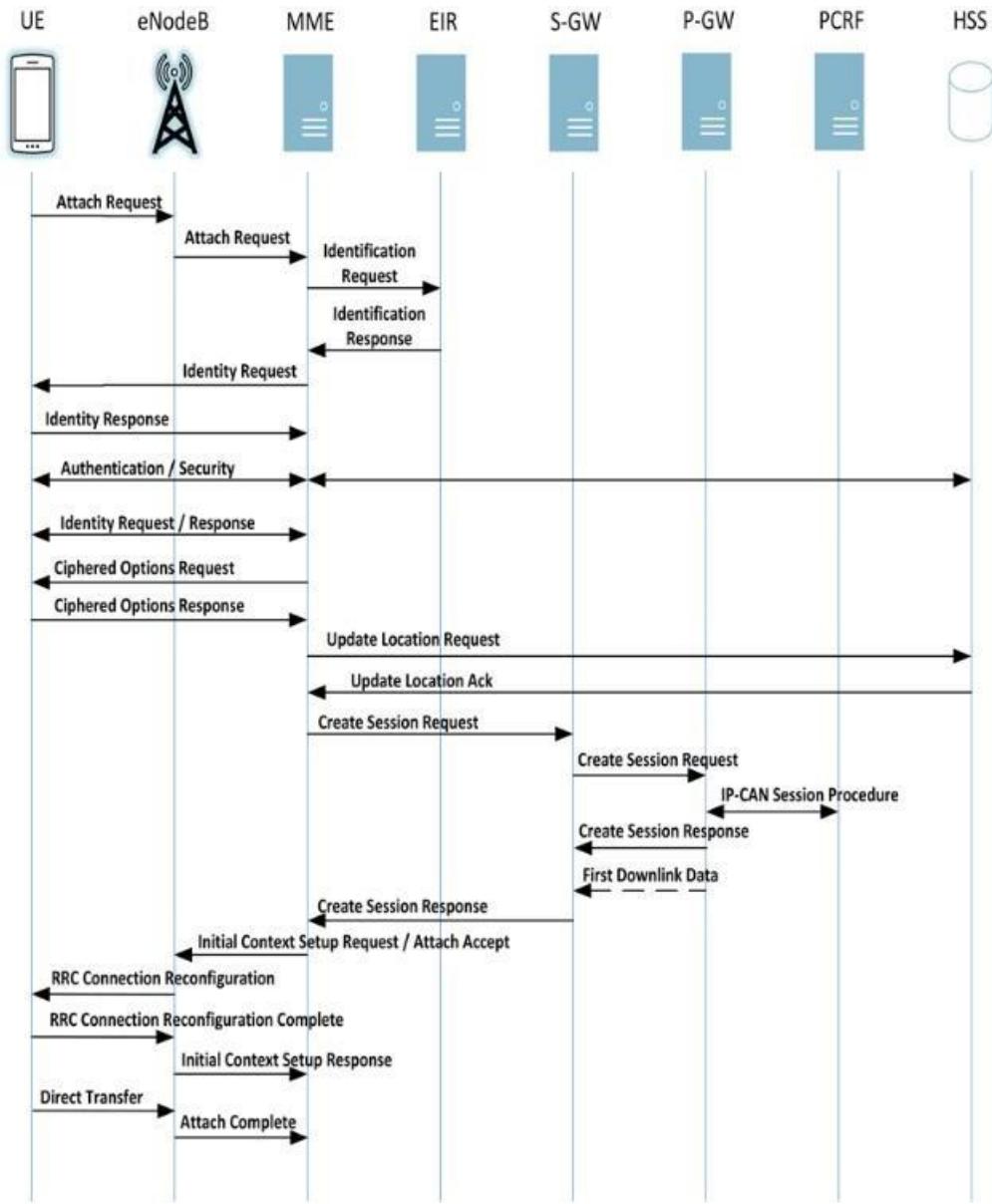


Figura 2.9 LTE Initial Attach

El lector podrá observar que el HSS será el encargado de gestionar la autenticación del UE, ya que es quien almacena las claves criptográficas necesarias.

Una vez se ha identificado el UE en la red LTE, es necesario que se produzca la Autenticación a través del *Key Agreement protocol* (AKA).

2.2.3 Protocolos de red

Para establecer comunicaciones en la red LTE, se usa una serie de protocolos agrupados en la denominada *air interface protocol stack*, los cuales suponen la base para todo el tráfico TCP/IP que opera por encima de estas capas:

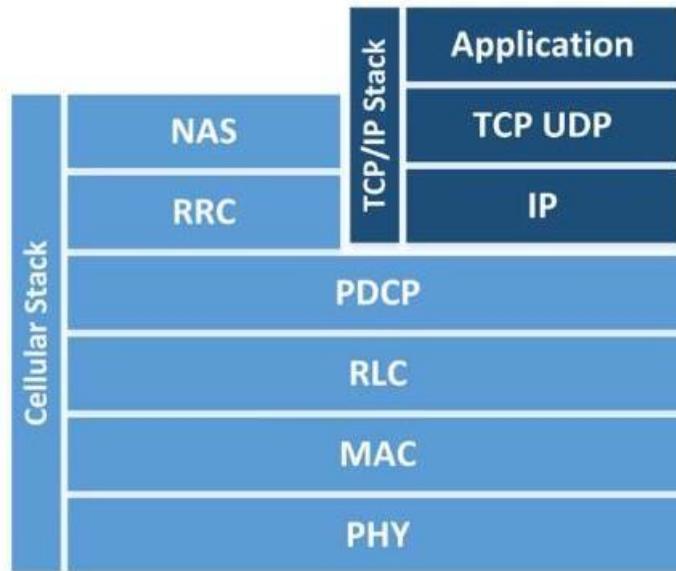


Figura 2.10 Air interface protocol stack

En la imagen anterior, se usa PHY para hacer referencia a *Physical Address*; PDCP para *Packet Data Convergence Protocol*; RLC para *Radio Link Control*; MAC para *Medium Access Control*; RRC para *Radio Resource Control*.

Como no resulta relevante en el ámbito criptográfico, no se va a detallar el objetivo ni el funcionamiento de ninguna de estas capas del *stack*.

2.2.4 Seguridad en LTE

El estándar LTE introdujo una serie de nuevos conjuntos de algoritmos criptográficos y modificó sustancialmente la estructura de claves que venía existiendo desde predecesores como GSM. Los de algoritmos aseguran confidencialidad e integridad y se denominan: *EPS Encryption Algorithm (EEA)* y *EPS Integrity Algorithms (EIA)*.

EEA1 y EIA1 están basados en el cifrado SNOW 3G (que ya se usaba en UMTS), mientras que EEA2 y EIA2 están basados en AES, usando los modos CTR y CMAC, respectivamente. EEA3 y EIA3 están basados en el cifrado chino ZUC.

Sin embargo, pese a que estos algoritmos más seguros y modernos se encuentran presentes en LTE, es común encontrar implementaciones con algoritmos antiguos para asegurar la retrocompatibilidad con redes y dispositivos que no soporten los nuevos.

La siguiente figura recoge la aplicación de las diversas claves en cada uno de los *stacks* de la comunicación LTE:

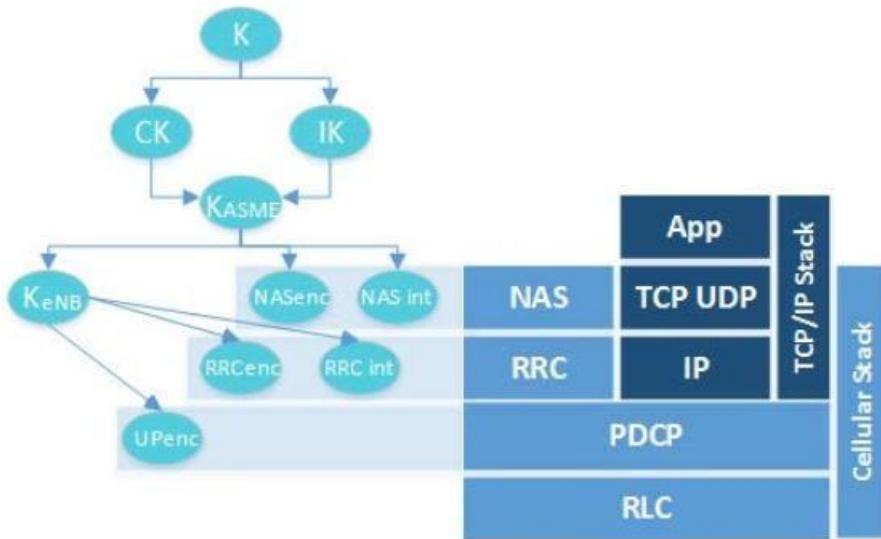


Figura 2.11 Claves en el stack LTE.

Y la siguiente tabla explica cuál es cada clave, su longitud y su procedencia (de qué se deriva esa clave):

Clave	Nombre	Longitud (bits)	Deriva de
K	Master Key	128	N/A: se comparte previamente
IK	Integrity Key	128	K
CK	Cipher Key	128	K
K_{ASME}	MME Base Key	256	CK, IK
NH	Next Hop	256	K_{ASME}
K_{eNB^*}	eNB Handover Key	256	K_{ASME}, K_{eNB}
K_{eNB}	eNB Base Key	256	K_{ASME}, NH
K_{NASint}	NAS Integrity Key	128	K_{ASME}
K_{NASenc}	NAS Confidentiality Key	128	K_{ASME}
RRC_{enc}	RRC Confidentiality Key	128	K_{eNB}, NH
RRC_{int}	RRC Integrity Key	128	K_{eNB}, NH
UP_{enc}	UP Confidentiality Key	128	K_{eNB}, NH

Figura 2.12 Tabla de claves LTE.

Por otro lado, las tarjetas SIM/UICC, almacenan claves criptográficas y credenciales necesarias para la comunicación. En concreto, en LTE las UICCs cuentan con una clave a largo plazo, compartida previamente, denominada K . Dicha clave se almacena, a su vez, en el HSS y nunca debe abandonar ninguna de sus ubicaciones: de ello depende la confidencialidad de las comunicaciones. Las tarjetas SIM modernas incorporan un PIN de activación para otorgar una capa extra de seguridad.

La autenticación de suscriptores y las operaciones criptográficas las realizan la UICC junto con el HSS y la MME.

2.2.4.1 Autenticación del UE

El *Authentication and Key Agreement (AKA) protocol* permite a un UE autenticarse en la red LTE. Para ello, se otorga una evidencia criptográfica de que el UICC y el operador conocen la clave K , siguiendo los siguientes pasos:

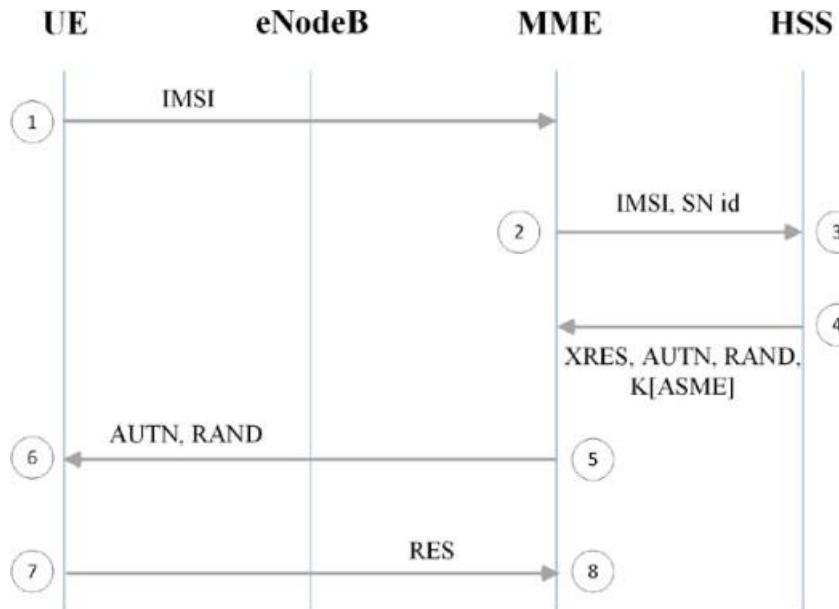


Figura 2.13 Protocolo de Acuerdo de Claves LTE.

- El UE le envía su identificador al MME. Este puede ser permanente (IMSI) o temporal (GUTI/TMSI).
- El MME le manda el identificador junto con algunos parámetros criptográficos y el ID de la red-servidor (SN id) al HSS.
- El HSS usa la información recibida para generar un vector de autenticación (AUTN) a partir de un *nonce* aleatorio, la clave secreta K y un número de secuencia SQN que actúan como parámetros de una función criptográfica. El resultado de dicha función es un par de valores que podrán usarse en la derivación de futuras claves, así como el resultado esperado (XRES) y el token AUTN.
- Se devuelve al MME el vector de autenticación.
- Tras extraer el valor esperado XRES del vector, se envían los datos al UE.
- El terminal usa AUTN, RAND, K y SQN como *inputs* de una función criptográfica similar a la usada por el HSS.
- El resultado de la función se envía al MME.
- Se comprueba que el resultado recibido (RES) es igual que el esperado (XRES).

2.2.4.2 Cifrado

En los estándares publicados para LTE, únicamente se contempla el cifrado entre el terminal y la estación base, y de forma opcional. Por lo tanto, en caso de querer proporcionar confidencialidad a la comunicación, será necesario establecer un cifrado a nivel de la capa de red (usando, por ejemplo, IPSec).

2.2.4.3 Amenazas

Estaciones falsas

Se trata de estaciones sin licencia ni autorización que no están manejadas por auténticos operadores. Emiten una red similar a las que emiten las estaciones legítimas, actuando como un nodo más. Esto hace que, cuando haya UEs cerca, puedan establecer conexiones con esta estación.

La principal amenaza consiste en que, como muchos dispositivos están diseñados para soportar retrocompatibilidad, las estaciones solicitan conexiones a través de protocolos vulnerables, como 2G GSM, explotando vulnerabilidades conocidas.

Privacidad

El uso de identificadores IMSI e IMEI permite establecer una relación entre un dispositivo y una persona física. Por lo tanto, la unicidad de estos códigos permite a atacantes que levanten una estación falsa conocer la ubicación exacta de un individuo que se conecte a su red.

Seguridad de la clave *K*

Como la clave *K* se usa como raíz de la que se derivan todas las demás, es importante mantenerla en secreto en todo momento. Una mala gestión de claves por parte de la UICC o del proveedor, así como el uso de claves criptográficamente inseguras, puede llegar a suponer un riesgo importante.

2.2.5 Conclusión

Las redes LTE no están diseñadas para ser criptográficamente robustas, sino que aportan un canal de comunicación sobre el que se espera que viaje información cifrada por encima de la capa de red. Es, por tanto, muy importante que se establezca un protocolo como IPsec para asegurar la confidencialidad de la comunicación.

La autenticación e integridad, por otro lado, sí que parecen estar aseguradas en este tipo de redes, aunque siempre es recomendable que toda aplicación establezca sus propios controles.

El NIST recomienda el uso de funciones criptográficas en todo momento en las comunicaciones LTE, aunque no deja de ser un requisito de seguridad opcional.

2.3 WLAN: WEP y WPA/WPA2

Una de las formas más comunes de acceso a Internet hoy en día es a través de redes inalámbricas locales (WLAN) que cuentan con una puerta de acceso (*gateway*) hacia la red. Este tipo de redes se implementan comúnmente usando la tecnología *WiFi* a través de un *router*.

Sin embargo, las redes WLAN no sólo conectan un terminal a Internet, sino que permiten la interacción directa entre terminales que se encuentran dentro de una misma red local. En concreto, es posible monitorizar todo el tráfico de una red local de forma relativamente sencilla haciendo, por ejemplo, un ataque MITM. Por lo tanto, resulta totalmente necesario establecer un protocolo que garantice la confidencialidad dentro de la red, así como la autenticación y la integridad.

Una primera solución, recogida en el estándar 802.11a en el año 1997, pasaba por usar el algoritmo criptográfico **WEP** (*Wired Equivalent Privacy*), que buscaba otorgar un grado de confidencialidad semejante al que existe en redes cableadas.

2.3.1 WEP

Con el objetivo de garantizar la confidencialidad y la integridad, WEP utilizaba los algoritmos RC4 y CRC-32, respectivamente.

Pese a que en un primer momento este algoritmo usaba claves de 64 bits (debido a las restricciones de exportación de tecnología criptográfica de los Estados Unidos), se acabó normalizando su uso con claves de 128 bits.

El algoritmo consiste en concatenar un IV de 24 bits con una clave de 40/104 bits (que se introducía en formato hexadecimal) y utilizar la cadena resultante como clave para RC4, obteniendo así un *keystream*. Mientras tanto, se calcula el ICV (*Integrity Check Value*) que garantiza la integridad del mensaje, y se concatena al final del mismo. Finalmente, basta con realizar un XOR entre el *keystream* y el mensaje junto con el ICV para obtener el texto cifrado.

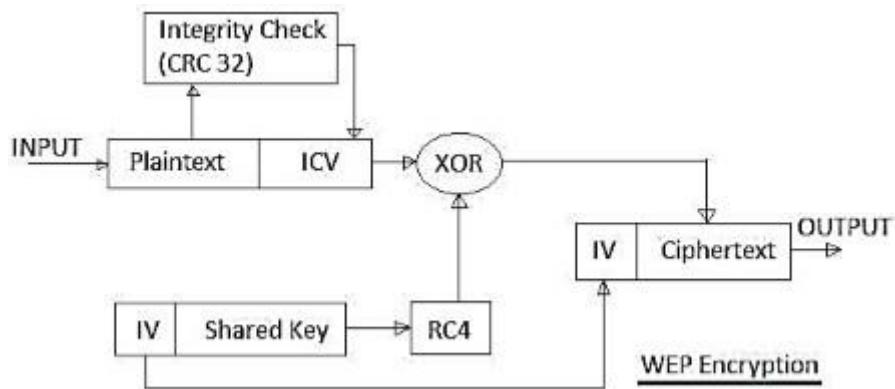


Figura 2.14 Esquema del protocolo WEP.

El protocolo definido en el IEEE 802.11a ofrecía dos métodos de autenticación sobre el cifrado WEP: *Open System authentication* y *Shared Key authentication*.

En el primero, el cliente no se autentica con el punto de acceso, sino que directamente se empareja y comienza a enviar datos cifrados usando la clave WEP.

En el segundo método, cuando un cliente solicita una autenticación al punto de acceso, este le envía un *challenge* en texto plano que deberá cifrar usando la clave WEP (que, por lo general, la introduce el usuario tras mirarla detrás del router). Esto permite al punto de acceso saber que el cliente, en efecto, pretende conectarse de forma legítima.

2.3.1.1 Vulnerabilidades

Aunque *a priori* parezca más seguro utilizar *Shared Key Authentication*, realmente este método presenta una vulnerabilidad crítica: el hecho de que el *challenge* se envíe en texto plano al cliente permite que un tercero lo intercepte, haciendo posible derivar el *keystream* utilizado para cifrar comparando el texto plano con el cifrado que envía el cliente.

Sin embargo, el diseño del propio algoritmo WEP presenta un problema de seguridad mucho mayor.

Al ser RC4 un algoritmo de cifrado en bloques, es imperativo que la clave usada para cifrar no se repita, ya que esto permitiría establecer relaciones entre mensajes y derivar información sobre el contenido. Para eso se incluye un IV que se concatena a la clave WEP. Sin embargo, este IV es demasiado corto (24 bits), por lo que sólo existen unos 17 millones de IVs posibles.

En un primer momento no parece un problema, pero cuando estamos enviando cientos e incluso miles de paquetes por minuto, vemos aumentar la probabilidad de que se repita un IV. De hecho, por la *paradoja del cumpleaños* sabemos que es bastante probable que, por cada 4096 paquetes enviados, dos de ellos comparten el mismo IV y, por lo tanto, se filtre información sobre la clave y el mensaje.

Este tipo de razonamiento, junto con el hecho de que RC4 presenta un conjunto importante de *weak keys* (esto es, claves que hacen que cierto algoritmo se comporte de una forma no deseada, por ejemplo, perdiendo entropía o revelando información) llevó a la creación de numerosas herramientas capaces de obtener una clave WEP en apenas unos minutos, complementando la estadística con ataques de fuerza bruta. En concreto, la suite de *aircrack* incluye una herramienta con esta funcionalidad.

2.3.2 WPA/WPA2

La facilidad con la que se podía *crackear* una clave WEP obligó a desarrollar un nuevo algoritmo criptográfico que lo sustituyera. Así nacieron, entre 2003 y 2004, los algoritmos *Wi-Fi Protected Access*: *WPA* y *WPA2*. El primero se ideó como una transición hacia WPA2 y no difiere demasiado en su funcionamiento, por lo que vamos a tomar WPA2 como referencia en este apartado. En concreto, vamos a basarnos en lo especificado en el estándar 802.11i.

2.3.2.1 Autenticación

Al igual que en el estándar 802.11a, es posible trabajar con dos modos de autenticación: *Open System auth.* y *Shared Key auth.* Sin embargo, en este nuevo estándar se denominan **WPA-Personal (o WPA-PSK)** y **WPA-Enterprise**, respectivamente. En ambos casos, el proceso de autenticación depende de una clave PMK (*Pairwise Maste Key*), que se deriva directamente de la contraseña introducida por el usuario.

WPA-Personal genera una clave de 256 bits, denominada *pre-shared key* (PSK) que, junto con la longitud del SSID (Service Set Identifier), se usará para generar la *Pairwise Master Key* (PMK), que será la clave empleada en el *handshake* y la generación de la clave de sesión, como $PMK = PBKDF2(HMAC-SHA1, PSK, SSID, 4096, 256)$.

WPA-Enterprise emplea un intercambio EAP (*Extensible Authentication Protocol*) para generar la PMK usando la función *hash PBKDF2-SHA1*. Este protocolo incluye funcionalidad para garantizar control de acceso, uso de certificados y algunas otras características ideales para su uso en empresas. Sin embargo, necesita de la inclusión de un servidor de autenticación en la red para funcionar correctamente.

La autenticación se realiza a través del *Four-Way Handshake*. Este procedimiento permite que tanto el punto de acceso como el terminal se demuestren mutuamente que conocen la PMK, sin llegar a revelarla, en tan solo cuatro pasos:

1. El punto de acceso envía un valor *ANonce* al cliente, el cual generará la denominada PTK (*Pairwise TransientKey*). Para ello tomará un valor *SNonce* y generará la clave como: $PTK = PMK + ANonce + SNonce + MAC_{cliente}$
2. El cliente envía ahora el *SNonce* al punto de acceso, junto con un *Message Integrity Code* (MIC) que garantice la integridad y autenticidad del mensaje.
3. El punto de acceso comprueba el MIC y genera la PTK (que coincidirá con la generada por el cliente) y la *Group Temporal Key* (GTK), que envía ahora al cliente junto con su correspondiente MIC.
4. Finalmente, el cliente comprueba el MIC y envía un *acknowledge* de verificación, tras el cual ambas partes instalarán las claves PTK y GTK.

