CRIPTOGRAFÍA Y SEGURIDAD EN COMPUTADORES

5ª Edición

Manuel J. Lucena López

Título: *Criptografía y Seguridad en Computadores*

Autor: Manuel J. Lucena López

Versión: 5-2.0.1

Fecha: 11 de febrero de 2022

—¿Qué significa *habla, amigo y entra*? —preguntó Merry.

—Es bastante claro —dijo Gimli—. Si eres un amigo, dices la contraseña y las puertas se abren y puedes entrar.

—Sí —dijo Gandalf—, es probable que estas puertas estén gobernadas por palabras…

J.R.R. Tolkien, *El Señor de Los Anillos*

Yo seguía sin entender. De pronto, tuve una iluminación:

—¡Super thronos viginti quatuor! ¡La inscripción! ¡Las palabras grabadas sobre el espejo!

—¡Vamos! —dijo Guillermo—. ¡Quizás aún estemos a tiempo de salvar una vida!

Umberto Eco, *El Nombre de la Rosa*

—¿Y qué? —preguntó un visitante de Washington—. ¿Qué significan otros números primos más?

—Tal vez significa que nos están enviando un dibujo. Este mensaje está compuesto por una enorme cantidad de bits de información. Supongamos que esa cantidad es el producto de tres números más pequeños (...). Entonces, el mensaje tendría tres dimensiones.

Carl Sagan, *Contact*

En la pantalla se formaban y volvían a formarse dibujos de hielo mientras él tanteaba en busca de brechas, esquivaba las trampas más obvias y trazaba la ruta que tomaría a través del hielo de la Senso/Red.

William Gibson, *Neuromante*

Agradecimientos

A mis alumnos, por aguantarme cada año.

A Borja Arberas, Jordi Barnes, Alejandro Benabén, Miguel Castro, Manuel Cátedra, Jesús Cea, Dextar, Gabriel Horacio Díez, Luis Escánez, Lázaro Escudero, Silvia Fandiño, Sacha Fuentes, Antonio J. Galisteo, Carlos García, David García, Luis García, Alfonso González, Germán Jácome, Juen Juen, Martín Knoblauch, Jorge Iglesias, Inkel, Ignacio Llatser, Daniel Lombraña, Juan Martín, Nicolás Mendia, Ana Celia Molina, Javier Nieto, Juan Andrés Olmo, Jorge Ordovás Juan Ramón Moral, Antonio Jesús Navarro, Alejandro Noguera, José A. Ramos, Carlos Romeo, Roberto Romero, Javier Rubio, Juan José Ruiz, Miguel Sánchez, Felipe Santos, Pablo Tellería, Manuel Vázquez, Rafael Vendrell, Víctor A. Villagra, y al resto de personas que, enviando sugerencias o correcciones, han ayudado a mejorar este libro.

A quienes alguna vez han compartido sus conocimientos, por enriquecer a toda la comunidad.

A todos los que durante todo este tiempo me han animado a seguir trabajando en esta obra.

Sobre esta obra

Ya han pasado más de 20 años desde que me hice cargo de la asignatura de *Criptografía y Seguridad en Computadores*, en la antigua Ingeniería Técnica en Informática de Gestión, impartida en la Escuela Politécnica Superior de Jaén. Por aquel entonces, Internet apenas estaba empezando a experimentar el crecimiento explosivo que la ha llevado a ser, entre otras cosas, una fuente inagotable de información –y, lamentablemente, ruido– sobre cualquier tema. La poquísima información disponible sobre criptografía se encontraba por aquel entonces en un puñado de libros, casi todos ellos en inglés. Justo antes de encargarme de la asignatura antes mencionada, tuve la oportunidad de participar en la elaboración de una *colección de apuntes* que editaba la propia Universidad de Jaén. Se trataba de un buen comienzo, pero resultaba insuficiente para una disciplina tan dinámica, y no era viable llevar a cabo sobre ella las actualizaciones imprescindibles que mantuvieran su utilidad año a año.

En estas condiciones, decidí elaborar una nueva colección de apuntes, en formato exclusivamente digital y partiendo esta vez de cero, lo cual me permitiría llevar a cabo actualizaciones frecuentes para mantener su vigencia. Esa colección fue creciendo y tomando forma hasta que alcanzó un estado razonablemente bueno como para ser usada como texto base en mi docencia. Fue entonces cuando, casi por casualidad, me encontré en la página de Kriptópolis una referencia a la antigua colección de apuntes, en la sección de recursos sobre Criptografía en castellano. Así que les envié este nuevo documento, ofreciéndolo

gratuitamente para su descarga. Al fin y al cabo, era fruto de un trabajo ya remunerado por mi universidad. La respuesta del público fue increíblemente positiva, y el número de comentarios, felicitaciones y aportaciones para mejorarlo me animó a seguir adelante con esta obra.

Con el tiempo mi labor, tanto docente como investigadora, se ha ido diversificando. Esta circunstancia, unida a la progresiva estabilización de los contenidos del libro, ha hecho que las actualizaciones sean cada vez menos frecuentes. Pero este proyecto sigue vivo. Al fin y al cabo me ha ayudado a conocer a tanta y tan buena gente, me ha abierto tantas puertas y, por qué no decirlo, le tengo tanto cariño, que aunque desde los puntos de vista de mi *carrera profesional* o económico me ha aportado bastante poco (mención aparte merecería algún caso de *referencias extensas* en libros comerciales, sin citarme como fuente *inspiradora)*, que acabo volviendo sobre él cada cierto tiempo, para corregirle una frase allí, aclararle un párrafo allá, o añadirle algún contenido nuevo.

Espero que disfruten de esta obra tanto como yo he disfrutado con su elaboración.

Enero de 2016

Índice general

Parte I

Preliminares

Capítulo 1

Introducción

1.1. Cómo leer esta obra

Esta obra ha sido organizada de la siguiente forma:

1. *Preliminares*. Aquí se incluyen todos los conceptos básicos y se introduce la terminología empleada en el resto del libro. Su lectura es recomendable incluso para las personas que ya conocen el tema, puesto que puede evitar cierta confusión en los términos empleados a lo largo de la obra.

2. *Fundamentos Teóricos de la Criptografía*. Se desarrollan brevemente los resultados teóricos sobre los que se van a apoyar las diferentes técnicas descritas en el libro. Si usted no domina las Matemáticas, o simplemente no tiene interés en estos fundamentos, puede pasar estos capítulos por alto.

3. *Algoritmos Criptográficos*. Este bloque está dedicado a los algoritmos de cifrado —simétricos y asimétricos— a las funciones resumen, y en general a las técnicas que permiten garantizar la seguridad de la información.

4. *Aplicaciones Criptográficas*. A lo largo de esta parte del libro estudiaremos distintas aplicaciones de la Criptografía, como la comunicación segura, los certificados digitales, etc.

5. *Apéndices*.

Este texto no tiene necesariamente que ser leído capítulo por capítulo, aunque se ha organizado de manera que los contenidos más básicos aparezcan primero. La parte de fundamentos teóricos está orientada a personas con unos conocimientos mínimos sobre Álgebra y Programación, pero puede ser ignorada si el lector está dispuesto a prescindir de las justificaciones matemáticas de lo que encuentre en posteriores capítulos. La recomendación del autor en este sentido es clara: si es su primer contacto con la Criptografía, deje los fundamentos teóricos justo para el final, o correrá el riesgo de perderse entre conceptos que, si de una parte son necesarios para una comprensión profunda del tema, no son imprescindibles a la hora de empezar a adentrarse en este apasionante mundo.

Se ha pretendido que todos los conceptos queden suficientemente claros con la sola lectura de este libro, pero se recomienda vivamente que si el lector tiene interés por profundizar en cualquiera de los aspectos tratados aquí, consulte la bibliografía para ampliar sus conocimientos, pudiendo emplear como punto de partida las propias referencias que aparecen al final de este libro aunque, por desgracia, algunas de las más interesantes están en inglés.

1.2. Algunas notas sobre la historia de la Criptografía

La Criptografía moderna nace al mismo tiempo que las computadoras. Durante la Segunda Guerra Mundial, en un lugar llamado Bletchley Park, un grupo de científicos entre los que se encontraba Alan Turing, trabajaba en el proyecto ULTRA tratando de descifrar

los mensajes enviados por el ejército alemán con los más sofisticados ingenios de codificación ideados hasta entonces: la máquina ENIGMA y el cifrado *Lorenz*. Este grupo de científicos diseñó y utilizó el primer computador de la Historia, denominado *Colossus* —aunque esta información permaneció en secreto hasta mediados de los 70—.

Desde entonces hasta hoy ha habido un crecimiento espectacular de la tecnología criptográfica, si bien una parte sustancial de estos avances se mantenían —y se siguen manteniendo, según algunos— en secreto. Financiadas fundamentalmente por la NSA (Agencia Nacional de Seguridad de los EE.UU.), la mayor parte de las investigaciones hasta hace relativamente poco tiempo han sido tratadas como secretos militares. Sin embargo, en los últimos años, investigaciones serias llevadas a cabo en universidades de todo el mundo han logrado que la Criptografía sea una ciencia al alcance de todos, y que se convierta en la piedra angular de asuntos tan importantes como el comercio electrónico, la telefonía móvil, o las plataformas de distribución de contenidos multimedia. Esta dualidad civil–militar ha dado lugar a una curiosa *doble historia* de la Criptografía, en la que los mismos algoritmos eran descubiertos, con pocos años de diferencia, por equipos de anónimos militares y posteriormente por matemáticos civiles, alcanzando únicamente estos últimos el reconocimiento público por sus trabajos.

Muchas son las voces que claman por la disponibilidad pública de la Criptografía. La experiencia ha demostrado que la única manera de tener buenos algoritmos es que éstos sean accesibles, para que puedan ser sometidos al escrutinio de toda la comunidad científica. Existe una máxima en Criptografía que afirma que cualquier persona —o equipo— es capaz de desarrollar un algoritmo criptográfico que ella misma no sea capaz de romper. Si la seguridad de nuestro sistema se basa en que nadie conozca su funcionamiento, práctica que que se conoce con el revelador nombre de *seguridad a través de la oscuridad*, se producen varias consecuencias perversas: por un lado, aquellos que quieran conocer su verdadera resistencia tendrán que confiar en nuestra palabra, y por otro, tendrán una falsa sensación de seguridad, ya que si algún

enemigo encuentra una falla en el sistema, es bastante probable que no la publique. En consecuencia, tal y como ya indicó Auguste Kerckhoffs en 1883, el único secreto que debe tener un sistema criptográfico es la clave. Ejemplos a lo largo de la historia sobre fracasos de esta política, por desgracia, hay muchos, algunos de ellos en ámbitos tan delicados como el voto electrónico.

Salvo honrosas excepciones[1], la Criptografía llega hasta nosotros en forma de programas informáticos. Un programa mal diseñado puede echar por tierra la seguridad de un buen algoritmo criptográfico, por lo que es necesario conocer cómo está escrito el programa en cuestión, para poder detectar y eliminar los fallos que aparezcan en él. En este sentido, el *Sofware Libre*, cuyo código fuente está a disposición de los usuarios —a diferencia del *software* privativo, que mantiene el código fuente en secreto— quizás sea el que brinda mejores resultados, ya que permite a cualquiera, además de asegurarse de que no contiene *puertas traseras*, estudiar y eventualmente corregir el código si encuentra fallos en él. Actualmente, una de las mayores amenazas sobre el *software* libre es la pretensión de establecer sistemas de patentes sobre los programas informáticos, con un claro perjuicio tanto para los usuarios como para las pequeñas empresas frente al poder de las grandes corporaciones. Por desgracia, este problema ha estado vigente en Europa durante décadas, y cada cierto tiempo se vuelve a intentar introducir patentes de *software* en la legislación de la Unión Europea.

Es imposible desligar la Criptografía moderna de todas las consideraciones políticas, filosóficas y morales que suscita. Hoy por hoy, tiene más poder quien más información controla, por lo que permitir que los ciudadanos empleen técnicas criptográficas para proteger su intimidad limita de forma efectiva ese poder. Con el pretexto de la seguridad se están aplicando medidas para ralentizar el acceso de los ciudadanos a la Criptografía *fuerte*, bien desprestigiando a quienes la usan, bien dificultando por distintos medios su adopción generalizada. Uno de los frentes de debate más llamativos en este sentido ha

[1]Como el algoritmo *Solitaire*, desarrollado por Bruce Schneier, para el que únicamente se necesita papel, lápiz, una baraja y algo de paciencia.

sido la intención de algunos gobiernos de almacenar todas las claves privadas de sus ciudadanos, necesarias para *firmar digitalmente*, y considerar ilegales aquellas que no estén registradas. Es como pedirnos a todos que le demos a la policía una copia de las llaves de nuestra casa. Esta corriente crea una situación extremadamente perversa: aquellos que quieren emplear la Criptografía para usos legítimos encuentran dificultades mientras que, por ejemplo, a un traficante de armas le tiene sin cuidado que sea ilegal usarla, con lo que no se frena su uso delictivo.

Existe un falaz argumento que algunos esgrimen en contra del uso privado de la Criptografía, proclamando que ellos nada tienen que ocultar. Estas personas insinúan que cualquiera que abogue por el uso libre de la Criptografía es poco menos que un delincuente, y que la necesita para encubrir sus crímenes. En ese caso, ¿por qué esas personas que *no tienen nada que ocultar* no envían todas sus cartas en tarjetas postales, para que todos leamos su contenido?, o ¿por qué se molestan si alguien escucha sus conversaciones telefónicas? Defender el ámbito de lo privado es un derecho inalienable de la persona, que en mi opinión debe prevalecer sobre la obligación que tienen los estados de perseguir a los delincuentes. Démosle a los gobiernos poder para entrometerse en nuestras vidas, y acabarán haciéndolo, no les quepa duda.

Uno de los elementos más polémicos acerca de los ataques indiscriminados a la intimidad es la red *Echelon*. Básicamente se trata de una red, creada por la NSA en 1980 —sus *precursoras* datan de 1952— en colaboración con Gran Bretaña, Australia y Nueva Zelanda, para monitorizar prácticamente todas las comunicaciones electrónicas —teléfono, e-mail y fax principalmente— del planeta, y buscar de manera automática ciertas palabras clave. La información obtenida iría a la NSA, que luego podría a su vez brindársela a otros países. El pretexto es, nuevamente, la lucha contra el terrorismo, pero podría ser empleada tanto para espionaje industrial —como presuntamente ha hecho durante años el Gobierno Francés, poniendo a disposición de sus propias compañías secretos robados a empresas extranjeras—, como para el *control* de aquellas personas que pueden representar amenazas políti-

cas a la *estabilidad* de la sociedad moderna. La Unión Europea reconoció la existencia de Echelon, pero hasta la fecha nadie ha exigido a ningún gobierno explicación alguna; es más, parece que los planes de la U.E. al respecto pasan por el despliegue de su propia red de vigilancia electrónica, llamada *Enfopol*. Si bien el proyecto se encuentra paralizado, es conveniente mantenerse en guardia, especialmente desde que los terribles atentados del 11 de septiembre de 2001 han propiciado una ola de limitación de las libertades civiles con el pretexto de la seguridad. Quizás algunos deberían recordar aquella famosa frase de Benjamin Franklin: *"Quienes son capaces de renunciar a la libertad esencial, a cambio de una seguridad transitoria, no son merecedores de la seguridad ni de la libertad."*

A la luz de las anteriores reflexiones, parece que adquiere un nuevo sentido el espectacular auge de las redes sociales. Ya no solo no tenemos *nada que ocultar*: es la propia sociedad la que nos anima a exhibir en público nuestra vida privada, proporcionando jugosos beneficios a empresas que trafican con nuestros datos, y facilitando la vigilancia de los ciudadanos, todo ello a través de servicios *gratuitos* que registran nuestra ubicación, actividades, gustos, interacciones sociales, etc. Podría decirse que hemos sacrificado una parte significativa de nuestra libertad a cambio de muy poco, y hemos dado pleno sentido a la frase que afirma que cuando alguien te ofrece un servicio gratis, es por que la mercancía eres tú.

Uno de los logros más importantes de la sociedad humana es la libertad de expresión. Naturalmente, lo ideal sería que todos pudiéramos expresar nuestros pensamientos con total libertad, y que cada cual se hiciera responsable de sus palabras. Sin embargo, todos sabemos que hay situaciones, incluso en ámbitos en los que supuestamente se respeta la libertad de expresión, en los que ciertas afirmaciones *inconvenientes* o políticamente incorrectas pueden dar lugar a represalias. Es necesario, por tanto, para poder garantizar la libertad, poder preservar también el anonimato. Por supuesto, es evidente que debemos otorgar menos crédito a aquellas cosas que se dicen bajo el paraguas del anonimato, muchas de ellas falsas e incluso dañinas, como las denominadas

fake news, pero el problema no viene de estas mentiras propiamente dichas, sino de su aceptación y propagación de manera acrítica por parte de la gente. No obstante, creo que sería peor no disponer de la posibilidad de poder expresarse en libertad. En este sentido la Criptografía, combinada con otras técnicas, es la única tecnología que puede permitirnos llegar a garantizar niveles razonables de anonimato. Después de todo, como dijo Thomas Jefferson, *"es preferible estar expuesto a los inconvenientes que surgen de un exceso de libertad que a los que provienen de una falta de ella."*

No cabe duda de que la información se está convirtiendo en la mayor fuente de poder que ha conocido la Humanidad, y que la Criptografía es una herramienta esencial para su control. Es necesario, pues, que los ciudadanos de a pie conozcamos sus ventajas e inconvenientes, sus peligros y leyendas. Dicen que vivimos en democracia pero, si a la gente no se le muestra toda la información relevante de manera honesta e imparcial, ¿cómo va a poder decidir su futuro? Esta obra pretende poner su pequeño granito de arena en ese sentido.

1.3. Números *grandes*

Los algoritmos criptográficos emplean claves con un elevado número de bits, y usualmente se mide su calidad por la cantidad de esfuerzo que se necesita para romperlos. El tipo de ataque más simple es la *fuerza bruta*, que simplemente trata de ir probando una a una todas las claves. Por ejemplo, el algoritmo DES tiene 2^{56} posibles claves. ¿Cuánto tiempo nos llevaría probarlas todas si, pongamos por caso, dispusiéramos de un computador capaz de hacer un millón de operaciones por segundo? Tardaríamos…¡más de 2200 años! Pero ¿y si la clave del ejemplo anterior tuviera 128 bits? El tiempo requerido sería de 10^{24} años.

Es interesante dedicar un apartado a tratar de fijar en nuestra imaginación la magnitud real de este tipo de números. En el cuadro 1.1

podemos observar algunos valores que nos ayudarán a comprender mejor la auténtica magnitud de muchos de los números que veremos en este texto. Observándola podremos apreciar que 10^{24} años es aproximadamente cien billones de veces la edad del universo (y eso con un ordenador capaz de ejecutar el algoritmo de codificación completo un millón de veces por segundo). Esto nos debería disuadir de emplear mecanismos basados en la fuerza bruta para *reventar* claves de 128 bits.

Para manejar la tabla con mayor rapidez, recordemos que un millón es aproximadamente 2^{20}, y que un año tiene más o menos 2^{24} segundos. Recorrer completamente un espacio de claves de, por ejemplo, 256 bits a razón de un millón por segundo supone $2^{256-44} = 2^{212}$ años de cálculo.

1.4. Acerca de la terminología empleada

En muchos libros sobre Criptografía y Seguridad se emplean términos como *encriptar* y *desencriptar*, adoptados con toda probabilidad del verbo anglosajón *encrypt*. Si bien estas expresiones ya están aceptadas por la Real Academia Española, hemos preferido emplear *cifrar–descifrar*, que estaba presente en nuestro idioma mucho antes. Otros vocablos, como la palabra *esteganografía*, hispanización del término inglés *steganography* —que a su vez proviene del título del libro '*Steganographia*', escrito por Johannes Trithemius en 1518—, no están presentes todavía en nuestro Diccionario. Nótese también que el término inglés *key* está traducido indistintamente mediante los vocablos *clave* o *llave*, que consideraremos equivalentes en la mayoría de los casos.

El lector podrá advertir que en este texto aparece el término *autentificación*, en lugar de *autenticación*. Quisiera hacer notar en este punto que ambos términos son correctos y están recogidos en el Diccionario de la Real Academia, y que aquí el uso del primero de ellos responde

Valor	Número
Probabilidad de ser fulminado por un rayo (por día)	1 entre 9.000.000.000 (2^{33})
Probabilidad de ganar la Lotería Primitiva Española	1 entre 13.983.816 (2^{23})
Probabilidad de ganar la Primitiva y caer fulminado por un rayo el mismo día	1 entre 2^{56}
Tiempo hasta la próxima glaciación	14.000 (2^{14}) años
Tiempo hasta que el Sol se extinga	10^9 (2^{30}) años
Edad del Planeta Tierra	10^9 (2^{30}) años
Edad del Universo	10^{10} (2^{34}) años
Duración de un año	10^{13} (2^{44}) microsegundos
Número de átomos en el Planeta Tierra	10^{51} (2^{170})
Número de átomos en el Sol	10^{57} (2^{189})
Número de átomos en la Vía Láctea	10^{67} (2^{223})
Número de átomos en el Universo (excluyendo materia oscura)	10^{77} (2^{255})
Masa de la Tierra	$5{,}9 \times 10^{24}$ (2^{82}) Kg
Masa del Sol	2×10^{30} (2^{100}) Kg
Masa de la Vía Láctea	2×10^{42} (2^{140}) Kg
Masa estimada del Universo (excluyendo materia oscura)	10^{50} (2^{166}) Kg
Volumen de la Tierra	10^{21} (2^{69}) m^3
Volumen del Sol	10^{27} (2^{89}) m^3
Volumen estimado del Universo	10^{82} (2^{272}) m^3

Cuadro 1.1: Algunos números *grandes*

simplemente a una cuestión de gustos personales.