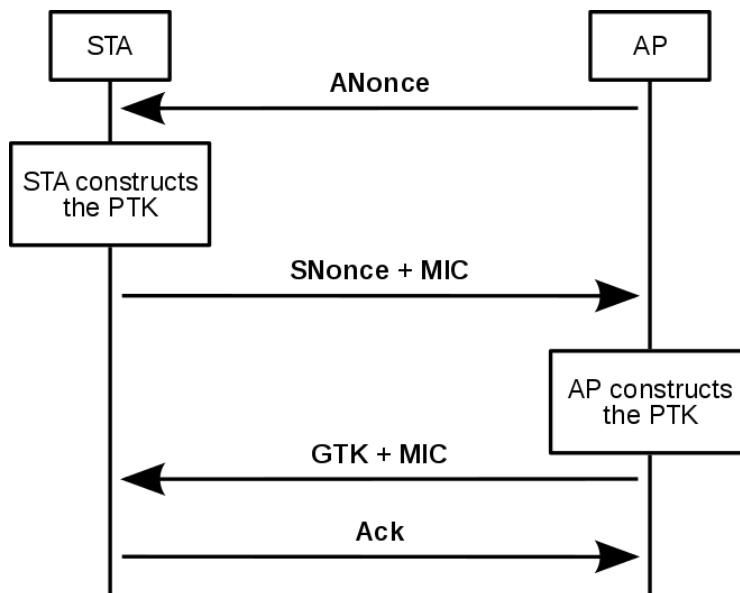


Figura 2.15 Four-Way Handshake en WPA2.

2.3.2.2 WPS

WPS (*Wi-Fi Protected Setup*) es un estándar que surgió en 2007, cuya función es facilitar el acceso a una red WLAN de forma segura y cómoda. En concreto, WPS define los métodos para intercambiar las credenciales necesarias para la autenticación (SSID y PSK).

Actualmente, este estándar es opcional en la mayoría de *routers* Wi-Fi, llegando incluso a estar activado por defecto al iniciar el *router* por primera vez.

En la arquitectura se definen tres roles diferentes:

- **Registrar:** dispositivo que genera o revoca credenciales en la red. Suele ser el punto de acceso (aunque puede ser un terminal).
- **Enrollee:** dispositivo que solicita el acceso a la red.
- **Authenticator:** proxy entre el *registry* y el *enrollee*.

WPS recoge varios métodos de autenticación en la red:

- **PIN:** todo dispositivo que quiera conectarse a la red WLAN deberá introducir un código de 8 dígitos que, por lo general, se encuentra en la parte posterior del *router*. Si se introduce correctamente, el punto de acceso le proveerá la clave PSK.
- **PBC:** las credenciales se generan e intercambian en el momento en que se pulsa un botón tanto en el punto de acceso como en el cliente. Sin embargo, durante el período de tiempo en el que el botón permanezca pulsado, cualquier otro dispositivo podrá conectarse.
- **NFC:** las credenciales se intercambian usando tecnología NFC.
- **USB:** las credenciales se transmiten usando una memoria externa desde el *registrar* al *enrollee*.

2.3.2.3 Cifrado

Una vez ambas partes han acordado una clave de transición (PTK), de esta derivan tres nuevas claves: la *Key Confirmation Key* (KCK), la *Key Encryption Key* (KEK) y la *Temporary Key* (TK). Las dos primeras se usarán para proteger la información intercambiada durante el *handshake*, mientras que la última es la que nos permitirá cifrar los paquetes de la comunicación.

En concreto, el algoritmo usado en WPA2 es AES-CCMP (aunque algunas versiones de WPA admiten TKIP, que es un algoritmo similar al usado en WEP pero que construye las claves de forma más segura), lo que garantiza confidencialidad e integridad de los mensajes.

La idea del cifrado WPA2 es que no se busca cifrar directamente el mensaje usando una clave, sino que se toma un *stream* de datos relacionados con la conexión (una serie de flags, direcciones MAC, contador de paquetes, número de paquete...) y se usa AES-CCMP para cifrar este *stream* con la clave derivada de la PTK, obteniendo lo que se conoce como *keystream*. Finalmente, se realiza un XOR entre el mensaje y el *keystream* para obtener el texto cifrado.

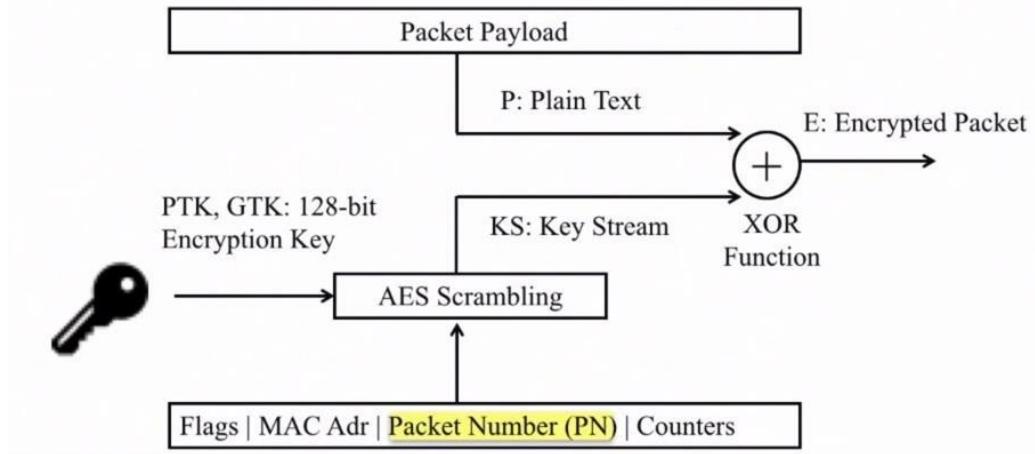


Figura 2.16 Cifrado en WPA2.

2.3.2.4 Vulnerabilidades

WPA2 fue objeto de estudio durante numerosos años, por lo que es considerado uno de los protocolos más seguros y más recomendables para usarse en redes WiFi. De hecho, prácticamente la totalidad de los *routers* actuales funcionan con este algoritmo. Sin embargo, a lo largo de los años han aparecido algunas vulnerabilidades que, aunque han sido *parcheadas* en muchos casos, todavía afectan a numerosos sistemas.

En primer lugar, aunque no se considera una vulnerabilidad como tal, si un atacante conoce la clave de acceso a la red, podrá obtener la PTK. Le bastará con interceptar los paquetes en los que los *nonces* se intercambian en texto plano y concatenarlos con la PMK (que puede derivar de la clave de acceso) y la MAC de la víctima.

Reutilización de *nonces*

Una vez ha tenido lugar el *handshake*, es imperativo que, cada vez que se cifre un mensaje, se utilice un *stream* distinto como entrada de la función AES-CCMP. En concreto, existe un valor denominado *Packet Number (PN)* que incrementa con cada paquete enviado, siendo el único factor no fijo del *stream*.

Si siempre se usase el mismo PN, entonces el *keystream* no vararía, por lo que tendríamos la siguiente igualdad (siendo E1 y E2 dos mensajes cifrados distintos, P1 y P2 los textos planos correspondientes): $E1 \text{ XOR } E2 = P1 \text{ XOR } P2$.

De esta forma, si un atacante conociese el contenido de un único mensaje, podría descifrar cualquier otro usando una simple operación: $P1 \text{ XOR } (P1 \text{ XOR } P2) = P2$.

Por lo tanto, aunque no se trata de un fallo en el diseño, si un desarrollador implementa este protocolo de forma incorrecta estará permitiendo que cualquier atacante descifre todos los mensajes de los clientes de una red.

KRACK

El ataque conocido como *CRACK (Key Reinstallation Attack)*, publicado en 2017, consiste en obligar a la víctima a reutilizar un *Packet Number* (o *nonce*) y reinstalar una clave que ya haya sido usada y permita a un atacante descifrar cualquier mensaje. Los investigadores que descubrieron esta vulnerabilidad se dieron cuenta de que, durante el *handshake*, si al punto de acceso no le llega el mensaje de *acknowledge* final, este repite el paso 3 (reenvía la clave GTK), haciendo que el terminal reinstale las claves (PTK y GTK) y, por lo tanto, reinicie los *nonces*.

En este punto ya se está produciendo una reutilización de *nonces* que nos lleva al ataque anterior. Sin embargo, en función del estándar que se esté utilizando, el dispositivo podrá manejar este reenvío de una manera o de otra. En concreto, la parte más crítica de este ataque se la lleva Android 6.0.1: se encontró un error de implementación que lleva al dispositivo a reiniciar la clave a cero cuando se produce un reenvío del paquete en concreto.

Ataque al PIN WPS

Como hemos explicado, WPS contempla la autenticación a través de un código de 8 dígitos llamado PIN. Ya de por sí, parece más fácil obtener este código que la clave de acceso como tal (que suele ser más larga y consta de caracteres alfanuméricos). Sin embargo, un ataque de fuerza bruta al PIN, aunque posible (la mayoría de *routers* no cuenta con una limitación de intentos de acceso), resulta poco viable al tener que probar 10^8 combinaciones.

El problema viene cuando nos damos cuenta de cómo se comprueba la validez del PIN: en vez de comprobar los 8 dígitos, se verifican los 4 primeros y, posteriormente, los 4 últimos, lo que deja $10^4 + 10^4$ combinaciones. Además, el último dígito no forma parte del PIN, sino que se usa como *checksum*, por lo que en total habría que probar $10^4 + 10^3 = 11,000$ combinaciones en total, lo que resulta bastante manejable.

Tras averiguar el PIN, resulta bastante sencillo obtener la clave PSK y acceder así a la red.

2.3.3 Conclusiones

WPA2 supuso una verdadera evolución frente al protocolo WEP en lo que a seguridad se refiere, al incluir algoritmos criptográficos modernos y robustos. Además, al ser un estándar muy estudiado, su diseño es bastante bueno y da lugar pocos problemas de seguridad, siempre y cuando se tomen algunas precauciones, como desactivar el acceso por WPS (especialmente en modo PIN).

Por otro lado, WEP se considera totalmente obsoleto hoy en día debido a la facilidad con la que se pueden romper las claves usando algunos programas como *reaver* o *pixiewps*. Por tanto, no debería usarse para la gestión de redes WLAN (aunque, sorprendentemente, todavía podemos encontrar algunas redes que utilizan este protocolo).

2.4 TLS

Transport Layer Security (TLS) es un protocolo criptográfico de seguridad diseñado para redes de comunicaciones. En la *Internet Protocol Suite* lo podemos encontrar en la capa de aplicación, al igual que otros protocolos como SSH o Telnet.

Uno de los usos principales de TLS es el de cifrar las comunicaciones en la Web, dando lugar al protocolo HTTPS, que implementa el clásico HTTP a través de un canal seguro. Por lo tanto, este apartado va a ir enfocado a las aplicaciones *cliente-servidor* que lo usan.

Originalmente, este protocolo se denominaba SSL y fue desarrollado por Netscape. Sin embargo, a lo largo de los años se descubrieron numerosas vulnerabilidades que afectaban directa o indirectamente a las diferentes versiones del protocolo:

- SSL se basaba en algoritmos que hoy en día se pueden romper, como RC4 o MD5. Además, las versiones previas a la 3.0 presentaban una vulnerabilidad que podía ser explotada mediante el ataque conocido como *POODLE*, que permitía obtener información sobre mensajes cifrados e incluso descifrarlos por completo.
- En la versión 1.0 de TLS, era posible efectuar un ataque contra el CBC al utilizar este un esquema inseguro para la generación de IVs, haciéndolos predecibles.
- Otro ataque que podía efectuarse en la versión 1.0 obligaba a cliente y servidor a establecer una comunicación empleando una versión de SSL insegura (*downgrade attack*).
- Las versiones 1.1, 1.2 y 1.3 no presentan vulnerabilidades severas, sino que surgen de la implantación de nuevos mecanismos criptográficos más seguros y eficientes.

Las versiones más utilizadas hoy en día son la 1.2 y la 1.3 al ser las más robustas. Además, grandes empresas como Apple, Microsoft, Mozilla y Google han llegado a ponerse de acuerdo para que sus navegadores interrumpan la compatibilidad con las versiones 1.0 y 1.1 a partir de 2020.

2.4.1 Detalles del protocolo

Cuando se establece una conexión TLS, el procedimiento es muy similar al del intercambio de claves explicado en capítulos anteriores: como la criptografía asimétrica es computacionalmente compleja, se usa únicamente para intercambiar una clave de sesión de forma segura. Esta clave de sesión será la que se emplee para cifrar toda la comunicación.

Uno de los mecanismos asimétricos de intercambio de clave que más se usa es ECDH, pero en TLS existe una variación de este algoritmo, denominada ECDH efímero o ECDHE. La diferencia es que ECDHE utiliza una clave pública distinta en cada conexión, por lo que el servidor la genera cada vez que un cliente se la solicita. ECDH, sin embargo, permite al servidor almacenar una única clave pública en su certificado, ahorrándose un paso (aunque, obviamente, es menos seguro).

El procedimiento de intercambio de información necesaria para establecer la conexión segura se denomina *handshake TLS* y consta de las siguientes fases:

1. El cliente envía un mensaje **ClientHello** al servidor, donde se especifica la última versión TLS que soporta, junto con un número aleatorio y el conjunto de funciones criptográficas de las que dispone.
2. El servidor responde con un **ServerHello** que contiene la versión escogida (la más alta soportada por ambos), un número aleatorio y las funciones criptográficas a utilizar (de entre las que dispone el cliente y se especifican en la versión acordada).
3. El servidor envía su clave pública al cliente, en un mensaje **Certificate**.
4. El servidor envía un mensaje **ServerKeyExchange** que contiene la clave efímera en caso de utilizar ECDHE.
5. El servidor envía un **CertificateRequest** al cliente para solicitar un certificado.
6. El servidor envía un **ServerHelloDone** para indicar el fin de la negociación.
7. El cliente responde con su propio **Certificate** y su **ClientKeyExchange**, cuyo contenido depende del cifrado escogido.
8. El cliente envía un **CertificateVerify**, que no es más que una firma digital sobre los mensajes previos.
9. El servidor verifica la firma usando la clave pública del cliente, pudiendo verificar la identidad del cliente.
10. Con la información intercambiada, cliente y servidor generan la misma clave simétrica, con la que cifrarán el resto de mensajes.
11. El cliente envía ahora un **ChangeCipherSpec** que indica al servidor que todos los mensajes que le lleguen estarán ahora cifrados.
12. El cliente envía un mensaje **Finished** cifrado con la clave simétrica, junto con un *hash* y un MAC de los mensajes intercambiados en el *handshake*. El servidor comprueba ahora la integridad y validez de todos los mensajes y, en caso de que no sean correctos, finaliza la conexión.
13. Finalmente, el servidor envía su propio **ChangeCipherSpec** y su **Finished** cifrado, de nuevo, con un *hash* y un MAC, para que el cliente verifique la información.
14. A partir de este punto, comienza la comunicación como tal, con todos los mensajes cifrados usando la clave simétrica generada y el algoritmo acordado.

2.4.2 Versiones

La única versión considerada segura en su totalidad es la v1.3, debido a que únicamente contempla un conjunto de algoritmos criptográficos totalmente seguros, mientras que la v1.1 y la v1.2 admiten una serie de algoritmos cuya robustez depende de cómo sean implementados por el cliente. En las siguientes tablas podemos ver los algoritmos admitidos por cada versión:

Intercambio de clave y autenticación				
Algoritmo	TLS 1.0	TLS 1.1	TLS 1.2	TLS 1.3
RSA	✓	✓	✓	✗
DH-RSA	✓	✓	✓	✗
DHE-RSA	✓	✓	✓	✓
ECDH-RSA	✓	✓	✓	✗
ECDHE-RSA	✓	✓	✓	✓
DH-DSS	✓	✓	✓	✗
DHE-DSS	✓	✓	✓	✓
ECDH-ECDSA	✓	✓	✓	✗
ECDHE-ECDSA	✓	✓	✓	✓

Figura 2.17 Intercambios de clave admitidos por las versiones TLS.

Algoritmos de cifrado					
Tipo	Nombre	Robustez	TLS 1.1	TLS 1.2	TLS 1.3
Cifrador en bloque con modo de operación	<i>AES GCM</i>	256, 128	N/A	Seguro	Seguro
	<i>AES CCM</i>		N/A	Seguro	Seguro
	<i>AES CBC</i>		N/A	N/A	N/A
	<i>Camellia GCM</i>		N/A	Seguro	N/A
	<i>Camellia CBC</i>		N/A	N/A	N/A
	<i>3DES</i>	112	Inseguro	Inseguro	N/A
	<i>IDEA CBC</i>	128	Inseguro	N/A	N/A
	<i>DES CBC</i>	56	Inseguro	N/A	N/A
	<i>RC2 CBC</i>	40	N/A	N/A	N/A
Cifrador en flujo	<i>ChaCha20-Poly1305</i>	256	N/A	Seguro	Seguro
	<i>RC4</i>	128	Inseguro	Inseguro	N/A

Figura 2.18 Robustez de los algoritmos de cifrado en versiones TLS.

Algoritmos de integridad de datos			
Nombre	TLS 1.1	TLS 1.2	TLS 1.3
HMAC-MD5	✓	✓	✗
HMAC-SHA1	✓	✓	✗
HMAC-SHA2/3	✗	✓	✗
AEAD	✗	✓	✓

Figura 2.19 Algoritmos de integridad soportados por TLS.

2.4.3 Seguridad

Actualmente, TLS es el protocolo más utilizado en aplicaciones cliente-servidor. De hecho, su aparición se tradujo en un aumento exponencial en la seguridad de la Web al aportar métodos de autenticación y cifrado que, hasta entonces, no existían. En concreto, HTTPS consiguió neutralizar los ataques *Man in the Middle* al impedir que el atacante pudiese acceder a la información por estarcifrada.

Sin embargo, existen algunos aspectos a tener en cuenta y algunas vulnerabilidades en ciertas versiones del protocolo:

POODLE

Como ya hemos mencionado anteriormente, el ataque POODLE clásico consiste en aprovechar un fallo en la negociación de la versión TLS para hacer que cliente y servidor establezcan una conexión usando SSL 3.0 (o alguna versión anterior) y aprovechar vulnerabilidades existentes en dicha versión. Este ataque se encuentra registrado como [CVE-2014-3566](#).

Heartbleed

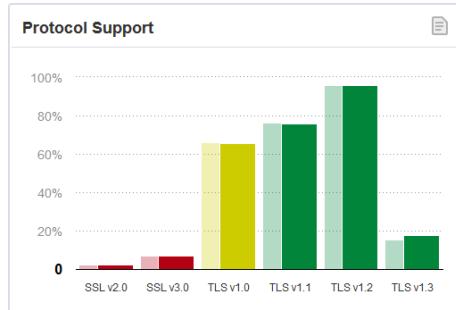
Se trata de una vulnerabilidad en las versiones 1.0.1-1.0.1f de OpenSSL, que permite leer la memoria de un servidor o cliente y obtener así claves privadas SSL/TLS. Se encuentra registrado como [CVE-2014-0160](#).

Lucky Thirteen

Un ataque *side-channel* criptográfico que afecta al protocolo TLS cuando usa cifrados con modo CBC. No tiene una importancia crítica, ya que actualmente no se usa este tipo de cifrados, pero existe la creencia de que ataques como este podrían mejorar y llegar a afectar a las versiones más modernas de TLS. Por lo tanto, se recomienda usar librerías criptográficas que implementen métodos para defenderse de este tipo de *side-channel attacks*.

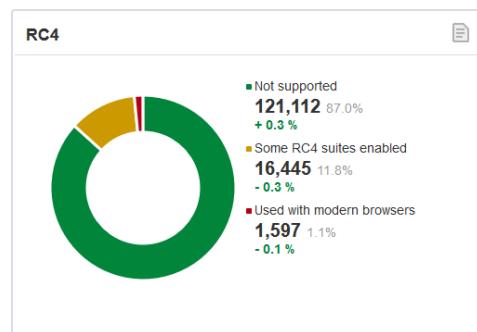
2.4.4 ssl-pulse

Existe una aplicación desarrollada por [ssl-labs](#), llamada [ssl-pulse](#), que efectúa escaneos mensuales para monitorizar la correcta implantación de SSL/TLS. En concreto, podemos encontrar datos como los siguientes (a fecha de **septiembre de 2019**):

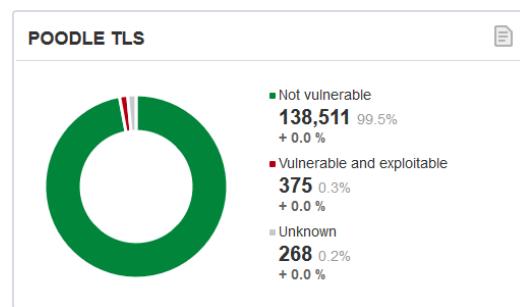


Existe todavía una alarmante cantidad de sitios web compatibles con la versión 1.0 de TLS, considerada insegura.

Por otro lado, se puede apreciar una lenta transición hacia la nueva versión 1.3.



Aunque el **87%** de los sitios encuestados ha dejado de soportar RC4, todavía existe una cierta cantidad de sitios web que la soportan.



El ataque Poodle ha sido prácticamente erradicado. Tan sólo un **0.3%** de las páginas encuestadas son vulnerables a este ataque.

2.4.5 HSTS

Si bien es cierto que al aparecer TLS y extenderse al protocolo HTTPS se neutralizaron los ataques MITM, los atacantes descubrieron que todavía era posible solicitar conexiones HTTP (no seguras) a los servidores. Por lo tanto, si se redirigía el tráfico de la víctima hacia un puerto HTTP, se podía “esquivar” esta capa de seguridad.

Como solución a este problema surgió la *HTTP Strict Transport Security (HSTS)*, que es una política de seguridad mediante la cual un servidor establece que los navegadores compatibles únicamente podrán establecer conexiones con él mediante HTTPS, denegando cualquier conexión no segura y avisando al usuario en ese caso.

2.5 Cifrados End-to-End

Hoy en día es bastante común oír hablar del *cifrado extremo a extremo* o “*end-to-end*” (*E2EE*). La idea subyacente bajo este concepto es la de proporcionar un canal de comunicación al que sólo tengan acceso los extremos. Es decir, si ponemos como ejemplo una aplicación de mensajería, únicamente los usuarios finales deberían tener acceso al contenido de los mensajes y, aunque estos pasen por un servidor intermedio, nadie más aparte de los usuarios pueda descifrarlos.

Actualmente existe una gran cantidad de aplicaciones (la mayoría de mensajería instantánea) que basan su funcionamiento en esta estructura, como *WhatsApp*, *Telegram*, *Signal* o *Skype*.

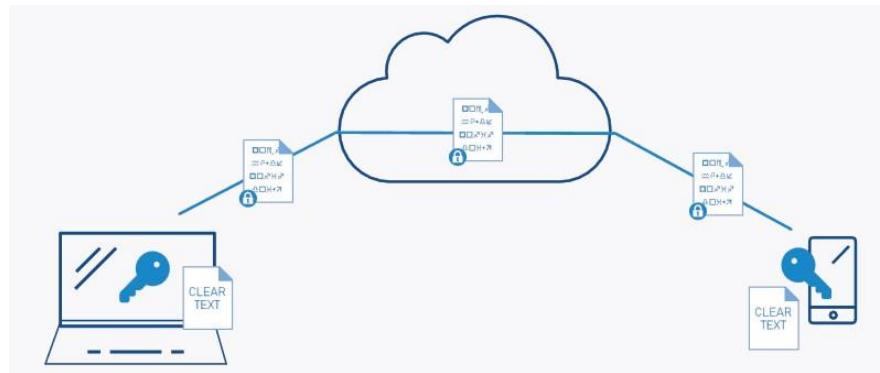


Figura 2.20 Cifrado end-to-end.

Este sistema garantiza a los usuarios un nivel bastante alto de privacidad, ya que teóricamente ni siquiera la compañía que ha desarrollado la aplicación de mensajería podrá leer los mensajes enviados.

Obviamente, alguien podría preguntarse: “¿Y por qué no nos deshacemos del servidor y así evitamos complicaciones y conectamos directamente a los usuarios?”. Está claro que esta solución sería la clave para obtener una privacidad total en el intercambio de mensajes. Sin embargo, ¿cómo puede Alice saber dónde se encuentra el móvil de Bob? Y si uno de los dos tiene el móvil apagado, ¿cómo pueden seguir enviándose mensajes?

La respuesta es que no podemos prescindir del servidor. Este almacena las direcciones IP de cada dispositivo, por lo que puede estar en contacto con ellos continuamente. Además, los servidores almacenan los mensajes cuando un dispositivo está desconectado y, cuando recupera la conexión, los envía. Es cierto que hay compañías con servidores que almacenan más información de la estrictamente necesaria con fines más o menos éticos. Sin embargo, hacen esto aprovechándose de que no existen soluciones que pasen por la inexistencia de un servidor intermedio.

2.5.1 Retos

Para crear una infraestructura que se amolde a estas características se debe satisfacer un conjunto de necesidades criptográficas (a las que denominaremos **retos**), como implementar un intercambio de claves seguro y eficiente, elegir un cifrado rápido y robusto o garantizar la correcta gestión de claves en los dispositivos.

Autenticación

Uno de los problemas más críticos en las comunicaciones son los ataque MITM, en los que un tercero intercepta los mensajes y accede a su contenido, vulnerando así la privacidad de los usuarios. Idealmente, E2EE asegura que nadie puede descifrar los mensajes sin conocer las claves que, en teoría, únicamente los extremos poseen. Sin embargo, es posible “engaños” a los usuarios que establecen una comunicación:

Supongamos que Alice y Bob quieren mantener una conversación usando una aplicación que implementa un sistema E2EE. Eve interceptaría los mensajes iniciales, en los que ambas partes intercambian claves para el cifrado. Haciendo esto, Eve podría intercambiar su propia clave con Bob, quien creería que en realidad procede de Alice, y seguiría un proceso análogo para la otra parte.

En este punto, cuando Alice le envíe un mensaje a Bob, lo hará usando la clave proporcionada por Eve, de forma que el atacante tendrá acceso al mensaje y, tras leerlo, lo cifrará con la clave que le ha proporcionado Bob. Entonces Eve le reenviará el mensaje a Bob y nadie sospechará que se ha producido una intrusión.

Para evitar este tipo de ataque, es necesario proporcionar un método de autenticación que garantice un intercambio de claves seguro. Para esto se pueden usar diversos enfoques, como el uso de certificados, una *web-of-trust*, sistemas *out-of-band* (es decir, externos a la aplicación)...

Seguridad en los extremos

En un sistema E2EE, no sólo existen riesgos en la comunicación. Si uno de los extremos ha sido atacado, es posible que un atacante obtenga las claves allí almacenadas y, en consecuencia, acceda a toda la información intercambiada. Por lo tanto, es necesario garantizar una gestión de claves segura, aislando estas del resto del sistema.

Uno de los ejemplos más llamativos de este tipo de seguridad es el **Google's Project Vault**, que pretende almacenar en una tarjeta MicroSD las claves criptográficas necesarias para la autenticación.

Backdoors

Una “puerta trasera” o *backdoor* es un fragmento de código insertado en el código fuente de un sistema que permite un acceso al mismo sorteando la seguridad su seguridad. En 2013, el analista de la NSA *Edward Snowden* filtró información que demostraba que la NSA incluyó *backdoors* en una gran cantidad de aplicaciones y sistemas informáticos para tener acceso a información privada. En concreto, se mostró que *Skype* incluía una forma de sortear el cifrado de sus mensajes.

Por lo tanto, otro de los retos a los que se enfrenta un sistema E2EE es el de demostrar que no se ha insertado ningún fragmento de código con estas características que pueda vulnerar la privacidad de los usuarios.

Eficiencia

Los sistemas E2EE encuentran su principal aplicación en la mensajería instantánea y, como tal, se debe tener especialmente en cuenta la eficiencia y la velocidad del cifrado. Por lo tanto, resulta crucial escoger algoritmos rápidos pero robustos que se adecuen a las características de la comunicación (cifrados en bloque para paquetes pequeños o de flujo para, por ejemplo, *streamings*).

2.5.2 Signal Protocol

La aplicación de mensajería *Signal* fue una de las primeras en incorporar el cifrado *end-to-end*, y lo hizo de la mano de un protocolo propio denominado *Signal Protocol* que permitía cifrar tanto llamadas de video y voz, como mensajes instantáneos.

El protocolo combina un algoritmo denominado *Double Ratchet* con un intercambio de claves basado en un triple ECDH (X3DH). Para el cifrado se usan las curvas X25519 y X448, así como las primitivas AES y HMAC-SHA256. Además, se define un nuevo algoritmo, denominado *Sesame*, que gestiona el uso de sesiones de cifrado de forma asíncrona para múltiples dispositivos, usando los algoritmos mencionados.

Generación de claves

Cuando un usuario se descarga e instala la aplicación, se generará una serie de pares de claves ECDH. De este modo, la primera vez que Bob se conecte al servidor, enviará sus claves públicas para que se almacenen y, cuando algún otro usuario quiera enviarle un mensaje, pueda usar dichas claves.

- **IPK:** *Identity Public Key*, es la clave pública asignada a un usuario de forma permanente.
- **SPK:** *Signed Public Key*, se usa para verificar la IPK.
- **OPK:** *One-Time Public Key*, se usa para establecer un primer acuerdo de claves simétricas con otro usuario.
- **EPK:** *Ephemeral Public Key*, par de claves efímero (de un único uso) para ECDH que se usa en la generación del secreto compartido.

A diferencia de la IPK y la SPK, que son únicas, se envía un conjunto de OPKs al servidor. Cada vez que un usuario nuevo quiera realizar un intercambio de claves con Bob, instalará una de las OPKs disponibles, y esta será eliminada del servidor. Este conjunto de claves será rellenado por Bob según sea necesario.

Obviamente, Bob almacenará de forma segura todas las claves privadas asociadas a cada una de las que se encuentran en el servidor.

Por último, Alice (que quiere enviarle un mensaje a Bob), deberá generar la EPK única para dicho mensaje.

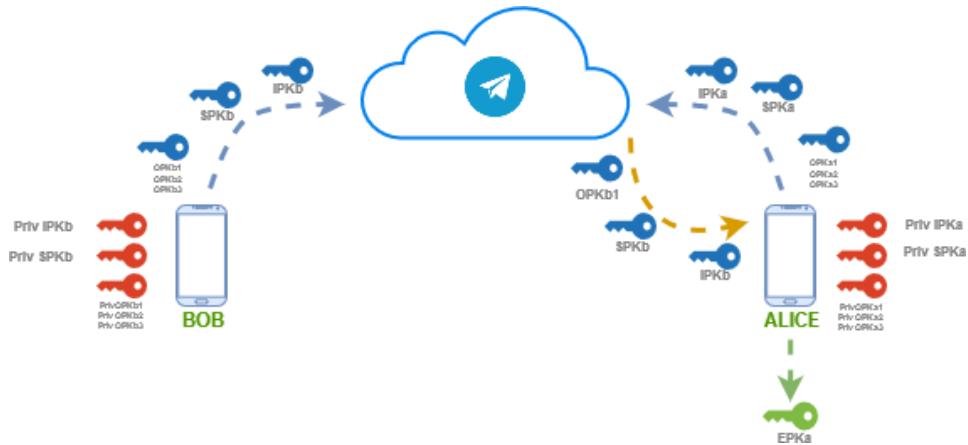


Figura 2.21 Generación de claves para Signal Protocol.

Triple Diffie-Hellman (X3DH)

A continuación, es necesario que ambas partes acuerden una clave simétrica con la que cifrar mensajes y, como viene siendo hasta ahora, se usará ECDH para ello. Sin embargo, el hecho de haber generado tantas claves no sirve únicamente para complicarlo todo, sino que todas ellas se emplean para衍生 una única clave: en total, se realizarán 4 Diffie-Hellman entre las claves compartidas, y se concatenarán para dar lugar al *secreto compartido*.

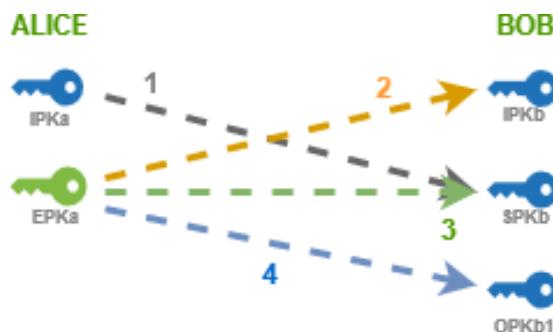


Figura 2.22 Triple Diffie-Hellman

El secreto compartido se obtendrá como:

$$\text{master_secret} = \text{KDF}(\text{ECDH}_1 \parallel \text{ECDH}_2 \parallel \text{ECDH}_3 \parallel \text{ECDH}_4)$$

Llegados a este punto, Alice y Bob comparten un secreto y pueden intercambiar mensajes cifrados. Sin embargo, obteniendo el secreto, se podría descifrar la conversación entera. Por lo tanto, es necesario incluir otra capa de seguridad que solvete este problema.

Double Ratchet

Double Ratchet (también conocido como *Axolotl*) es la clave de la seguridad de Signal Protocol. Gracias a este algoritmo, podemos衍生 claves de forma sincronizada entre dos dispositivos, y así obtener claves únicas para cada mensaje, de modo que aunque un atacante logre obtener una de las claves, no pueda descifrar más de un mensaje (ni pasado ni futuro).

El concepto *ratchet* (literalmente, “rueda dentada”) se emplea para hacer referencia a un tipo de función unidireccional empleada para衍生 claves, denominada *ratchet function*. La idea es que, mientras emisor y receptor tengan una *ratchet* sincronizada, podrán衍生 claves que les permitan cifrar y descifrar mensajes compartidos.

Por lo tanto, el algoritmo *Double Ratchet* introduce dos *ratchets* en cada extremo: uno para enviar y otro para recibir. Cuando Alice le envía un mensaje a Bob por primera vez, sus *ratchets* estarán inicializados con una misma clave inicial (derivada de las claves que han intercambiado con el servidor durante el registro). En este caso, Alice derivará su clave AS1 (para enviar) y Bob derivará su clave BR1 (para recibir), obteniendo cada uno una clave nueva. Tras derivar las claves, se deben eliminar las anteriores (salvo casos excepcionales que veremos más adelante).

En la siguiente figura se muestra un ejemplo del funcionamiento, donde ASx y BSx hacen referencia a las claves para enviar, y ARx y BRx, para recibir.

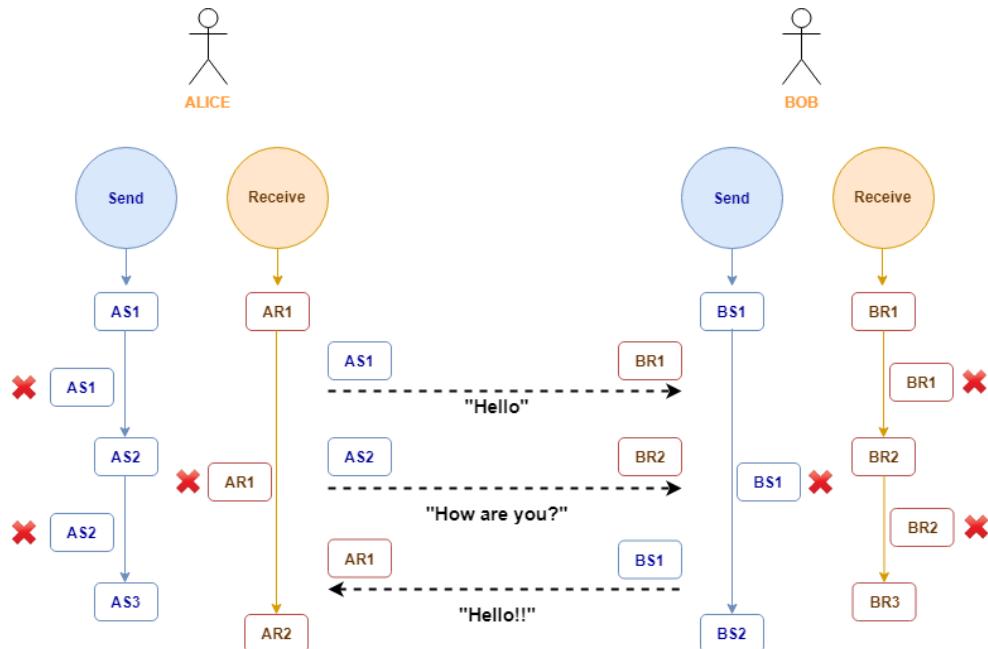


Figura 2.23 Single Ratchet.

Al procedimiento anterior se le conoce como *Single Ratchet*, y presenta un serio problema: si un atacante intercepta alguna de las claves, no podrá descifrar los mensajes anteriores, pero sí será capaz de derivarla para obtener las siguientes.

Para evitar este problema, se añade lo que se entiende como *segundo ratchet*: a cada cierto número de mensajes intercambiados, se efectúa un nuevo acuerdo de claves ECDH que reinicia los valores de los *ratchets*. De este modo, a no ser que el atacante obtenga también las claves del ECDH, no podrá obtener ninguna información. En la mayoría de aplicaciones (como *Whatsapp*), este intercambio se realiza con cada mensaje.

Por último, queda solucionar el problema de que se pierda la sincronización entre *ratchets* (por ejemplo, si se pierde un mensaje o si llegan en distinto orden). En estos casos, se puede optar por una de estas dos soluciones:

1. El mensaje que llega fuera del orden (todo mensaje lleva información sobre el orden en que debería llegar) se almacena a la espera de recibir todos los que faltan.
 2. Se deriva la clave tantas veces como mensajes falten por llegar, pero no se eliminan las claves intermedias, sino que se almacenan a la espera de sus mensajes correspondientes.

Por lo general, la opción por la que apuestan las grandes aplicaciones es la segunda, ya que permite acceder a los mensajes de forma asíncrona y facilita la resolución de la pérdida de mensajes. Sin embargo, esta decisión obliga a mantener un estricto control sobre el almacenamiento y la eliminación de las claves.

Autenticación

Sólo queda un problema pendiente para entender el protocolo al completo: ¿Cómo sé realmente que la IPKa que le envía el servidor a Bob, realmente procede de Alice? ¿Cómo sé que ningún atacante está efectuando un MITM en nuestra conversación?

La respuesta es sencilla: compruébalo tú mismo. *Signal Protocol* no efectúa una comprobación de la identidad como tal, pero sí te provee de los medios necesarios para que lo hagas tú. Para comprobar la autenticidad de las claves de identidad de otro usuario, se emplea un método *out-of-band* que consiste en verificar la firma de esa clave que se encuentra almacenada en un servidor externo.

Whatsapp, por ejemplo, te permite obtener esa firma y compararla con la del usuario real, e incluso te muestra un *Código QR* para ello.

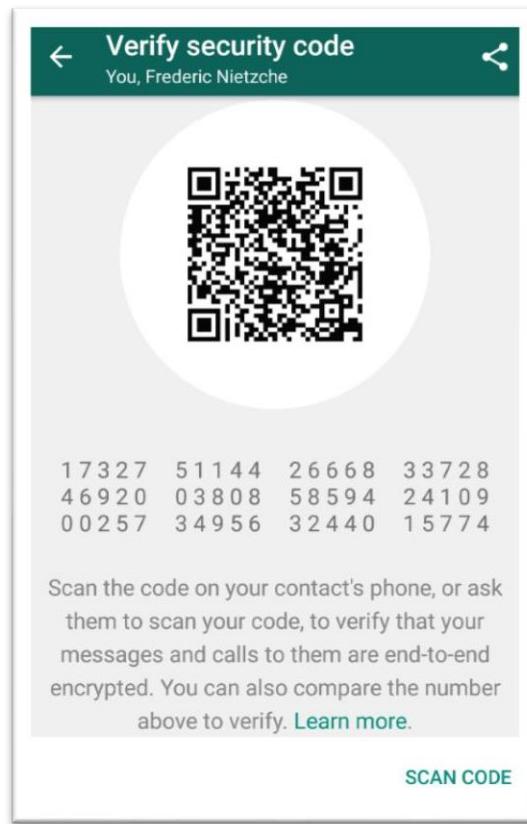


Figura 2.24 Verificación de identidad en WhatsApp.

Seguridad y privacidad

Signal Protocol garantiza la confidencialidad, la integridad, la autenticación, el no repudio y una larga lista de propiedades similares. Todo esto trabajando de forma asíncrona y permitiendo su uso en sistemas de comunicación instantáneos, incluyendo videollamadas y *streamings*.

En cuanto a la autenticación, se emplean claves públicas que pueden verificarse mediante el uso de *fingerprints* a través de un canal externo. Esto quiere decir que un usuario puede tomar una cadena de *bytes* única para cada clave pública (normalmente se usa una función *hash* sobre la misma clave) y compararla usando un servicio externo de confianza. De esta forma, dos usuarios pueden estar seguros de que la comunicación establecida no es objeto de un MITM.

Además, desde la página oficial de Signal se puede acceder a un *GitHub* que contiene el código fuente del protocolo, con **licencia Open Source**. Esta transparencia ha permitido comprobar que no existen *backdoors* que vulneren la privacidad de sus usuarios.

Sin embargo, todavía es posible para terceros conocer cuándo y con quién se comunica un usuario, por lo que la privacidad en este sentido depende de las políticas de la empresa que incorpore el protocolo.

Influencia

Las características de *Signal Protocol* son tan llamativas que la mayoría de los cifrados extremo a extremo existentes en la actualidad se basan en él, como por ejemplo el protocolo E2EE que incluye WhatsApp, así como los de Viber o Wire.

Además, el hecho de que se haya publicado con licencia *Open Source* con la idea de generalizar su implantación gratuita en todo tipo de proyectos, lo convierte en una idea que siempre se deberá considerar a la hora de diseñar un proyecto que requiera cifrados *end-to-end*.

2.5.3 WhatsApp

La aplicación de mensajería más utilizada a día de hoy incluyó en 2016 su propio servicio E2EE basado en el *Signal Protocol* que permite cifrar mensajes, llamadas de voz y video y cualquier archivo multimedia enviado con la aplicación.

En detalle

WhatsApp distingue dos tipos de claves de cifrado: las asimétricas y las de sesión. Entre las del primer tipo se encuentran la ***Identity Key Pair*** (par de claves para la Curva25519 generado en la instalación y de uso a largo plazo), la ***Signed Pre Key*** (par de claves para Curva25519 generados en la instalación y renovados de forma periódica) y las ***One-Time Pre Keys*** (un conjunto de pares de claves para Curva 25519 de un único uso, generado durante la instalación, que se va rellenando según se necesita). Entre las claves de sesión, por otro lado, encontramos la ***Root Key*** (un valor de 32 bytes usado para generar *Chain Keys*), la ***Chain Key*** (se deriva de la *Root Key* y se usa para crear *Message Keys*) y la ***Message Key*** (un valor de 80 bytes usado para cifrar mensajes: 32 bytes para AES-256, 32 para HMAC-SHA256 y 16 como IV).

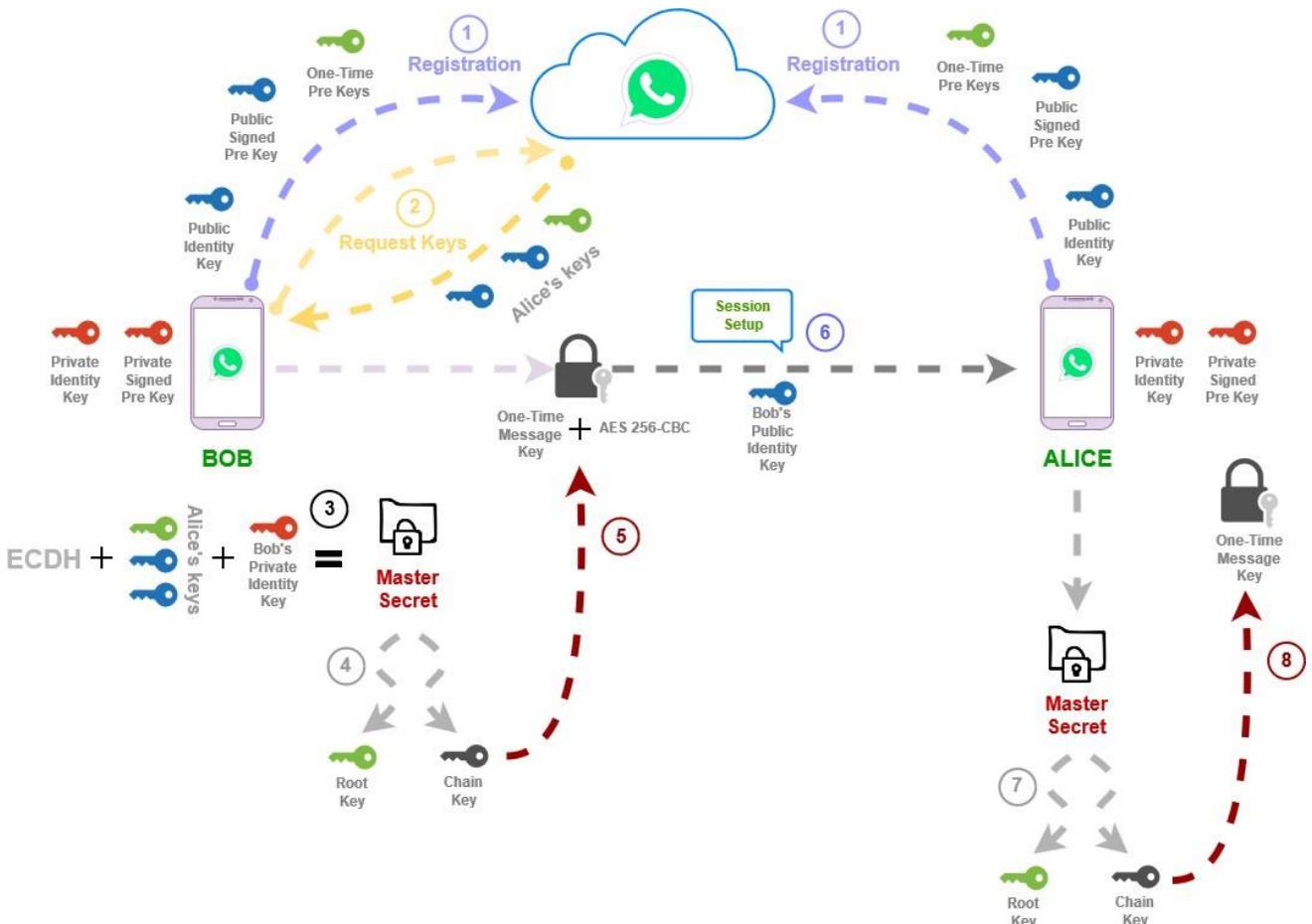


Figura 2.25 Protocolo E2EE de WhatsApp.

En la imagen superior se indican los pasos del protocolo, que vamos a explicar a continuación haciendo referencia al número de cada paso.

Registro y creación de la sesión de cifrado

Antes de comenzar a intercambiar mensajes, es necesario registrar al cliente en el servidor e iniciar una sesión de cifrado extremo a extremo entre ambas partes (emisor y receptor).

Cuando se registra un cliente (1), este envía su *Identity Key* pública, su *Signed Pre Key* pública y un conjunto de *One-Time Pre Keys* al servidor, el cual las almacena, pero en ningún momento tiene acceso a las claves privadas (que permanecen en el cliente).