1.5. Notación algorítmica

En este libro se describen diversos algoritmos de interés en Criptografía. La notación empleada en ellos es muy similar a la del lenguaje de programación $\mathcal{C}$, con objeto de que sea accesible al mayor número de personas posible. Si usted no conoce este lenguaje, siempre puede acudir a cualquier tutorial básico para poder entender los algoritmos de este libro, y después llevar a cabo sus propias implementaciones en cualquier otro lenguaje de programación. Sin embargo, aunque la notación que uso es parecida, no es exactamente la misma: allí donde el empleo de un $\mathcal{C}$ *puro* ponía en peligro la claridad en la descripción de los algoritmos, me he permitido pequeñas licencias. Tampoco he tenido en cuenta ni mucho menos la eficiencia de tiempo o memoria para estos algoritmos, por lo que mi sincero consejo es que no intenten *cortar y pegar* para realizar sus propias implementaciones.

Capítulo 2

Conceptos básicos

2.1. Criptografía

Según el Diccionario de la Real Academia, la palabra *criptografía* proviene de la unión de los términos griegos $\kappa\rho\upsilon\pi\tau\acute{o}\varsigma$ (oculto) y $\gamma\rho\acute{\alpha}\varphi\epsilon\iota\nu$ (escritura), y su definición es: *"Arte de escribir con clave secreta o de un modo enigmático"*. Obviamente la Criptografía hace años que dejó de ser un arte para convertirse en una técnica, o más bien un conglomerado de técnicas, que tratan sobre la protección —ocultamiento frente a observadores no autorizados— de la información. Entre las disciplinas que engloba cabe destacar la Teoría de la Información, la Teoría de Números —o Matemática Discreta, que estudia las propiedades de los números enteros—, y la Complejidad Algorítmica.

Existen dos trabajos fundamentales sobre los que se apoya prácticamente toda la teoría criptográfica actual. Uno de ellos, desarrollado por Claude Shannon en sus artículos *"A Mathematical Theory of Communication"* (1948) y *"Communication Theory of Secrecy Systems"* (1949), sienta las bases de la Teoría de la Información y de la Criptografía moderna. El segundo, publicado por Whitfield Diffie y Martin Hellman en 1976, se titulaba *"New directions in Cryptography"*, e introducía el concepto de Criptografía Asimétrica, abriendo enormemente el abani-

co de aplicación de esta disciplina.

Conviene hacer notar que la palabra Criptografía sólo hace referencia al uso de códigos, por lo que no engloba a las técnicas que se usan para romper dichos códigos, conocidas en su conjunto como *Criptoanálisis*. En cualquier caso ambas disciplinas están íntimamente ligadas; no olvidemos que cuando se diseña un sistema para cifrar información, hay que tener muy presente su posible criptoanálisis, ya que en caso contrario podríamos llevarnos desagradables sorpresas.

Finalmente, el término *Criptología*, aunque no está recogido aún en el Diccionario, se emplea habitualmente para agrupar Criptografía y Criptoanálisis.

2.2. Criptosistema

Definiremos un criptosistema como una quíntupla (M, C, K, E, D), donde:

- M representa el conjunto de todos los mensajes sin cifrar (lo que se denomina texto claro, o *plaintext*) que pueden ser enviados.
- C representa el conjunto de todos los posibles mensajes cifrados, o criptogramas.
- K representa el conjunto de claves que se pueden emplear en el criptosistema.
- E es el conjunto de *transformaciones de cifrado* o familia de funciones que se aplica a cada elemento de M para obtener un elemento de C. Existe una transformación diferente E_k para cada valor posible de la clave k.
- D es el conjunto de *transformaciones de descifrado*, análogo a E.

Todo criptosistema ha de cumplir la siguiente condición:

$$D_k(E_k(m)) = m \tag{2.1}$$

es decir, que si tenemos un mensaje m, lo ciframos empleando la clave k y luego lo desciframos empleando la misma clave, obtenemos de nuevo el mensaje original m.

Existen dos tipos fundamentales de criptosistemas:

- *Criptosistemas simétricos o de clave privada*. Son aquellos que emplean la misma clave k tanto para cifrar como para descifrar. Presentan el inconveniente de que para ser empleados en comunicaciones la clave k debe estar tanto en el emisor como en el receptor, lo cual nos lleva preguntarnos cómo transmitir la clave de forma segura.

- *Criptosistemas asimétricos o de llave pública*, que emplean una doble clave (k_p, k_P). k_p se conoce como *clave privada* y k_P se conoce como *clave pública*. Una de ellas sirve para la transformación E de cifrado y la otra para la transformación D de descifrado. En muchos casos son intercambiables, esto es, si empleamos una para cifrar la otra sirve para descifrar y viceversa. Estos criptosistemas deben cumplir además que el conocimiento de la clave pública k_P no permita calcular la clave privada k_p. Ofrecen un abanico superior de posibilidades, pudiendo emplearse para establecer comunicaciones seguras por canales inseguros —puesto que únicamente viaja por el canal la clave pública—, o para llevar a cabo autentificaciones.

En la práctica se emplea una combinación de estos dos tipos de criptosistemas, puesto que los segundos presentan el inconveniente de ser computacionalmente mucho más costosos que los primeros. En el *mundo real* se codifican los mensajes (largos) mediante algoritmos simétricos, que suelen ser muy eficientes, y luego se hace uso de la *criptografía asimétrica* para codificar las claves simétricas (cortas).

Claves *débiles*

En la inmensa mayoría de los casos los conjuntos M y C definidos anteriormente son iguales. Esto quiere decir que tanto los textos claros como los textos cifrados se representan empleando el mismo alfabeto —por ejemplo, cuando se usa el algoritmo DES, ambos son cadenas de 64 bits—. Por esta razón puede darse la posibilidad de que exista algún $k \in K$ tal que $E_k(M) = M$, lo cual sería catastrófico para nuestros propósitos, puesto que el empleo de esas claves dejaría todos nuestros mensajes... ¡sin codificar!

También puede darse el caso de que ciertas claves concretas generen textos cifrados de *poca calidad*. Una posibilidad bastante común en ciertos algoritmos es que algunas claves tengan la siguiente propiedad: $E_k(E_k(M)) = M$, lo cual quiere decir que basta con volver a codificar el criptograma para recuperar el texto claro original. Estas circunstancias podrían llegar a simplificar enormemente un intento de violar nuestro sistema, por lo que también habrá que evitarlas a toda costa.

La existencia de claves con estas características, como es natural, depende en gran medida de las peculiaridades de cada algoritmo en concreto, y en muchos casos también de los parámetros escogidos a la hora de aplicarlo. Llamaremos en general a las claves que no codifican *correctamente* los mensajes *claves débiles (weak keys* en inglés). Normalmente en un buen criptosistema la cantidad de claves débiles es cero o muy pequeña en comparación con el número total de claves posibles. No obstante, conviene conocer esta circunstancia para poder evitar en la medida de lo posible sus consecuencias.

2.3. Esteganografía

La esteganografía —o empleo de *canales subliminales*— consiste en ocultar en el interior de una información, aparentemente inocua, otro tipo de información (cifrada o no). Este método ha cobrado bastante

importancia últimamente debido a que permite burlar diferentes sistemas de control. Supongamos que un disidente político quiere enviar un mensaje fuera de su país, evitando la censura. Si lo codifica, las autoridades jamás permitirán que el mensaje atraviese las fronteras independientemente de que puedan acceder a su contenido, mientras que si ese mismo mensaje viaja camuflado en el interior de una imagen digital para una inocente felicitación navideña, tendrá muchas más posibilidades de llegar a su destino.

2.4. Criptoanálisis

El *criptoanálisis* consiste en comprometer la seguridad de un criptosistema. Esto se puede hacer descifrando un mensaje sin conocer la llave, o bien obteniendo a partir de uno o más criptogramas la clave que ha sido empleada en su codificación. No se considera criptoanálisis el descubrimiento de un algoritmo secreto de cifrado; hemos de suponer por el contrario que los algoritmos siempre son conocidos.

En general el criptoanálisis se suele llevar a cabo estudiando grandes cantidades de pares mensaje–criptograma generados con la misma clave. El mecanismo que se emplee para obtenerlos es indiferente, y puede ser resultado de *escuchar* un canal de comunicaciones, o de la posibilidad de que el objeto de nuestro ataque *responda* con un criptograma cuando le enviemos un mensaje. Obviamente, cuanto mayor sea la cantidad de pares, más probabilidades de éxito tendrá el criptoanálisis.

Uno de los tipos de análisis más interesantes es el de *texto claro escogido*, que parte de que conocemos una serie de pares de textos claros —elegidos por nosotros— y sus criptogramas correspondientes. Esta situación se suele dar cuando tenemos acceso al dispositivo de cifrado y éste nos permite efectuar operaciones, pero no nos permite leer su clave —por ejemplo, las tarjetas de los teléfonos móviles GSM—. El número de pares necesarios para obtener la clave desciende entonces

significativamente. Cuando el sistema es débil, pueden ser suficientes unos cientos de mensajes para obtener información que permita deducir la clave empleada.

También podemos tratar de criptoanalizar un sistema aplicando el algoritmo de descifrado, con todas y cada una de las claves, a un mensaje codificado que poseemos y comprobar cuáles de las salidas que se obtienen *tienen sentido* como posible texto claro. En general, todas las técnicas que buscan exhaustivamente por el espacio de claves K se denominan *de fuerza bruta*, y no suelen considerarse como auténticas técnicas de criptoanálisis, reservándose este término para aquellos mecanismos que explotan posibles debilidades intrínsecas en el algoritmo de cifrado. En general, se denomina *ataque* a cualquier técnica que permita recuperar un mensaje cifrado empleando menos esfuerzo computacional que el que se usaría por la fuerza bruta. Se da por supuesto que el espacio de claves para cualquier criptosistema digno de interés ha de ser suficientemente grande como para que los métodos basados en la fuerza bruta sean inviables. Hemos de tener en cuenta no obstante que la capacidad de cálculo de las computadoras crece a gran velocidad, por lo que algoritmos que hace unos años eran resistentes a la fuerza bruta hoy pueden resultar inseguros, como es el caso de DES. Sin embargo, existen longitudes de clave para las que resultaría imposible a todas luces, empleando computación *tradicional*, aplicar un método de este tipo. Por ejemplo, si diseñáramos una máquina capaz de recorrer todas las combinaciones que pueden tomar 256 bits, cuyo consumo fuera mínimo cada vez que se cambia el valor de uno de ellos[1], consumiría para completar su tarea una cantidad de energía equivalente a la que generan millones de soles como el nuestro a lo largo de toda su vida.

Un par de métodos de criptoanálisis que han dado interesantes resultados son el *análisis diferencial* y el *análisis lineal* (ver sección 10.7). El primero de ellos, partiendo de pares de mensajes con diferencias

[1]Según las Leyes de la Termodinámica, en condiciones ideales, la cantidad mínima de energía necesaria para poder modificar el estado de un sistema físico es de $3,76 \times 10^{-23}$ julios.

mínimas —usualmente de un bit—, estudia las variaciones que existen entre los mensajes cifrados correspondientes, tratando de identificar patrones comunes. El segundo emplea operaciones XOR entre algunos bits del texto claro y algunos bits del texto cifrado, obteniendo finalmente un único bit. Si realizamos esto con muchos pares de texto claro–texto cifrado podemos obtener una probabilidad p en ese bit que calculamos. Si p está suficientemente sesgada (no se aproxima a $\frac{1}{2}$), tendremos la posibilidad de recuperar la clave.

Otro tipo de análisis, esta vez para los algoritmos asimétricos, consistiría en tratar de deducir la llave privada a partir de la pública. Suelen ser técnicas analíticas que básicamente intentan resolver los problemas de elevado coste computacional en los que se apoyan estos criptosistemas: factorización, logaritmos discretos, etc. Mientras estos problemas genéricos permanezcan sin solución eficiente, podremos seguir confiando en estos algoritmos.

La Criptografía no sólo se emplea para proteger información, también se utiliza para permitir su autentificación, es decir, para identificar al autor de un mensaje e impedir que nadie suplante su personalidad. En estos casos surge un nuevo tipo de criptoanálisis que está encaminado únicamente a permitir que elementos falsos pasen por buenos. Puede que ni siquiera nos interese descifrar el mensaje original, sino simplemente poder sustituirlo por otro falso y que supere las pruebas de autentificación.

Como se puede apreciar, la gran variedad de sistemas criptográficos produce necesariamente gran variedad de técnicas de criptoanálisis, cada una de ellas adaptada a un algoritmo o familia de ellos. Con toda seguridad, cuando en el futuro aparezcan nuevos mecanismos de protección de la información, surgirán con ellos nuevos métodos de criptoanálisis. De hecho, la investigación en este campo es tan importante como el desarrollo de algoritmos criptográficos, y esto es debido a que, mientras que la presencia de fallos en un sistema es posible demostrarla, su ausencia es por definición indemostrable.

2.5. Compromiso entre criptosistema y criptoanálisis

En la sección 3.5 (pág. 54) veremos que pueden existir sistemas idealmente seguros, capaces de resistir cualquier ataque. También veremos que estos sistemas en la práctica carecen de interés, lo cual nos lleva a tener que adoptar un compromiso entre el coste del sistema —tanto computacional como de almacenamiento, e incluso económico— frente a su resistencia a diferentes ataques criptográficos.

La información posee un tiempo de vida, y pierde su valor transcurrido éste. Los datos sobre la estrategia de inversiones a largo plazo de una gran empresa, por ejemplo, tienen un mayor periodo de validez que la exclusiva periodística de una sentencia judicial que se va a hacer pública al día siguiente. Será suficiente, pues, tener un sistema que garantice que el tiempo que se puede tardar en comprometer su seguridad es mayor que el tiempo de vida de la propia información que éste alberga. Esto no suele ser fácil, sobre todo porque no tardará lo mismo un *oponente* que disponga de una única computadora de capacidad modesta, que otro que emplee una red de supercomputadores. Por eso también ha de tenerse en cuenta si la información que queremos proteger vale más que el esfuerzo de criptoanálisis que va a necesitar, porque entonces puede que no esté segura. La seguridad de los criptosistemas se suele medir en términos del número de computadoras y del tiempo necesarios para romperlos, y a veces simplemente en función del dinero necesario para llevar a cabo esta tarea con garantías de éxito.

En cualquier caso hoy por hoy existen sistemas que son muy poco costosos —o incluso gratuitos, como algunas versiones de PGP—, y que nos garantizan un nivel de protección tal que toda la potencia de cálculo que actualmente hay en el planeta sería insuficiente para romperlos.

Tampoco conviene depositar excesiva confianza en el algoritmo de cifrado, puesto que en el proceso de protección de la información exis-

ten otros puntos débiles que deben ser tratados con un cuidado exquisito. Por ejemplo, no tiene sentido emplear algoritmos con niveles de seguridad extremadamente elevados si luego escogemos contraseñas (*passwords*) ridículamente fáciles de adivinar. Una práctica muy extendida por desgracia es la de escoger palabras clave que contengan fechas, nombres de familiares, nombres de personajes o lugares de ficción, etc. Son las primeras que un atacante avispado probaría. Tampoco es una práctica recomendable anotarlas o decírselas a nadie, puesto que si la clave cae en malas manos, todo nuestro sistema queda comprometido, por buenos que sean los algoritmos empleados.

2.6. Seguridad en sistemas informáticos

Todo sistema que procese, almacene o transmita información tiene que cumplir una serie de requisitos. En primer lugar, ha de preservar la información frente a alteraciones tanto fortuitas como deliberadas, debidas a fallos en el *software* o en el *hardware*, provocadas por agentes externos —incendios, interrupciones en el suministro eléctrico, etc.— o por los propios usuarios. En segundo lugar, es necesario evitar accesos no autorizados tanto al sistema como a su contenido. Finalmente, el sistema debe garantizar que la información esté disponible cuando sea necesario. Estos tres requerimientos quedan recogidos en los conceptos de *integridad*, *confidencialidad* y *disponibilidad* de la información respectivamente, y son los que hacen que podamos considerar *seguro* a un sistema.

Por lo tanto, garantizar la seguridad de un sistema informático es un objetivo mucho más amplio y complejo que la simple protección de los datos mediante técnicas criptográficas. De hecho, hemos de tener en cuenta múltiples factores, tanto internos como externos. En esta sección comentaremos algunos de los más relevantes, de manera no exhaustiva.

Quizás la primera pregunta que haya que responder a la hora de

identificar los requerimientos de seguridad de un sistema sea la siguiente: *¿está conectado con el exterior?* En este sentido podemos hacer la siguiente subdivisión:

1. *Sistemas aislados*. Son los que no tienen acceso a ningún tipo de red. De unos años a esta parte se han convertido en minoría, debido al auge que han experimentado las redes, especialmente Internet. En ellos suele ser suficiente la implementación de mecanismos de control de acceso físico —cerraduras, videovigilancia, etc.—, junto con protocolos adecuados de gestión de los privilegios de cada usuario, si es que hay más de uno.

2. *Sistemas interconectados*. Constituyen el caso más general y extendido. De hecho, hoy por hoy casi cualquier ordenador está conectado a alguna red —y cada vez más dispositivos de uso cotidiano son auténticas computadoras: consolas de videojuegos, teléfonos celulares, reproductores multimedia, etc.—, enviando y recogiendo información del exterior casi constantemente. Esto hace que las redes de ordenadores sean cada día más complejas, y presenten auténticos desafíos de cara a gestionarlos adecuadamente.

En cuanto a las cuestiones de seguridad propiamente dichas, citaremos algunas de las más relevantes:

1. *Seguridad física*. Englobaremos dentro de esta categoría a todos los asuntos relacionados con la salvaguarda de los soportes físicos de la información, más que de la información propiamente dicha. En este nivel estarían, entre otras, las medidas contra incendios y sobrecargas eléctricas, la prevención de ataques terroristas, las políticas de copias de respaldo (*backups*), etc. También se suelen tener en cuenta dentro de este punto aspectos relacionados con la restricción del acceso físico a las computadoras.

2. *Seguridad de los canales de comunicación*. Los canales de comunicación rara vez se consideran seguros. Debido a que normalmente

escapan a nuestro control, ya que pertenecen a terceros, resulta imposible asegurarse de que no están siendo escuchados o intervenidos. En la inmensa mayoría de los casos tendremos que establecer mecanismos de protección de la información capaces de cumplir su cometido en canales manipulados, e incluso *hostiles*.

3. *Control de acceso a los datos*. Como ya hemos dicho, un sistema informático debe permitir acceder a la información únicamente a agentes autorizados. Generalmente, diferentes usuarios tendrán acceso a distinta información, por lo que una simple restricción del acceso al sistema no será suficiente, sino que habrá que establecer privilegios individualizados, así como mecanismos que, como el cifrado, permitan preservar la confidencialidad incluso frente a accesos físicos a los dispositivos de almacenamiento.

4. *Autentificación*. Para garantizar su correcto funcionamiento, es necesario poder verificar de forma fiable la autenticidad de los distintos elementos que interactúan en un sistema informático: la información que se recibe, envía y almacena, los usuarios que acceden a él, y eventualmente los dispositivos que se comunican con el mismo. En los dos últimos casos, hemos de evitar a toda costa que se produzcan problemas de *suplantación de identidad*.

5. *No repudio*. Cuando se recibe un mensaje no sólo es necesario poder identificar de forma unívoca al remitente, sino que éste asuma todas las responsabilidades derivadas de la información que haya podido enviar, por ejemplo en la firma de un contrato o en una transacción comercial. En este sentido es fundamental impedir que el emisor pueda *repudiar* un mensaje, es decir, negar su autoría sobre el mismo.

6. *Anonimato*. Es, en cierta manera, el concepto opuesto al de no repudio. En determinadas aplicaciones, como puede ser un proceso electoral o la denuncia de violaciones de los derechos humanos en entornos dictatoriales, es crucial garantizar el anonimato del ciudadano para poder preservar su intimidad y su li-

bertad. Sin embargo, el anonimato también puede ser empleado para practicar actividades delictivas con total impunidad, lo cual lo convierte en una auténtica arma de doble filo. En cualquier caso, se trata una característica realmente difícil de conseguir, y que no goza de muy buena fama, especialmente en países donde prima la *seguridad nacional* sobre la libertad y la intimidad de los ciudadanos. Si a eso le sumamos el interés que para muchas empresas tiene conocer los perfiles de actividad de sus clientes, de cara a personalizar sus ofertas, entenderemos por qué apenas hay iniciativas serias en la industria para proporcionar servicios de este tipo.