A la hora de establecer una sesión entre clientes, uno de los dos (denominado *Initiator*) debe pedir el inicio de la misma (2), solicitando las claves públicas del receptor registradas en el servidor, así como una de las *One-Time Pre Keys* que, tras ser usada una vez, se descarta. A continuación, el *Initiator* genera una Curva25519 efímera, a la que llamaremos E_{Init} , y carga las *Identity Keys* de ambas partes (I_{Init} e I_{Recip}), la *Signed Pre Key* del receptor (S_{Recip}) y la *One-Time Pre Key* (O_{Recip}).

Usando las claves almacenadas, genera un *master secret* (3) mediante el X3DH:

$$\text{master_secret} = \text{ECDH}(I_{Init}, S_{Recip}) \parallel \text{ECDH}(E_{Init}, I_{Recip}) \parallel \text{ECDH}(E_{Init}, S_{Recip}) \parallel \text{ECDH}(E_{Init}, O_{Recip})$$

Donde el operador “ \parallel ” representa la concatenación.

Finalmente, el *Initiator* usa la función de derivación de clave HKDF para generar una *Root Key* y *Chain Keys* a partir del *master secret* (4). Cuando al receptor le llega el primer mensaje, este incluye información de establecimiento (6) de sesión que le permite calcular la *Root Key* y las *Chain Keys* correspondientes (7).

Intercambio de mensajes

Llegados a este punto, emisor y receptor poseen las claves necesarias para intercambiar mensajes cifrados *end-to-end*. Para ello, utilizan una de las *Chain Keys* para derivar la *Message Key* como $message_{key} = HMAC_{SHA256}(ChainKey, 0x01)$. La *Message Key* se usará para cifrar un único mensaje usando AES256-CBC (5).

Para que el receptor pueda descifrar los mensajes, tendrá que derivar las *Message Key* de la *Chain Key* correspondiente (8).

Regeneración de claves

Cada vez que se envía un mensaje, se anuncia el uso de una nueva clave pública efímera de la Curva25519, tal y como se especifica en el *Double Ratchet*. De esta forma, cuando el emisor del mensaje recibe la confirmación de que el mensaje se ha recibido, a su vez recibe la nueva clave pública que va a usar la otra parte. De esta forma, cada vez que se finaliza el ciclo de un mensaje (emisión + confirmación de recepción), se establecen las claves necesarias (*Chain Key* y *Root Key*) para generar una nueva *Message Key* para el próximo mensaje, evitando la reutilización de las mismas.

Gestión de claves

Ya hemos comentado que uno de los detalles más importantes a tener en cuenta en un criptosistema es la gestión de claves: ¿cómo podemos almacenarlas de modo que no dejen de ser accesibles para el usuario/aplicación, pero de forma segura?

La solución de *WhatsApp* es bastante buena: cuando se inicia la aplicación, se levanta un *sandbox* únicamente accesible por la misma. De esta forma, las claves se generan en dicho *sandbox* en el momento de la instalación, y permanecen en él indefinidamente. Por lo tanto, únicamente cuando la app de *WhatsApp* levante el *sandbox* podrá utilizar las claves.

Esto explica, por ejemplo, que sea necesario generar claves nuevas cada vez que se reinstala la aplicación: por seguridad, las claves originales no deben abandonar el *sandbox* anterior.

Conclusión

El cifrado *end-to-end* que ofrece WhatsApp es uno de los más completos y seguros del mercado. Está basado en el famoso *Signal Protocol* diseñado en gran parte por Mixie Marlinspike, que ya se encontraba previamente implementado en la aplicación de mensajería *Signal*.

Este sistema ofrece protección total frente a los ataques MITM, y además impide que las terceras partes que mantienen y administran la conexión puedan tener acceso a la comunicación, garantizando la privacidad de los usuarios.

Para agregar una capa extra de seguridad, se modifican las claves usadas tras cada mensaje enviado por lo que, aunque se interceptase una de las claves privadas, sería imposible recomponer la conversación al completo.

Pero...

Pese a que *WhatsApp* (y *Facebook*) aseguran que ni ellos ni terceros pueden acceder al contenido de los mensajes (de hecho, así lo ratificó Zuckerberg en su comparecencia ante el Congreso por el caso de *Cambridge Analytica*), un estudio intensivo de la aplicación nos hace sospechar lo contrario.

Supongamos que configuramos la app para guardar copias de seguridad en Google Drive (o en una carpeta local), opciones que están activadas por defecto.

Si un usuario se compra un móvil nuevo e instala *WhatsApp*, podrá recuperar sus mensajes usando una de estas copias de seguridad, pero en principio no tiene acceso a las claves privadas almacenadas en el dispositivo anterior (de hecho, al instalar la aplicación de nuevo, se generan claves distintas). Resulta que, para proporcionar esta funcionalidad, se crea una nueva clave usada únicamente para cifrar/descifrar la base de datos de la copia de seguridad, y esta clave **sí que se almacena en los servidores de WhatsApp**. De este modo, cuando un usuario solicite recuperar una copia de seguridad (autenticándose previamente mediante SMS o llamada), el servidor le enviará dicha clave.

Si bien es cierto que ningún atacante podrá acceder (a priori) a la clave de cifrado de la copia de seguridad, en este punto se pierde la privacidad absoluta de los usuarios: si una entidad con poder suficiente (pongamos, el FBI) solicita la clave y la copia de seguridad a *WhatsApp* o *Google* respectivamente, tendrá acceso a todos los mensajes enviados y recibidos por el usuario.

Sin embargo, este hecho resulta todavía más preocupante cuando nos encontramos con herramientas como *WhatsApp* que, usando un emulador con acceso físico al dispositivo, instala una nueva instancia de la aplicación, registrándose con el número de teléfono real. Si un atacante tiene acceso a la base de datos de mensajes (que se encuentra accesible en un directorio del dispositivo, aunque cifrada), y hace creer a la aplicación que cuenta con una copia de seguridad, solicitando la clave de recuperación, podrá recuperar todos los mensajes.

2.6 Bases de datos

En la actualidad, el activo más importante de las grandes empresas es la información y, como tal, esta debe almacenarse de manera eficiente y segura. Una de las formas más comunes de almacenamiento es mediante el empleo de bases de datos, que permiten un acceso rápido, estratificado y organizado a la información.

Sin embargo, cuando se trata de seguridad, es necesario prestar atención a una gran cantidad de detalles al trabajar con estas estructuras. Por ejemplo, hay que definir una política de accesos rigurosa, cifrar la información sensible, proteger el acceso físico a la base de datos...

Como este documento enfoca la seguridad desde el punto de vista de la criptografía, no pretendemos que sea tomado como una referencia suficiente para proteger una base de datos, sino necesaria.

2.6.1 Cifrado de una base de datos

Alahora de cifrar el contenido de una base de datos, tenemos que tener en cuenta tanto las ventajas y los inconvenientes de nuestras acciones, como nuestro objetivo final.

Por un lado, debemos establecer una separación entre los datos realmente sensibles (cuya filtración supone un grave problema) y los datos corrientes. Esto es porque cuanta más información cifremos, más difícil y lento será el acceso a la base de datos. Obviamente, si ciframos la base de datos al completo estaremos alcanzando un nivel óptimo de seguridad, pero estaremos sacrificando velocidad para ello.

Si miramos la estructura de una base de datos, una primera aproximación consiste en cifrar una columna. Esta es una funcionalidad que muchos gestores incorporan hoy en día: el administrador aporta una contraseña de la que se deriva una clave que sirve para cifrar/descifrar una columna en concreto.

Por otro lado, tenemos que tener en cuenta que hay cierta información que no merece la pena cifrar, ya que se puede trabajar con ella de formas más eficientes y seguras. Este es el caso, por ejemplo, de las contraseñas. Si el cliente envía la contraseña a la base de datos, obviamente se trata de información sensible que no debe almacenarse en texto plano.

2.6.2 Gestión de contraseñas

Uno de los usos más comunes de las bases de datos es el de autenticar a un usuario, por ejemplo, mediante un sistema de *login* basado en una tupla usuario:contraseña. Por lo tanto, como acabamos de comentar, es especialmente importante tratar adecuadamente las contraseñas en nuestra base de datos.

En general, la idea en la que se basa la correcta gestión de contraseñas es que **nadie** (ni siquiera el administrador) debe ser capaz de extraer una contraseña de la base de datos. Por lo tanto, no podemos almacenarlas ni en texto plano, ni usando una función criptográfica reversible (como uncifrado).

La forma más segura de almacenar este tipo de información es usando una función *hash*, de modo que no se pueda recuperar su contenido. Sin embargo, las funciones *hash* son deterministas (la misma entrada siempre genera la misma salida), por lo que dos usuarios que usen la misma contraseña generarán el mismo *hash*, dando pistas al atacante. Además, ese determinismo hace que sea mucho más fácil aplicar métodos de fuerza bruta para obtener la contraseña asociada a un *hash*.

La solución que resuelve este problema es el uso de *salts*, que son secuencias aleatorias que se introducen al principio de la contraseña. Estos *salts* pueden almacenarse en texto plano sin suponer un problema, de modo que para comparar la contraseña, se aplica el *hash* a la secuencia “*salt //password*”.

Por otro lado, para evitar ataques de fuerza bruta, nos interesa añadir un factor de complejidad que afecte exponencialmente a la eficiencia del ataque. A este procedimiento se le denomina *key stretching*. Existen algoritmos que implementan este *key stretching*, como PBKDF2, el cual utiliza una función *hash* sobre un *salty* una contraseña, y al resultado le vuelve a aplicar el *hash* junto con el mismo *salt*, tantas veces como queramos.

Por lo tanto, una posible gestión segura de contraseñas puede resumirse en los siguientes pasos:

1. El servidor recibe la contraseña por primera vez, pero no la almacena en texto plano.
2. Se calcula un *salt* aleatorio.
3. Se calcula $PBKDF2(Salt, Password, nTimes)$ n-veces (idealmente, del orden de decenas de miles).
4. Se almacena en la base de datos el *Salt*, la *Password* y el valor *nTimes* que indica el número de iteraciones de *PBKDF2*.
5. Para corroborar un inicio de sesión, el servidor recibe una contraseña en texto plano y calcula el valor $PBKDF2(Salt, Password, nTimes)$, comparándolo con el almacenado.

2.6.3 Seguridad de la conexión

Observando detenidamente los pasos descritos para almacenar de forma segura las contraseñas, nos daremos cuenta de que el servidor las recibe en texto plano (junto con toda la información que se intercambia con la base de datos). Por lo tanto, resulta crucial establecer conexiones seguras cliente-servidor para evitar ataques MITM que tiren por los suelos nuestros intentos de proteger la información.

2.6.4 Gestión de claves

De nuevo, al cifrar el contenido de una base de datos nos enfrentamos al problema de la gestión de claves. Debemos almacenarlas en un entorno aislado y protegido para evitar su filtración. Más adelante hablaremos de los gestores de claves (KMS), pero básicamente se trata de servicios que almacenan claves de forma segura, y que además permiten la creación y el intercambio de lasmismas.

Un ejemplo de uso de un KMS sería el siguiente:

1. Hemos cifrado una columna sensible de nuestra base de datos usando una clave `secret_key` generada aleatoriamente.
2. Ciframos la `secret_key` usando el KMS.
3. Almacenamos en la base de datos la `secret_key` cifrada.
4. Para descifrar de nuevo el contenido, desciframos la `secret_key` usando de nuevo el KMS.

Existen, sin embargo, otros mecanismos para utilizar claves de forma segura, como el uso de contraseñas (que idealmente sólo las conoce el usuario y no se almacenan en ninguna otra parte).

2.6.5 Integridad de la base de datos

Otro de los problemas que preocupa a los administradores es el de asegurar que la información que se encuentra en la base de datos no ha sido modificada sin autorización. Para ello, se busca incorporar mecanismos de integridad, como los códigos MAC.

Un enfoque bastante bueno consiste en incluir un MAC en cada fila que controle todos los elementos de la misma (no sólo los sensibles).

Pongamos como ejemplo una base de datos que almacene los campos: `USER_ID`, `USERNAME`, `HASHED_PASSWORD`, `BIRTHDATE`. Una forma de garantizar la integridad de los mismos sería añadir una nueva columna, denominada MAC, cuyo valor se calcule como `HMAC(USER_ID, USERNAME, HASHED_PASSWORD, BIRTHDATE)`.

De este modo, cuando un usuario acceda a la información, podrá comprobar que su contenido no ha sido modificado maliciosamente.

2.6.6 Seguridad externa

Hasta ahora nos hemos centrado en cifrar información almacenada en la base de datos, pero resulta igual de importante establecer ciertos criterios de seguridad externos a la misma.

Por ejemplo, se puede plantear cifrar la base de datos al completo (usando herramientas de cifrado a nivel del sistema de ficheros) o incluso cifrar el medio físico en el que se encuentra la información usando, por ejemplo, BitLocker.

Sin embargo, estas soluciones, pese a aportar un grado de seguridad todavía mayor, hacen que muchas veces resulte complicado trabajar con la base de datos, por lo que es común evitar su implantación.

2.7 Ransomware

Ransom (“rescate”) *ware* (“software”) es un término que hace referencia a uno de los tipos de *malware* más relevante de los últimos años. Este tipo de programa cifra ficheros del disco duro de una víctima, solicitando un rescate (generalmente, en forma de criptomonedas) a cambio de las claves de descifrado. En la mayoría de los casos, este tipo de *malware* cuenta también con una serie de métodos de propagación.

Entre los casos más sonados se encuentran nombres como *Petya*, *Reventon*, *CryptoLocker* y *WannaCry* (siendo este último el que afectó a numerosas empresas españolas, como Telefónica, en el año 2017).

2.7.1 Técnicas de cifrado empleadas

El *Ransomware* no ha sido siempre tan sofisticado como lo es hoy en día. De hecho, en la mayoría de los ataques producidos hasta la fecha se han encontrado errores de implementación que han llevado a detener la expansión del virus. En concreto, muchos de los errores dependían del método de cifrado empleado:

1. **Cifrado simétrico puro:** este primer intento empleaba cifrados simétricos. Cuando se instalaba el virus, se generaban claves aleatorias para cada archivo a cifrar y se almacenaban en algún directorio “oculto” en texto plano. Tenía la ventaja de que el proceso de cifrado podía realizarse sin conexión, por lo que desconectar el ordenador de la red no detenía el ataque. Sin embargo, resultaba bastante sencillo para los analistas recuperar las claves y descifrar los archivos infectados.
2. **Cifrado asimétrico en el cliente:** esta fue una de las soluciones menos eficientes documentadas. Consistía en generar un par de claves asimétricas en el cliente, el cual enviaba la clave privada al servidor del atacante y se deshacía de ella. De esta forma, podía cifrar los archivos con la clave pública, pero sería necesario que el atacante enviase la clave privada para poder descifrar. Este sistema requería que tanto la víctima como el atacante estuviesen *online* de forma simultánea para cifrar y, si alguno perdía la conexión, o bien se detenía el cifrado, o bien se perdían las claves privadas, haciendo imposible recuperar el contenido cifrado.
3. **Cifrado asimétrico en el servidor:** en este esquema, el servidor genera un par de claves pública-privada e incluye la pública en el código del cliente del *ransomware*. Por lo tanto, cuando la víctima paga el rescate, el atacante le envía la clave privada para poder descifrar. Este enfoque tampoco resulta práctico, ya que basta con que un único usuario pague el rescate para obtener la clave privada y distribuirla, haciendo posible el descifrado de todos los terminales infectados.
4. **Cifrado híbrido:** esta es la solución que emplea la mayoría de los *ransomwares* actuales. El servidor genera un par de claves, **Spub** y **Spriv**, e incluye **Spub** en el código del *malware*. Cuando un terminal es infectado, el cliente genera otro par de claves **Cpub** y **Cpriv**, así como una clave simétrica **key** por cada fichero a cifrar. A continuación, cifra **key** usando **Cpub**, y cifra **Cpriv** usando **Spub**. Los ficheros se cifran usando un algoritmo simétrico con la clave **key**. Tanto **key** como **Cpriv** se encuentran en el terminal, por lo que el cifrado y el descifrado se ejecutan rápidamente y sin necesidad de conexión a Internet, pero resulta muy complicado para un analista recuperar las claves, ya que ambas están cifradas y, para recuperarlas, es necesario tener acceso a la clave **Spriv** que se encuentra exclusivamente en el servidor atacante (y nunca se envía). Cuando se produce el pago, el cliente envía **Cpriv** al servidor, el cual la descifra y se la envía al cliente, que podrá recuperar sus archivos.

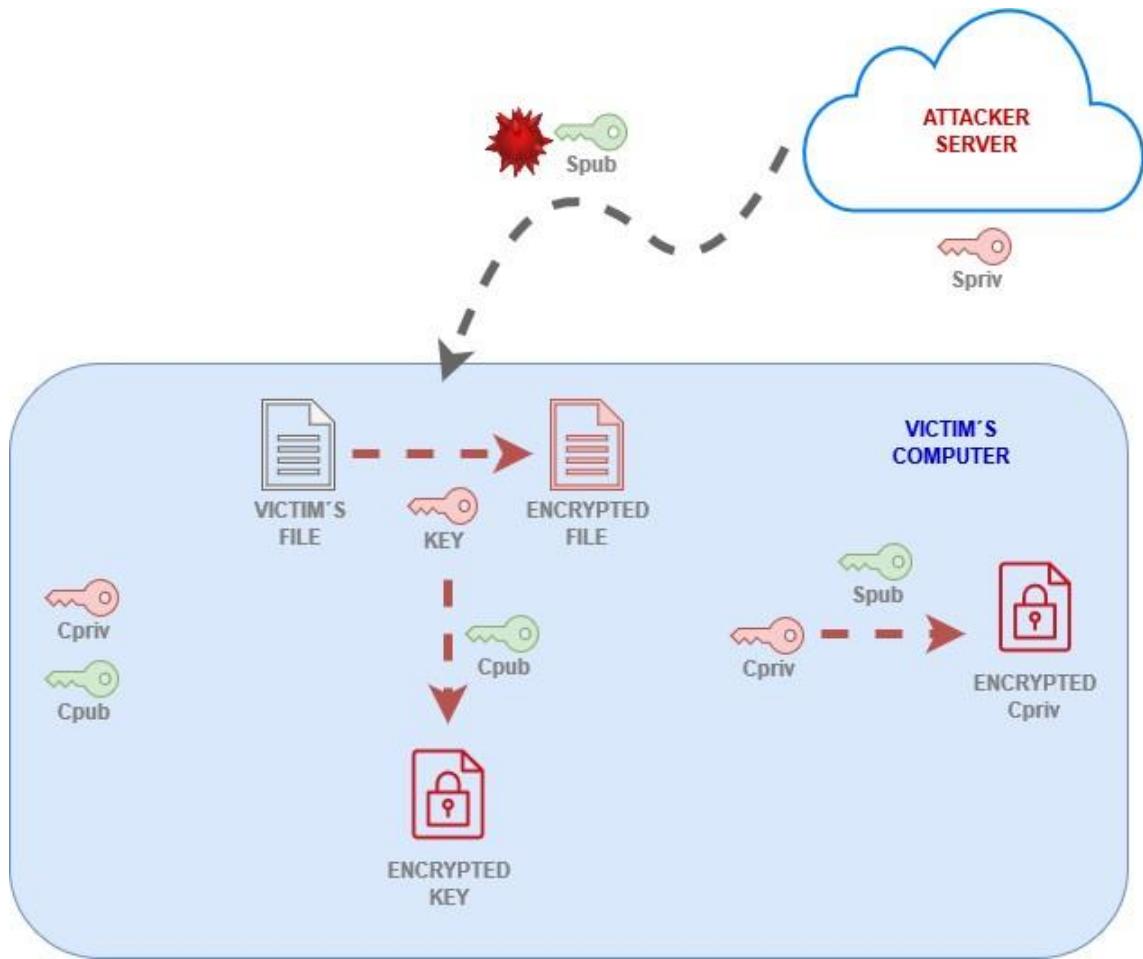


Figura 2.26 Esquema de cifrado del ransomware actual.

2.7.2 WannaCry

WannaCry ha sido uno de los ataques *ransomware* más conocidos hasta la fecha. Se produjo en 2017 y afectó a numerosas empresas y países, entre ellos España y Telefónica. En su día se pensaba que era uno de los *malwares* más sofisticados jamás creados, pero un estudio más detallado permitió encontrar algunos fallos importantes en el diseño del mismo, lo que llevó a detener su expansión e incluso, en algunas ocasiones, a revertir el cifrado.

Funcionamiento

El *malware* incluye dos claves públicas: una usada para el ataque como tal y otra para cifrar/descifrar un número limitado de archivos, a modo de demostración para la víctima. Una vez se instala el programa en el ordenador de la víctima, el cliente genera un nuevo par de claves único. En ambos casos, las claves con las que se trabaja son para RSA 2048. La clave pública del cliente se guarda en un archivo local, denominado 00000000.pky. A continuación, se cifra la clave privada del cliente usando la pública del servidor (*hardcodeada* en el código) y se elimina la clave sincifrar.

El siguiente paso es enumerar los archivos que merece la pena cifrar y, tras generar una clave de 128 bits por cada archivo, los cifra usando AES 128. Estas claves de 128 bits se almacenan cifradas usando la clave pública del cliente, y se eliminan del disco las versiones sin cifrar.

Errores

La principal característica de *WannaCry*, descubierta por el investigador conocido como *MalwareTech*, que permitió frenar el famoso ataque, fue que el *malware* comprobaba si un dominio concreto se había registrado. En caso negativo, continuaba el ataque. Por lo tanto, en el momento que *MalwareTech* registró el dominio, dejaron de producirse ataques. Más que un error de diseño, se cree que se trataba de un “botón de apagado” que el atacante había incluido en el código para detener el ataque en caso necesario.

Más adelante, también se descubrió que, siempre y cuando el ataque acabara de producirse (y no se hubiese apagado el ordenador desde entonces), era posible recuperar las claves utilizadas para el cifrado de la memoria del ordenador víctima, ya que la librería utilizada para eliminarlas no lo hacía inmediatamente, sino que dejaba residuos en memoria.

2.7.3 Conclusiones

El *ransomware* es uno de los mayores temores de las empresas en la actualidad. Sin embargo, un exhaustivo trabajo de ingeniería inversa y del esquema criptográfico empleado por el atacante puede permitirnos revertir el ataque sin que se produzcan pérdidas.

Desgraciadamente, nos sirve de poco conocer el funcionamiento de virus antiguos como *WannaCry* para protegernos, ya que cada ataque presenta unas características nuevas, haciendo imposible aplicar las técnicas de recuperación que funcionaron en el ataque anterior. Es, por lo tanto, bastante conveniente contar con profesionales formados en el análisis y la investigación en este campo que sean capaces de descubrir nuevos errores en *malwares* venideros.

3 Implementaciones

3.1 Introducción

El mundo de la criptografía se divide en dos campos: la teoría y la práctica. Cuando se diseña un algoritmo, se presta especial atención a los detalles de implementación del mismo, pero de una forma demasiado concreta. A la hora de incluir funcionalidad criptográfica en la producción de software hay que cuidar, y mucho, detalles como la gestión de claves y certificados, la programación segura o la elección de librerías.

Prácticamente nadie implementa una función criptográfica para un proyecto, sino que se recurre a librerías especializadas y probadas que optimizan los algoritmos y, si se usan correctamente, aportan un alto nivel de seguridad. Por este motivo, en el apartado actual vamos a presentar algunas de las librerías más utilizadas en varios lenguajes para el desarrollo de software.

Sin embargo, debemos advertir al lector que el código que proporcionamos no tiene como objetivo su inclusión en una aplicación, sino que se presenta con fin didáctico. De hecho, para mostrar el funcionamiento de la librería y facilitar su comprensión, hemos incluido gran cantidad de funciones que en la práctica no son seguras (como funciones de salida estándar, por ejemplo). Es necesario amoldar el código proporcionado a los requerimientos de seguridad de la aplicación en desarrollo.

Por último, como complemento a este documento, todo el código referenciado en las siguientes páginas puede encontrarse en el repositorio de GitHub <https://github.com/Nagomez97/Cryptography>.

3.2 Gestión de claves

Uno de los mayores problemas que presentan estas librerías (y la criptografía teórica, en general) es que no se especifica cómo deben manejarse las claves (tanto simétricas como asimétricas). Estas deben permanecer seguras en todo momento, pero también deben poder compartirse (en el caso de las simétricas) y ser accesibles para el descifrado.

Existe una serie de protocolos definidos para esta gestión, recogidos en estándares como los [FIPS SP 800-57](#) (partes 1 y 2). Estos protocolos establecen una serie de medidas a llevar a cabo para proteger nuestras claves, desde cifrarlas hasta incluirlas en un gestor de claves y controlar el acceso al mismo. En este apartado vamos a tratar algunas de estas medidas, pero, de nuevo, recomendamos encarecidamente al lector que acceda a la documentación oficial para implementar correctamente los protocolos necesarios.

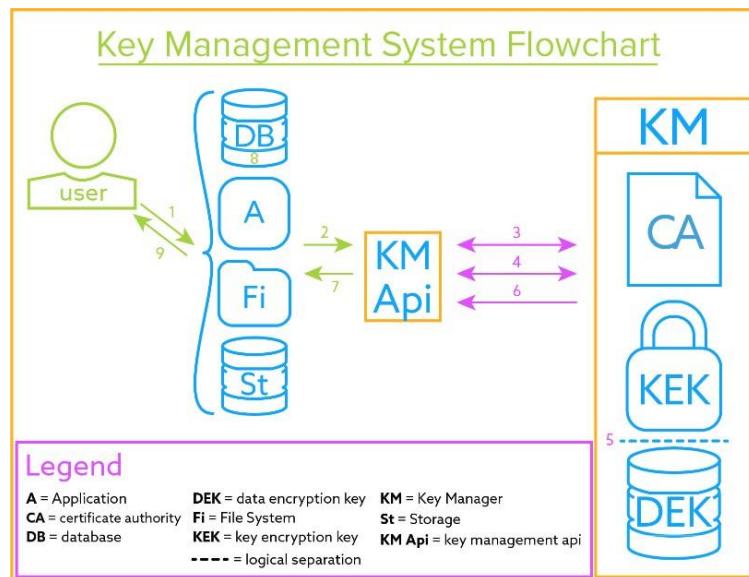


Figura 3.1 Diagrama de flujo de un Sistema de Gestión de Claves.

Como se puede apreciar en el diagrama superior, el flujo de un Sistema de Gestión de Claves (KMS, por sus siglas en inglés) es el siguiente:

1. Un usuario solicita un recurso que se encuentra cifrado.
2. La aplicación que contiene el recurso solicita a la API del *KeyManager (KM)* la Clave de Cifrado de Datos (*DEK*).
3. A continuación, la *KM API* y el gestor *KM* verifican sus certificados para autenticarse mutuamente.
4. Se establece una conexión TLS entre la API y el gestor.
5. El gestor, que almacena las *DEK* cifradas, la descifra usando una Clave de Cifrado de Claves (*KEK*).
6. El *KM* le envía la *DEK* a la API a través de la conexión segura.
7. La API le envía la *DEK* a la aplicación que lo solicitó.
8. Puede que la aplicación almacene esta clave en caché para, por ejemplo, mantener una sesión de cifrado simétrico.
9. La aplicación puede ahora descifrar el recurso solicitado por el usuario.

3.2.1 Ciclo de vida de las claves

El NIST define el ciclo de vida de las claves dividiéndolo en cuatro etapas: pre-operacional, operacional, post-operacional y eliminación. Establece también la necesidad de definir un *criptoperíodo* de la clave, tiempo durante el cual el uso de dicha clave está autorizado. Este criptoperíodo estará determinado por la combinación del tiempo durante el cual el emisor estará cifrando datos (*Originator Usage Period, OUP*) y el tiempo durante el cual el receptor los descifrará (*Recipient Usage Period, RUP*).

En general, cuanto más sensible sea la información a almacenar, más efímero será su criptoperíodo. De hecho, puede darse el caso de que el tiempo de vida de una clave sea inferior al tiempo durante el cual un usuario tiene acceso a un recurso, por lo que es necesario almacenar las claves caducadas para poder usarlas únicamente en el proceso de descifrado (esta sería la fase post-operacional). Sin embargo, si se ha usado una clave caducada para acceder a un recurso, este se volverá a almacenar usando la clave actual del sistema.

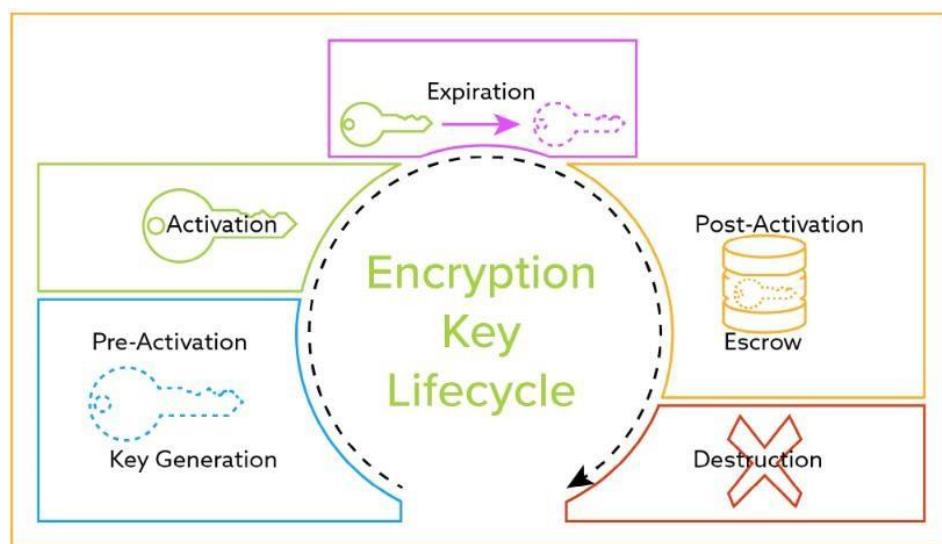


Figura 3.2 Ciclo de vida de una clave.

Por otro lado, es importante gestionar la cantidad de recursos controlados por la misma clave: cuantos menos sean, mejor. Esto se debe a que, en caso de filtrarse una clave, esta debe tener acceso al menor número de recursos posible.

Además, un administrador debe ser capaz de acceder al gestor para revocar una clave que considere insegura o que haya sido comprometida.

3.2.2 Seguridad del Gestor de Claves

Es crucial que el gestor ofrezca un entorno seguro donde almacenar las claves. Esto incluye seguridad tanto física como lógica.

SEGURIDAD FÍSICA

En las publicaciones del NIST se especifican los criterios que se deben cumplir para garantizar la seguridad física de un recurso, en concreto de un gestor de claves. Estos criterios incluyen:

- Control del acceso físico a los recursos, reduciéndolo al menor número de gente posible.
- Uso dedicado de puertos físicos.
- Integridad estructural frente a amenazas ambientales, como terremotos, incendios, inundaciones...
- Alternativas frente al fallo utilitario, como cortes de luz o exceso de temperatura.
- Cifrado apropiado de todas las transmisiones.
- Establecer permisos estrictos para el acceso remoto.
- Emplear hardware dedicado (*Hardware Security Modules, HSMs*).

SEGURIDAD LÓGICA

La idea de los criterios de seguridad lógica radica en la separación de recursos. La clave que controla un recurso debe estar separada (preferiblemente, de forma física) del recurso como tal. Del mismo modo, la clave que controla otras claves debe estar igualmente aislada.

A parte de estos dos conceptos de seguridad, es necesario establecer una serie de roles de usuario para el acceso a los recursos. El NIST promulga el uso del privilegio mínimo, donde cada usuario tiene acceso a los recursos estrictamente necesarios para desempeñar su función.

3.2.3 Gestores de Claves

Llegados a este punto, es necesario que nos planteemos el tipo de gestor que queremos usar.

Existen librerías que incorporan gestores, como el proporcionado por Java (JKS), lo que facilita bastante su uso. Sin embargo, no es la solución más segura para una aplicación grande o sensible, ya que en estos casos se tiende a obviar los criterios de aislamiento (muchas veces la aplicación se ejecuta en el mismo entorno que el gestor de claves). Por otro lado, el gestor proporcionado por la librería necesita una clave de acceso que, o bien la introduce el usuario cada vez que quiera acceder a un recurso, o bien la *hardcodeamos* en la propia aplicación (lo cual, obviamente, supone un problema de seguridad abismal).

Es por todo esto que los KMS incorporados y autogestionados pueden ser una buena idea para proyectos pequeños o pruebas de concepto, pero rara vez deberán ser utilizados en producción (a no ser que se establezca una arquitectura de gestión de claves segura, siguiendo por ejemplo las indicaciones del NIST).

Por lo general, una solución más sencilla consiste en contratar un servicio de gestión de claves externo. Esta clase de servicios, si han sido validados por una entidad confiable, cumplen con todos los requisitos mencionados anteriormente y, además, aseguran la privacidad del contenido. Se trata de una muy buena opción en desarrollo, ya que no requiere dedicar infraestructura para gestionar un KMS propio y aporta un nivel bastante alto de protección, ya que la aplicación únicamente tiene acceso a la clave de un recurso en concreto y durante un tiempo muy limitado. Algunos ejemplos de servicios KMS son *Amazon Web Services KMS* o *Amazon Web Services Vault*.

3.3 Java 11

3.3.1 javax.crypto (JCA)

La librería **javax.crypto** está incluida por defecto en todas las versiones posteriores a la JDK 1.4 y forma parte de la *Java Cryptographic Architecture (JCA)*. La idea tras esta librería es que una aplicación no necesite implementar sus propias funciones criptográficas, sino que el desarrollador pueda confiar en que la plataforma Java las ejecute de manera segura. Además, como todas las clases que emplea forman parte del *Java Development Kit (JDK)*, no necesita instalar ningún paquete extra para funcionar.

La arquitectura se basa en **Cryptographic Service Providers** (CSP), que son paquetes que implementan servicios criptográficos (como algoritmos de cifrado o de firma, funciones hash...), de modo que el programador tendrá acceso a una interfaz que le permita interactuar con el CSP seleccionado (por defecto, Java incluye SunJCE). Esta estructura permite modificar de forma transparente toda la lógica criptográfica, sin necesidad de cambiar el código de nuestras aplicaciones. Además, los CSPs son interoperables, por lo que los recursos generados por uno pueden ser utilizados por cualquier otro.

Debido a su extensión y complejidad, no vamos a entrar en detalle en la arquitectura de la librería. Sin embargo, recomendamos encarecidamente al lector recurrir a la documentación oficial de Oracle antes de incluir esta librería en su aplicación.

El proveedor por defecto de Java JCE, llamado **SunJCE** incluye las siguientes implementaciones criptográficas (entre paréntesis aparecen los nombres con los que las reconoce el proveedor):

Algoritmos de Cifrado Simétrico

1. DES para claves de 56 bits (DES)
2. AES para claves de 128, 192 y 256 bits. (AES)
3. RC2 (RC2).
4. Triple DES para claves de 112 bits (DESede).
5. Blowfish para claves de 56 bits (Blowfish).
6. ChaCha20 (ChaCha20).
7. ChaCha20 + Poly1305 para MAC (ChaCha20-Poly1305).

Esquemas de relleno

1. PKCS5 (PKCS5Padding)
2. ISO10126 (ISO10126Padding)

Modos de Cifrado

1. ECB
2. CBC
3. CFB
4. OFB
5. PCBC

Algoritmos de Cifrado Asimétrico

1. RSA.
2. Diffie-Hellman con claves de 104 bits por defecto.

Hashing

1. MD5 de 64 bytes por defecto.
2. SHA1 de 64 bytes por defecto.

DETALLES DE LA LIBRERÍA

Vale, sí... Esta librería está muy bien pero, ¿cómo se usa?

En primer lugar, es Java. Será necesario trabajar con clases e importar un montón de librerías, pero merecerá la pena. En segundo lugar, **OWASP** tiene [documentación que recoge el uso correcto de esta librería](#), así que vamos a basarnos en sus ejemplos y justificaciones.

En general, cuando queramos implementar un cifrado simétrico, debemos generar una clave simétrica *segura*, lo que implica usar un generador de claves que incluya una fuente de aleatoriedad. La buena noticia es que JCE incorpora su propio *KeyGenerator* que, además de generar claves seguras, funciona para todos los algoritmos simétricos que admite nuestro *provider*.

Otro detalle bastante importante es almacenar la clave simétrica: debe permanecer protegida en todo momento. Para ello, *java.security* cuenta con un gestor de claves denominado JKS (Java Key Stores), que se implementa usando la clase *KeyStore*. Este gestor admite formatos como el PKCS12, basado en el estándar. Además, SunJCE cuenta con su propio gestor (que admite más algoritmos), JCEKS, que se encuentra en *com.sun.crypto.provider.JceKeyStore*. Además, almacena las claves usando cifrado TripleDES. Existe una gran variedad de gestores de claves, cada uno ideal para una situación distinta. Por ejemplo, PKCS12 es un gestor portable, capaz de operar con Java, C, C++ o C#; PKCS11 es otro gestor que ofrece una interfaz para conectar con gestores hardware, como *SafeNet's Luna o nCipher*; Windows-MY, por su parte, es un gestor operado por Windows, aunque necesita incorporar una API externa para funcionar.

Por otro lado, la mayoría de modos de operación simétricos requieren el uso de un vector de inicialización que, como ya explicamos, debe ser generado con una fuente de entropía y aleatoriedad máximas. Java cuenta con una herramienta llamada *SecureRandom*, incluida en *java.security.SecureRandom* que soluciona este problema con bastante eficacia. De hecho, la implementación de **OWASP** genera los IVs usando esta herramienta.

En caso de usar un vector de inicialización, este debe almacenarse para el descifrado. Por eso, en general, se suele concatenar el texto cifrado a continuación del IV, para después codificarlo y enviarlo/almacenarlo.

Por último, es importante destacar el uso de la clase *Cipher*, que nos aporta la instancia para cifrar un mensaje, y nos permite especificar el algoritmo a utilizar, el modo de operación y hasta el esquema de relleno que queremos usar.

3.3.2 *Bouncy Castle*

Este paquete de Java incluye una amplia gama de implementaciones de algoritmos criptográficos, siendo generalmente la librería más empleada en criptografía en Java. Recibe actualizaciones de forma habitual y cuenta con licencia *Open Source*.

En comparación con JCA, *Bouncy Castle* tiene la ventaja de incorporar muchos más algoritmos, sin la necesidad de depender de diversos proveedores. Sin embargo, como este paquete no forma parte del JDK, es necesario incluirlo en el proyecto. Para ello, basta con añadir el *.jar* de la librería al proyecto e incluir los paquetes *java.security.Security* y *org.bouncycastle.jce.provider.BouncyCastleProvider*.

```
wget https://www.bouncycastle.org/download/bcprov-jdk15on-154.jar
mv bcprov-jdk15on-154.jar bouncycastle.jar
```

A la hora de correr el código java, tendremos que crear las clases con el comando:

```
javac -cp ../bouncycastle.jar <nombre del fichero .java>
```

Y ejecutar de la siguiente forma:

```
java -cp ../bouncycastle.jar:. GCMencrypt <mensaje a cifrar>
```

Por otro lado, la JCA ha sido objeto de numerosos estándares y estudios de diseño seguro. Por lo tanto, en la medida de lo posible se recomienda usar la JCA para producción, salvo que el algoritmo requerido no se encuentre en un proveedor adecuado.

Algoritmos de cifrado simétrico (no soportados por SunJCE)

1. Camellia
2. IDEA
3. RC4
4. Grain
5. Salsa20
6. ChaCha20 (64 bits de IV)

Algoritmos de cifrado asimétrico / Intercambio de clave

1. ECDH
2. DH
3. ElGamal
4. NTRU

Hashing

1. MD5
2. SHA1
3. SHA2
4. SHA3
5. RIPEMD
6. Whirlpool
7. Streebog
8. BLAKE2

3.3.3 Comparación

Proveedor	Ventajas	Inconvenientes
SunJCE	<ul style="list-style-type: none"> - Incluido en la JCA (no necesita instalación). - Actualización periódica. - Incluye los algoritmos más utilizados (RSA y AES-GCM). 	<ul style="list-style-type: none"> - Pocos algoritmos. - Longitudes de clave limitadas (algunos algoritmos están diseñados para admitir ciertas longitudes que SunJCE no permite).
BouncyCastle	<ul style="list-style-type: none"> - Incluye casi todos los algoritmos existentes. - Incluye otros modos de operación, como GCM. - Actualización periódica. - Se puede incorporar como un proveedor de la JCA (usándose de manera idéntica a JCE). 	<ul style="list-style-type: none"> - Requiere descargar e instalar el paquete .jar de la librería.

3.3.4 AES usando SunJCE

Como prueba de concepto del proveedor SunJCE hemos implementado AES, ya que los demás algoritmos simétricos presentes en la librería se usan de forma bastante similar, y AES es el más empleado de todos. Hemos decidido tomar claves de 256 bits y usar el modo CFB (por defecto se usa ECB, pero no es seguro, así que no vamos a tenerlo en cuenta para la implementación). Este modo requiere añadir un IV aleatorio.

Por otro lado, tenemos la opción de incluir un esquema de relleno o no hacerlo. La única diferencia es que al añadir relleno al mensaje, el atacante no tendrá información de la longitud del mismo. Por eso, en la medida de lo posible, es recomendable usarlo. En este caso hemos tomado PKCS7 como esquema de padding, pero debido a un error en la librería, aparece como PKCS5.

La implementación planteada se encuentra en el repositorio de github mencionado al inicio del capítulo, en el directorio `/Java/JCE/AES`, con los nombres `AESencrypt.java` y `AESdecrypt.java`.

Cifrado

En primer lugar, debemos generar una clave secreta segura, para lo que necesitamos crear una instancia del *KeyGenerator* especificando el algoritmo que estamos usando en el cifrado (AES). Después de eso bastará llamar a la función *generateKey()* para obtener una clave, habiendo configurado previamente la longitud de la clave (256 bits).

```
// Step 1: generate keyLength-key
KeyGenerator keyGen = KeyGenerator.getInstance(encMode);
keyGen.init(keyLength);
SecretKey secretKey = keyGen.generateKey();
```

Figura 3.3 Generación de claves AES usando JCE.

A continuación debemos instanciar el *Cipher*, especificando las transformaciones que queremos emplear (AES/CFB/PKCS5PADDING) e inicializar el IV usando la funcionalidad del *SecureRandom* para aportar la entropía suficiente.

```
// Step 2: get the cipher instance with the selected mode
Cipher aesCipherForEncryption = Cipher.getInstance(transformationString);

// Step 3: Initialize the IV using a secure random function.
byte[] iv = new byte[aesCipherForEncryption.getBlockSize()];
SecureRandom prng = new SecureRandom();
prng.nextBytes(iv);
```

Figura 3.4 Inicialización del IV en AES usando JCE.

Por último, inicializamos el cifrador en modo *ENCRYPT*, añadiéndole la clave secreta y el IV, de modo que llamando al método *doFinal* obtendremos el mensaje cifrado. Como ya indicamos, al usar un modo de operación que emplea IVs, es necesario concatenar el texto cifrado tras el IV y codificar el resultado, de forma que el receptor pueda recuperar el vector y así descifrar correctamente el mensaje.

```
// Step 4
aesCipherForEncryption.init(Cipher.ENCRYPT_MODE, secretKey, new IvParameterSpec(iv))

// Step 5
byte[] encrypted = aesCipherForEncryption.doFinal(message.getBytes(charEnc));
ByteBuffer cipherData = ByteBuffer.allocate(iv.length + encrypted.length);
cipherData.put(iv);
cipherData.put(encrypted);
cipherText = new String(Base64.getEncoder().encode(cipherData.array()), charEnc);
System.out.println("Encrypted and encoded message is: " + cipherText);
```

Figura 3.5 Código de ejemplo del cifrado de AES.

Transportando la clave

En el código de ejemplo, el ejecutable *AESencrypt* muestra la clave secreta por pantalla codificándola en base 64, de forma que al llamar a *AESdecrypt* basta con introducirle como argumento la clave y el mismo ejecutable la decodificará y generará un objeto *SecretKey* con el que puede trabajar nuestro *Cipher*.

Como ya hemos mencionado, esto no es nada recomendable, ya que asumimos que en el trayecto entre el cifrado y el descifrado la clave se encuentra en texto plano. Existen APIs que se pueden llamar desde estos ejecutables para exportar e importar claves desde un gestor de forma bastante sencilla.

Descifrado

El proceso es bastante similar al de cifrado, con la diferencia de que, en nuestro caso, recibimos el texto cifrado y la clave como argumentos, y debemos decodificarlos para su uso. En el caso de la clave, es necesario convertirla a la clase `SecretKey` usando la función `SecretKeySpec` e indicando el algoritmo empleado (AES).

```
//Step 1: Get the key and decode it.
keyBytes = Base64.getDecoder().decode(encodedKey.getBytes(charEnc));
secretKeySpec = new SecretKeySpec(keyBytes, encMode);
```

Figura 3.6 Obteniendo la clave en AES.

Una vez decodificado el texto cifrado, recordamos que al comienzo aparecerá el IV, por lo que debemos separarlo del mensaje cifrado y emplearlo en el proceso de descifrado. A partir de este momento, el funcionamiento es idéntico al del cifrado, usando el `Cipher` en modo `DECRYPT_MODE`.

```
// Step 2: get the cipher instance.
Cipher aesCipherForDecryption = Cipher.getInstance(transformationString);

// Step 3: get the decoded cipher data and the IV (first block)
ByteBuffer cipherData = ByteBuffer.wrap(Base64.getDecoder().decode(cipherText.getBytes(charEnc)));
byte[] iv = new byte[aesCipherForDecryption.getBlockSize()];
cipherData.get(iv);
encrypted = new byte[cipherData.remaining()];
cipherData.get(encrypted);
aesCipherForDecryption.init(Cipher.DECRYPT_MODE, secretKey, new IvParameterSpec(iv));

// Step 4: decrypt the message.
byte[] decrypted = aesCipherForDecryption.doFinal(encrypted);
System.out.println("Decrypted text message is: " + new String(decrypted, charEnc));
```

Figura 3.7 Código de ejemplo para el descifrado en AES.

3.3.5 RSA usando SunJCE

El caso de RSA es ligeramente distinto al de los demás algoritmos incluidos en este proveedor, principalmente por el uso de criptografía asimétrica para el intercambio de claves.

Existen diversos usos de RSA: como cifrado asimétrico, como método para intercambiar claves, como algoritmo de firma digital... Sin embargo, el más extendido es una combinación de todos ellos, tal y como se recoge en muchas implementaciones de TLS. Normalmente, se emplea el algoritmo asimétrico como una forma de intercambiar una clave simétrica que pueda usarse durante una sesión de comunicación completa. Para ello, suponemos que cliente y servidor han intercambiado ya sus certificados y los han validado, por lo que cada uno tiene la clave pública del contrario.