2.6.1. Tipos de autentificación

Como ya se ha dicho, el concepto de autentificación viene asociado a la comprobación del origen de la información, y de la identidad de los agentes que interactúan con un sistema. En general, y debido a los diferentes escenarios que pueden darse, distinguiremos tres tipos de autentificación:

- Autentificación *de mensaje*. Queremos garantizar la procedencia de un mensaje conocido, de forma que podamos asegurarnos de que no es una falsificación. Este proceso es el que subyace en las *firmas digitales*, o en los sistemas de credenciales a través de los cuales ciertos elementos de una red se identifican frente a otros.

- Autentificación *de usuario mediante contraseña*. En este caso se trata de garantizar la presencia física de un usuario legal en algún punto del sistema. Para ello deberá hacer uso de una información secreta —o *contraseña*—, que le permita identificarse.

- Autentificación *de dispositivo*. Se trata de garantizar la presencia frente al sistema de un dispositivo concreto. Este dispositivo puede ser autónomo e identificarse por sí mismo para interactuar con el sistema, o tratarse de una *llave electrónica* que sustituya o

complemente a la contraseña para facilitar la entrada a un usuario.

Nótese que la autentificación de usuario por medio de alguna característica biométrica, como pueden ser las huellas digitales, la retina, el iris, la voz, etc. puede reducirse a un problema de autentificación de dispositivo, solo que el *dispositivo* en este caso es el propio usuario.

Parte II

Fundamentos teóricos de la Criptografía

Capítulo 3

Teoría de la información

Comenzaremos el estudio de los fundamentos teóricos de la Criptografía dando una serie de nociones básicas sobre Teoría de la Información, introducida por Claude Shannon a finales de los años cuarenta. Esta disciplina permite efectuar una aproximación formal al estudio de la seguridad de cualquier algoritmo criptográfico, proporcionando incluso la demostración de que existen sistemas invulnerables frente a cualquier tipo de ataque, aún disponiendo de capacidad de computación infinita.

3.1. Cantidad de información

Vamos a introducir este concepto partiendo de su idea intuitiva. Para ello analizaremos el siguiente ejemplo: supongamos que tenemos una bolsa con nueve bolas negras y una blanca. ¿Cuánta información obtenemos si alguien nos dice que ha sacado una bola blanca de la bolsa? ¿Y cuánta obtenemos si después saca otra y nos dice que es negra?

Obviamente, la respuesta a la primera pregunta es que aporta bastante información, puesto que estábamos *casi seguros* de que la bola

tenía que salir negra. Análogamente si hubiera salido negra diríamos que ese suceso *no nos extraña* (nos suministra poca información). En cuanto a la segunda pregunta, claramente podemos contestar que el suceso no proporciona ninguna información, ya que al no quedar bolas blancas *sabíamos* que iba a salir negra.

Podemos fijarnos en la cantidad de información como una medida de la disminución de incertidumbre acerca de un suceso. Por ejemplo, si nos dicen que el número que ha salido en un dado es menor que dos, estamos recibiendo más información que si nos dicen que el número que ha salido es par.

Intuitivamente, se puede decir que la cantidad de información que obtenemos al observar un suceso crece cuando el número de posibilidades que éste presenta es mayor. Si existen diez posibilidades, la observación nos proporciona más información que si inicialmente tuviéramos dos. Por ejemplo, supone mayor información conocer la combinación ganadora del próximo sorteo de la Lotería Primitiva, que saber si una moneda lanzada al aire va a caer con la cara o la cruz hacia arriba. Claramente es más fácil acertar en el segundo caso, puesto que el número de posibilidades *a priori* —y por tanto la incertidumbre, suponiendo sucesos equiprobables— es menor.

También se puede observar que la cantidad de información varía según la probabilidad inicial de un suceso. En el caso de las bolas pueden pasar dos cosas: sacar bola negra, que es más probable, y sacar bola blanca, que es menos probable. Sacar una bola negra aumenta nuestro grado de *certeza* inicial de un 90 % a un 100 %, proporcionándonos una ganancia del 10 %. Sacar una bola blanca aumenta esa misma certeza en un 90 % —puesto que partimos de un 10 %—. Podemos considerar la disminución de incertidumbre proporcional al aumento de certeza, por lo cual diremos que el primer suceso —*sacar bola negra*— aporta menos información.

A partir de ahora, con objeto de simplificar la notación, vamos a emplear una variable aleatoria V para representar los posibles sucesos que podemos encontrar. Notaremos el suceso i-ésimo como x_i, $P(x_i)$

será la probabilidad asociada a dicho suceso, y n será el número de sucesos posibles.

Supongamos ahora que sabemos con toda seguridad que el único valor que puede tomar V es x_i. Saber el valor de V no va a aportar ninguna información, ya que lo conocemos de antemano. Por el contrario, si tenemos una certeza del 99 % sobre la posible ocurrencia de un valor cualquiera x_i, el hecho de obtener un x_j diferente proporciona bastante información, como ya hemos visto. Este concepto de información es cuantificable y se puede definir de la siguiente forma:

$$I_i = -\log_2 \left(P(x_i)\right) \tag{3.1}$$

siendo $P(x_i)$ la probabilidad del estado x_i. Obsérvese que si la probabilidad de un estado fuera 1 (máxima), la cantidad de información que aporta sería igual a 0, mientras que si su probabilidad se acercara a 0, tendería a $+\infty$ —esto es lógico, un suceso que no puede suceder nos aportaría una cantidad infinita de información si llegara a ocurrir—.

3.2. Entropía

Efectuando una suma ponderada de las cantidades de información de todos los posibles estados de una variable aleatoria V, obtenemos:

$$H(V) = -\sum_{i=1}^{n} P(x_i) \log_2 \left[P(x_i)\right] = \sum_{i=1}^{n} P(x_i) \log_2 \left[\frac{1}{P(x_i)}\right] \tag{3.2}$$

Esta magnitud $H(V)$ se conoce como la *entropía* de la variable aleatoria V. Sus propiedades son las siguientes:

I. $0 \leq H(V) \leq \log_2(N)$

II. $H(V) = 0 \iff \exists i$ tal que $P(x_i) = 1$ y $P(x_j) = 0 \; \forall j \neq i$

III. $H(x_1, x_2 \dots x_n) = H(x_1, x_2 \dots x_n, x_{n+1})$ si $P(x_{n+1}) = 0$

Como ejercicio vamos a demostrar la propiedad (I). Para ello emplearemos el *Lema de Gibbs*, que dice que dados dos sistemas de números $p_1, \dots p_n$ y $q_1, \dots q_n$ no negativos tales que

$$\sum_{i=1}^{n} p_i = \sum_{i=1}^{n} q_i$$

se verifica que

$$-\sum_{i=1}^{n} p_i \log_2(p_i) \leq -\sum_{i=1}^{n} p_i \log_2(q_i) \tag{3.3}$$

Entonces, si tomamos $p_i = P(x_i)$ y $q_i = \frac{1}{N}$, resulta que

$$-\sum_{i=1}^{n} p_i \log_2(p_i) \leq -\sum_{i=1}^{n} p_i \log_2\left(\frac{1}{N}\right)$$

y por lo tanto

$$H(X) \leq -\log_2\left(\frac{1}{N}\right) \sum_{i=1}^{n} p_i = \log_2(N)$$

Obsérvese que la entropía es proporcional a la longitud media de los mensajes que se necesitaría para codificar una serie de valores de V de manera óptima dado un alfabeto cualquiera. Esto quiere decir que cuanto más probable sea un valor individual, aportará menos información cuando aparezca, y podremos codificarlo empleando un mensaje más corto. Si $P(x_i) = 1$ no necesitaríamos ningún mensaje, puesto que sabemos de antemano que V va a tomar el valor x_i, mientras que si $P(x_i) = 0{,}9$ parece más lógico emplear mensajes cortos para representar el suceso x_i y largos para los x_j restantes, ya que el valor que más

veces va a aparecer en una secuencia de sucesos es precisamente x_i. Volveremos sobre este punto un poco más adelante.

Veamos unos cuantos ejemplos más:

- La entropía de la variable aleatoria asociada a lanzar una moneda al aire es la siguiente:

$$H(M) = -(0{,}5 \log_2(0{,}5) + 0{,}5 \log_2(0{,}5)) = 1$$

  Este suceso aporta exactamente una unidad de información.

- Si la moneda está trucada (60 % de probabilidades para cara, 40 % para cruz), se obtiene:

$$H(M_t) = -(0{,}6 \log_2(0{,}6) + 0{,}4 \log_2(0{,}4)) = 0{,}970$$

- Veamos el ejemplo de las bolas (nueve negras y una blanca):

$$H(B) = -(0{,}9 \log_2(0{,}9) + 0{,}1 \log_2(0{,}1)) = 0{,}468$$

La cantidad de información asociada al suceso más simple, que consta únicamente de dos posibilidades equiprobables —como el caso de la moneda sin trucar—, será nuestra unidad a la hora de medir esta magnitud, y la denominaremos *bit*. Esta es precisamente la razón por la que empleamos logaritmos base 2, para que la cantidad de información del suceso más simple sea igual a la unidad.

Podemos decir que la entropía de una variable aleatoria es el número medio de bits que necesitaremos para codificar cada uno de los estados de la variable, suponiendo que expresemos cada suceso empleando un mensaje escrito en un alfabeto binario. Imaginemos ahora que queremos representar los diez dígitos decimales usando secuencias de bits: con tres bits no tenemos suficiente, así que necesitaremos más, pero ¿cuántos más? Si usamos cuatro bits para representar todos los dígitos tal vez nos estemos pasando... Veamos cuánta entropía tienen diez sucesos equiprobables:

$$H = -\sum_{i=1}^{10} \frac{1}{10} \log_2 \left(\frac{1}{10}\right) = -\log_2 \left(\frac{1}{10}\right) = 3{,}32 bits$$

El valor que acabamos de calcular es el límite teórico, que normalmente no se puede alcanzar. Lo único que podemos decir es que no existe ninguna codificación que emplee longitudes promedio de mensaje inferiores al número que acabamos de calcular. Veamos la siguiente codificación: 000 para 0, 001 para 1, 010 para 2, 011 para 3, 100 para 4, 101 para 5 ,1100 para 6, 1101 para 7, 1110 para 8, y 1111 para 9. Con esta codificación empleamos, como media

$$\frac{3 \cdot 6 + 4 \cdot 4}{10} = 3{,}4 bits$$

para representar cada mensaje. Nótese que este esquema permite codificar una secuencia de números por simple yuxtaposición, sin ambigüedades, por lo que no necesitaremos símbolos que actúen de separadores, ya que éstos alargarían la longitud media de los mensajes. El denominado *Método de Huffman*, uno de los más utilizados en transmisión de datos, permite obtener codificaciones binarias que se aproximan bastante al óptimo teórico de una forma sencilla y eficiente.

3.3. Entropía condicionada

Supongamos que tenemos ahora una variable aleatoria bidimensional (X, Y). Recordemos las distribuciones de probabilidad más usuales que podemos definir sobre dicha variable, teniendo n posibles casos para X y m para Y:

1. Distribución conjunta de (X, Y):

$$P(x_i, y_j)$$

2. Distribuciones marginales de X e Y:

$$P(x_i) = \sum_{j=1}^{m} P(x_i, y_j) \qquad P(y_j) = \sum_{i=1}^{n} P(x_i, y_j)$$

3. Distribuciones condicionales de X sobre Y y viceversa:

$$P(x_i/y_j) = \frac{P(x_i, y_j)}{P(y_j)} \qquad P(y_j/x_i) = \frac{P(x_i, y_j)}{P(x_i)}$$

Definiremos la entropía de las distribuciones que acabamos de referir:

$$H(X,Y) = -\sum_{i=1}^{n}\sum_{j=1}^{m} P(x_i, y_j) \log_2 \left(P(x_i, y_j)\right)$$

$$H(X/Y = y_j) = -\sum_{i=1}^{n} P(x_i/y_j) \log_2 \left(P(x_i/y_j)\right)$$

Haciendo la suma ponderada de los $H(X/Y = y_j)$ obtenemos la expresión de la *Entropía Condicionada* de X sobre Y:

$$\begin{aligned} H(X/Y) &= -\sum_{i=1}^{n}\sum_{j=1}^{m} P(y_j)P(x_i/y_j) \log_2(P(x_i/y_j)) = \\ &= -\sum_{i=1}^{n}\sum_{j=1}^{m} P(x_i, y_j) \log_2(P(x_i/y_j)) \end{aligned} \tag{3.4}$$

Así como existe una *Ley de la Probabilidad Total*, análogamente se define la *Ley de Entropías Totales*:

$$H(X,Y) = H(X) + H(Y/X) \tag{3.5}$$

cumpliéndose además, si X e Y son variables independientes:

$$H(X,Y) = H(X) + H(Y) \tag{3.6}$$

Teorema de disminución de la entropía: La entropía de una variable X condicionada por otra Y es menor o igual a la entropía de X, alcanzándose la igualdad si y sólo si las variables X e Y son independientes.

Este teorema representa una idea intuitiva bien clara: conocer algo acerca de la variable Y puede que nos ayude a saber más sobre X —lo cual se debería traducir en una reducción de su entropía—, pero en ningún caso podrá hacer que aumente nuestra incertidumbre.

3.4. Cantidad de información entre dos variables

Shannon propuso una medida para la cantidad de información que aporta sobre una variable el conocimiento de otra. Se definirá, pues, la *cantidad de información de Shannon que la variable X contiene sobre Y* como:

$$I(X,Y) = H(Y) - H(Y/X) \tag{3.7}$$

La explicación intuitiva de esta magnitud es la siguiente. Inicialmente, nosotros poseemos un grado determinado de incertidumbre sobre la variable aleatoria Y. Si antes de medir una realización concreta de Y, medimos la de otra variable X, parece lógico que nuestra incertidumbre sobre Y se reduzca o permanezca igual. Por ejemplo, supongamos que Y representa la situación meteorológica (lluvia, sol, viento, nieve, etc.), mientras que X representa el atuendo de una persona que entra en nuestra misma habitación. Inicialmente tendremos un nivel determinado de entropía sobre Y. Si, acto seguido, la citada

persona aparece con un paraguas mojado, seguramente para nosotros aumentará la probabilidad para el valor *lluvia* de Y, modificando su entropía. Esa disminución —o no— de entropía es precisamente lo que mide $I(X, Y)$.

Las propiedades de la cantidad de información entre dos variables son las siguientes:

I. $I(X, Y) = I(Y, X)$
II. $I(X, Y) \geq 0$

3.5. Criptosistema seguro de Shannon

Diremos que un criptosistema es *seguro* si la cantidad de información que aporta el hecho de conocer el mensaje cifrado c sobre la entropía del texto claro m vale cero. En concreto, consideraremos una variable aleatoria C, compuesta por todos los criptogramas posibles, y cuya observación corresponderá el valor concreto c del criptograma enviado, y otra variable M, definida análogamente para los textos en claro m. En ese caso, tendremos que:

$$I(C, M) = 0 \tag{3.8}$$

Esto significa sencillamente que la distribución de probabilidad que nos inducen todos los posibles mensajes en claro —el conjunto M— no cambia si conocemos el mensaje cifrado. Para entenderlo mejor supongamos que sí se modifica dicha distribución: El hecho de conocer un mensaje cifrado, al variar la distribución de probabilidad sobre M haría unos mensajes más probables que otros, y por consiguiente unas claves de cifrado más probables que otras. Repitiendo esta operación muchas veces con mensajes diferentes, cifrados con la misma clave, podríamos ir modificando la distribución de probabilidad sobre la clave empleada hasta obtener un valor de clave

mucho más probable que todos los demás, permitiéndonos romper el criptosistema.

Si por el contrario el sistema cumpliera la condición (3.8), jamás podríamos romperlo, ni siquiera empleando una máquina con capacidad de proceso infinita. Por ello los criptosistemas que cumplen la condición de Shannon se denominan también *criptosistemas ideales*.

Se puede demostrar también que para que un sistema sea criptoseguro según el criterio de Shannon, la cardinalidad del espacio de claves ha de ser al menos igual que la del espacio de mensajes. En otras palabras, que la clave ha de ser al menos tan larga como el mensaje que queramos cifrar. Esto vuelve inútiles a estos criptosistemas en la práctica, porque si la clave es tanto o más larga que el mensaje, a la hora de protegerla nos encontraremos con el mismo problema que teníamos para proteger el mensaje.

Un ejemplo clásico de criptosistema seguro es el algoritmo inventado por Mauborgne y Vernam en 1917, que consistía en emplear como clave de codificación una secuencia de letras tan larga como el mensaje original, y usar cada carácter de la clave para cifrar exactamente una letra del mensaje, haciendo la suma módulo 26. Este sistema dio lugar a las secuencias de un solo uso (*one-time pads*): cadenas de longitud arbitraria que se combinan byte a byte con el mensaje original mediante la operación *or-exclusivo* u otra similar para obtener el criptograma.

3.6. Redundancia

Si una persona lee un mensaje en el que faltan algunas letras, normalmente puede reconstruirlo. Esto ocurre porque casi todos los símbolos de un mensaje en lenguaje natural contienen información que se puede extraer de los símbolos de alrededor —información que, en la práctica, se está enviando *dos o más veces*—, o en otras palabras, porque el lenguaje natural es *redundante*. Puesto que tenemos mecanismos para definir la cantidad de información que presenta un suceso,

podemos intentar medir el exceso de información (redundancia) de un lenguaje. Para ello vamos a dar una serie de definiciones:

- *Índice de un lenguaje*. Definiremos el índice de un lenguaje L para mensajes de longitud k como:

$$r_k = \frac{H_k(M)}{k} \tag{3.9}$$

  siendo $H_k(M)$ la entropía de todos los mensajes de longitud k que tienen sentido en L —los que no tienen sentido tienen probabilidad 0, y por lo tanto no contribuyen a la entropía resultante, como ya hemos visto—. Estamos midiendo el número de bits de información que transporta cada carácter en mensajes de una longitud determinada. Para idiomas como el castellano, r_k suele valer alrededor de $1{,}4$ bits por letra para valores pequeños de k.

- *Índice absoluto de un lenguaje*. Es el máximo número de bits de información que pueden ser codificados en cada carácter, asumiendo que todas las combinaciones de caracteres son igualmente probables. Suponiendo m símbolos diferentes en nuestro alfabeto este índice vale:

$$R = \frac{\log_2(m^k)}{k} = \frac{k \log_2(m)}{k} = \log_2(m)$$

  Nótese que el índice R es independiente de la longitud k de los mensajes. En el caso del castellano, puesto que tenemos 27 símbolos, podríamos codificar $4{,}7$ bits por cada letra aproximadamente, luego parece que el nivel de redundancia de los lenguajes *naturales* es alto.

- Finalmente, la *redundancia* de un lenguaje se define como la diferencia entre las dos magnitudes anteriores:

$$D = R - r_k$$

También se define el *índice de redundancia* como el siguiente cociente:

$$I = \frac{D}{R}$$

Para medir la auténtica redundancia de un lenguaje, hemos de tener en cuenta secuencias de cualquier número de caracteres, por lo que la expresión (3.9) debería calcularse en realidad como:

$$r_\infty = \lim_{n\to\infty} \frac{H_n(M)}{n} \tag{3.10}$$

Hay principalmente dos aplicaciones fundamentales de la Teoría de la Información, relacionadas directamente con la redundancia:

- *Compresión de datos*: simplemente trata de eliminar la redundancia dentro de un archivo, considerando cada byte como un mensaje elemental, y codificándolo con más o menos bits según su frecuencia de aparición. En este sentido se trata de codificar exactamente la misma información que transporta el archivo original, pero empleando un número de bits lo más pequeño posible.
- *Códigos de Redundancia Cíclica* (CRC): permiten introducir un campo de longitud mínima en el mensaje, tal que éste proporcione la mayor redundancia posible. Así, si el mensaje original resultase alterado, la probabilidad de que el CRC añadido siga siendo correcto es mínima.

Nótese que, conocidos los patrones de redundancia de un lenguaje, es posible dar de forma automática una estimación de si una cadena de símbolos corresponde o no a dicho lenguaje. Esta característica es aprovechada para efectuar ataques por la fuerza bruta, ya que ha de asignarse una probabilidad a cada clave individual en función de las características del mensaje obtenido al decodificar el criptograma con

dicha clave. El número de claves suele ser tan elevado que resulta imposible una inspección visual. Una estrategia bastante interesante para protegerse contra este tipo de ataques, y que suele emplearse con frecuencia, consiste en comprimir los mensajes antes de codificarlos. De esa manera eliminamos la redundancia y hacemos más difícil a un atacante apoyarse en las características del mensaje original para recuperar la clave.

3.7. Desinformación y distancia de unicidad

Definiremos *desinformación* de un sistema criptográfico como la entropía condicionada del conjunto M de posibles mensajes sobre el conjunto C de posibles criptogramas:

$$H(M/C) = -\sum_{m \in M} \sum_{c \in C} P(c)P(m/c)log_2(P(m/c)) \qquad (3.11)$$

Esta expresión permite saber la incertidumbre que queda sobre cuál ha sido el mensaje enviado m, suponiendo que conocemos su criptograma asociado c. Si esa incertidumbre fuera la misma que teníamos cuando desconocíamos el valor de c —en cuyo caso se cumpliría que $H(M) = H(M/C)$—, nos encontraríamos con que C y M son variables estadísticamente independientes, y por lo tanto estaríamos frente a un criptosistema seguro de Shannon, ya que jamás podríamos disminuir nuestra incertidumbre acerca de m a partir de los valores de c. Lo habitual no obstante es que exista relación estadística entre C y M (a través del espacio de claves K), por lo que $H(M/C) < H(M)$.

Adicionalmente, si el valor de $H(M/C)$ fuera muy pequeño con respecto a $H(M)$, significaría que el hecho de conocer c proporciona mucha información sobre m, lo cual quiere decir que nuestro criptosistema es inseguro. El peor de los casos sería que $H(M/C) = 0$, puesto que entonces, conociendo el valor de c tendríamos absoluta certeza sobre el valor de m.

Esta magnitud se puede medir también en función del conjunto K de claves, y entonces representará la incertidumbre que nos queda sobre k conocida c:

$$H(K/C) = -\sum_{k \in K} \sum_{c \in C} P(c)P(k/c)log_2(P(k/c)) \tag{3.12}$$

Definiremos finalmente la *distancia de unicidad* de un criptosistema como la longitud mínima de mensaje cifrado que aproxima el valor $H(K/C)$ a cero. En otras palabras, es la cantidad de texto cifrado que necesitamos para poder descubrir la clave. Los criptosistemas seguros de Shannon tienen distancia de unicidad infinita. Nuestro objetivo a la hora de diseñar un sistema criptográfico será que la distancia de unicidad sea lo más grande posible.

3.8. Confusión y difusión

Según la Teoría de Shannon, las dos técnicas básicas para ocultar la redundancia en un texto claro son la *confusión* y la *difusión*. Estos conceptos, a pesar de su antigüedad, poseen una importancia clave en Criptografía moderna.

- *Confusión*. Trata de ocultar la relación entre el texto claro y el texto cifrado. Recordemos que esa relación existe y se da a partir de la clave k empleada, puesto que si no existiera jamás podríamos descifrar los mensajes. El mecanismo más simple de confusión es la sustitución, que consiste en cambiar cada ocurrencia de un símbolo en el texto claro por otro. La sustitución puede ser tan simple o tan compleja como queramos.
- *Difusión*. Diluye la redundancia del texto claro *repartiéndola* a lo largo de todo el texto cifrado. El mecanismo más elemental para llevar a cabo una difusión es la transposición, que consiste en cambiar de sitio elementos individuales del texto claro.

3.9. Ejercicios resueltos

1. Calcule la información que proporciona el hecho de que en un dado no cargado salga un número par.

   Solución: La probabilidad de que en un dado no cargado salga un número par es $\frac{1}{2}$. Por lo tanto, empleando la expresión (3.1) tenemos que la información asociada al suceso vale:

$$I_{par} = -\log_2\left(\frac{1}{2}\right) = 1\ bit$$

2. Calcule la entropía que tiene un dado que presenta doble probabilidad para el número tres que para el resto.

   Solución: El dado presenta la siguiente distribución de probabilidad:

$$P(x=3) = \frac{2}{7}; \qquad P(x \neq 3) = \frac{1}{7}$$

   Su entropía será, pues

$$H(X) = -\frac{2}{7}\log_2\left(\frac{2}{7}\right) - 5\cdot\frac{1}{7}\log_2\left(\frac{1}{7}\right) = 0{,}5163 + 2{,}0052 = 2{,}5215$$

3. Demuestre el Lema de Gibbs, teniendo en cuenta la siguiente propiedad:

$$\forall x,\ 1 \geq x > 0 \Longrightarrow \log_2(x) \leq x - 1$$

   Solución: Sea el cociente $\frac{q_i}{p_i}$. Puesto que tanto p_i como q_i son positivos, su cociente también lo será, luego

$$\log_2\left(\frac{q_i}{p_i}\right) = \log_2(q_i) - \log_2(p_i) \leq \frac{q_i}{p_i} - 1$$

Multiplicando ambos miembros de la desigualdad por p_i se tiene

$$p_i \log_2(q_i) - p_i \log_2(p_i) \leq q_i - p_i$$

Puesto que p_i es positivo, se mantiene el sentido de la desigualdad. Ahora sumemos todas las desigualdades y obtendremos lo siguiente:

$$\sum_{i=1}^{n} p_i \log_2(q_i) - \sum_{i=1}^{n} p_i \log_2(p_i) \leq \sum_{i=1}^{n} q_i - \sum_{i=1}^{n} p_i = 0$$

Reorganizando los términos obtenemos finalmente la expresión buscada

$$-\sum_{i=1}^{n} p_i \log_2(p_i) \leq -\sum_{i=1}^{n} p_i \log_2(q_i)$$

4. Demuestre la Ley de Entropías Totales.

   Solución: Desarrollemos el valor de $H(Y/X)$, según la expresión (3.4):

$$H(Y/X) = \left[-\sum_{j=1}^{m} \sum_{i=1}^{n} P(x_i, y_j) \log_2(P(y_j/x_i)) \right]$$

   La Ley de la Probabilidad Total dice que

$$P(X, Y) = P(X) \cdot P(Y/X)$$

   por lo que nuestra expresión se convierte en

$$\left[-\sum_{j=1}^{m} \sum_{i=1}^{n} P(x_i, y_j) \log_2 \left(\frac{P(x_i, y_j)}{P(x_i)} \right) \right]$$

Descomponiendo el logaritmo del cociente como la diferencia de logaritmos se obtiene

$$\left[-\sum_{j=1}^{m}\sum_{i=1}^{n} P(x_i, y_j)\left[\log_2(P(x_i, y_j)) - \log_2(P(x_i))\right]\right]$$

Si desarrollamos la expresión anterior tenemos

$$\left[-\sum_{j=1}^{m}\sum_{i=1}^{n} P(x_i, y_j)\log_2(P(x_i, y_j))\right] + \\ + \left[\sum_{j=1}^{m}\sum_{i=1}^{n} P(x_i, y_j)\log_2(P(x_i))\right]$$

El primer sumando es igual a $H(X, Y)$. Observemos el último sumando:

$$\left[\sum_{j=1}^{m}\sum_{i=1}^{n} P(x_i, y_j)\log_2(P(x_i))\right] = \\ = \left[\sum_{i=1}^{n}\log_2(P(x_i))\sum_{j=1}^{m} P(x_i, y_j)\right] = \\ = \left[\sum_{i=1}^{n}\log_2(P(x_i))P(x_i)\right] = -H(X)$$

Luego $H(Y/X) = H(X, Y) - H(X)$. Reorganizando los términos, llegamos finalmente a la expresión de la Ley de Entropías Totales:

$$H(X, Y) = H(X) + H(Y/X)$$

5. Suponga un equipo de fútbol que nunca empata, que cuando no llueve vence el 65 % de sus partidos, y que si llueve sólo gana el 35 % de las veces. La probabilidad de que llueva en un partido es del 15 %. ¿Cuál es la cantidad de información que aporta la variable aleatoria *lluvia* sobre la variable *ganar un partido*?

   Solución: Sea G la variable aleatoria que representa los partidos. Sea g_s el suceso correspondiente a que el equipo gane el partido, y g_n el suceso asociado a que lo pierda. Análogamente, definiremos la variable L, asociada a que llueva o no. Tendremos, pues:

$$\begin{array}{lcl}
P(l_s) & = & 0{,}15 \\
P(l_n) & = & 0{,}85 \\
P(g_s, l_s) & = & 0{,}35 \cdot 0{,}15 = 0{,}0525 \\
P(g_s, l_n) & = & 0{,}65 \cdot 0{,}85 = 0{,}5525 \\
P(g_n, l_s) & = & 0{,}65 \cdot 0{,}15 = 0{,}0975 \\
P(g_n, l_n) & = & 0{,}35 \cdot 0{,}85 = 0{,}2975 \\
P(g_s/L = l_s) & = & 0{,}35 \\
P(g_s/L = l_n) & = & 0{,}65 \\
P(g_n/L = l_s) & = & 0{,}65 \\
P(g_n/L = l_n) & = & 0{,}35 \\
P(g_s) & = & P(l_n) \cdot P(g_s/L = l_n) + \\
 & & + P(l_s) \cdot P(g_s/L = l_s) = 0{,}605 \\
P(g_n) & = & P(l_n) \cdot P(g_n/L = l_n) + \\
 & & + P(l_s) \cdot P(g_n/L = l_s) = 0{,}395
\end{array}$$

   Calculemos ahora las entropías:

$$\begin{array}{lcl}
H(G) & = & -P(g_s)\log_2(P(g_s)) - P(g_n)\log_2(P(g_n)) = 0{,}9679 \\
H(G/L) & = & -P(g_s, l_s)\log_2(P(g_s/L = l_s)) - \\
 & & -P(g_s, l_n)\log_2(P(g_s/L = l_n)) - \\
 & & -P(g_n, l_s)\log_2(P(g_n/L = l_s)) - \\
 & & -P(g_n, l_n)\log_2(P(g_n/L = l_n)) = \\
 & = & -0{,}0525 \cdot \log_2(0{,}35) - 0{,}5525 \cdot \log_2(0{,}65) - \\
 & & -0{,}0975 \cdot \log_2(0{,}65) - 0{,}2975 \cdot \log_2(0{,}35) = 0{,}9333
\end{array}$$

La cantidad de información entre G y L es, finalmente

$$H(G) - H(G/L) = 0{,}9679 - 0{,}9333 = 0{,}0346\ bits$$

6. Suponga un conjunto de 20 mensajes equiprobables. ¿Cuál será la longitud media de cada mensaje para una transmisión óptima? Escriba un código binario que aproxime su longitud media de mensaje a ese valor óptimo.

   Solución:La longitud media óptima de los mensajes, cuando éstos son equiprobables, es el logaritmo base dos del número de mensajes, por tanto

   $$\log_2(20) = 4{,}3219$$

   Una posible codificación, con una longitud media de $4{,}4\ bits$ por mensaje, sería la siguiente:

m_0	00000	m_{10}	0110
m_1	00001	m_{11}	0111
m_2	00010	m_{12}	1000
m_3	00011	m_{13}	1001
m_4	00100	m_{14}	1010
m_5	00101	m_{15}	1011
m_6	00110	m_{16}	1100
m_7	00111	m_{17}	1101
m_8	0100	m_{18}	1110
m_9	0101	m_{19}	1111

7. Considere un conjunto de 11 mensajes, el primero con probabilidad 50 %, y el resto con probabilidad 5 %. Calcule su entropía.

   Solución: La entropía de los mensajes es igual a:

   $$H(X) = -0{,}5 \cdot \log_2(0{,}5) - 10 \cdot 0{,}05 \cdot \log_2(0{,}05) = 2{,}660\ bits$$

3.10. Ejercicios propuestos

1. Calcule la cantidad de información asociada a conocer el ganador de una carrera en la que compiten quince atletas, si suponemos que *a priori* todos los corredores tienen las mismas probabilidades de ganar. Calcule también la cantidad de información asociada a conocer también quiénes quedan en segundo y tercer puesto respectivamente.

2. Suponga que lanzamos dos dados y sumamos las puntuaciones obtenidas. Calcule la entropía asociada a dicho experimento.

3. Calcule el índice absoluto de un lenguaje con 32 símbolos. Calcule la redundancia de dicho lenguaje, sabiendo que su índice es de $2 bits/letra$.

Capítulo 4

Complejidad algorítmica

Cuando diseñamos un algoritmo criptográfico, pretendemos plantear a un posible atacante un problema que éste sea incapaz de resolver. Pero, ¿bajo qué circunstancias podemos considerar que un problema es *intratable*? Evidentemente, queremos que nuestro *fisgón* se enfrente a unos requerimientos de computación que no pueda asumir. La cuestión es cómo modelizar y cuantificar la capacidad de cálculo necesaria para abordar un problema. En este capítulo efectuaremos un breve repaso de las herramientas formales que nos van a permitir dar respuesta a estos interrogantes.

4.1. Concepto de algoritmo

En la actualidad, la práctica totalidad de las aplicaciones criptográficas emplean computadoras en sus cálculos, y las computadoras convencionales están diseñadas para ejecutar *algoritmos*. Definiremos algoritmo como una *secuencia finita y ordenada de instrucciones elementales que, dados los valores de entrada de un problema, en algún momento finaliza y devuelve la solución*.

En efecto, las computadoras actuales poseen una memoria, que les

sirve para almacenar datos, unos dispositivos de entrada y salida que les permiten comunicarse con el exterior, una unidad capaz de hacer operaciones aritméticas y lógicas, y una unidad de control, capaz de leer, interpretar y *ejecutar* un programa o secuencia de instrucciones. Habitualmente, las unidades aritmético–lógica y de control se suelen encapsular en un único circuito integrado, que se conoce por *microprocesador* o *CPU*.

Cuando nosotros diseñamos un algoritmo de cifrado, estamos expresando, de un modo más o menos formal, la estructura que ha de tener la secuencia de instrucciones concreta que permita implementar dicho algoritmo en cada computadora particular. Habrá computadoras con más o menos memoria, velocidad o incluso número de microprocesadores —capaces de ejecutar varios programas al mismo tiempo—, pero en esencia todas obedecerán al concepto de algoritmo.

La Teoría de Algoritmos es una ciencia que estudia cómo construir algoritmos para resolver diferentes problemas. En muchas ocasiones no basta con encontrar una forma de solucionar el problema: la solución ha de ser óptima. En este sentido la Teoría de Algoritmos también proporciona herramientas formales que nos van a permitir decidir qué algoritmo es mejor en cada caso, independientemente de las características particulares[1] de la computadora concreta en la que queramos implantarlo.

La Criptografía depende en gran medida de la Teoría de Algoritmos, ya que por un lado hemos de asegurar que el usuario legítimo, que posee la clave, puede cifrar y descifrar la información de forma rápida y cómoda, mientras que por otro hemos de garantizar que un atacante no dispondrá de ningún algoritmo eficiente capaz de comprometer el sistema.

Cabría plantearnos ahora la siguiente cuestión: si un mismo algoritmo puede resultar más rápido en una computadora que en otra,

[1] En algunos casos, sobre todo cuando se trata de computadoras con muchos microprocesadores, se estudian algoritmos específicos para aprovechar las peculiaridades de la máquina sobre la que se van a implantar.

¿podría existir una computadora capaz de ejecutar de forma eficiente algoritmos que sabemos que no lo son?. Existe un principio fundamental en Teoría de Algoritmos, llamado *principio de invarianza*, que dice que si dos implementaciones del mismo algoritmo consumen $t_1(n)$ y $t_2(n)$ segundos respectivamente, siendo n el tamaño de los datos de entrada, entonces existe una constante positiva c tal que $t_1(n) \leq c \cdot t_2(n)$, siempre que n sea lo suficientemente grande. En otras palabras, que aunque podamos encontrar una computadora más rápida, o una implementación mejor, la evolución del tiempo de ejecución del algoritmo en función del tamaño del problema permanecerá constante, por lo tanto la respuesta a la pregunta anterior es, afortunadamente, negativa. Eso nos permite centrarnos por completo en el algoritmo en sí y olvidarnos de la implementación concreta a la hora de hacer nuestro estudio.

En muchas ocasiones, el tiempo de ejecución de un algoritmo viene dado por las entradas concretas que le introduzcamos. Por ejemplo, se necesitan menos operaciones elementales para ordenar de menor a mayor la secuencia $\{1, 2, 3, 4, 6, 5\}$ que $\{6, 5, 3, 2, 1, 4\}$. Eso nos llevará a distinguir entre tres alternativas:

- *Mejor caso*: Es el número de operaciones necesario cuando los datos se encuentran distribuidos de la mejor forma posible para el algoritmo. Evidentemente este caso no es muy práctico, puesto que un algoritmo puede tener un mejor caso muy bueno y comportarse muy mal en el resto.
- *Peor caso*: Es el número de operaciones necesario para la distribución más pesimista de los datos de entrada. Nos permitirá obtener una cota superior del tiempo de ejecución necesario. Un algoritmo que se comporte bien en el peor caso, será siempre un buen algoritmo.
- *Caso promedio*: Muchas veces, hay algoritmos que en el peor caso no funcionan bien, pero en *la mayoría* de los casos que se presentan habitualmente tienen un comportamiento razonablemente eficiente. De hecho, algunos algoritmos típicos de ordenación

necesitan el mismo número de operaciones en el peor caso, pero se diferencian considerablemente en el caso promedio.

4.2. Complejidad algorítmica

En la mayoría de los casos carece de interés calcular el tiempo de ejecución concreto de un algoritmo en una computadora, e incluso algunas veces simplemente resulta imposible. En su lugar emplearemos una notación de tipo asintótico, que nos permitirá acotar dicha magnitud. Normalmente consideraremos el tiempo de ejecución del algoritmo como una función $f(n)$ del tamaño n de la entrada, y la llamaremos *orden de complejidad* del algoritmo. f debe estar definida para los números naturales y devolver valores en $\mathbb{R}^+$.

Dada la función $f(n)$, haremos las siguientes definiciones:

- *Límite superior asintótico*: $f(n) = O(g(n))$ si existe una constante positiva c y un número entero positivo n_0 tales que $0 \leq f(n) \leq cg(n)\ \forall n \geq n_0$.
- *Límite inferior asintótico*: $f(n) = \Omega(g(n))$ si existe una constante positiva c y un número entero positivo n_0 tales que $0 \leq cg(n) \leq f(n)\ \forall n \geq n_0$.
- *Límite exacto asintótico*: $f(n) = \Theta(g(n))$ si existen dos constantes positivas c_1, c_2 y un número entero positivo n_0 tales que $c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0$.
- *Notación o*: $f(n) = o(g(n))$ si para cualquier constante positiva c existe un número entero positivo $n_0 > 0$ tal que $0 \leq f(n) \leq cg(n)\ \forall n \geq n_0$.

Intuitivamente, $f(n) = O(g(n))$ significa que $f(n)$ crece asintóticamente no más rápido que $g(n)$ multiplicada por una constante. Análogamente $f(n) = \Omega(g(n))$ quiere decir que $f(n)$ crece asintóticamente al

menos tan rápido como $g(n)$ multiplicada por una constante. Definiremos ahora algunas propiedades sobre la notación que acabamos de introducir:

a) $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$.

b) $f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$.

c) Si $f(n) = O(h(n)) \wedge g(n) = O(h(n))$, entonces $(f + g)(n) = O(h(n))$.

d) Si $f(n) = O(h(n)) \wedge g(n) = O(l(n))$, entonces $(f \cdot g)(n) = O(h(n)l(n))$.

e) $f(n) = O(f(n))$.

f) Si $f(n) = O(g(n)) \wedge g(n) = O(h(n))$, entonces $f(n) = O(h(n))$.

Para algunas funciones de uso común, podemos definir directamente su orden de complejidad:

- *Funciones polinomiales*: Si $f(n)$ es un polinomio de grado k, y su coeficiente de mayor grado es positivo, entonces $f(n) = \Theta(n^k)$.
- *Funciones logarítmicas*: Para cualquier constante $c > 0$, $\log_c(n) = \Theta(\ln(n))$.
- *Factoriales*: $n! = \Omega(2^n)$.
- *Logaritmo de un factorial*: $\ln(n!) = \Theta(n \ln(n))$.

Veamos un ejemplo: supongamos que tenemos un algoritmo que necesita llevar a cabo $f(n) = 20n^2 + 10n + 1000$ operaciones elementales. Podemos decir que ese algoritmo tiene un orden de ejecución $\Theta(n^2)$, es decir, que el tiempo de ejecución crece, de forma asintótica, proporcionalmente al cuadrado del tamaño de la entrada. Otro algoritmo que necesite $g(n) = n^3 + 1$ operaciones efectuará menos cálculos para una entrada pequeña, pero su orden es $\Theta(n^3)$, por lo que crecerá mucho más rápidamente que el anterior y, en consecuencia, será menos eficiente.

4.2.1. Operaciones *elementales*

Hasta ahora hemos empleado el término *operaciones elementales* sin especificar su significado concreto. Podemos considerar una operación elemental como aquella que se ejecuta siempre en tiempo constante. Evidentemente, en función de las características concretas de la computadora que estemos manejando, habrá operaciones que podrán considerarse elementales o no. Por ejemplo, en una computadora que pueda operar únicamente con números de 16 bits, no podrá considerarse elemental una operación con números de 32 bits.

En general, el tamaño de la entrada a un algoritmo se mide en bits, y se consideran en principio elementales únicamente las operaciones a nivel de bit. Sean a y b dos números enteros positivos, ambos menores o iguales que n. Necesitaremos, pues, aproximadamente $\log_2(n)$ bits para representarlos —nótese que, en este caso, $\log_2(n)$ es el tamaño de la entrada—. Según este criterio, las operaciones aritméticas, llevadas a cabo mediante los algoritmos tradicionales, presentan los siguientes órdenes de complejidad:

- Suma ($a + b$): $O(\log_2(a) + \log_2(b)) = O(\log_2(n))$
- Resta ($a - b$): $O(\log_2(a) + \log_2(b)) = O(\log_2(n))$
- Multiplicación ($a \cdot b$): $O(\log_2(a) \cdot \log_2(b)) = O((\log_2(n))^2)$
- División (a/b): $O(\log_2(a) \cdot \log_2(b)) = O((\log_2(n))^2)$

Recordemos que el orden de complejidad de un logaritmo es independiente de la base, por lo que la capacidad de realizar en tiempo constante operaciones aritméticas con números de más bits únicamente introducirá un factor de proporcionalidad —recuérdese que $\log_a(x) = \log_b(x) \cdot \log_a(b)$—. Dicho factor no afecta al orden de complejidad obtenido, por lo que podemos considerar que estas operaciones se efectúan en grupos de bits de tamaño arbitrario. En otras palabras, una computadora que realice operaciones con números de 32 bits debería tardar la mitad en ejecutar el mismo algoritmo que otra que sólo

pueda operar con números de 16 bits pero, asintóticamente, el crecimiento del tiempo de ejecución en función del tamaño de la entrada será el mismo para ambas.