En el modelo que proponemos a continuación, el emisor del mensaje genera la clave secreta y la cifra usando la clave pública del receptor, de forma que se asegura la confidencialidad del contenido. Es cierto que en este modelo no se autentica al emisor, pero eso es algo que TLS arregla en su protocolo de *handshake*. Una forma de asegurar esta autenticidad sería que el emisor intercambiase una firma digital con el receptor, por ejemplo cifrando la clave secreta usando su propia clave privada, y posteriormente cifrase el resultado usando la clave pública del receptor.

Generación de las claves

El ejecutable *RSAkeygen.java* generará dos ficheros: *RSAkey.key* y *RSAkey.pub*; una clave privada y una pública, respectivamente. Para ello hará uso de una instancia de *KeyPairGenerator* y codificará los resultados para poder exportar e importar las claves desde Java.

Es importante tener en cuenta que las claves generadas tienen formato *PKCS8* (la privada) y *X.509* (la pública).

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance(encMode);
kpg.initialize(keyLength);
KeyPair kp = kpg.generateKeyPair();
Key pub = kp.getPublic();
Key pvt = kp.getPrivate();

//Exporting keys in binary format
String outFile = "./RSAkey";
FileOutputStream out = new FileOutputStream(outFile + ".key");
out.write(pvt.getEncoded());
out.close();
out = new FileOutputStream(outFile + ".pub");
out.write(pub.getEncoded());
out.close();
```

Figura 3.8 Generación de claves RSA.

Cifrado

Como hemos explicado, el modelo que estamos implementando necesita generar una clave simétrica para AES-256 (de forma idéntica al ejemplo anterior) para cifrar el mensaje. Esta clave simétrica será cifrada con la clave pública del receptor y enviada (en nuestro caso, impresa por pantalla). En este ejemplo estamos usando RSA con un esquema de relleno PKCS1Padding y el modo de operación ECB (en RSA no resulta inseguro usar ECB, ya que el cifrado del mensaje se realiza usando AES/CFB).

Es necesario obtener la clave pública del destinatario del mensaje, que importaremos en formato X.509 y transformaremos a un objeto del tipo *PublicKey*. Esta clave será la que usemos para cifrar la clave simétrica.

Una vez cifrada la clave simétrica, procedemos a cifrar el mensaje usando la misma metodología que en el ejemplo de AES.

```

//Generating AES key and IV
KeyGenerator kgen = KeyGenerator.getInstance(encMode);
kgen.init(symmetricKeyLength);
SecretKey skey = kgen.generateKey();
String cipherText;

SecureRandom srandom = new SecureRandom();
byte[] iv = new byte[symmetricKeyLength/8];
srandom.nextBytes(iv);
IvParameterSpec ivspec = new IvParameterSpec(iv);

//Loading RSA key and encrypting secret key
byte[] bytes = Files.readAllBytes(Paths.get(keyFile + ".pub"));
X509EncodedKeySpec ks = new X509EncodedKeySpec(bytes);
KeyFactory kf = KeyFactory.getInstance("RSA");
PublicKey pub = kf.generatePublic(ks);

Cipher cipher = Cipher.getInstance(cipherInstance);
cipher.init(Cipher.ENCRYPT_MODE, pub);

byte[] b = cipher.doFinal(skey.getEncoded());

```

Figura 3.9 Cifrado de la clave simétrica en RSA.

Descifrado

El descifrado en RSA requiere contar con la clave privada del receptor, por lo que únicamente esta persona podrá acceder al contenido. El primer paso es importar la clave privada en formato PKCS8.

```

//Loading RSA public key
byte[] bytes = Files.readAllBytes(Paths.get(keyFile + ".key"));
PKCS8EncodedKeySpec ks = new PKCS8EncodedKeySpec(bytes);
KeyFactory kf = KeyFactory.getInstance("RSA");
PrivateKey pvt = kf.generatePrivate(ks);

```

Figura 3.10 Importando la clave privada en Java para RSA.

A continuación, se descifra la clave secreta, permitiéndonos acceder al mensaje. Esta última parte de descifrado es idéntica a la presentada en el apartado anterior, ya que hemos estado usando AES-256 para tratar el contenido.

```

Cipher cipher = Cipher.getInstance(cipherInstance);
cipher.init(Cipher.DECRYPT_MODE, pvt);

//Decrypting AES secret key using public key
keyBytes = Base64.getDecoder().decode(encodedSecretKey.getBytes(charEnc));
SecretKey skey = new SecretKeySpec(cipher.doFinal(keyBytes), "AES");

//AES decryption
Cipher aesCipherForDecryption = Cipher.getInstance(transformationString);

ByteBuffer cipherData = ByteBuffer.wrap(Base64.getDecoder().decode(cipherText.getBytes(charEnc)));
byte[] iv = new byte[128/8];
cipherData.get(iv);
encrypted = new byte[cipherData.remaining()];
cipherData.get(encrypted);
aesCipherForDecryption.init(Cipher.DECRYPT_MODE, skey, new IvParameterSpec(iv));

byte[] decrypted = aesCipherForDecryption.doFinal(encrypted);
System.out.println("Decrypted text message is: " + new String(decrypted, charEnc));

```

Figura 3.11 Descifrado de RSA usando Java.

3.3.6 AES-GCM usando BouncyCastle

Una de las ventajas de BouncyCastle es que incluye una gran cantidad de algoritmos y variaciones que no están implementadas en el proveedor SunJCE. Entre ellos encontramos el algoritmo AES-GCM, que como ya vimos es un tipo de AEAD (*Authenticated Encryption with Associated Data*), lo que garantiza la integridad del mensaje. Además, esta variación permite tratar AES como un cifrador de flujo, por lo que es uno de los métodos más usados en tecnologías como TLS.

En general, AES-GCM usa un *nonce* junto con un contador a modo de IV, que es lo que cifra usando una clave simétrica, mientras que los bloques del mensaje se añaden con un XOR y se vuelven a cifrar. El resultado, además del texto cifrado, incluye un *auth tag* que garantiza su integridad. De esta forma, si al descifrar el mensaje se obtiene un *auth tag* distinto, sabremos que algo ha ido mal.

BouncyCastle automatiza esta funcionalidad, por lo que únicamente debemos preocuparnos del mensaje, el IV (que, al igual que en los casos anteriores, se encuentra al comienzo del texto enviado) y la clave, de forma que la herramienta lanzará una excepción en caso de observar un cambio en el *tag*.

Cifrado

El hecho de incorporar un nuevo proveedor modifica ligeramente el código, aunque la estructura sigue siendo la misma que la del AES de SunJCE (ambos proveedores trabajan sobre la JCA). La principal diferencia radica en la necesidad de decirle a Java que queremos usar un nuevo proveedor.

```
Security.addProvider( new org.bouncycastle.jce.provider.BouncyCastleProvider() );
```

Figura 3.12 Gestiónando el proveedor BouncyCastle.

Además, tanto en la instanciación del *KeyGenerator* como en la del *Cipher* debemos especificar el uso del nuevo proveedor *BC*.

```
// Step 1: generate the key
KeyGenerator keyGen = KeyGenerator.getInstance(encMode, "BC");
keyGen.init(keyLength);
SecretKey secretKey = keyGen.generateKey();

System.out.println("Symmetric key used: " + new String(Base64.getEncoder().encode(secretKey.getEncoded()), charEnc));

// Step 2: Get the cipher instance with the selected mode
Cipher aesCipherForEncryption = Cipher.getInstance(transformationString, "BC");
```

Figura 3.13 Especificación del proveedor en las instancias.

A continuación, generaremos el *nonce* de 12 bytes usando la función *SecureRandom()*. Es la propia librería la que se encarga de añadir un contador y generar el IV, aunque debemos especificarlo así en la inicialización del *cipher*:

```
// Step 4
aesCipherForEncryption.init(Cipher.ENCRYPT_MODE, secretKey, new GCMParameterSpec(128, nonce));
```

Figura 3.14 Inicialización del cifrador para AES-GCM.

Aquí, **128** es la longitud en bits del *auth tag*. Sólo se asegura el correcto funcionamiento del algoritmo para valores de {96, 104, 112, 120, 128}, aunque se pueden definir otros valores.

El resto del código es idéntico al presentado en AES: llamada a *doFinal()* para obtener el texto cifrado, adición del *nonce* y codificación en Base64 para el envío.

3.3.7 Otros algoritmos

Como muchos de los algoritmos existentes se implementan de manera muy similar, no hemos incluido apartados específicos para cada uno.

Por ejemplo, Camellia y ChaCha20 usan la misma estructura que AES en BouncyCastle, cambiando únicamente el nombre del cifrado a utilizar, la longitud de los *nonces/IVs* o la longitud de clave.

En concreto, en el repositorio se incluye una implementación de ChaCha20 usando Poly1305 como MAC, con el proveedor SunJCE. La estructura de este algoritmo es idéntica a la de AES, pero es necesario trabajar con claves de 256 bits y IVs de 12 bytes.

Por otro lado, si queremos implementar algún otro método de generación de pares de claves pública/privada, como ECDH, basta con tomar la estructura de *RSAkeygen*. En el repositorio hemos incluido un ejemplo de generación de claves usando ECDH.

Es por ese motivo que creemos que, con las nociones explicadas en los apartados correspondientes, el lector tiene material suficiente para implementar cualquier algoritmo presente en la librería.

En los siguientes enlaces se encuentran las especificaciones de ambos proveedores, incluyendo los identificadores de los algoritmos aceptados por cada uno.

BouncyCastle

<https://www.bouncycastle.org/specifications.html>

SunJCE

<https://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html#SunJCEProvider>

Sin embargo, es necesario tener en cuenta que la documentación no está completa: por ejemplo, las especificaciones de SunJCE no incluyen los algoritmos ChaCha20 y Poly1305, pero ya hemos mencionado que se pueden implementar usando dicho proveedor. Creemos que se trata de una desactualización del documento, por lo que recomendamos al lector indagar un poco antes de añadir estas librerías a su proyecto.

3.4 EverCrypt

Se trata de una de las herramientas más prometedoras que existen. Desarrollada en el marco del Proyecto Everest (*Everest VERified End-to-end Secure Transport*) liderado por Micorosft e Inria cuyo objetivo principal es el de desarrollar herramientas para la evolución del ecosistema HTTPS y la seguridad de las comunicaciones.

EverCrypt hace uso de HACL*, que es una librería de primitivas criptográficas escritas en F* (un lenguaje enfocado a la verificación de programas). De este modo, se construye un proveedor criptográfico, dando lugar a una API *agile*, multiplataforma y autoconfigurable.

La principal ventaja que propone este proyecto es que, mientras que normalmente el ciclo de software pasa por el diseño, el desarrollo y la fase de *testing* (en ese orden), HACL* se diseña y desarrolla mientras se prueba. Por lo tanto, está íntegramente enfocado a validar toda clase de características desseguridad.

Actualmente, ofrece soporte para algoritmos como AES-GCM, ChachaPoly, MD5, SHA1/2/3, HMAC, Poly1305, HKDF, Curve25519, Ed25519...

Nos encontramos ante una herramienta criptográfica segura por definición, invulnerable a ataques de todo tipo: *side-channel*, criptoanálisis, *leaks* de información... Además, aseguran que esta librería iguala e incluso mejora la eficiencia y velocidad de las demás existentes.

Sin embargo, la herramienta se encuentra actualmente en fase de desarrollo y, aunque ya existen versiones funcionales, todavía no se recomienda su inclusión en proyectos reales. Por lo tanto, creo que es una buena idea que el lector permanezca al tanto de los avances en EverCrypt, ya que promete convertirse en la nueva librería estándar de desarrollo criptográfico.

4 Conclusiones

A lo largo de este documento hemos aprendido a identificar los distintos algoritmos y técnicas criptográficas que más relevancia han tenido en la historia: cuáles han sido fuente de graves problemas al verse comprometidos, cuáles se siguen usando hoy en día e incluso cuáles acabarán quedando obsoletos en un futuro no muy lejano.

Sin embargo, hay una serie de puntos destacables en los que merece la pena reincidir, ya que el principal objetivo de este proyecto es ayudar al lector a implementar sistemas criptográficamente seguros y a analizar un criptosistema para detectar posibles errores o vulnerabilidades.

4.1 ¿Qué algoritmo debo elegir?

Algo en lo que se ha hecho especial hincapié a lo largo del documento es que no se debe confundir algoritmo criptográfico con algoritmo de cifrado. Mientras el primero aporta un abanico de *features* de seguridad (confidencialidad, integridad, autenticación, no repudio...), el segundo únicamente garantiza la confidencialidad del mensaje. Por lo tanto, el primer paso que debemos dar es el de elegir la finalidad de nuestro sistema.

¿Bloques o flujo?

Supongamos que únicamente queremos cifrar cierta información. En ese caso, debemos tener claro qué tipo de información vamos a cifrar, y en qué contexto. Por ejemplo: si queremos cifrar un documento de texto, probablemente nos decantaremos por un algoritmo clásico de cifrado en bloque, como puede ser AES o 3DES. Si, por otro lado, buscamos desarrollar una aplicación de *streaming* de video o videollamadas (por ejemplo, Skype), lo más sensato es decantarse por un algoritmo de cifrado en flujo, siendo el más común AES-CBC (ya que aporta las características de un cifrado en flujo sin perder la rapidez y seguridad de AES), aunque ChaCha20 es una buena alternativa.

Una vez nos hemos decantado por un tipo de algoritmo, parece obvio pensar que el algoritmo más robusto es el más indicado. Sin embargo, esto no siempre es así. Debemos tener en cuenta muchos factores como, por ejemplo, el *hardware* sobre el que vamos a trabajar. Hoy en día, la mayoría de los sistemas cuentan con *hardware* optimizado para el algoritmo Rijndael (AES), ya que es el más utilizado. Por este motivo, en la mayoría de las ocasiones querremos utilizar AES.

Sin embargo, puede que trabajemos con sistemas cuyas características sean poco comunes y que requieran, por ejemplo, un consumo mínimo de energía o memoria, o una gran velocidad. En este caso, probablemente deberemos buscar un algoritmo concreto, ligero, que se acomode a nuestras necesidades.

Además, ChaCha20 ha demostrado ser un algoritmo (de flujo) bastante prometedor, y proyectos como el TLSv1.3 de Google o SSH incorporan ya a ChaCha20-Poly1305 como cifrado estándar. Esto se debe a que la simplicidad de sus operaciones (*Addition-Rotation-XOR*) lo hace algo más rápido que AES (en procesadores no dedicados) e invulnerable a ataques *side-channel* como los de *cache-timing*.

¿Simétrico o asimétrico?

Otra de las preguntas más importantes que nos tenemos que plantear antes de elegir un algoritmo de cifrado, es si queremos que sea simétrico, asimétrico o híbrido. Por lo general, los criptosistemas que nos encontramos utilizarán criptografía híbrida, ya que permite un intercambio de claves seguro entre dos partes (usando criptografía asimétrica), sin perder la velocidad que aporta un sistema simétrico. Sin embargo, esta solución no siempre es la mejor.

Si, por ejemplo, queremos cifrar un documento o un archivo .zip usando una contraseña que solo nosotros conocemos, querremos usar un cifrado simétrico, ya que no será necesario realizar un intercambio de clave.

¿Integridad?

Por lo general, un buen criptosistema no se centra únicamente en la confidencialidad, sino que permite al usuario saber si la información recibida ha sido modificada sin permiso. Para eso utilizaremos sistemas que incorporen, por ejemplo, funciones MAC. De este modo, es posible obtener algoritmos de cifrado + integridad (como CHACHA20-POLY1305-SHA256 o AES256-GCM-SHA256) o únicamente integridad (como HMAC-SHA256).

¿Autenticación?

Otra de las características de un criptosistema seguro es la autenticación: los usuarios deben poder verificar que se están comunicando con quien de verdad esperan, sin suplantaciones de identidad. Para ello hemos visto diversos métodos (*TLS handshake*, emparejamiento *Bluetooth*...) que dependen del diseño del protocolo de comunicación utilizado. Sin embargo, la criptografía asimétrica, y más concretamente, las infraestructuras de clave pública o PKIs, son una solución bastante eficiente.

El desarrollador deberá sopesar las ventajas e inconvenientes de levantar su propia PKI (coste de la gestión de certificados, servidores dedicados...) o de confiar en una externa (usando certificados firmados por CAs de confianza) o, por otro lado, utilizar protocolos que aporten soluciones específicas para el contexto en que queramos implantar la autenticación (no es lo mismo autenticar dispositivos en una red local que autenticar usuarios o empresas en Internet).

Estándar

Independientemente del tipo elegido, siempre vamos a recomendar el empleo de algoritmos que pertenezcan a un estándar actualizado. Por ejemplo, puede que AES y DES nos solucionen el mismo problema, pero AES es un estándar activo, mientras que DES es un algoritmo muy poco robusto y con claves pequeñas.

Longitud de clave

Por otro lado, puede que nos encontremos con que un mismo algoritmo nos ofrece distintas longitudes de clave. Este es un factor importante, ya que la seguridad de un sistema puede depender exclusivamente de ello. Por ejemplo, si usamos AES con claves de 128 bits, estamos facilitando que un atacante rompa nuestro algoritmo con un ataque de fuerza bruta que, aunque le llevará tiempo, tendrá éxito. Si, en su lugar, usamos claves de por lo menos 256 bits, estaremos consiguiendo que la fuerza bruta sea incapaz de vulnerar nuestro sistema.

En el Anexo 3 hemos incluido una comparativa de algoritmos de cifrado donde el lector podrá comparar las características principales de cada uno.

4.2 ¿Cómo incluyo un algoritmo en mi proyecto?

Una vez hemos decidido qué algoritmo vamos a utilizar, nos encontramos con el problema de que no sabemos cómo implementarlo. Obviamente, la solución no es coger el pseudocódigo del *paper* y escribirlo en nuestro lenguaje, sino que existen librerías que contienen una gran cantidad de algoritmos implementados y, lo que es más importante, optimizados y revisados.

Es necesario informarse previamente acerca de la librería que vamos a usar, ya que es posible que cuente con malas implementaciones o vulnerabilidades que puedan afectar a nuestro proyecto.

Además, no todas las librerías incluyen todos los algoritmos ni están implementadas en todos los lenguajes. Antes de nada, debemos tener claro si queremos priorizar el algoritmo o el lenguaje. Si, por ejemplo, estamos trabajando en Python pero queremos usar el cifrado *Grain*, nos encontraremos con que ninguna de las librerías comunes lo implementa.

En el Anexo 6 incluimos una comparativa entre las librerías más comunes en C, C++, Java y Python y los lenguajes que implementa cada una.

4.3 ¿Qué no debo hacer?

Crear mi propio algoritmo

Esta es una de las peores ideas que puede tener un desarrollador. Nunca se debe crear un algoritmo de cifrado con el fin de incluirlo directamente en producción. Esto es porque todos los algoritmos conocidos se publicaron previamente y fueron objeto de estudio durante muchos años antes de aceptarse como algoritmos estándar e incluirse en proyectos reales.

Escoger un algoritmo “sobre la marcha”

No es una buena idea esperar hasta el último momento para elegir un algoritmo criptográfico, ya que ninguno de ellos fue diseñado con la idea de ser portable. Es necesario crear un ambiente óptimo para el sistema elegido y no dar lugar a cabos sueltos o errores de diseño que permitan ataques de algún tipo.

Tomar esta guía como referencia ciega

Este documento no pretende ser un manual metódico sobre cómo diseñar un sistema criptográfico seguro, sino proveer al lector de las herramientas necesarias para hacerlo. De hecho, no existe ninguna guía que te explique paso a paso todo el proceso de diseño, ya que depende exclusivamente de las circunstancias en las que se quiera implantar.

4.4 ¿Qué sí debo hacer?

Documentación

Es necesario llevar a cabo una fase previa al diseño del criptosistema que pase por recoger las necesidades y objetivos del mismo, así como sus limitaciones. De nuevo, repetimos que es *esencial* conocer todos estos detalles para que nuestro sistema se adapte al proyecto.

El amplio abanico de protocolos y algoritmos hace que parezca muy difícil tomar una decisión, pero en la mayoría de los casos basta con tomar como referencia los estándares ofrecidos por entidades como el NIST, por lo que recomendamos contrastar la documentación de los estándares más conocidos.

Simple es mejor

Muchas veces, elegir un algoritmo en base a su robustez o su complejidad se traduce en una gran pérdida de eficiencia de nuestro sistema. Por lo tanto, si creemos que AES o RSA nos pueden solucionar la vida, no será necesario ir más allá.

Elegir el algoritmo más utilizado

Puede que pensemos que algoritmos nuevos y desconocidos sean más seguros o funcionen mejor para nuestro proyecto. Sin embargo, ante la duda, siempre es mejor recurrir a los clásicos (AES, RSA, 3DES...). Se trata de algoritmos que llevan años siendo objeto de estudio por las mentes más brillantes del planeta en esta área y, hasta hoy, no se han conseguido romper. Además, su amplia implantación hace que, en general, la mayoría de los sistemas cuenten con *hardware* y *software* preparados para trabajar optimizando estos métodos.

4.5 Trabajo futuro

Desde el comienzo del proyecto tuvimos la intención de aportar una base teórica sólida que englobara los principales algoritmos criptográficos, permitiendo al lector avanzar en esta área por su cuenta y poder incluir criptosistemas en proyectos con conocimiento de causa.

Como se ha dejado ver en varios puntos del documento, uno de los objetivos era el de incluir un capítulo dedicado al criptoanálisis, sus técnicas e implicaciones en el mundo de la criptografía moderna. Además, esperábamos hablar de las posibles formas de *romper* un algoritmo moderno de cara a poder realizar auditorías desde el punto de vista criptográfico (un gran olvidado por los expertos en *pentesting* y *White Hacking*).

Sin embargo, tras una investigación más profunda, nos hemos dado cuenta de que el criptoanálisis moderno no ha sobrepasado la barrera de lo teórico, y únicamente en situaciones demasiado concretas resulta factible llevar a cabo uno de los ataques propuestos contra sistemas modernos.

Pensándolo detenidamente, esto tiene bastante sentido ya que, si se llega a encontrar una vulnerabilidad fácil de explotar en un cifrado como AES, se tardaría bastante poco en desarrollar una herramienta con ese fin (como pasó, por ejemplo, con WEP) y, por lo tanto, nos veríamos obligados a actualizar todos los sistemas que incluyan ese algoritmo. Por lo tanto, podemos asumir que los algoritmos más utilizados hoy en día son, en cierto modo, invulnerables (de momento).

Por lo tanto, queda como proyecto personal y futuro completar este documento con información de carácter teórico que incluya, entre otras cosas, las principales técnicas de criptoanálisis.

Además, somos conscientes de la incompletitud de algunos capítulos como, por ejemplo, el capítulo 3. En él habríamos querido incluir ejemplos prácticos que mostrasen cómo se deben implementar ciertas construcciones criptográficas (uso de librerías en los principales lenguajes, por ejemplo). Sin embargo, también nos dimos cuenta de que estas implementaciones son demasiado relativas y de que, por cada lenguaje, existen numerosas librerías.

Por lo tanto, decidimos que carecía de interés dar un ejemplo de uso de cada librería, ya que, al fin y al cabo, íbamos a aportar poco más que la propia documentación de la misma. Además, también observamos que la implementación es realmente similar entre una y otra, y que era más importante centrarse en el *por qué* de los parámetros (para lo cual es necesario comprender el algoritmo) en vez de en el *cómo*.

Es por eso que, con menor prioridad, dejamos este contenido para futuras revisiones, esperando que el desarrollador se guíe por nuestra comparativa de librerías y lleve a cabo una profunda inspección antes de incluir ninguna en su proyecto.

4.6 Criptografía post-cuántica

En su momento hablamos del Algoritmo de Shor y de las implicaciones del mismo: un algoritmo capaz de resolver la factorización de enteros muy grandes en tiempo polinomial $O((\log n)^3)$ utilizando computación cuántica, que supondría el “fin de la criptografía actual” (tal y como puede aparecer en los titulares de los diarios más sensacionalistas).

Sin embargo, no es necesario alarmarse. Es cierto que el Algoritmo de Shor simplifica mucho el problema de factorización y, por lo tanto, permite romper cifrados como RSA en un período de tiempo mucho menor. Pero en el capítulo 1.11 ya hablamos de dos (de muchos) algoritmos *Post-Cuánticos*, es decir, que no se verían afectados por la aparición de los computadores cuánticos. Además, para poder romper nuestra criptografía, sería necesario alcanzar una capacidad de computación equivalente a miles de *qubits*, cuando IBM acaba de anunciar (y ya es toda una revolución, a fecha de septiembre de 2019) el primer computador con 53 *qubits*.

Por lo tanto, podemos sacar dos conclusiones: la primera es que todavía falta mucho para alcanzar la supremacía cuántica y, por tanto, para romper cifrados como AES. La segunda es que, a día de hoy, ya contamos con medidas de prevención para evitar un colapso cuando esto ocurra.

5 Anexos

5.1 Anexo 1: Cajas-S para Redes Feistel

s1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

s2

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

s3

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

s4

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

s5

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

s6

12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

s7

4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

s8

13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

5.2 Anexo 2: Rijndael en profundidad

5.2.1 Conceptos matemáticos

El algoritmo Rijndael engloba cierta cantidad de conceptos matemáticos avanzados, especialmente relacionados con el álgebra y la teoría de grupos. Por tanto, en este primer apartado vamos a familiarizar al lector con dichos conceptos. Para ello, vamos a seguir una estructura de definición-proposición-teorema característica de los desarrollos matemáticos habituales.

Definición: Un *anillo*, A , es un conjunto cerrado por la suma ($+$) y la multiplicación (\cdot), tal que:

- i) A es conmutativo (abeliano) con respecto a la suma.
- ii) El producto cumple la propiedad asociativa.
- iii) Se cumple la propiedad distributiva entre la suma y el producto:

$$(a + b) \cdot c = (a \cdot c) + (b \cdot c) \quad y \quad c \cdot (a + b) = (c \cdot a) + (c \cdot b)$$

Si, además, el producto es conmutativo, se dice que A es un *anillo conmutativo* y, si el producto tiene un elemento neutro (unidad), se dice que A es un *anillo con unidad*.

Definición: Un *grupo* es un par ordenado, formado por un *conjunto* de elementos y una *operación binaria* cerrada en el conjunto. Esto quiere decir que al operar dos elementos cualesquiera del conjunto, se obtiene otro elemento del mismo conjunto

Definición: un *cuerpo*, K , es un anillo tal que $K - \{0\}$ es un grupo abeliano respecto a la multiplicación.

Definición: un *cuerpo finito* es un cuerpo definido sobre un conjunto finito de elementos.

Proposición: todo cuerpo finito tiene orden $q = p^n$, donde p es un primo y n es un entero positivo.

Proposición: para cada cardinalidad q , existe una única manera de definir un cuerpo finito con q elementos. Por tanto, todos los campos finitos del mismo orden son isomorfos entre sí.

Para este caso en concreto, vamos a tratar con el grupo finito $GF(2^8)$ y su representación polinómica. Es decir, podremos representar un *byte* como un elemento de este grupo, de la forma:

$$b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x^1 + b_0$$

Por ejemplo, el byte con representación hexadecimal '57' (en binario, 0101 0111), se corresponde con el polinomio:

$$x^6 + x^4 + x^2 + x + 1$$

Observación: La operación *suma* equivale al *bitwise EXOR*. Es decir, es el resultado de sumar los bits uno a uno, módulo 2. Por ese motivo, cada elemento es su propio inverso respecto a la suma, ya que todo byte sumado a sí mismo equivale al byte 00 (elemento neutro).

Definición: Un polinomio es *irreducible* si no tiene divisores, salvo la unidad. En RIJNDAEL, dicho polinomio se denomina $m(x)$ y viene dado por:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \quad (o \text{ '01 01 B' en hexadecimal})$$

Observación: A la hora de multiplicar, es posible que obtengamos resultados con polinomios de grado mayor que 7, es decir, que se saldrían de la longitud de un byte. Por eso mismo, vamos a tratar en el espacio cociente $\mathbb{P}[x] / m(x)$. Esto quiere decir que vamos elegir representantes para cada polinomio dentro de los que tienen grado 7 o menor. Es decir, un polinomio q de grado mayor que 7 tendrá como representante a $q \text{ modulo } x^8 + x^4 + x^3 + x + 1$.

$$\begin{aligned}\text{Ejemplo: } & (x^6 + x^4 + x^2 + x + 1) \cdot (x^7 + x + 1) = \\ & = (x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1) \text{ modulo } (x^8 + x^4 + x^3 + x + 1) = \\ & = x^7 + x^6 + 1\end{aligned}$$

Observación: no existe una operación simple a nivel de bit para implementar la multiplicación. Además, el producto aquí definido es asociativo y cuenta con elemento neutro ('01'). Además, empleando el Algoritmo de Euclides, se puede demostrar que todo elemento tiene un inverso multiplicativo.

En consecuencia a lo aquí observado, se puede concluir que el set de 256 posibles bytes, con la operación XOR como suma y la multiplicación aquí definida, cumple con la estructura del cuerpo finito $GF(2^8)$.

Multiplicación por x

Si multiplicamos un polinomio p por x (en hexadecimal, '10'), es fácil observar que, por la definición del producto previamente dada, estaremos desplazando todos los elementos del polinomio una posición a la izquierda. Es decir:

$$(b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x^1 + b_0) \cdot x = b_7 x^8 + b_6 x^7 + b_5 x^6 + b_4 x^5 + b_3 x^4 + b_2 x^3 + b_1 x^2 + b_0$$

El resultado, al reducir módulo $m(x)$, se obtendría, en caso de que $b_7=0$, aplicando la identidad. En caso de que $b_7=1$, sustrayendo $m(x)$. Portanto, la operación $p(x) \cdot x$ se puede implementar como un *left-shift* seguido de un XOR condicional y, de esta forma, podremos transformar la multiplicación por cualquier elemento en una cadena de multiplicaciones por x .

$$'57' \cdot '13' = '57' \cdot ('01' + '02' + '10') = '57' \cdot '01' + '57' \cdot '02' + '57' \cdot '10'$$

Polinomios con coeficientes en $GF(2^8)$

Es posible trabajar con vectores de 4 bytes usando polinomios con coeficientes en $GF(2^8)$. En este caso, la suma se traduciría a una suma byte por byte que, internamente, actuaría como el *bitwise XOR* anteriormente definido.

Sin embargo, la multiplicación es más compleja:

Sean $a(x), b(x) \in GF(2^8)[x]$ tales que:

$$\begin{aligned}a(x) = & a_3 x^3 + a_2 x^2 + a_1 x + a_0 \\ b(x) = & b_3 x^3 + b_2 x^2 \\ & + b_1 x + b_0\end{aligned}$$

El producto $c(x) = a(x)b(x)$ viene dado por:

$$\begin{aligned}
c(x) &= c_6 x^6 + c_5 x^5 + c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0 && \text{with} \\
c_0 &= a_0 \cdot b_0 & c_4 &= a_3 \cdot b_1 \oplus a_2 \cdot b_2 \oplus a_1 \cdot b_3 \\
c_1 &= a_1 \cdot b_0 \oplus a_0 \cdot b_1 & c_5 &= a_3 \cdot b_2 \oplus a_2 \cdot b_3 \\
c_2 &= a_2 \cdot b_0 \oplus a_1 \cdot b_1 \oplus a_0 \cdot b_2 & c_6 &= a_3 \cdot b_3 \\
c_3 &= a_3 \cdot b_0 \oplus a_2 \cdot b_1 \oplus a_1 \cdot b_2 \oplus a_0 \cdot b_3
\end{aligned}$$

En este caso, $c(x)$ no puede representarse por un vector 4-dimensional, ya que tiene 7 coeficientes. Por ello, vamos a reducir $c(x)$ módulo un polinomio de índice 4. En RIJNDAEL, el polinomio empleado es $M(x) = x^4 + 1$. De esta forma, $x^i \bmod x^4 + 1 = x^{i \bmod 4}$.

Por consiguiente, $c(x)$ quedaría de la siguiente forma:

$$c(x) = c_3 x^3 + (c_6 + c_2) x^2 + (c_5 + c_1) x + (c_4 + c_0) = d_3 x^3 + d_2 x^2 + d_1 x + d_0$$

Esta operación puede verse como un producto de matrices, empleando una matriz "circular":

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Figura 5.1 Matriz circular.

Si ahora contemplamos multiplicar un polinomio $b(x)$ de estas características por x , obtenemos el siguiente resultado:

$$(b_3 x^3 + b_2 x^2 + b_1 x + b_0) \cdot x = b_3 x^4 + b_2 x^3 + b_1 x^2 + b_0 x$$

El cual, al aplicar la reducción módulo $x^4 + 1$, se convierte en:

$$b_2 x^3 + b_1 x^2 + b_0 x + b_3$$

Observamos ahora que el vector es el resultado de aplicar un left-shift a cada coeficiente. Por tanto, se puede expresar la multiplicación por x como:

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 00 & 00 & 00 & 01 \\ 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Figura 5.2 Matriz left-shift.

5.2.2 Base de diseño

Se siguieron tres criterios a la hora de desarrollar el algoritmo:

- Resistencia contra todo tipo de ataque conocido.
- Velocidad y código compacto en una gran variedad de plataformas.
- Simplicidad del diseño.

La mayoría de cifrados empleados hasta la época basaban sus rondas de transformación en la denominada *Red de Feistel*, ya vista en el algoritmo DES. Sin embargo, es importante destacar que RIJNDAEL no emplea esta estructura, sino que cuenta con tres transformaciones uniformes e invertibles, denominadas *capas*. Dichas capas basan su funcionamiento en la aplicación de la *Wide Trail Strategy*, según la cual cada capa tiene su propia función:

Capa de mezcla lineal: garantiza una alta difusión.

Capa no lineal: aplicación en paralelo de Cajas-S.

Capa de adición de la clave: XOR de la clave de la ronda actual (se llamará *Round Key*).

La *Wide Trail Strategy* supuso la base de la robustez de Rijndael, puesto que enfocó el desarrollo del algoritmo en conseguir que las técnicas de criptoanálisis lineal y diferencial resulten inútiles. Debido a la complejidad de la estrategia y su justificación, dejamos como fuente para el lector un *paper* con dicha información:

[The Wide Trail Design Strategy \(Joan Daemen, Vincent Rijmen\)](#)

5.2.3 Especificaciones

Rijndael es un algoritmo de cifrado en bloque iterativo, con longitud de bloque variable y una longitud de clave de 128, 192 o 256 bits.

¿POR QUÉ CLAVES DE 128, 192 O 256 BITS?

En primer lugar, es importante que las claves sean múltiplos de 32 para aplicar las operaciones que describiremos a continuación. Por otro lado, como hemos explicado inicialmente, todos los elementos con los que trabajemos deben vivir en $GF(2^8)$ para poder aplicarse el álgebra definida. En caso contrario, la teoría de anillos no sería aplicable y, por lo tanto, muchas de las asunciones hechas a lo largo de este desarrollo no tendrían sentido. El número de clave más grande que vive en nuestro anillo de polinomios es, por tanto, 256 bits.

ESTADO, CLAVE DE CIFRADO Y NÚMERO DE RONDAS

Definición: entendemos por *Estado* al resultado intermedio del cifrado.

El *Estado* se puede ver como una matriz rectangular de bytes, con cuatro filas y un número de columnas Nb equivalente a la longitud del bloque dividido entre 32.

La *Clave de Cifrado*, de manera similar, se corresponde con una matriz rectangular de cuatro filas, con tantas columnas como la longitud de clave dividida entre 32, Nk .

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Figura 5.3 Ejemplo de Estado ($Nb = 6$) y Clave de Cifrado ($Nk = 4$).

La entrada y la salida del algoritmo se considerará como un array unidimensional formado por bytes (8bits) numerados del 0 al $Nb * 4 - 1$. Portanto, estos bloques tendrán una longitud de 16, 24 o 32 bytes. La clave de cifrado se tratará de la misma forma, con bytes numerados del 0 al $Nk * 4 - 1$.

Los bytes de la entrada se corresponderán con los del *Estado* en el orden $a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0}, a_{0,1}, \dots$ Y, de forma similar, los de la Clave de Cifrado se corresponderán con los del array de la clave, en el mismo orden que el *Estado*.

Entonces, sea n el número del byte en el array unidimensional, sea i el número de columna y j el número de fila, tendremos las siguientes relaciones:

$$i = n \bmod 4; \quad j = \lfloor n/4 \rfloor; \quad n = i + 4 * j;$$

Además, i será el índice del byte dentro de una palabra de 4 bytes, y j será el índice del vector/palabra dentro del bloque actual.

El número de rondas se denota por Nr y viene dado por la siguiente tabla:

Nr	Nb = 4	Nb = 6	Nb = 8
Nk = 4	10	12	14
Nk = 6	12	12	14
Nk = 8	14	14	14

Figura 5.4 Tabla de rondas para el algoritmo.

TRANSFORMACIÓN

Para comprender mayor el algoritmo, haremos uso de *pseudocódigo* en C. Cada ronda de la transformación se representa como:

```
Round (State, RoundKey)
{
    ByteSub (State) ;
    ShiftRow (State) ;
    MixColumn (State) ;
    AddRoundKey (State, RoundKey) ;
}
```

Figura 5.5 Pseudocódigo de una ronda en Rijndael.

La ronda final del algoritmo suprime la función *MixColumn*, quedando así:

```
FinalRound (State, RoundKey)
{
    ByteSub (State) ;
    ShiftRow (State) ;
    AddRoundKey (State, RoundKey) ;
}
```

Figura 5.6 Pseudocódigo de la ronda final en Rijndael.

Función ByteSub(State)

Se trata de una transformación por sustitución, no lineal, que opera en cada uno de los bytes del Estado de forma independiente. La tabla de sustitución (Caja-S) empleada es invertible, fruto de la composición de dos transformaciones:

1. Se toma el inverso multiplicativo dentro de $GF(2^8)$, empleando la representación definida en el apartado 1 del apéndice.
2. Se aplica una transformación afín sobre $GF(2^8)$ definida como:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Figura 5.7 Transformación afín.

Para invertir la función ByteSub(), basta con aplicar inicialmente la inversa de la transformación definida arriba, y seguidamente tomar el inverso multiplicativo en GF(2⁸).

Función ShiftRow(State)

En este caso, las filas del Estado se desplazan a la izquierda tantas posiciones como indique la siguiente tabla, en función del número de columnas del Estado, **Nb**:

Nb	C1	C2	C3
4	1	2	3
6	1	2	3
8	1	3	4

Figura 5.8 Tabla de desplazamientos de ShiftRow en Rijndael.

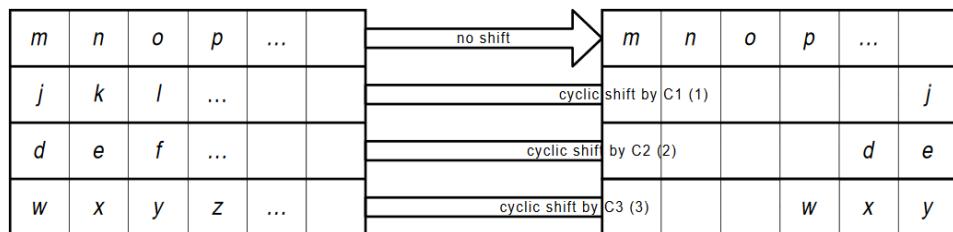


Figura 5.9 Ejemplo de empleo de ShiftRow en Rijndael.

La inversa de esta función se obtiene haciendo tantos desplazamientos a la izquierda como **Nb** – C1, **Nb** – C2 y **Nb** – C3.

Función MixColumn(State)

En esta función, las columnas del Estado se consideran polinomios con coeficientes en GF(2⁸) y se multiplicarán por un polinomio fijo $c(x)$ módulo $x^4 + 1$, donde:

$$c(x) = '03' x^3 + '01' x^2 + '01' x + '02'$$

Este polinomio es coprimo con $x^4 + 1$ (ejercicio para el lector) y, por lo tanto, invertible dentro de nuestro anillo de polinomios. Como explicamos en el apartado inicial, esta multiplicación puede escribirse como un producto matricial. Sea $b(x) = c(x) \cdot a(x)$:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Figura 5.10 Multiplicación de matrices en MixColumn().

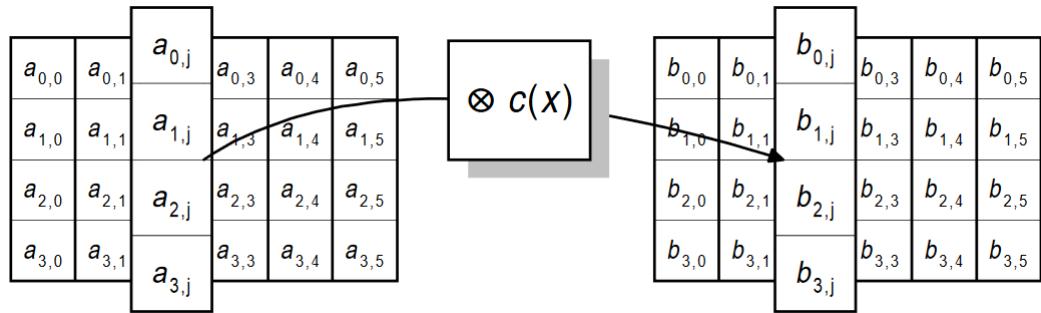


Figura 5.11 Actuación de MixColumn() sobre las columnas del Estado.

La inversa de esta función es similar, salvo porque emplea un nuevo polinomio, $d(x)$, que es el inverso de $c(x)$:

$$d(x) = '0B' x^3 + '0D' x^2 + '09' x + '0E'$$

ADICIÓN DE LA CLAVE

En este caso, una *Round Key* se aplica al estado usando un simple *bitwise XOR*. La *Round Key* se derivará de la Clave de Cifrado, y tendrá una longitud equivalente a la longitud **Nb**.

5.2.4 Obteniendo las RoundKeys

Para obtener las *RoundKeys*, se siguen dos pasos: Expansión de la Clave y Selección de la *Round Key*. Será necesaria una longitud de *Round Key* igual al número de bits en un bloque, multiplicado por el número de rondas más 1.

EXPANSIÓN DE LA CLAVE

La Clave Expandida es un array lineal formado por palabras de 4 bytes, de longitud $Nb^*(Nr+1)$. Las primeras Nk palabras contendrán la Clave de Cifrado (haciendo cálculos, para $Nk = 6$, la longitud de la Clave de Cifrado es de 192 bits).

Existen dos versiones del algoritmo de expansión: uno para claves de longitud menor o igual a 192 bits ($Nk \leq 6$), y otro para claves mayores a 192 bits.

```
KeyExpansion(byte Key[4*Nk] word W[Nb*(Nr+1)])
{
    for(i = 0; i < Nk; i++)
        W[i] = (Key[4*i], Key[4*i+1], Key[4*i+2], Key[4*i+3]);

    for(i = Nk; i < Nb * (Nr + 1); i++)
    {
        temp = W[i - 1];
        if (i % Nk == 0)
            temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
        W[i] = W[i - Nk] ^ temp;
    }
}
```

Figura 5.12 Expansión de clave para $Nk \leq 6$.

```
KeyExpansion(byte Key[4*Nk] word W[Nb*(Nr+1)])
{
    for(i = 0; i < Nk; i++)
        W[i] = (key[4*i], key[4*i+1], key[4*i+2], key[4*i+3]);

    for(i = Nk; i < Nb * (Nr + 1); i++)
    {
        temp = W[i - 1];
        if (i % Nk == 0)
            temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
        else if (i % Nk == 4)
            temp = SubByte(temp);
        W[i] = W[i - Nk] ^ temp;
    }
}
```

Figura 5.13 Expansión de clave para $Nk > 6$.

En el código se emplea la función *SubByte(W)*, que devuelve una palabra de 4 bytes en la que a cada byte se le ha aplicado la Caja-S de Rijndael (la explicada en la función *ByteSub*) en la posición correspondiente. También se emplea *RotByte(W)*, que devuelve una palabra cuyos bytes son el resultado de una permutación cíclica a la izquierda, de modo que (a,b,c,d) se convierta en (b,c,d,a).

Se puede ver que los Nk primeros bytes se engloban a la Clave de Cifrado, mientras que los bytes sucesivos van a ser el fruto de realizar un XOR entre la palabra anterior (a la que se le aplican las funciones anteriormente mencionadas) y la que se encuentra Nk posiciones antes.

También aparece una constante $Rcon[i]$, definida por:

$$Rcon[i] = (RC[i], '00', '00', '00')$$

Donde, además, $RC[i]$ representa un elemento en $GF(2^8)$.

SELECCIÓN DE LA ROUND KEY

La Round Key i -ésima viene dada por las palabras en la clave extendida que están entre las posiciones $W[Nb^*i]$ y $W[Nb^*(i+1)]$. Por ejemplo, para $Nb = 6$ y $Nk = 4$:

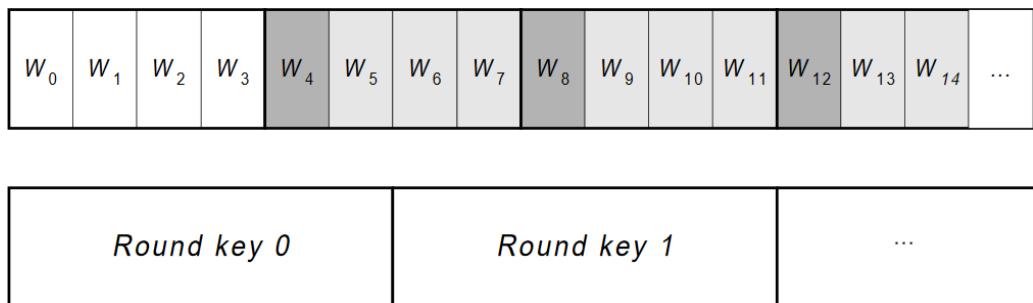


Figura 5.14 Round Key para $Nb = 6$ y $Nk = 4$.

5.2.5 El cifrado

El cifrado consta de tres fases: una fase de adición de la Round Key inicial, $Nr - 1$ rondas y una ronda final, de acuerdo a los algoritmos vistos previamente.