4.3. Algoritmos polinomiales, exponenciales y subexponenciales

Diremos que un algoritmo es *polinomial* si su peor caso de ejecución es de orden $O(n^k)$, donde n es el tamaño de la entrada y k es una constante. Adicionalmente, cualquier algoritmo que no pueda ser acotado por una función polinomial, se conoce como *exponencial*. En general, los algoritmos polinomiales se consideran eficientes, mientras que los exponenciales se consideran ineficientes.

Un algoritmo se denomina *subexponencial* si en el peor de los casos, la función de ejecución es de la forma $e^{o(n)}$, donde n es el tamaño de la entrada. Son asintóticamente más rápidos que los exponenciales puros, pero más lentos que los polinomiales.

4.4. Clases de complejidad

Para simplificar la notación, en muchas ocasiones se suele reducir el problema de la complejidad algorítmica a un simple problema de decisión, de forma que se considera un algoritmo como un mecanismo que permite obtener una respuesta *sí* o *no* a un problema concreto.

- La *clase de complejidad* **P** es el conjunto de todos los problemas de decisión que pueden ser resueltos en tiempo polinomial.
- La *clase de complejidad* **NP** es el conjunto de todos los problemas para los cuales una respuesta afirmativa puede ser verificada en

tiempo polinomial, empleando alguna información extra, denominada *certificado*.

- La *clase de complejidad* **co–NP** es el conjunto de todos los problemas para los cuales una respuesta negativa puede ser verificada en tiempo polinomial, usando un certificado apropiado.

Nótese que el hecho de que un problema sea **NP**, no quiere decir necesariamente que el certificado correspondiente sea fácil de obtener, sino que, dado éste último, puede verificarse la respuesta afirmativa en tiempo polinomial. Una observación análoga puede llevarse a cabo sobre los problemas **co–NP**.

Sabemos que **P** $\subseteq$ **NP** y que **P** $\subseteq$ **co-NP**. Sin embargo, aún no se sabe si **P** = **NP**, si **NP** = **co–NP**, o si **P** = **NP** $\cap$ **co–NP**. Si bien muchos expertos consideran que ninguna de estas tres igualdades se cumple, este punto no ha podido ser demostrado matemáticamente.

Dentro de la clase **NP**, existe un subconjunto de problemas que se llaman **NP**–completos, y cuya clase se nota como **NPC**. Estos problemas tienen la peculiaridad de que todos ellos son equivalentes, es decir, se pueden reducir unos en otros, y si lográramos resolver alguno de ellos en tiempo polinomial, los habríamos resuelto todos. También se puede decir que cualquier problema **NP**–completo es al menos tan difícil de resolver como cualquier otro problema **NP**, lo cual hace a la clase **NPC** la de los problemas más difíciles de resolver computacionalmente.

Sea $A = \{a_1, a_2, \ldots a_n\}$ un conjunto de números enteros positivos, y s otro número entero positivo. El problema de determinar si existe un subconjunto de A cuyos elementos sumen s es un problema **NP**–completo, y, como ya se ha dicho, todos los problemas de esta clase pueden ser reducidos a una instancia de este. Nótese que dado un subconjunto de A, es muy fácil verificar si suma s, y que dado *un subconjunto* de A que sume s —que desempeñaría el papel de certificado—, se puede verificar fácilmente que la respuesta al problema es afirmativa.

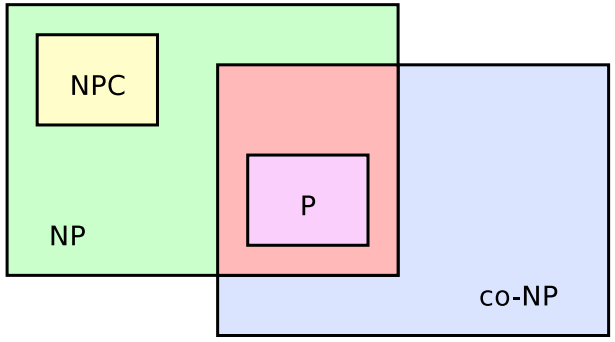

Figura 4.1: Relación entre las clases de complejidad **P, NP, co–NP y NPC.**

En la figura 4.1 puede observarse gráficamente la relación existente entre las distintas clases de complejidad que acabamos de definir.

Finalmente, apuntaremos que existe una clase de problemas, los denominados **NP**–duros —esta clase se define sobre los problemas en general, no sólo sobre los de decisión—, y que contiene la versión computacional del problema definido anteriormente, que consistiría en encontrar el subconjunto de A cuyos elementos suman s.

4.5. Algoritmos probabilísticos

Hasta ahora hemos estudiado la complejidad de algoritmos de tipo *determinístico*, que siempre siguen el mismo camino de ejecución y que siempre llegan —si lo hacen— a la misma solución. Sin embargo, existen problemas para los cuales puede ser más interesante emplear algoritmos de tipo no determinístico, también llamados probabilísticos o aleatorizados. Este tipo de algoritmos maneja algún tipo de parámetro aleatorio, lo cual hace que dos ejecuciones diferentes con los mismos datos de entrada no tengan por qué ser idénticas. En algunos casos,

métodos de este tipo permiten obtener soluciones en una cantidad de tiempo considerablemente inferior a la necesaria si se emplean algoritmos determinísticos (ver sección 5.7).

Podemos clasificar los algoritmos no determinísticos según la probabilidad con la que devuelvan la solución correcta. Sea A un algoritmo aleatorizado para el problema de decisión L, y sea I una instancia arbitraria de L. Sea $P1$ la probabilidad de que A devuelva *cierto* cuando I es cierto, y $P2$ la probabilidad de que A devuelva *cierto* cuando I es falso.

- A es de tipo *error nulo* si $P1 = 1$ y $P2 = 0$.
- A es de tipo *error simple* si $P1 \geq c$, siendo c una constante positiva, y $P2 = 0$
- A es de tipo *error doble* si $P1 \geq \frac{1}{2} + \epsilon$, y $P2 \leq \frac{1}{2} - \epsilon$

Definiremos también el tiempo esperado de ejecución de un algoritmo aleatorizado como el límite superior del *tiempo de ejecución esperado* para cada entrada, expresado en función del tamaño de la entrada. El tiempo de ejecución esperado para cada entrada será la media de los tiempos obtenidos para esa entrada y todas las posibles salidas del generador aleatorio.

Las clases de complejidad probabilística son las siguientes:

- Clase **ZPP**: conjunto de todos los problemas de decisión para los cuales existe un algoritmo de tipo *error nulo* que se ejecuta en un tiempo esperado de ejecución polinomial.
- Clase **RP**: conjunto de los problemas de decisión para los cuales existe un algoritmo de tipo *error simple* que se ejecuta en el peor caso en tiempo polinomial.
- Clase **BPP**: conjunto de los problemas de decisión para los cuales existe un algoritmo de tipo *error doble* que se ejecuta en el peor caso en tiempo polinomial.

Finalmente, diremos que $\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP} \subseteq \mathbf{BPP}$ y $\mathbf{RP} \subseteq \mathbf{NP}$.

4.6. Conclusiones

En este capítulo hemos contemplado únicamente aquellos problemas para los que existe una solución algorítmica —el programa *finaliza* siempre, aunque necesite un número astronómico de operaciones elementales—, y hemos dejado a un lado deliberadamente aquellos problemas para los cuales no existen algoritmos cuya finalización esté garantizada (problemas *no–decidibles* y *semidecicibles*), ya que en principio escapan al propósito de este libro.

Se han repasado las clases genéricas de problemas que se pueden afrontar, en función del tipo de algoritmos que permiten resolverlos, y se ha descrito una notación general para expresar de forma precisa la complejidad de un algoritmo concreto. Se ha puesto de manifiesto asimismo que un algoritmo ineficiente, cuando el tamaño de la entrada es lo suficientemente grande, es totalmente inabordable incluso para la más potente de las computadoras, al menos con la tecnología actual.

El hecho de que no se conozca un algoritmo eficiente para resolver un problema no quiere decir que éste no exista, y por eso es tan importante la Teoría de Algoritmos para la Criptografía. Si, por ejemplo, se lograra descubrir un método eficiente capaz de resolver logaritmos discretos (ver sección 5.4), algunos de los algoritmos asimétricos más populares en la actualidad dejarían de ser seguros. De hecho, la continua reducción del tiempo de ejecución necesario para resolver ciertos problemas, propiciada por la aparición de algoritmos más eficientes, junto con el avance de las prestaciones del *hardware* disponible, obliga con relativa frecuencia a actualizar las previsiones sobre la seguridad de muchos sistemas criptográficos.

Capítulo 5

Aritmética modular

5.1. Concepto de Aritmética modular

La aritmética modular es una parte de las Matemáticas extremadamente útil en Criptografía, ya que permite realizar cálculos complejos y plantear problemas interesantes, manteniendo siempre una representación numérica compacta y definida, puesto que sólo maneja un conjunto finito de números enteros. Mucha gente la conoce como la *aritmética del reloj*, debido a su parecido con la forma que tenemos de contar el tiempo. Por ejemplo, si son las 19:13:59 y pasa un segundo, decimos que son las 19:14:00, y no las 19:13:60. Como vemos, los segundos —al igual que los minutos—, se expresan empleando sesenta valores cíclicamente, de forma que tras el 59 viene de nuevo el 0. Desde el punto de vista matemático diríamos que los segundos se expresan *módulo 60*.

Empleemos ahora un punto de vista más formal y riguroso: Dados los números $a \in \mathbb{Z}$, $n, b \in \mathbb{N}$, con $b < n$, decimos que a *es congruente con* b *módulo* n, y se escribe:

$$a \equiv b \pmod{n}$$

si se cumple:

$$a = b + kn, \text{ para algún } k \in \mathbb{Z}$$

Por ejemplo, $37 \equiv 5 \pmod 8$, ya que $37 = 5 + 4 \cdot 8$. De hecho, los números 5, -3, 13, -11, 21, -19, 29... son todos equivalentes en la aritmética *módulo 8*, es decir, forman una *clase de equivalencia*. Como se puede apreciar, cualquier número entero pertenecerá necesariamente a alguna de esas clases, y en general, tendremos n clases de equivalencia *módulo* n (números congruentes con 0, números congruentes con 1, ..., números congruentes con $n-1$). Por razones de simplicidad, representaremos cada clase de equivalencia por un número comprendido entre 0 y $n-1$. De esta forma, en nuestro ejemplo (módulo 8) tendremos el conjunto de clases de equivalencia $\{0, 1, 2, 3, 4, 5, 6, 7\}$, al que denominaremos $\mathbb{Z}_8$. Podemos definir ahora las operaciones suma y producto en este tipo de conjuntos:

- $a + b \equiv c \pmod n \iff a + b = c + kn \qquad k \in \mathbb{Z}$
- $ab \equiv c \pmod n \iff ab = c + kn \qquad k \in \mathbb{Z}$

Propiedades de la suma:

- *Asociativa*: $\forall a, b, c \in \mathbb{Z}_n \quad (a + b) + c \equiv a + (b + c) \pmod n$
- *Conmutativa*: $\forall a, b \in \mathbb{Z}_n \quad a + b \equiv b + a \pmod n$
- *Elemento Neutro*: $\forall a \in \mathbb{Z}_n \quad \exists\, 0$ tal que $a + 0 \equiv a \pmod n$
- *Elemento Simétrico (opuesto)*: $\forall a \in \mathbb{Z}_n \quad \exists\, b$ tal que $a + b \equiv 0 \pmod n$

Propiedades del producto:

- *Asociativa*: $\forall a, b, c \in \mathbb{Z}_n \quad (a \cdot b) \cdot c \equiv a \cdot (b \cdot c) \pmod n$
- *Conmutativa*: $\forall a, b \in \mathbb{Z}_n \quad a \cdot b \equiv b \cdot a \pmod n$

- *Elemento Neutro*: $\forall a \in \mathbb{Z}_n \quad \exists\, 1$ tal que $a \cdot 1 \equiv a \pmod{n}$

Propiedades del producto con respecto de la suma:

- *Distributiva*: $\forall a, b, c \in \mathbb{Z}_n \quad (a + b) \cdot c \equiv (a \cdot c) + (b \cdot c) \pmod{n}$

La operación suma cumple las propiedades asociativa y conmutativa y posee elementos neutro y simétrico. Podemos decir por tanto que el conjunto $\mathbb{Z}_n$, junto con esta operación, tiene estructura de grupo conmutativo. A partir de ahora llamaremos grupo finito inducido por n a dicho conjunto.

Con la operación producto se cumplen las propiedades asociativa y conmutativa, y tiene elemento neutro, pero no necesariamente simétrico —recordemos que al elemento simétrico para el producto se le suele denominar *inverso*—. La estructura del conjunto con las operaciones suma y producto es, pues, de anillo conmutativo. Más adelante veremos bajo qué condiciones existe el elemento simétrico para el producto.

5.1.1. Algoritmo de Euclides

Quizá sea el algoritmo más antiguo que se conoce, y a la vez uno de los más útiles. Permite obtener de forma eficiente el mayor número entero que divide simultáneamente a otros dos números enteros o, lo que es lo mismo, su máximo común divisor.

Sean a y b dos números enteros de los que queremos calcular su máximo común divisor m. El Algoritmo de Euclides explota la siguiente propiedad:

$$m|a \wedge m|b \Longrightarrow m|(a - kb) \text{ con } k \in \mathbb{Z} \Longrightarrow m|(a \bmod b)$$

$a|b$ quiere decir que a *divide a* b, o en otras palabras, que b es múltiplo de a, mientras que $(a \bmod b)$ representa el resto de dividir a entre b.

En esencia estamos diciendo, que, puesto que m divide tanto a a como a b, debe dividir a su diferencia. Entonces si restamos k veces b de a, llegará un momento en el que obtengamos el resto de dividir a por b, o sea $a \text{ mód } b$.

Si llamamos c a $(a \text{ mód } b)$, podemos aplicar de nuevo la propiedad anterior y tenemos:

$$m|(b \text{ mód } c)$$

Sabemos, pues, que m tiene que dividir a todos los restos que vayamos obteniendo. Es evidente que el último de ellos será cero, puesto que los restos siempre son inferiores al divisor. El penúltimo valor obtenido es el mayor número que divide tanto a a como a b, o sea, el máximo común divisor de ambos. El algoritmo queda entonces como sigue:

```
int euclides(int a, int b)
  { int i;
    int g[];

    g[0]=a;
    g[1]=b;
    i=1;
    while (g[i]!=0)
      { g[i+1]=g[i-1]%g[i];
        i++;
      }
    return(g[i-1]);
  }
```

El invariante —condición que se mantiene en cada iteración— del Algoritmo de Euclides es el siguiente:

$$g_{i+1} = g_{i-1} \pmod{g_i}$$

y su orden de complejidad será de $O((log_2(n))^2)$ operaciones a nivel de bit, siendo n una cota superior de a y b.

5.1.2. Complejidad de las operaciones aritméticas en $\mathbb{Z}_n$

La complejidad algorítmica de las operaciones aritméticas modulares es la misma que la de las no modulares:

- Suma modular ($(a + b) \bmod n$): Basta con sumar a y b y, si el resultado es mayor que n, restarle n. Por lo tanto, su orden de complejidad es $O(\log_2(n))$.
- Resta modular ($(a-b) \bmod n$): Se realizaría igual que con la suma, solo que en este caso, si el resultado es negativo, sumamos n, por lo que tenemos un orden de complejidad de $O(\log_2(n))$.
- Multiplicación modular ($(a \cdot b) \bmod n$): En este caso, tendríamos la multiplicación entera, seguida de una división para obtener el resto, ambas operaciones con complejidad $O((\log_2(n))^2)$.

5.2. Cálculo de inversas en $\mathbb{Z}_n$

5.2.1. Existencia de la inversa

Hemos comentado en la sección 5.1 que los elementos de un grupo finito no tienen por qué tener inversa —elemento simétrico para el producto—. En este apartado veremos qué condiciones han de cumplirse para que exista la inversa de un número dentro de un grupo finito.

Definición: Un número entero $p \geq 2$ se dice *primo* si sus únicos divisores positivos son 1 y p. En caso contrario se denomina *compuesto*.

Definición: Dos números enteros a y b se denominan *primos entre sí* (o *coprimos*, o *primos relativos*), si $\text{mcd}(a, b) = 1$.

Lema: Dados a,$n \in \mathbb{N}$

$$\text{mcd}(a,n) = 1 \Longrightarrow ai \neq aj \pmod{n} \qquad \forall\, i \neq j \quad 0 < i,j < n \tag{5.1}$$

Demostración: Vamos a suponer que el lema es falso, es decir, que $\text{mcd}(a,n) = 1$, y que existen $i \neq j$ tales que $ai \equiv aj \pmod{n}$. Se cumple, pues:

$$n|(ai - aj) \Longrightarrow n|a(i-j)$$

puesto que a y n son primos entre sí, n no puede dividir a a, luego

$$n|(i-j) \Longrightarrow i \equiv j \pmod{n}$$

con lo que hemos alcanzado una contradicción. Por lo tanto, el lema no puede ser falso.

Ahora podemos hacer la siguiente reflexión: Si $ai \neq aj$ para cualesquiera $i \neq j$, multiplicar a por todos los elementos del grupo finito módulo n producirá una permutación de los elementos del grupo (exceptuando el cero), por lo que forzosamente ha de existir un valor tal que al multiplicarlo por a nos dé 1. Eso nos conduce al siguiente teorema:

Teorema: Si $\text{mcd}(a,n) = 1$, a tiene inversa módulo n.

Corolario: Si n es primo, el grupo finito que genera, con respecto a la suma y el producto, tiene estructura de cuerpo —todos sus elementos tienen inversa para el producto excepto el cero—. Estos cuerpos finitos tienen una gran importancia en Matemáticas, se denominan *Cuerpos de Galois*, y su notación es $GF(n)$.

5.2.2. Función de Euler

Llamaremos *conjunto reducido de residuos módulo* n —y lo notaremos $\mathbb{Z}_n^*$— al conjunto de números primos relativos con n. En otras palabras,

$\mathbb{Z}_n^*$ es el conjunto de todos los números que tienen inversa módulo n. Por ejemplo, si n fuera 12, su conjunto reducido de residuos sería:

$$\{1, 5, 7, 11\}.$$

Nótese que, en el caso de que n sea primo, el conjunto $\mathbb{Z}_n^*$ contendrá todos los valores comprendidos entre 1 y $n - 1$. Este conjunto tiene estructura de grupo, por lo que también se le conoce como *grupo multiplicativo* de $\mathbb{Z}_n$.

La siguiente expresión permite calcular el número de elementos, o cardinal, de $\mathbb{Z}_n^*$:

$$|\mathbb{Z}_n^*| = \prod_{i=1}^{n} p_i^{e_i - 1}(p_i - 1) \tag{5.2}$$

siendo p_i los factores primos de n y e_i su multiplicidad. En el caso de que n fuera el producto de dos números primos p y q,

$$|\mathbb{Z}_n^*| = (p - 1)(q - 1).$$

Ejemplo: Sea el conjunto reducido de residuos módulo 15, $\mathbb{Z}_{15}^*$. Como $15 = 3 \cdot 5$, el tamaño de este conjunto debería ser

$$\phi(15) = 3^0 \cdot (3 - 1) \cdot 5^0 \cdot (5 - 1) = 8.$$

Efectivamente, este conjunto tiene tamaño 8 y es igual a

$$\{1, 2, 4, 7, 8, 11, 13, 14\}.$$

Definición: La *función de Euler sobre* n, que notaremos como $\phi(n)$, es el cardinal de $\mathbb{Z}_n^*$, es decir:

$$\phi(n) = |\mathbb{Z}_n^*|$$

Teorema de Euler-Fermat: Si $\text{mcd}(a, n) = 1$:

$$a^{\phi(n)} \equiv 1 \pmod{n} \tag{5.3}$$

Demostración: Puesto que a y n son primos entre sí, a multiplicado por cualquier elemento del conjunto reducido de residuos módulo n $\{r_1, \ldots, r_{\phi(n)}\}$ ha de ser también primo con n, por lo tanto el conjunto $\{ar_1, \ldots, ar_{\phi(n)}\}$ no es más que una permutación del conjunto anterior, lo cual nos lleva a:

$$\prod_{i=1}^{\phi(n)} r_i = \prod_{i=1}^{\phi(n)} ar_i = a^{\phi(n)} \prod_{i=1}^{\phi(n)} r_i \Longrightarrow a^{\phi(n)} \equiv 1 \pmod{n}$$

Después de todo lo expuesto, queda claro que uno de los posibles métodos para calcular inversas módulo n, es precisamente la Función de Euler, puesto que:

$$a^{\phi(n)} = aa^{\phi(n)-1} \equiv 1 \pmod{n} \Longrightarrow a^{-1} \equiv a^{\phi(n)-1} \pmod{n}$$

No obstante, este método tiene el inconveniente de que necesitamos conocer los factores primos de n.

5.2.3. Algoritmo extendido de Euclides

El Algoritmo extendido de Euclides también puede ser empleado para calcular inversas. Es una ampliación del de Euclides, que posee el mismo orden de complejidad, y que se obtiene simplemente al tener en cuenta los cocientes además de los restos en cada paso. El invariante que mantiene es el siguiente, suponiendo que se le pasen como parámetros n y a:

$$g_i = nu_i + av_i$$

Los valores iniciales del algoritmo son $g_0 = n, u_0 = 1, v_0 = 0, g_1 = a, u_1 = 0$ y $v_1 = 1$, que cumplen el invariante de forma evidente. Siendo $\lfloor \cdot \rfloor$ el operador que nos devuelve la parte entera de un número, se calcula:

$$c = \left\lfloor \frac{g_{i-1}}{g_i} \right\rfloor$$

$$g_{i+1} = g_{i-1} \text{ mód } g_i$$

$$u_{i+1} = u_{i-1} - c \cdot u_i$$

$$v_{i+1} = v_{i-1} - c \cdot v_i$$

El último valor de g_i será el máximo común divisor entre a y n, que valdrá 1 si estos números son primos relativos, por lo que tendremos:

$$1 = nu_i + av_i$$

o sea,

$$av_i \equiv 1 \pmod{n}$$

luego $(v_i \text{ mód } n)$ será la inversa de a módulo n.

Nuestra segunda alternativa para calcular inversas, cuando desconozcamos $\phi(n)$, será pues el Algoritmo Extendido de Euclides. En la implementación que damos, como puede apreciarse, calculamos tanto los u_i como los v_i, aunque luego en la práctica sólo empleemos estos últimos. Obsérvese también la segunda cláusula `while`, que tiene como único fin que el valor devuelto esté comprendido entre 0 y $n-1$.

```
int inversa(int n, int a)
  { g[0]=n; g[1]=a;
    u[0]=1; u[1]=0;
    v[0]=0; v[1]=1;
    i=1;
    while (g[i]!=0)
      { c=g[i-1]/g[i];
        g[i+1]=g[i-1]%g[i];
        u[i+1]=u[i-1]-c*u[i];
        v[i+1]=v[i-1]-c*v[i];
        i++;
```

```
    }
  while (v[i-1]<0) v[i-1]=v[i-1]+n;
  return(v[i-1]%n);
}
```

5.3. Teorema chino del resto

El Teorema chino del resto es una potente herramienta matemática, que posee interesantes aplicaciones criptográficas.

Teorema: Sea $p_1, \ldots p_r$ una serie de números primos entre sí, y $n = p_1 \cdot p_2 \cdot \ldots \cdot p_r$, entonces el sistema de ecuaciones en congruencias

$$x \equiv x_i \pmod{p_i} \qquad i = 1, \ldots, r \tag{5.4}$$

tiene una única solución común en $[0, n-1]$, que viene dada por la expresión:

$$x = \sum_{i=1}^{r} \frac{n}{p_i} \left[(n/p_i)^{-1} \pmod{p_i} \right] x_i \pmod{n} \tag{5.5}$$

Demostración: Para cada i, mcd $\left[p_i, \frac{n}{p_i}\right] = 1$. Por lo tanto, cada $\frac{n}{p_i}$ debe tener una inversa y_i tal que

$$\left[\frac{n}{p_i}\right] y_i \equiv 1 \pmod{p_i}$$

También se cumple

$$\left[\frac{n}{p_i}\right] y_i \equiv 0 \pmod{p_j} \qquad \forall i \neq j$$

ya que $\frac{n}{p_i}$ es múltiplo de cada p_j.

Sea $x = \sum_{i=1}^{r} \frac{n}{p_i} y_i x_i \pmod{n}$. Entonces x es una solución a (5.4), ya que

$$x = \sum_{k \neq i} \frac{n}{p_k} y_k x_k + \frac{n}{p_i} y_i x_i = 0 + 1 \cdot x_i \equiv x_i \pmod{p_i}.$$

Como puede apreciarse, esta demostración nos proporciona además una solución al sistema de ecuaciones (5.4), lo cual puede resultarnos de gran utilidad para ciertas aplicaciones, como por ejemplo, el algoritmo RSA (ver sección 12.3).

5.4. Exponenciación. Logaritmos discretos

Como todos sabemos, la operación de exponenciación *(elevar* un número a a otro entero b), consiste en multiplicar a por sí mismo b veces, y se escribe a^b. En $\mathbb{Z}_n$ esta operación es exactamente igual, tomando como resultado el resto módulo n de la multiplicación. La operación inversa, denominada logaritmo discreto, presenta una dificultad intrínseca en la que se apoyan muchos algoritmos criptográficos.

5.4.1. Exponenciación en grupos finitos

(Pequeño) Teorema de Fermat: Si p es primo, entonces

$$a^{p-1} \equiv 1 \pmod{p} \tag{5.6}$$

Este teorema es consecuencia directa del Teorema de Euler-Fermat (expresión 5.3), ya que $\phi(p) = p - 1$. También nos permite deducir que si p es un número primo y

$$r \equiv s \pmod{p-1},$$

entonces

$$a^r \equiv a^s \pmod{p},$$

sea cual sea el valor de a. Por lo tanto, cuando trabajamos módulo p, siendo p primo, los exponentes pueden ser reducidos módulo $p-1$.

Definición: Sea $a \in \mathbb{Z}_n^*$. Se define el orden de a, denotado ord(a), como el menor entero positivo t tal que $a^t \equiv 1 \pmod{n}$.

Existe una interesante propiedad de ord(a). Si $a^s \equiv 1 \pmod{n}$, entonces ord(a) divide a s. En particular, tenemos que ord(a) siempre divide a $\phi(n)$.

Ejemplo: Sabemos que $\phi(15) = 8$, y que $\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$. Podemos comprobar que:

- ord$(1) = 1$.
- ord$(2) = 4$, ya que $2^4 = 16 \equiv 1 \pmod{15}$.
- ord$(4) = 2$, ya que $4^2 = 16 \equiv 1 \pmod{15}$.
- ord$(7) = 4$, ya que $7^4 = 2401 \equiv 1 \pmod{15}$.
- ord$(8) = 4$, ya que $8^4 = 4096 \equiv 1 \pmod{15}$.
- ord$(11) = 2$, ya que $11^2 = 121 \equiv 1 \pmod{15}$.
- ord$(13) = 4$, ya que $13^4 = 28561 \equiv 1 \pmod{15}$.
- ord$(14) = 2$, ya que $14^2 = 196 \equiv 1 \pmod{15}$.

De hecho, los conjuntos generados por los valores $a^k \pmod{n}$, con $a \in \mathbb{Z}_n^*$, se denominan cíclicos, ya que tienen un elemento generador a. Estos conjuntos se notan $\langle a \rangle$, y desempeñan un papel importante para determinados algoritmos de cifrado asimétrico, como el de Diffie Hellman (Capítulo 12).

Llamaremos *raíz primitiva módulo* n a cualquier valor $a \in \mathbb{Z}_n^*$ tal que $\langle a \rangle = \mathbb{Z}_n^*$. En ese caso, el orden de a tiene que ser exactamente $\phi(n)$.

Si observamos el ejemplo anterior, veremos que no existe ninguna raíz primitiva módulo 15.

Ejemplo: Sea $\mathbb{Z}_{14}^* = \{1, 3, 5, 9, 11, 13\}$. El conjunto $\langle 5 \rangle$, correspondiente a todos los valores $5^k \pmod{14}$, con k entero, es el siguiente:

$$\langle 5 \rangle = \{5^1 \equiv 5; 5^2 \equiv 11; 5^3 \equiv 13; 5^4 \equiv 9; 5^5 \equiv 3; 5^6 \equiv 1\}$$

Como puede observarse, $\langle 5 \rangle$ tiene exactamente los mismos elementos que $\mathbb{Z}_{14}^*$, solo que en diferente orden, lo cual equivale a decir que 5 es una raíz primitiva módulo 14 o, lo que es lo mismo, que 5 es un generador de $\mathbb{Z}_{14}^*$.

5.4.2. Símbolo de Legendre

El símbolo de Legendre permite saber si un número entero a es un residuo cuadrático módulo p, siendo p un número primo. En otras palabras, si existe un entero x tal que

$$x^2 \equiv a \pmod{p}. \tag{5.7}$$

Su notación es $\left(\frac{a}{p}\right)$ y solo puede tomar tres valores:

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{si } a \text{ es un residuo cuadrático módulo } p \\ -1 & \text{si } a \text{ no es un residuo cuadrático módulo } p \\ 0 & \text{si } a \text{ es múltiplo de } p. \end{cases} \tag{5.8}$$

El valor de $\left(\frac{a}{p}\right)$ puede calcularse mediante la siguiente expresión:

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p} \tag{5.9}$$

La justificación de esta fórmula se deriva directamente del Teorema de Fermat (ecuación 5.6). El caso de que a sea múltiplo de p es trivial,

así que supondremos que no lo es. Puesto que p es primo, $a^{(p-1)} \equiv 1 \pmod{p}$, por lo tanto, siempre se cumple que

$$a^{(p-1)} \equiv \left(a^{\frac{p-1}{2}}\right)^2 \equiv 1 \pmod{p}$$

De aquí se deduce que, puesto que su cuadrado es igual a 1, $a^{\frac{p-1}{2}}$ solo puede valer 1 o -1 módulo p.

Supongamos ahora que a es un residuo cuadrático, es decir, que $x^2 \equiv a \pmod{p}$. En ese caso, por el Teorema de Fermat:

$$\left(\frac{a}{p}\right) \equiv \left(x^2\right)^{\frac{p-1}{2}} \equiv x^{(p-1)} \equiv 1 \pmod{p}.$$

Supongamos ahora que a fuera un residuo cuadrático y que su símbolo de Legendre fuera -1. Tendríamos entonces lo siguiente:

$$\left(\frac{a}{p}\right) \equiv \left(x^2\right)^{\frac{p-1}{2}} \equiv x^{(p-1)} \equiv -1 \pmod{p},$$

lo cual es imposible por el mismo Teorema. Por lo tanto, si $\left(\frac{a}{p}\right) \equiv -1$, a no puede ser un residuo cuadrático módulo p

5.4.3. Algoritmo rápido de exponenciación

Muchos de los cifrados de clave pública emplean exponenciaciones dentro de grupos finitos para codificar los mensajes. Tanto las bases como los exponentes en esos casos son números astronómicos, incluso de miles de bits de longitud. Efectuar las exponenciaciones mediante multiplicaciones reiterativas de la base sería inviable. En esta sección veremos mecanismos eficientes para llevar a cabo estas operaciones.

Supongamos que tenemos dos números naturales a y b, y queremos calcular a^b. El mecanismo más sencillo sería multiplicar a por sí mismo

b veces. Sin embargo, para valores muy grandes de b este algoritmo no nos sirve.

Tomemos la representación binaria de b:

$$b = 2^0 b_0 + 2^1 b_1 + 2^2 b_2 + \ldots + 2^n b_n$$

Expresemos la potencia que vamos a calcular en función de dicha representación:

$$a^b = a^{2^0 b_0 + 2^1 b_1 + 2^2 b_2 + \ldots + 2^n b_n} = \prod_{i=0}^{n} a^{2^i b_i}$$

recordemos que los b_i sólo pueden valer 0 ó 1, por tanto para calcular a^b sólo hemos de multiplicar los a^{2^i} correspondientes a los dígitos binarios de b que valgan 1.

Nótese, además, que $a^{2^i} = (a^{2^{i-1}})^2$, por lo que, partiendo de a, podemos calcular el siguiente valor de esta serie elevando al cuadrado el anterior. El Algoritmo Rápido de Exponenciación queda como sigue:

```
int exp_rapida(int a, int b)
    { z=b;
      x=a;
      resul=1;
      while (z>0)
        { if (z%2==1)
             resul=resul*x;
          x=x*x;
          z=z/2;
        }
      return(resul);
    }
```

La variable `z` se inicializa con el valor de b y se va dividiendo por 2 en cada paso para tener siempre el i-ésimo bit de b en el menos significativo de `z`. En la variable `x` se almacenan los valores de a^{2^i}.

La extensión a $\mathbb{Z}_n$ de este algoritmo es muy simple, pues bastaría sustituir las operaciones producto por el producto módulo n, mientras que su orden de complejidad, siendo n una cota superior de a, b y a^b es de $O(\log(n))$ multiplicaciones sobre números de tamaño $\log(n)$, por lo que nos queda $O((\log(n))^3)$ operaciones a nivel de bit.

5.4.4. El problema de los logaritmos discretos

El problema inverso de la exponenciación es el cálculo de logaritmos discretos. Dados dos números a, b y el módulo n, se define el logaritmo discreto de a en base b módulo n como:

$$c = \log_b(a) \pmod{n} \iff a \equiv b^c \pmod{n} \tag{5.10}$$

En la actualidad no existen algoritmos eficientes que sean capaces de calcular en tiempo razonable logaritmos de esta naturaleza, y muchos esquemas criptográficos basan su resistencia en esta circunstancia. El problema de los logaritmos discretos está íntimamente relacionado con el de la factorización, de hecho está demostrado que si se puede calcular un logaritmo, entonces se puede factorizar fácilmente (el recíproco aún no se ha podido demostrar).

5.4.5. El problema de Diffie-Hellman

El problema de Diffie-Hellman está íntimamente relacionado con el problema de los Logaritmos Discretos, y es la base de algunos sistemas criptográficos de clave pública, como el de Diffie-Hellman (apartado 12.4) y el de ElGamal (apartado 12.5.1).

Antes de enunciarlo definiremos el término *generador*. Dado el conjunto $\mathbb{Z}_p^*$, con p primo, diremos que $\alpha \in \mathbb{Z}_p^*$ es un generador de $\mathbb{Z}_p^*$, si se cumple

$$\forall\, b \in \mathbb{Z}_p^*,\ \exists\, i \text{ tal que } \alpha^i = b$$

es decir, que $\langle \alpha \rangle = \mathbb{Z}_p^*$.

El enunciado del problema es el siguiente: dado un número primo p, un número α que sea un generador de $\mathbb{Z}_p^*$, y los elementos α^a y α^b, encontrar $\alpha^{ab} \pmod{p}$.

Nótese que nosotros conocemos α^a y α^b, pero no el valor de a ni el de b. De hecho, si pudiésemos efectuar de forma eficiente logaritmos discretos, sería suficiente con calcular a y luego $(\alpha^b)^a = \alpha^{ab}$.

5.4.6. El problema de los residuos cuadráticos

Otro problema interesante dentro de la aritmética modular, cuya solución es computacionalmente muy costosa, y que se emplea en algunas aplicaciones criptográficas, es el de los residuos cuadráticos.

Formalmente, dados dos números enteros a y n, decimos que a es un residuo cuadrático módulo n, si existe un número entero x tal que

$$a \equiv x^2 \pmod{n}. \tag{5.11}$$

El problema consiste en decidir, conocidos a y n, si existe x. En el caso de que n sea primo, podemos solucionarlo fácilmente mediante el símbolo de Legendre (sección 5.4.2), pero si n es el producto de dos números primos p_1 y p_2, a solo sería un residuo cuadrático módulo n si también lo fuera módulo p_1 y p_2. Por lo tanto, tendríamos que calcular los símbolos de Legendre $\left(\frac{a}{p_1}\right)$ y $\left(\frac{a}{p_2}\right)$, lo que nos obligaría a conocer la factorización de n, lo cual, como veremos más adelante, es un problema computacionalmente costoso.

5.5. Importancia de los números primos

Para explotar la dificultad de cálculo de logaritmos discretos, muchos algoritmos criptográficos de clave pública se basan en operaciones de exponenciación en grupos finitos. Dichos conjuntos deben cumplir la propiedad de que su módulo n sea un número muy grande con pocos factores —usualmente dos—. Estos algoritmos funcionan si se conoce n y sus factores se mantienen en secreto. Habitualmente para obtener n se calculan primero dos números primos muy grandes, que posteriormente se multiplican. Necesitaremos pues mecanismos para poder calcular esos números primos *grandes*.

La factorización es el problema inverso a la multiplicación: dado n, se trata de buscar un conjunto de números tales que su producto valga n. Normalmente, y para que la solución sea única, se impone la condición de que los factores de n que obtengamos sean todos primos elevados a alguna potencia. Al igual que para el problema de los logaritmos discretos, no existen algoritmos eficientes para efectuar este tipo de cálculos. Esto nos permite confiar en que, en la práctica, será imposible calcular los factores de n , incluso disponiendo de elevados recursos computacionales.

En cuanto al cálculo de primos grandes, bastaría con aplicar un algoritmo de factorización para saber si un número es primo o no. Este mecanismo es inviable, puesto que acabamos de decir que no hay algoritmos eficientes de factorización. Por suerte, sí que existen algoritmos probabilísticos que permiten decir con un grado de certeza bastante elevado si un número cualquiera es primo o compuesto.

Cabría preguntarse, dado que para los algoritmos asimétricos de cifrado necesitaremos generar muchos números primos, si realmente hay *suficientes*. De hecho se puede pensar que, a fuerza de generar números, llegará un momento en el que repitamos un primo generado con anterioridad. Podemos estar tranquilos, porque si a cada átomo del universo le asignáramos mil millones de números primos cada microsegundo desde su origen hasta hoy, harían falta un total de 10^{109}

números primos diferentes, mientras que el total estimado de números primos de 512 bits o menos es aproximadamente de 10^{151}.

También podríamos pensar en calcular indiscriminadamente números primos para luego emplearlos en algún algoritmo de factorización rápida. Por desgracia, si quisiéramos construir un disco duro que albergara diez mil GBytes por cada gramo de masa y milímetro cúbico para almacenar todos los primos de 512 bits o menos, el artilugio pesaría más de 10^{135} Kg y ocuparía casi 10^{130} metros cúbicos, es decir, sería miles de billones de veces más grande y pesado que la Vía Láctea.

5.6. Algoritmos de factorización

Como bien es sabido, la descomposición de un número entero $n = p_1^{e_1} \cdot p_2^{e_2} \ldots p_k^{e_k}$, siendo p_i números primos y e_i números enteros mayores que 0, es única. Cuando tratamos de obtener la factorización de n, normalmente nos conformamos con alcanzar una descomposición $n = a \cdot b$ no trivial —la descomposición trivial es aquella en la que $a = n$ y $b = 1$—. En tal caso, y puesto que tanto a como b son menores que n, podemos aplicar el mismo algoritmo de forma recursiva hasta que recuperemos todos los factores primos. Esta es la razón por la que los algoritmos de factorización suelen limitarse a dividir n en dos factores.

También conviene apuntar el hecho de que, como se verá en la sección 5.7, es mucho más eficiente comprobar si un número es primo que tratar de factorizarlo, por lo que normalmente se recomienda aplicar primero un test de primalidad para asegurarse de que el número puede descomponerse realmente de alguna manera no trivial.

Finalmente, queda la posibilidad de que n tenga un único factor, elevado a una potencia superior a 1. Afortunadamente, existen métodos capaces de verificar si n es una potencia perfecta x^k, con $k > 1$, por lo que todos los algoritmos que comentaremos en esta sección partirán

de la suposición de que n tiene al menos dos factores primos diferentes.

El algoritmo más sencillo e intuitivo para tratar de factorizar un número n es probar a dividirlo por todos los números enteros positivos comprendidos entre 2 y $\sqrt{n}$. Evidentemente, este método es del todo inaceptable en cuanto n alcanza valores elevados, y ha sido ampliamente mejorado por otras técnicas que, sin llegar a ser realmente *eficientes*, son mucho más rápidas que la fuerza bruta. En esta sección haremos un breve repaso a algunos de los métodos más interesantes aparecidos hasta la fecha.

5.6.1. La criba de Eratóstenes

Es uno de los métodos más antiguos que se conoce, y resulta muy útil para calcular todos los números primos inferiores a una cota n. Por supuesto, carece de utilidad práctica para números grandes, ya que sus requerimientos de tiempo y memoria resultan inasumibles.

La idea es crear un vector con n valores, e inicializarlos todos a 1. Empezando en la posición 2 del vector, hacemos el siguiente razonamiento: si el valor de la posición i del vector vale 1, es que el número i es primo; en ese caso, ponemos a 0 todos los valores del vector en posiciones múltiplo de i.