```
Rijndael(State, CipherKey)
{
    KeyExpansion(CipherKey, ExpandedKey) ;
    AddRoundKey(State, ExpandedKey) ;
    For( i=1 ; i<Nr ; i++ ) Round(State, ExpandedKey + Nb*i) ;
    FinalRound(State, ExpandedKey + Nb*Nr) ;
}
```

Figura 5.15 Pseudocódigo de Rijndael.

5.2.6 Aspectos de implementación

Debido a las operaciones con bytes y palabras, es importante prestar especial atención al tipo de procesador empleado. El uso de procesadores de 32 bits permite optimizar algunas de las tareas necesarias para el algoritmo.

Otro aspecto importante de este algoritmo es que permite paralelizar bastantes tareas, mejorando su rendimiento considerablemente.

Sin embargo, la optimización de este algoritmo requiere del uso de un hardware específico, diseñado para mejorar el rendimiento de todas las tareas.

En cuanto al número de rondas a emplear, se propone el uso de 10 rondas en el algoritmo. Esto se debe a que, para longitudes de clave de 128 bits, no se han encontrado ataques *shortcut* (se entiende por ataque *shortcut* a todo aquél más eficiente que la fuerza bruta) efectivos por encima de las 6 rondas. Por lo tanto, se han añadido 4 rondas de margen para aportar un enfoque conservador.

Sin embargo, para bloques por encima de los 128 bits, se recomienda incrementar el número de rondas para evitar, entre otras cosas, la aparición de patrones.

5.2.7 Descifrado

Rijndael está pensado para que el descifrado sea simple y rápido. Por ese motivo, basta con aplicar los mismos pasos que en el cifrado, modificando algunas funciones por sus inversas. El primer paso es invertir una ronda, para lo cual basta con recorrer sus pasos en sentido contrario, aplicando la transformada inversa en cada paso.

```
InvRound (State, RoundKey)
{
    AddRoundKey (State, RoundKey) ;
    InvMixColumn (State) ;
    InvShiftRow (State) ;
    InvByteSub (State) ;
}
```

Figura 5.16 Ejemplo de ronda inversa para descifrado de Rijndael.

Es importante destacar que las funciones *ShiftRow* y *ByteSub* se pueden aplicar en orden inverso gracias a la propiedad de conmutatividad que presentan entre sí. Esto se debe a que, mientras la primera modifica posiciones sin alterar los bytes, la segunda únicamente altera bytes, sin importar su posición.

Por otro lado, ya explicamos que la función *MixColumn* puede expresarse como una operación lineal con matrices. Es por ello que se considera una aplicación lineal y, como tal, cumple que $A(a+b)=A(a)+A(b)$. Como consecuencia de esta propiedad, también podremos invertir el orden de la secuencia:

AddRoundKey(State, RoundKey);

InvMixColumn(State);

Por lo siguiente:

InvMixColumn(State);

AddRoundKey(State, InvRoundKey);

En este caso, *InvRoundkey* es el resultado de aplicar *InvMixColumn* a la Round Key, cumpliendo con la propiedad lineal de la función.

Llegados a este punto, contamos con una definición para la inversa de una ronda cualquiera, y de la ronda final:

```
I_Round (State, I_RoundKey)
{
    InvByteSub (State) ;
    InvShiftRow (State) ;
    InvMixColumn (State) ;
    AddRoundKey (State, I_RoundKey) ;
}

I_FinalRound (State, I_RoundKey)
{
    InvByteSub (State) ;
    InvShiftRow (State) ;
    AddRoundKey (State, RoundKey0) ;
}
```

Figura 5.17 Funciones inversa para las rondas del descifrado de Rijndael.

En conclusión, el inverso del cifrado de Rijndael se puede expresar como:

```
I_Rijndael (State, CipherKey)
{
    I_KeyExpansion (CipherKey, I_ExpandedKey) ;
    AddRoundKey (State, I_ExpandedKey+ Nb*Nr) ;
    For( i=Nr-1 ; i>0 ; i-- ) Round (State, I_ExpandedKey+ Nb*i) ;
    FinalRound (State, I_ExpandedKey) ;
}
```

Figura 5.18 Inversa del cifrado Rijndael.

En este caso, la inversa de la función de expansión de la clave se supone de la siguiente forma:

```
I_KeyExpansion(CipherKey, I_ExpandedKey)
{
    KeyExpansion(CipherKey, I_ExpandedKey);
    for( i=1 ; i < Nr ; i++ )
        InvMixColumn(I_ExpandedKey + Nb*i) ;
}
```

Figura 5.19 Inversa de la expansión de clave en Rijndael.

5.2.8 Ventajas e inconvenientes

VENTAJAS

- Rijndael puede ser implementado para correr a velocidad muy superiores a las de otros algoritmos de cifrado en bloque. Además, permite la parallelización por diseño y, debido a la ausencia de operaciones aritméticas, funciona igual en procesadores *Big-Endian* que en *Little-Endian*.
- El cifrado es *autocontenido*, por lo que no emplea componentes pertenecientes a ningún otro algoritmo.
- Permite el uso de longitudes de bloque y de clave variables.

INCONVENIENTES

- El proceso de descifrado presenta ciertas limitaciones, ya que las funciones inversas no puede implementarse de forma abreviada: requieren el uso de todos los pasos del algoritmo como tal.
- El cifrado y su inversa utilizan código distinto y tablas distintas.

5.2.9 Comentarios

Queda visto que, como cifrador en bloque, Rijndael ha demostrado ser muy eficiente y seguro. Por este motivo fue elegido el estándar AES en su día. Sin embargo, en el paper original de *Joan Daemen et al.* se introducen algunas otras formas de empleo del algoritmo como, por ejemplo, función hash o generador de números pseudaleatorios.

5.3 Anexo 3: Comparación de algoritmos de cifrado

5.3.1 Comparación general de algoritmos

Algoritmo	Clasificación	Fundamento del algoritmo	Longitud de clave	Patente / Licencia necesaria	Ataques conocidos	CERTIFICACIÓN	SITUACIÓN
DES	Simétrico, bloques	Feistel	64 bits	NO	Fuerza Bruta	✗	Vulnerable
Triple-DES	Simétrico, bloques	Feistel	168 bits	NO	Fuerza Bruta, Texto Elegido, Texto Conocido	✓ FIPS 140-2 (Sólo en CBC y ECB, en cifrado y en MAC)	Robusto
Blowfish	Simétrico, bloques	Feistel	64-448 bits	NO	Ataque por Diccionario	✗	Robusto
Twofish	Simétrico, bloques	Feistel	128,192,256 bits	NO	Ninguno práctico	✗ Finalista AES	Robusto
IDEA	Simétrico, bloques	Lai-Massey scheme	128 bits	NO	Ninguno práctico	✗	Robusto
AES (Rijndael)	Simétrico, bloques	Wide Trail Strategy (sustitución – permutación)	128,192,56 bits	NO	Side-channel, Cache Timing Attack, (ninguno práctico)	✓ FIPS 140-2 (En cifrado y en MAC)	Robusto
RC6	Simétrico, bloques	Feistel (modificado)	128, 192, 256 bits (extensible)	NO	Ninguno práctico	✗ Finalista AES	Robusto
Serpent	Simétrico, bloques	Red sustitución-permutación	128, 192, 256 bits (extensible)	NO	Ninguno práctico	✗ Finalista AES	Robusto
ChaCha20	Simétrico, flujo	Add-Rotate-XOR	128, 256 bits	NO	Ninguno práctico	✓ eSTREAM portfolio	Robusto

Camellia	Simétrico, bloques	Feistel	128,192,256 bits	SI	Ninguno práctico	✓ CRYPTREC, NESSIE	Robusto
Diffie-Hellman	Asimétrico, Intercambio de Claves	Logartimo Discreto	Cualquiera	NO	<i>Man-In-The-Middle</i> (Autenticación)	✓ FIPS 140-2 (Sólo para intercambio de claves, con 2048-5012 bits de clave)	Robusto
RSA	Asimétrico	Factorización de Enteros	Cualquiera	NO	Algoritmo Cuántico de Shor	✓ FIPS 140-2 (Sólo claves mayores que 2048 bits, usado con funciones hash SHA1 y SHA2)	Robusto
ElGamal	Asimétrico	Logaritmo Discreto	Cualquiera	NO	<i>Chosen-Ciphertext</i> (Si no se escoge un buen padding scheme)	✗	Depende de implementación
ECDH	Asimétrico, Intercambio de Claves	Logaritmo Discreto	Cualquiera	NO	Ninguno práctico (en un futuro, criptografía cuántica)	✓ FIPS 140-2 (Sólo claves mayores que 112 bits)	Robusto
Cramer-Shoup	Asimétrico	Logaritmo Discreto	Cualquiera	NO	Ninguno práctico	✗	Robusto
McEliece	Asimétrico	Decodificación de Códigos Lineales	Cualquiera (en forma de matrices muy grandes)	NO	Ninguno práctico	✗	Robusto
NTRU	Asimétrico	Factorización de Polinomios	Cualquiera	GPL	Ninguno práctico	✗	Robusto (bajo estudio)

5.3.2 Comparación de seguridad en algoritmos de clave simétrica

Los siguientes datos representan una estimación del margen de seguridad de diversos algoritmos tal y como se recogieron en el informe final sobre el resultado del concurso para AES. Todos los cálculos se realizaron sobre implementaciones de 128 bits.

R representa el número de rondas propuestas para el candidato.

M_1 representa el mínimo número de rondas necesario para garantizar la seguridad del candidato.

M_2 representa el máximo número de rondas esperado para un ataque (ya sea real o estimado).

S_1 es el margen de seguridad asociado a M_1 .

S_2 es el margen de seguridad asociado a M_2 .

Algoritmo	R	M_1	M_2	S_1	S_2
Rijndael	10	8	6	25%	66%
Serpent	32	17	15	88%	113%
RC6	20	21	16	-5%	25%
Twofish	16	14	6	14%	166%

Figura 5.20 Tabla comparativa de márgenes de seguridad de algoritmos.

Según los datos recogidos, RC6 parece ser el algoritmo menos robusto de los estudiados. Esto se debe, entre otros motivos, a que incluye operaciones de multiplicación y rotación variables, que resultan más difíciles de defender que otras más simples como las operaciones Booleanas, las rotaciones fijas o las sumas a nivel de bit.

En el documento de resolución se plantea también la existencia de ciertos ataques teóricos que afectan a los algoritmos tal y como están implementados. Por ejemplo, un ataque basado en el análisis de la potencia consumida podría revelar información de los algoritmos Rijndael, Serpent y Twofish, mientras que RC6 parece más robusto. Sin embargo, este tipo de ataque es muy difícil de implementar, y requiere de unas condiciones demasiado específicas como para ser considerado crítico.

Al aplicar técnicas de enmascaramiento para la defensa frente a ataques como el de análisis de potencia, RC6 resulta ser que se ve más afectado en términos de eficiencia, mientras que Rijndael ofrece la mejor combinación de velocidad y memoria empleada.

En conclusión, puede que algunos algoritmos como Serpent pudieran ser considerados más robustos que Rijndael. Sin embargo, el análisis de otros muchos factores, como el *throughput* generado, la eficiencia del algoritmo o la memoria empleada, situó a Rijndael como el algoritmo favorito para el estándar AES. Además, como Rijndael se postuló como estándar, la mayoría de procesadores actuales cuentan con hardware dedicado a este algoritmo (es el más utilizado para cifrados simétricos), por lo que es muy difícil que algún otro algoritmo alcance una eficiencia similar.

REFERENCIAS

[NIST-1] [*Journal of Research of the National Institute of Standards and Technology*](#)

5.4 Anexo 4: Matemáticas de RSA

5.4.1 Conceptos matemáticos

Definición 1: dos números p y q son *coprimos* si no tienen divisores comunes, salvo el 1. Es decir, su *máximo común divisor* es 1.

Definición 2: sea n un número entero positivo, entonces la función *phi de Euler*, denotada por $\varphi(n)$, se define como el número de enteros positivos menores o iguales a n , que sean *coprimos* con n .

Propiedades de la phi de Euler:

- $\varphi(1) = 1$.
- $\varphi(p^k) = (p - 1)p^{k-1}$ si p es primo y k es un número natural.
- Si n y m son *coprimos*, entonces se cumple $\varphi(mn) = \varphi(m)\varphi(n)$.

Teorema 1 (Teorema de Euler): Sean a y n dos números enteros *coprimos*, y sea $\varphi(n)$ la función *phi de Euler*, entonces se cumple:

$$a^{\varphi(n)} \equiv 1 \pmod{n}$$

Teorema 2 (Teorema Chino del Resto): Sean n_1, \dots, n_k enteros coprimos entonces, dados a_1, \dots, a_k , existe un entero x que resuelve el siguiente sistema de congruencias:

$$x \equiv a_1 \pmod{n_1}$$

...

$$x \equiv a_k \pmod{n_k}$$

Además, todas las posibles soluciones x son congruentes módulo el mínimo común múltiplo de los n_i .

5.4.2 Demostración de la propiedad principal de RSA

Recapitulando, tenemos dos pares de claves pública y privada: (n, e) y (n, d) , respectivamente. Se han calculado con las siguientes propiedades:

- $n = p * q$, con p y q primos.
- Se toma un número e como clave pública, que cumpla las condiciones de ser mayor que 1 pero menor que $\varphi(n)$. Es necesario que e y $\varphi(n)$ sean coprimos.
- Se calcula la clave privada d , tal que $e \cdot d \equiv 1 \pmod{\varphi(n)}$. Es decir, d es el inverso multiplicativo de e . De esta forma, tenemos que $\varphi(n) | d \cdot e - 1$.
- El mensaje cifrado se calcula como $c = m^e \pmod{n}$.
- A partir del mensaje cifrado, podemos obtener el texto como $m = c^d \pmod{n}$.

Llegados a este punto, queremos demostrar la efectividad de la función de descifrado. Para ello, tendremos que ver que, efectivamente, $m = m^{ed} \pmod{n}$.

Inicialmente, aplicando el Teorema de Euler tenemos que $m^{\varphi(n)} \equiv 1 \pmod{n}$. Y más en concreto, tenemos que $m^{p-1} \equiv 1 \pmod{p}$. Además, observamos que $(p-1) \mid \varphi(n)$ (el símbolo $|$ se lee como “divide a”). A esta propiedad vamos a denominarla (1).

Como hemos calculado e y d de forma que sean inversos en el anillo multiplicativo modular, tenemos que $e \cdot d \equiv 1 \pmod{\varphi(n)}$. Por lo tanto, podemos asumir la siguiente igualdad: $e \cdot d = k \cdot \varphi(n) + 1$ para un cierto k . Así llegamos a la siguiente fórmula:

$$m^{ed} \equiv m^{k \cdot \varphi(n)+1} \pmod{n}$$

Usando ahora (1), vemos que:

$$m^{k \cdot \varphi(n)+1} = m^{k \cdot \varphi(n)} \cdot m = (m^{p-1})^{(q-1) \cdot k} \cdot m$$

Y recurriendo de nuevo al Teorema de Euler, tenemos que:

$$m^{p-1} \equiv 1 \pmod{p}$$

$$(m^{p-1})^{(q-1) \cdot k} \cdot m \equiv 1 \cdot m \equiv m \pmod{p}$$

De forma análoga, pero aplicando $(q-1)$ en vez de $(p-q)$ podemos llegar a ver que:

$$(m^{q-1})^{(p-1) \cdot k} \cdot m \equiv 1 \cdot m \equiv m \pmod{q}$$

Y aplicando ahora el Teorema Chino del Resto (observando que $\text{mcm}(p, q) = n$), podemos concluir que:

$$m^{k \cdot \varphi(n)+1} \equiv m \pmod{n}$$

Y por lo tanto, $m^{ed} \equiv m \pmod{n}$.

Es importante resaltar que en todo momento se asume $0 \leq m < n$.

5.5 Anexo 5: Pseudocódigo de MD5

```

//Note: All variables are unsigned 32 bit and wrap modulo 2^32 when calculating
var int[64] s, K
var int i

//s specifies the per-round shift amounts
s[ 0..15] := { 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22 }
s[16..31] := { 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20 }
s[32..47] := { 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23 }
s[48..63] := { 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21 }

//Use binary integer part of the sines of integers (Radians) as constants:
for i from 0 to 63
K[i] := floor(2^32 * abs(sin(i + 1)))
end for
//(Or just use the following precomputed table):
K[ 0.. 3] := { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee }
K[ 4.. 7] := { 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501 }
K[ 8..11] := { 0x698098d8, 0xb44f7af, 0xfffff5bb1, 0x895cd7be }
K[12..15] := { 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821 }
K[16..19] := { 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa }
K[20..23] := { 0xd62f105d, 0x02441453, 0xd8ale681, 0xe7d3fbcc8 }
K[24..27] := { 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed }
K[28..31] := { 0xa9e3e905, 0xfcfa3f8, 0x676f02d9, 0x8d2a4c8a }
K[32..35] := { 0xffffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c }
K[36..39] := { 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xebefbc70 }
K[40..43] := { 0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x4881d05 }
K[44..47] := { 0xd9d4d039, 0x6db99e5, 0x1fa27cf8, 0xc4ac5665 }
K[48..51] := { 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039 }
K[52..55] := { 0x655b59c3, 0x8f0ccc92, 0xffffeff47d, 0x85845dd1 }
K[56..59] := { 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0xe0811a1 }
K[60..63] := { 0x7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391 }

//Initialize variables:
var int a0 := 0x67452301 //A
var int b0 := 0xefcdab89 //B
var int c0 := 0x98badcfe //C
var int d0 := 0x10325476 //D
//Pre-processing: adding a single 1 bit
append "1" bit to message
// Notice: the input bytes are considered as bits strings,
// where the first bit is the most significant bit of the byte.1501

//Pre-processing: padding with zeros
append "0" bit until message length in bits ≡ 448 (mod 512)
append original length in bits mod 2^64 to message
//Process the message in successive 512-bit chunks:
for each 512-bit chunk of padded message
break chunk into sixteen 32-bit words M[j], 0 ≤ j ≤ 15
//Initialize hash value for this chunk:
var int A := a0
var int B := b0
var int C := c0
var int D := d0
//Main loop:
for i from 0 to 63
var int F, g
if 0 ≤ i ≤ 15 then
F := (B and C) or ((not B) and D)
g := i
else if 16 ≤ i ≤ 31 then
F := (D and B) or ((not D) and C)
g := (5*i + 1) mod 16
else if 32 ≤ i ≤ 47 then
F := B xor C xor D
g := (3*i + 5) mod 16
else if 48 ≤ i ≤ 63 then
F := C xor (B or (not D))
g := (7*i) mod 16
//Be wary of the below definitions of a,b,c,d
F := F + A + K[i] + M[g]
A := D
D := C
C := B
B := B + leftrotate(F, s[i])
end for
//Add this chunk's hash to result so far:
a0 := a0 + A
b0 := b0 + B
c0 := c0 + C
d0 := d0 + D
end for
var char digest[16] := a0 append b0 append c0 append d0 // (Output is in little-

```

5.6 Anexo 6: Comparación de librerías criptográficas

En este anexo presentamos una comparativa entre las principales librerías criptográficas. Tendremos en cuenta los lenguajes soportados, los distintos algoritmos que implementan, si se actualizan regularmente o no, qué estándares incluyen, qué licencia necesitan...

5.6.1 Principales librerías

Nombre	Lenguajes soportados	Open Source	Licencia	FIPS 140	Última actualización (conocida en agosto de 2019)
Botan	C++	✓	<i>Simplified BSD</i>	✗	Julio de 2019
Bouncy Castle	Java, C++	✓	<i>MIT License</i>	✓	Enero de 2019
SunJCE	Java	✓	Oracle	✗	2019
Crypto++	C++	✓	<i>Boost Software License</i>	✗	Febrero de 2019
Libgcrypt	C	✓	<i>GNU LGPL</i>	✓	Octubre de 2018
GnuTLS	C	✓	<i>GNU LGPL</i>	✓	Mayo de 2019
Network Security Services (Mozilla)	C	✓	<i>MPL 2.0</i>	✓	Julio de 2019
OpenSSL	C	✓	<i>Apache License 1.0</i>	✓	Mayo de 2019
WolfCrypt	C	✓	<i>GPL v2</i>	✓	Marzo de 2019
NaCl	C	✓	<i>Public Domain</i>	✗	Febrero 2011
pyCrypto	Python	✓	<i>Public Domain</i>	✗	2014
pyCryptodome	Python	✓	<i>Public Domain</i>	✗	2019
Cryptography (Python)	Python	✓	<i>Public Domain</i>	✗	Mayo de 2019

5.6.2 Algoritmos de intercambio de clave

Librería	E CDH	D H	D SA	R SA	ElGa mal	N TRU	D SS
Botan	✓	✓	✓	✓	✓	✗	✓
Bouncy Castle	✓	✓	✓	✓	✓	✓	✓
SunJCE	✗	✓	✓	✓	✗	✗	✗
Crypto++	✓	✓	✓	✓	✓	✗	✓
Libgcrypt	✓	✓	✓	✓	✓	✗	✓
GnuTLS	✓	✓	✓	✓	✗	✗	✓
Network Security Services	✓	✓	✗	✓	✗	✗	✓
OpenSSL	✓	✓	✓	✓	✗	✗	✗
WolfCrypt	✓	✓	✓	✓	✗	✓	✓
NaCl	✓	✗	✗	✗	✗	✗	✗
pyCryptodome	✗	✗	✓	✓	✓	✗	✓
Cryptography	✓	✓	✓	✓	✗	✗	✓

5.6.3 Algoritmos simétricos

Librería	Triple DES	AES	IDEA	Twofish	RC4	Grain	Salsa 20	ChaCha	CameLLia
Botan	✓	✓	✓	✓	✓	✗	✓	✓	✓
Bouncy Castle	✓	✓	✓	✓	✓	✓	✓	✓	✓
SunJCE	✓	✓	✗	✗	✓	✗	✗	✗	✗
Crypto++	✓	✓	✓	✓	✓	✗	✓	✓	✓
Libgrypt	✓	✓	✓	✓	✓	✗	✓	✓	✓
GnuTLS	✓	✓	✗	✗	✓	✗	✓	✓	✓
Network Security Services	✓	✓	?	?	✓	✗	?	?	?
OpenSSL	✓	✓	✓	✗	✓	✗	✗	✓	✓
WolfCrypt	✓	✓	✓	✗	✓	✗	✓	✓	✓
NaCl	✗	✓	✗	✗	✗	✗	✓	✓	✗
pyCryptodome	✓	✓	✗	✗	✓	✗	✓	✓	✗
Cryptography	✓	✓	✓	✗	✓	✗	✗	✓	✓

5.6.4 Funciones hash

Librería	MD 5	SHA1	SHA2	SHA3	RIPEMD	BLAKE2	Whirlpool	Streebog
Botan	✓	✓	✓	✓	✓	✓	✓	✓
Bouncy Castle	✓	✓	✓	✓	✓	✓	✓	✓
Crypto++	✓	✓	✓	✓	✓	✓	✓	✗
Libgcrypt	✓	✓	✓	✓	✓	✓	✓	✓
GnuTLS	✓	✓	✓	✓	✗	✗	✗	✓
Network Security Services	✓	✓	✓	✓	?	?	?	?
OpenSSL	✓	✓	✓	✓	✓	✓	✓	✗
WolfCrypt	✓	✓	✓	✓	✓	✓	✗	✗
NaCl	✗	✗	✓	✗	✗	✓	✗	✗
pyCryptodome	✓	✓	✓	✓	✓	✓	✗	✗
Cryptography	✓	✓	✓	✓	✗	✓	✗	✗

5.6.5 Algoritmos MAC

Librería	HMA C-MD5	HMAC- SHA1	HMAC- SHA2	Poly13 05-AES	BLAKE2 -MAC
Botan	✓	✓	✓	✓	✓
Bouncy Castle	✓	✓	✓	✓	✓
Crypto++	✓	✓	✓	✓	✓
Libgcrypt	✓	✓	✓	✓	✓
GnuTLS	✓	✓	✓	✓	✗
Network Security Services	?	?	?	?	?
OpenSSL	✓	✓	✓	✓	✓
WolfCrypt	✓	✓	✓	✓	✓
NaCl	✗	✗	✓	✓	✓
pyCryptodome	✓	✓	✓	✓	✓
Cryptography	✓	✓	✓	✓	✓