Puesto que, al poner todos los múltiplos de cada primo a 0, estamos marcando esos números como compuestos, solo valdrán 1 las posiciones del vector que no sean múltiplos de ningún número primo anterior, es decir, que sean números primos. De esta forma, al finalizar el algoritmo tendremos un 1 en todas las posiciones del vector correspondientes con números primos.

```
for (i=2;i<=n;i++) {
  b[i]=1;
}
```

```
i=2;
while (i*i<=n) {
  /* i es primo */
  for (p=2;p*i<n;p++)
    b[p*i]=0;
  do {
    i++;
    } while (b[i]==0);
}
```

5.6.2. Método de Fermat

Para factorizar n, el método de Fermat intenta representarlo mediante la expresión

$$n = x^2 - y^2 \tag{5.12}$$

con $x, y \in \mathbb{Z}$, $x, y \geq 1$. Es fácil ver que

$$n = (x + y)(x - y) = a \cdot b$$

donde a y b serán dos factores de n. El método de Fermat empieza tomando x_0 como el primer entero mayor que $\sqrt{n}$. Se comprueba entonces que $y_0 = x_0^2 - n$ es un cuadrado perfecto, y en caso contrario se calcula $x_{i+1} = x_i + 1$. Usando la siguiente expresión:

$$y_{i+1} = x_{i+1}^2 - n = (x_i + 1)^2 - n = x_i^2 - n + 2x_i + 1 = y_i + 2x_i + 1$$

se puede obtener el siguiente y_i haciendo uso únicamente de operaciones sencillas. En cuanto encontremos un y_i que sea un cuadrado perfecto, habremos dado con una factorización de n. Por ejemplo, vamos a intentar factorizar el número 481:

$$\begin{array}{lll} x_0 = 22 & y_0 = 3 & 2x_0 + 1 = 45 \\ x_1 = 23 & y_1 = 48 & 2x_1 + 1 = 47 \\ x_2 = 24 & y_2 = 95 & 2x_2 + 1 = 49 \\ x_3 = 25 & y_3 = 144 & \end{array}$$

Como puede verse, y_3 es el cuadrado de 12, luego podemos poner:

$$481 = (25 + 12)(25 - 12) = 13 \cdot 37$$

Este método permite aún varios refinamientos, pero en cualquier caso resulta inviable cuando el número n a factorizar es lo suficientemente grande, ya que presenta un orden de complejidad para el peor caso de $O(n)$ —nótese que al ser lineal en n, resulta exponencial en el tamaño de n—.

5.6.3. Método $p - 1$ de Pollard

Este método se basa en poseer un múltiplo cualquiera m de $p - 1$, siendo p un factor primo de n. Todo ello, por supuesto, sin conocer el valor de p. Para ello necesitaremos definir el concepto de *uniformidad*. Diremos que n es B*–uniforme* si todos sus factores primos son menores o iguales a B.

Llegados a este punto, suponemos que p es un factor de n y $p-1$ es B_1*–uniforme*, con B_1 suficientemente pequeño. Calcularemos m como el producto de todos los números primos inferiores a B_1, elevados a la máxima potencia que los deje por debajo de n. De esta forma, garantizamos que m es un múltiplo de $p - 1$, aunque no conozcamos el valor de p. Una vez obtenido el valor de m, el algoritmo de factorización queda como sigue:

1. Escoger un número a aleatorio dentro del conjunto $\{2, \ldots, n - 1\}$.
2. Calcular $d = \text{mcd}(a, n)$. Si $d > 1$, d es un factor de n. Fin.
3. Calcular $x = a^m \pmod{n}$.

4. Calcular $d = \text{mcd}(x-1, n)$. Si $d > 1$, d es un factor de n. Fin.

5. Devolver fallo en la búsqueda de factores de n. Fin.

Nótese que, en el paso 3, puesto que m es múltiplo de $p-1$, x debería ser congruente con 1 módulo p, luego $x-1$ debería ser múltiplo de p, por lo que el paso 4 debería devolver p.

Está demostrado que, si se estima bien el valor de B, este algoritmo tiene una probabilidad significativa de encontrar un valor de a que permita obtener un factor de n. En cualquier caso, ejecutándolo varias veces, es bastante probable que podamos hallar algún factor de n. Sin embargo, cuanto mayor sea B mayor carga computacional presenta este método.

Como ejemplo, vamos a tratar de factorizar el número 187, suponiendo que alguno de sus factores es 3–uniforme. En tal caso $m = 2^7 \cdot 3^4 = 10368$. Sea $a = 2$, entonces $x = (2^{10368} \text{ mód } 187) = 69$. Calculando $\text{mcd}(68, 187)$ nos queda 17, que divide a 187, por lo que $187 = 17 \cdot 13$.

El orden de eficiencia de este algoritmo es de $O(B \log_B(n))$ operaciones de multiplicación modular, suponiendo que n tiene un factor p tal que $p-1$ es B–uniforme.

5.6.4. Métodos cuadráticos de factorización

Los métodos cuadráticos de factorización se basan en la ecuación

$$x^2 \equiv y^2 \pmod{n} \tag{5.13}$$

Siempre y cuando $x \not\equiv \pm y \pmod{n}$, tenemos que $(x^2 - y^2)$ es múltiplo de n, y por lo tanto

$$n | (x-y)(x+y) \tag{5.14}$$

Adicionalmente, puesto que tanto x como y son menores que n, n no puede ser divisor de $(x+y)$ ni de $(x-y)$. En consecuencia, n ha de tener

factores comunes tanto con $(x+y)$ como con $(x-y)$, por lo que el valor $d = \text{mcd}(n, x-y)$ debe ser un divisor de n. Se puede demostrar que si n es impar, no potencia de primo y compuesto, entonces siempre se pueden encontrar x e y.

Para localizar un par de números satisfactorio, en primer lugar elegiremos un conjunto

$$F = \{p_0, p_1, \ldots, p_{t-1}\}$$

formado por t números primos diferentes, con la salvedad de que p_0 puede ser igual a -1. Buscaremos ahora ecuaciones en congruencias con la forma

$$x_i^2 \equiv z_i \pmod{n} \tag{5.15}$$

tales que z_i se pueda factorizar completamente a partir de los elementos de F. El siguiente paso consiste en buscar un subconjunto de los z_i tal que el producto de todos sus elementos, al que llamaremos z, sea un cuadrado perfecto. Puesto que tenemos la factorización de los z_i, basta con escoger estos de forma que la multiplicidad de sus factores sea par. Este problema equivale a resolver un sistema de ecuaciones lineales con coeficientes en $\mathbb{Z}_2$. Multiplicando los x_i^2 correspondientes a los factores de z escogidos, tendremos una ecuación del tipo que necesitamos, y por lo tanto una factorización de n.

Criba cuadrática

Este método se basa en emplear un polinomio de la forma

$$q(x) = (x+m)^2 - n$$

siendo $m = \lfloor \sqrt{n} \rfloor$, donde $\lfloor x \rfloor$ representa la parte entera de x. Puede comprobarse que

$$q(x) = x^2 + 2mx + m^2 - n \approx x^2 + 2mx$$

es un valor pequeño en relación con n, siempre y cuando x en valor absoluto sea pequeño. Si escogemos $x_i = a_i + m$ y $z_i = q(a_i)$, tendremos que se cumple la relación (5.15).

Lo único que nos queda es comprobar si z_i puede descomponerse totalmente con los elementos de F. Esto se consigue con la fase de criba, pero antes nos fijaremos en que si $p_i \in F$ divide a $q(x)$, también dividirá a $q(x + kp)$. Calcularemos la solución de la ecuación

$$q(x) \equiv 0 \pmod{p} \tag{5.16}$$

obteniendo una o dos series —dependiendo del número de soluciones que tenga la ecuación— de valores y tales que p divide a $q(y)$.

La criba propiamente dicha se lleva a cabo definiendo un vector $Q[x]$, con $-M \leq x \leq M$, que se inicializa según la expresión $Q[x] = \lfloor \log |q(x)| \rfloor$. Sean x_1, x_2 las soluciones a (5.16). Entonces restamos el valor $\lfloor \log(p) \rfloor$ a aquellas entradas $Q[x]$ tales que x sea igual a algún valor de las series de soluciones obtenidas en el paso anterior. Finalmente, los valores de $Q[x]$ que se aproximen a cero son los más susceptibles de ser descompuestos con los elementos de F, propiedad que se puede verificar de forma directa tratando de dividirlos.

Criba del cuerpo de números

Hoy por hoy es el algoritmo de factorización más rápido que se conoce, y fue empleado con éxito en 1996 para factorizar un número de 130 dígitos decimales. Es una extensión de la criba cuadrática, que emplea una segunda base de factores, esta vez formada por polinomios irreducibles. Los detalles de este método de factorización requieren unos conocimientos algebraicos que escapan a los contenidos de este libro, por lo que se recomienda al lector que acuda a la bibliografía si desea conocer más a fondo este algoritmo de factorización.

5.7. Tests de primalidad

Como ya hemos dicho, no es viable tratar de factorizar un número para saber si es o no primo, pero existen métodos probabilísticos que

nos pueden decir con un alto grado de certeza si un número es o no compuesto. En esta sección veremos algunos de los algoritmos más comunes para verificar que un número sea primo.

5.7.1. Método de Lehmann

Es uno de los tests más sencillos para saber si un número p es o no primo:

1. Escoger un número aleatorio $a < p$.
2. Calcular $b = a^{(p-1)/2} \pmod{p}$.
3. Si $b \neq 1 \pmod{p}$ y $b \neq -1 \pmod{p}$, p no es primo.
4. Si $b \equiv 1 \pmod{p}$ ó $b \equiv -1 \pmod{p}$, la probabilidad de que p sea primo es igual o superior al 50 %.

Repitiendo el algoritmo n veces, la probabilidad de que p supere el test y sea compuesto —es decir, no primo— será de 1 contra 2^n.

5.7.2. Método de Rabin-Miller

Es el algoritmo más empleado, debido a su facilidad de implementación. Sea p el número que queremos saber si es primo. Se calcula b, siendo b el número de veces que 2 divide a $(p-1)$, es decir, 2^b es la mayor potencia de 2 que divide a $(p-1)$. Calculamos entonces m, tal que $p = 1 + 2^b m$.

1. Escoger un número aleatorio $a < p$.
2. Sea $j = 0$ y $z = a^m \pmod{p}$.
3. Si $z = 1$, o $z = p - 1$, entonces p pasa el test y puede ser primo.

4. Si $j > 0$ y $z = 1$, p no es primo.

5. Sea $j = j + 1$. Si $j = b$ y $z \neq p - 1$, p no es primo.

6. Si $j < b$ y $z \neq p - 1$, $z = z^2 \pmod{p}$. Volver al paso (4).

7. Si $j < b$ y $z = p - 1$, entonces p pasa el test y puede ser primo.

8. p no es primo.

Para justificar este algoritmo, debemos tener en cuenta el hecho de que, si p es primo, no existe ningún valor x inferior a p, salvo 1 y $p - 1$, tal que $x^2 \equiv 1 \pmod{p}$. A modo de demostración, supongamos que existe un número a diferente de 1 y $p - 1$ tal que

$$a^2 \equiv 1 \pmod{p}$$

o, lo que es lo mismo:

$$a^2 - 1 = kp, \qquad k \in \mathbb{Z}.$$

Puesto que

$$a^2 - 1 = (a + 1)(a - 1)$$

tendríamos que p (que hemos supuesto que es primo) tendría que dividir a $(a+1)$ o a $(a-1)$, lo cual es una contradicción, ya que $a < p-1$.

Llegados a este punto, cabe recordar que, si p fuera primo, por el pequeño Teorema de Fermat, se cumple que $a^{\phi(p)} \equiv 1 \pmod{p}$, y que $\phi(p) = (p - 1)$. Puesto que $(p - 1)$ es un número par, lo podemos descomponer de la siguiente forma:

$$p - 1 = m2^b.$$

De esta manera, elevar cualquier número a a $\phi(p)$ es igual a calcular primero a^m y luego ir elevando al cuadrado el resultado b veces. Veamos cuáles son las alternativas posibles:

1. Que $a^m = 1$ o $a^m = (p-1)$. En ese caso todos los cuadrados de la serie serán iguales a 1, por lo que el número podría ser primo y pasa el test.

2. En caso contrario, si al elevar al cuadrado primero aparece un 1, tendríamos que existe un valor diferente de 1 y $p-1$, tal que su cuadrado es igual a 1 módulo p, por lo que p no puede ser primo.

3. Si el primer valor distinto de 1 que aparece es $p-1$, entonces la serie acabará en 1, y el número podría ser primo.

4. Si llegamos al penúltimo valor sin haber encontrado 1 ni $p-1$, el número ha de ser compuesto, ya que el cuadrado del penúltimo valor, diferente de 1 y $p-1$, sería igual a 1.

La probabilidad de que un número compuesto pase este algoritmo para un número a es del 25 %. Esto quiere decir que necesitaremos menos pasos para llegar al mismo nivel de confianza que el obtenido con el algoritmo de Lehmann.

5.7.3. Consideraciones prácticas

A efectos prácticos, el algoritmo que se suele emplear para generar aleatoriamente un número primo p es el siguiente:

1. Generar un número aleatorio p de n bits.

2. Poner a uno el bit más significativo —garantizamos que el número es de n bits— y el menos significativo —debe ser impar para poder ser primo—.

3. Intentar dividir p por una tabla de primos precalculados (usualmente aquellos que sean menores que 2000). Esto elimina gran cantidad de números no primos de una forma muy rápida. Baste decir a título informativo que más del 99.8 % de los números

impares no primos es divisible por algún número primo menor que 2000.

4. Ejecutar el test de Rabin-Miller sobre p como mínimo cinco veces.

5. Si el test falla, incrementar p en dos unidades y volver al paso 3.

5.7.4. Primos *fuertes*

Debido a que muchos algoritmos de tipo asimétrico (ver capítulo 12) basan su potencia en la dificultad para factorizar números enteros *grandes*, a lo largo de los años se propusieron diversas condiciones que debían cumplir los números empleados en aplicaciones criptográficas para que no fueran fáciles de factorizar. Se empezó entonces a hablar de *números primos fuertes*.

Sin embargo, en diciembre de 1998, Ronald Rivest y Robert Silverman publicaron un trabajo en el que quedaba demostrado que no era necesario emplear *primos fuertes* para los algoritmos asimétricos. En él se argumentaba que la supuesta necesidad de números de este tipo surgió para dificultar la factorización mediante ciertos métodos —como por ejemplo, el método $p - 1$—, pero la aparición de técnicas más modernas como la de Lenstra, basada en curvas elípticas, o la criba cuadrática, hacía que se ganase poco o nada con el empleo de este tipo de números primos.

No obstante, existen otros algoritmos criptográficos, basados en el problema del logaritmo discreto, para los que sí resulta conveniente que los números primos empleados cumplan determinadas condiciones, como el de Diffie-Hellman (sección **??**), o el algoritmo de firma digital DSA (sección 12.5.3).

5.8. Anillos de polinomios

Definición: Si tenemos un anillo conmutativo R, entonces un polinomio con variable x sobre el anillo R tiene la siguiente forma

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

donde cada $a_i \in R$ y $n \geq 0$. El elemento a_i se denomina *coeficiente i–ésimo* de $f(x)$, y el mayor m para el cual $a_m \neq 0$ se denomina *grado* de $f(x)$. Si $f(x) = a_0$ con $a_0 \neq 0$, entonces se dice que $f(x)$ tiene grado 0. Si todos los coeficientes de $f(x)$ valen 0, se dice que el grado de $f(x)$ es $-\infty$. Finalmente, un polinomio se dice *mónico* si su coeficiente de mayor grado vale 1.

Podemos definir las operaciones suma y producto de polinomios de la siguiente forma, siendo $f(x) = a_n x^n + \cdots + a_0$ y $g(x) = b_m x^m + \cdots + b_0$:

- *Suma*: $f(x) + g(x) = \sum c_r x^r$, donde $c_i = a_i + b_i$.
- *Producto*: $f(x) \cdot g(x) = \sum c_r x^r$, donde $c_i = \sum a_j b_k$, tal que $j + k = i$.

La suma de polinomios cumple las propiedades asociativa, conmutativa, elemento neutro y elemento simétrico, mientras que el producto cumple la asociativa, conmutativa y elemento neutro. El conjunto de polinomios definidos en un anillo R, que notaremos $R[x]$, con las operaciones suma y producto, tiene en consecuencia estructura de anillo conmutativo.

Dados $f(x), g(x) \in R[x]$, existen dos polinomios únicos $c(x)$ y $r(x)$, tales que $f(x) = g(x)c(x) + r(x)$. Esta operación es la división de polinomios, donde $c(x)$ desempeña el papel de cociente, y $r(x)$ el de resto, y tiene propiedades análogas a la de enteros. Eso nos permite definir una aritmética modular sobre polinomios, igual que la que ya conocemos para números enteros.

Definición: Se dice que $g(x)$ es congruente con $h(x)$ módulo $f(x)$, y se nota

$$g(x) \equiv h(x) \pmod{f(x)}$$

si

$$g(x) = h(x) + k(x)f(x), \text{ para algún } k(x) \in R[x].$$

Definición: Un polinomio $f(x)$ en $R[x]$ induce un conjunto de clases de equivalencia de polinomios en $R[x]$, donde cada clase posee al menos un representante de grado menor que el de $f(x)$. La suma y multiplicación pueden llevarse a cabo, por tanto, módulo $f(x)$, y tienen estructura de anillo conmutativo.

Definición: Decimos que un polinomio $f(x) \in R[x]$ de grado mayor o igual a 1 es *irreducible* si no puede ser puesto como el producto de otros dos polinomios de grado positivo en $R[x]$.

Aunque no lo demostraremos aquí, se puede deducir que si un polinomio es irreducible, el conjunto de clases de equivalencia que genera tiene estructura de cuerpo. Nótese que en este caso, el papel que desempeñaba un número primo es ahora ocupado por los polinomios irreducibles.

5.8.1. Polinomios en $\mathbb{Z}_n$

Puesto que, como ya sabemos, $\mathbb{Z}_n$ es un anillo conmutativo, podemos definir el conjunto $\mathbb{Z}_n[x]$ de polinomios con coeficientes en $\mathbb{Z}_n$.

Vamos a centrarnos ahora en el conjunto $\mathbb{Z}_2[x]$. En este caso, todos los coeficientes de los polinomios pueden valer únicamente 0 ó 1, por lo que un polinomio puede ser representado mediante una secuencia de bits. Por ejemplo, $f(x) = x^3 + x + 1$ podría representarse mediante el número binario 1011, y $g(x) = x^2 + 1$ vendría dado por el número 101.

Podemos ver que $f(x) + g(x) = x^3 + x^2 + x$, que viene dado por el número 1110. Puesto que las operaciones se realizan en $\mathbb{Z}_2$, esta

suma podría haber sido realizada mediante una simple operación *or-exclusivo* entre los números binarios que representan a $f(x)$ y $g(x)$. Como vemos, sería muy fácil implementar estas operaciones mediante *hardware*, y ésta es una de las principales ventajas de trabajar en $\mathbb{Z}_2[x]$.

Si escogemos un polinomio irreducible en $\mathbb{Z}_2$, podemos generar un cuerpo finito, o sea, un cuerpo de Galois. Dicho conjunto se representa como $GF(2^n)$, siendo n el grado del polinomio irreducible que lo genera, y tiene gran importancia en Criptografía, ya que algunos algoritmos de cifrado simétrico, como el estándar de cifrado AES, se basan en operaciones en $GF(2^n)$ (ver sección 10.5).

A modo de ejemplo, veamos cómo funciona la operación producto dentro de estos conjuntos. Tomemos el polinomio $f(x) = x^8+x^4+x^3+x+1$, que es irreducible en $\mathbb{Z}_2[x]$, y genera un cuerpo de Galois $GF(2^8)$. Vamos a multiplicar dos polinomios:

$$(x^5+x)\cdot(x^4+x^3+x^2+1) = x^9+x^8+x^7+x^5+x^5+x^4+x^3+x = \\ = x^9+x^8+x^7+x^4+x^3+x$$

Nótese que $x^5+x^5=0$, dado que los coeficientes están en $\mathbb{Z}_2$. Ahora hemos de tomar el resto módulo $f(x)$. Para ello emplearemos el siguiente truco:

$$x^8+x^4+x^3+x+1 \equiv 0 \pmod{f(x)} \Longrightarrow x^8 \equiv x^4+x^3+x+1 \pmod{f(x)}$$

luego

$$x^9+x^8+x^7+x^4+x^3+x = x(x^8)+x^8+x^7+x^4+x^3+x = \\ = x(x^4+x^3+x+1)+(x^4+x^3+x+1)+x^7+x^4+x^3+x = \\ = x^5+x^4+x^2+x+x^4+x^3+x+1+x^7+x^4+x^3+x = \\ = x^7+x^5+x^4+x^4+x^4+x^3+x^3+x^2+x+x+x+1 = \\ = x^7+x^5+x^4+x^2+x+1$$

La ventaja esencial que posee este tipo de conjuntos es que permite llevar a cabo implementaciones muy sencillas y paralelizables de los algoritmos aritméticos. En realidad, aunque el orden de complejidad

sea el mismo, se logra multiplicar la velocidad por una constante (y simplificar el diseño de los circuitos, si se trata de implementaciones por *hardware)*, por lo que se obtienen sistemas con mayores prestaciones, y a la vez más baratos.

5.9. Ejercicios resueltos

1. Comprobar las propiedades de la suma en grupos finitos.

   Solución: La suma en grupos finitos cumple las propiedades conmutativa y asociativa, además de la existencia de elementos neutro y simétrico. Tendremos en cuenta que:

   $$a \equiv b \pmod{n} \iff \exists k \in \mathbb{Z} \quad \text{tal que} \quad a = b + k \cdot n$$

   - *Propiedad conmutativa*: Puesto que $a+b = b+a$, tenemos que

     $$a + b = b + a + k \cdot n \quad \text{si} \quad k = 0, \text{ luego}$$

     $$a + b \equiv b + a \pmod{n}$$

   - *Propiedad asociativa*: Puesto que $a + (b + c) = (a + b) + c$, tenemos que

     $$a + (b + c) = (a + b) + c + k \cdot n \quad \text{si} \quad k = 0, \text{ luego}$$

     $$a + (b + c) \equiv (a + b) + c \pmod{n}$$

   - *Elemento neutro*: Trivialmente,

     $$a + 0 = a + k \cdot n \quad \text{si} \quad k = 0, \text{ luego}$$

     $$a + 0 \equiv a \pmod{n}$$

     por lo tanto, 0 es el elemento neutro para la suma.

- *Elemento simétrico*: Sea $a \in \mathbb{Z}_n$ y $b = n - a$, tenemos

$$a + b = a + (n - a) = k \cdot n \quad \text{si} \quad k = 1, \text{luego}$$

$$a + b \equiv 0 \pmod{n}$$

por tanto, b es el inverso de a para la suma.

2. Comprobar las propiedades del producto en grupos finitos.

Solución: El producto en grupos finitos cumple las propiedades conmutativa y asociativa, además de la existencia de elemento neutro. Tendremos en cuenta, al igual que en el ejercicio anterior, que:

$$a \equiv b \pmod{n} \iff \exists k \in \mathbb{Z} \quad \text{tal que} \quad a = b + k \cdot n$$

- *Propiedad conmutativa*: Puesto que $a \cdot b = b \cdot a$, tenemos que

$$a \cdot b = b \cdot a + k \cdot n \quad \text{si} \quad k = 0, \text{luego}$$

$$a \cdot b \equiv b \cdot a \pmod{n}$$

- *Propiedad asociativa*: Puesto que $a \cdot (b \cdot c) = (a \cdot b) \cdot c$, tenemos que

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c + k \cdot n \quad \text{si} \quad k = 0, \text{luego}$$

$$a \cdot (b \cdot c) \equiv (a \cdot b) \cdot c \pmod{n}$$

- *Elemento neutro*: Trivialmente,

$$a \cdot 1 = a + k \cdot n \quad \text{si} \quad k = 0, \text{luego}$$

$$a \cdot 1 \equiv a \pmod{n}$$

por lo tanto, 1 es el elemento neutro para el producto.

3. Calcular el valor de la función ϕ de Euler para los siguientes números: 64, 611, 2197, 5, 10000.

Solución: Para calcular el valor de la función ϕ emplearemos la expresión (5.2):

$64 = 2^6$	$\phi(64) = 2^5 \cdot (2-1) = 32$
$611 = 13 \cdot 47$	$\phi(611) = (13-1) \cdot (47-1) = 552$
$2197 = 13^3$	$\phi(2197) = 13^2 \cdot 12 = 2028$
5 es primo	$\phi(5) = 5 - 1 = 4$
$10000 = 2^4 \cdot 5^4$	$\phi(10000) = 2^3 \cdot 1 \cdot 5^3 \cdot 4 = 4000$

4. Resolver el siguiente sistema de ecuaciones en congruencias:

$$\begin{aligned} x &\equiv 12 \pmod{17} \\ x &\equiv 13 \pmod{64} \\ x &\equiv 8 \pmod{27} \end{aligned}$$

Solución: Emplearemos la expresión (5.5) para resolver el sistema:

$$\begin{aligned} x &\equiv 12 \pmod{17} \\ x &\equiv 13 \pmod{64} \\ x &\equiv 8 \pmod{27} \end{aligned}$$

Puesto que $n = 17 \cdot 64 \cdot 27 = 29376$, tenemos

$$\begin{aligned} x = \; & (29376/17)[1728^{-1} \pmod{17}] \cdot 12 + \\ & +(29376/64)(459^{-1} \pmod{64}) \cdot 13 + \\ & +(29376/27)(1088^{-1} \pmod{27}) \cdot 8 \end{aligned}$$

Calculamos ahora las inversas:

$$\begin{aligned} 1728 &\equiv 11 \pmod{17}, \quad 11^{-1} \pmod{17} = 14 \\ 459 &\equiv 11 \pmod{64}, \quad 11^{-1} \pmod{64} = 35 \\ 1088 &\equiv 8 \pmod{27}, \quad 8^{-1} \pmod{27} = 17 \end{aligned}$$

Sustituyendo los valores, nos queda

$$x = 1728 \cdot 14 \cdot 12 + 459 \cdot 35 \cdot 13 + 1088 \cdot 17 \cdot 8 = 647117 \equiv 845 \pmod{29376}$$

5. ¿Cómo calcularía el valor de $(2^{10368} \bmod 187)$, empleando únicamente lápiz, papel y calculadora?

Solución: Para calcular el valor de $(2^{10368} \bmod 187)$, se puede emplear el algoritmo de exponenciación rápida (apartado 5.4.3):

$r =$	1	$z =$	10368	$x =$	2
$r =$	1	$z =$	5184	$x =$	4
$r =$	1	$z =$	2592	$x =$	16
$r =$	1	$z =$	1296	$x =$	256 $(\text{mód } 187) = 69$
$r =$	1	$z =$	648	$x =$	4761 $(\text{mód } 187) = 86$
$r =$	1	$z =$	324	$x =$	7396 $(\text{mód } 187) = 103$
$r =$	1	$z =$	162	$x =$	10609 $(\text{mód } 187) = 137$
$r =$	1	$z =$	81	$x =$	18769 $(\text{mód } 187) = 69$
$r =$	69	$z =$	40	$x =$	4761 $(\text{mód } 187) = 86$
$r =$	69	$z =$	20	$x =$	7396 $(\text{mód } 187) = 103$
$r =$	69	$z =$	10	$x =$	10609 $(\text{mód } 187) = 137$
$r =$	69	$z =$	5	$x =$	18769 $(\text{mód } 187) = 69$
$r =$	4761	$z =$	2	$x =$	4761 $(\text{mód } 187) = 86$
$r =$	86	$z =$	1	$x =$	7396 $(\text{mód } 187) = 103$
$r =$	8858				

6. Calcule la suma y el producto de los polinomios correspondientes a los números binarios 100101 y 1011, dentro del $GF(2^6)$ definido por el polinomio irreducible $f(x) = x^6 + x + 1$.

   Solución: Para calcular la suma, es suficiente con aplicar un *or-exclusivo* entre ambos, por lo tanto:

   $$100101 \oplus 1011 = 101110 = x^5 + x^3 + x^2 + x$$

   En cuanto al producto, tenemos lo siguiente:

   $$\begin{aligned}&(x^5 + x^2 + 1)(x^3 + x + 1) = \\ &= x^8 + x^6 + x^5 + x^5 + x^3 + x^2 + x^3 + x + 1 = \\ &= x^8 + x^6 + x^2 + x + 1\end{aligned}$$

   Ahora nos queda calcular el módulo $x^6 + x + 1$. Para ello aplicaremos la propiedad

   $$x^6 + x + 1 \equiv 0 \Longrightarrow x^6 \equiv x + 1$$

que nos deja

$$
\begin{aligned}
& x^8 + x^6 + x^2 + x + 1 = \\
& = x^2 \cdot x^6 + x^6 + x^2 + x + 1 = \\
& = x^2(x+1) + (x+1) + x^2 + x + 1 = \\
& = x^3 + x^2 + x + 1 + x^2 + x + 1 = \\
& = x^3
\end{aligned}
$$

5.10. Ejercicios propuestos

1. Calcule el máximo común divisor de 1026 y 304 aplicando el algoritmo de Euclides, e indique todos los valores intermedios.

2. Calcule el valor de la función ϕ de Euler para los números: 1024, 748, y 5000.

3. Calcule la inversa de 126 módulo 325, empleando el algoritmo Extendido de Euclides, e indique todos los valores intermedios obtenidos.

4. Calcule todos los elementos de los conjuntos $\langle 3 \rangle$ y $\langle 9 \rangle$ en $\mathbb{Z}_{14}^*$. ¿Cuántos elementos tiene cada uno? ¿Qué relación tienen sus tamaños con $\phi(14)$?

5. Sin hacer el cálculo, indique si se cumple que $2^{27} \equiv 1 \pmod{71}$. Justifique su respuesta.

6. Resuelva el siguiente sistema de ecuaciones en congruencias:

$$
\begin{aligned}
x &\equiv 1 \pmod{13} \\
x &\equiv 2 \pmod{24} \\
x &\equiv 20 \pmod{125}
\end{aligned}
$$

7. Calcule el valor de $3^{50000} \pmod{211}$.

8. Factorice, empleando el método de Fermat, el número 161.

9. Realice una traza del algoritmo de Rabin Miller, con $p = 13$, y cogiendo los valores $a = 3$, $a = 5$ y $a = 7$.

10. Calcule la suma y el producto de los polinomios correspondientes a los números binarios 100011 y 10011, dentro del $GF(2^6)$ definido por el polinomio irreducible $f(x) = x^6 + x + 1$.

Capítulo 6

Curvas elípticas en Criptografía

La Criptografía de Curva Elíptica es una de las disciplinas con mayor importancia en el campo de los cifrados asimétricos. Las curvas elípticas constituyen un formalismo matemático conocido y estudiado desde hace más de 150 años, y presentan una serie de propiedades que da lugar a problemas *difíciles* (ver sección 5.4) análogos a los de la aritmética modular, lo cual permite adaptar a ellas algunos de los algoritmos asimétricos más conocidos (ver capítulo 12). Las primeras propuestas de uso de las curvas elípticas en Criptografía fueron hechas por Neal Koblitz y Victor Miller en 1985. Si bien su estructura algebraica es algo compleja, su implementación suele resultar tanto o más eficiente que la de la aritmética modular, con la ventaja adicional de que se pueden alcanzar niveles de seguridad análogos son claves mucho más cortas.

Para introducir el concepto de Curva Elíptica, vamos a establecer un paralelismo con otro formalismo mucho más cercano e intuitivo: los números enteros. Como ya vimos en el capítulo 5, los números enteros constituyen un conjunto sobre el que podemos definir una serie de operaciones, con unas propiedades concretas. Estos conjuntos y operaciones presentan una estructura que hace surgir problemas computacionalmente difíciles de tratar. Vamos a hacer exactamente lo mismo

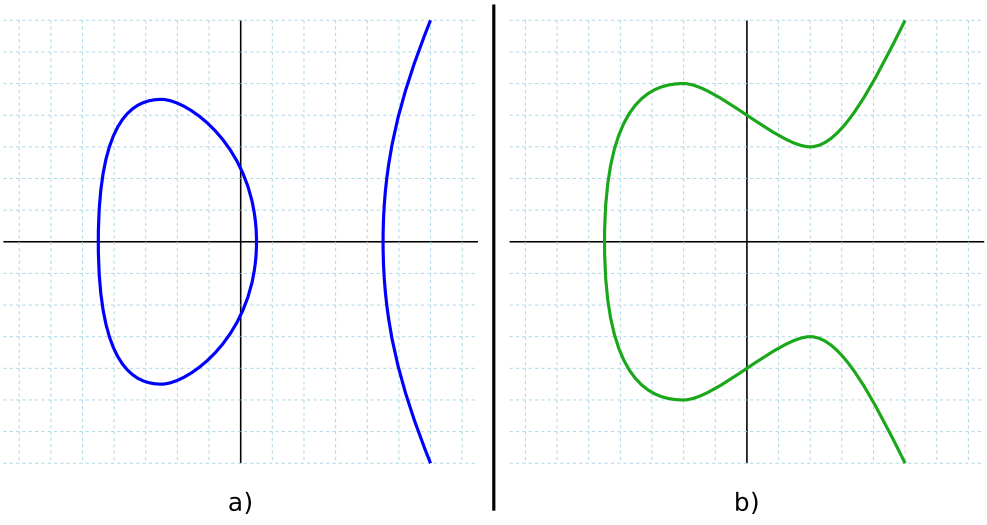

Figura 6.1: Gráficas de curvas elípticas. a) $y^2 = x^3 - 5x + 1$; b) $y^2 = x^3 - 3x + 4$.

con las curvas elípticas.

6.1. Curvas elípticas en $\mathbb{R}$

Definición: Una curva elíptica sobre $\mathbb{R}$ es el conjunto de puntos del plano (x, y) que cumplen la siguiente ecuación:

$$y^2 = x^3 + ax + b \tag{6.1}$$

Los coeficientes a y b caracterizan unívocamente cada curva. En la figura 6.1 puede verse la representación gráfica de dos de ellas —nótese que la curva se extenderá hacia la derecha hasta el infinito—.

Si $x^3 + ax + b$ no tiene raíces múltiples, lo cual es equivalente a que $4a^3 + 27b^2 \neq 0$, entonces la curva correspondiente, en conjunción con un

punto especial $\mathcal{O}$, llamado *punto en el infinito*, más la operación suma que definiremos más adelante, es lo que vamos a denominar *grupo de curva elíptica* $E(\mathbb{R})$. Hay que recalcar que $\mathcal{O}$ es un punto imaginario situado por encima del eje de abscisas a una distancia infinita, y que por lo tanto no tiene un valor concreto.

6.1.1. Suma en $E(\mathbb{R})$

Ya tenemos un conjunto sobre el que trabajar. Nuestro siguiente paso será definir una ley de composición interna que, dados dos elementos cualesquiera, nos devuelva otro que también pertenezca al conjunto. Denominaremos *suma* a esta operación y la representaremos mediante el signo '+', de forma totalmente análoga a lo que hacíamos con $\mathbb{Z}$.

Sean los puntos $\mathbf{r} = (r_x, r_y),\ \mathbf{s} = (s_x, s_y),\ \mathbf{p} = (p_x, p_y),\ \mathbf{t} = (t_x, t_y) \in E(\mathbb{R})$, la operación suma se define de la siguiente forma:

- $\mathbf{r} + \mathcal{O} = \mathcal{O} + \mathbf{r} = \mathbf{r}$, sea cual sea el valor de $\mathbf{r}$. Esto quiere decir que $\mathcal{O}$ desempeña el papel de *elemento neutro* para la suma.
- Si $r_x = s_x$ y $r_y = -s_y$, decimos que $\mathbf{r}$ es el opuesto de $\mathbf{s}$, escribimos $\mathbf{r} = -\mathbf{s}$, y además $\mathbf{r} + \mathbf{s} = \mathbf{s} + \mathbf{r} = \mathcal{O}$ por definición.
- Si $\mathbf{r} \neq \mathbf{s}$ y $\mathbf{r} \neq -\mathbf{s}$, entonces para sumarlos se traza la recta que une $\mathbf{r}$ con $\mathbf{s}$. Dicha recta cortará la curva en un punto. La suma $\mathbf{t}$ de $\mathbf{r}$ y $\mathbf{s}$ será el opuesto de dicho punto (ver figura 6.2a).
- Para sumar un punto $\mathbf{p}$ consigo mismo, se emplea la tangente a la curva en $\mathbf{p}$. Si $p_y \neq 0$, dicha tangente cortará a la curva en un único punto. La suma $\mathbf{t} = \mathbf{p} + \mathbf{p}$ será el opuesto de dicho punto (ver figura 6.2b).
- Para sumar un punto $\mathbf{p}$ consigo mismo, cuando $p_y = 0$, la tangente a la curva será perpendicular al eje de abscisas, por lo que

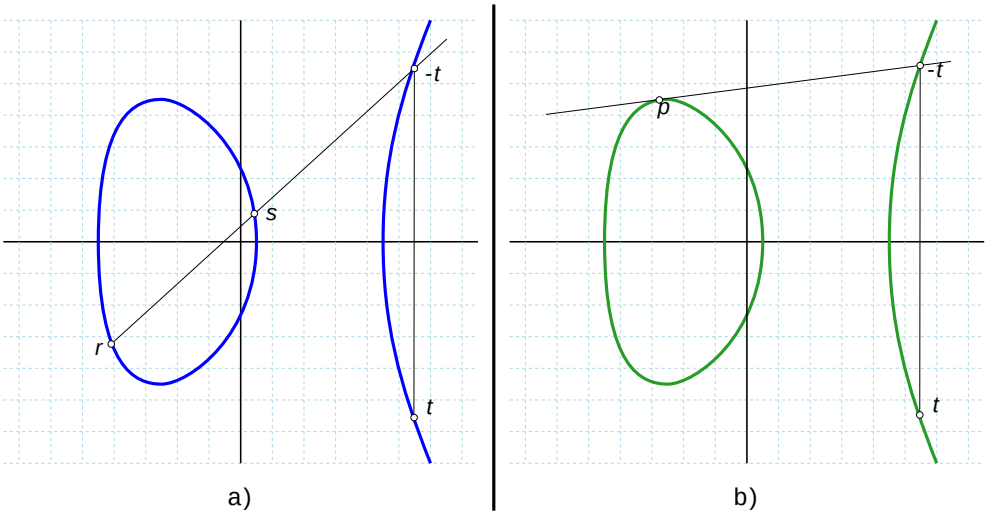

Figura 6.2: Interpretación gráfica de la suma de dos puntos en una curva elíptica.

podemos considerar que corta a la curva en el infinito. Por lo tanto, $\mathbf{p} + \mathbf{p} = \mathcal{O}$ si $p_y = 0$.

Por razones de simplicidad en la notación diremos que sumar un punto $\mathbf{p}$ consigo mismo k veces, es como *multiplicar* dicho punto por el escalar k, y lo notaremos $k\mathbf{p}$

Nótese que, cuando se suma $\mathbf{r}$ y $-\mathbf{r}$, la recta que los une resulta perpendicular al eje de abcisas, por lo que cortará a la curva en el infinito, dando como resultado $\mathcal{O}$. Compruébese, además, que cuando $r_y = 0$, se cumple:

$$2\mathbf{r} = \mathbf{r} + \mathbf{r} = \mathcal{O}$$
$$3\mathbf{r} = 2\mathbf{r} + \mathbf{r} = \mathcal{O} + \mathbf{r} = \mathbf{r}$$
$$4\mathbf{r} = 3\mathbf{r} + \mathbf{r} = \mathbf{r} + \mathbf{r} = \mathcal{O}$$
$$\dots$$

Algebraicamente, la suma de puntos en curvas elípticas se puede definir de la siguiente forma: sea $\mathbf{r} = (r_x, r_y)$ y $\mathbf{s} = (s_x, s_y)$, donde $\mathbf{r} \neq -\mathbf{s}$, entonces $\mathbf{r} + \mathbf{s} = (t_x, t_y)$ según las expresiones

$$d = \frac{r_y - s_y}{r_x - s_x}; \qquad t_x = d^2 - r_x - s_x; \qquad t_y = -r_y + d(r_x - t_x) \tag{6.2}$$

En el caso de que queramos sumar un punto consigo mismo, tenemos que $2\mathbf{p} = (t_x, t_y)$ donde

$$d = \frac{3p_x^2 + a}{2p_y}; \qquad t_x = d^2 - 2p_x; \qquad t_y = -p_y + d(p_x - t_x) \tag{6.3}$$

Si nos fijamos un poco, podremos observar que d representa a la pendiente de la recta que une $\mathbf{r}$ y $\mathbf{s}$, o bien a la tangente en el punto $\mathbf{p}$.

De manera trivial podemos observar que la operación suma definida cumple la propiedad conmutativa. También cumple la propiedad asociativa, si bien la demostración de esto último es algo más complicada. Eso, unido a la existencia de elementos neutro y simétrico, confiere a la curva elíptica junto con la operación suma, la estructura de grupo conmutativo.

Obsérvese que cuando introdujimos los grupos finitos en $\mathbb{Z}$, seleccionábamos un subconjunto de elementos de $\mathbb{Z}$ y definíamos la operación suma, junto con sus propiedades, para este subconjunto. Con las curvas elípticas hemos hecho exactamente lo mismo, sólo que el subconjunto es extraído del plano $\mathbb{R}^2$ y la operación suma es ligeramente más complicada.

6.1.2. Producto por enteros en $E(\mathbb{R})$

Ya hemos dicho que a sumar k veces un punto de una curva lo hemos llamado *multiplicación* de ese punto por el entero k. Como es

obvio, no se trata ni siquiera de una ley de composición interna, ya que esta operación combina un punto $\mathbf{p}$ de la curva con un número entero, por lo que no podemos establecer ninguna analogía en este caso entre esta operación y la multiplicación de números enteros. Como veremos más adelante, la analogía más apropiada para esta operación es la exponenciación de enteros.

En cualquier caso, dado que la suma de puntos en curvas elípticas cumple las propiedades conmutativa y asociativa, es posible reducir la suma de un punto $\mathbf{p}$ consigo mismo k veces a un número de operaciones de suma elemental proporcional al $\log_2(k)$, igual que hicimos con el algoritmo rápido de exponenciación (sección 5.4.3), sustituyendo los productos por sumas de puntos.

El algoritmo para calcular $k\mathbf{p}$ quedaría por lo tanto de la siguiente manera:

1. $\mathbf{x} \longleftarrow \mathbf{p}$
2. $z \longleftarrow k$
3. $r \longleftarrow \mathcal{O}$
4. Mientras $z > 0$ repetir:
   - 4.1. Si $z \bmod 2 \neq 0$, $\mathbf{r} \longleftarrow \mathbf{r} + \mathbf{x}$
   - 4.2. $\mathbf{x} \longleftarrow \mathbf{x} + \mathbf{x}$
   - 4.3. $z \longleftarrow z \text{ div } 2$
5. Devolver $\mathbf{r}$

6.2. Curvas elípticas en $GF(n)$

Recordemos que un cuerpo de Galois $GF(n)$ es el grupo finito generado por n, siendo n un número primo. En dicho conjunto todos los

elementos menos el cero tienen inversa, por lo que podemos sumar, restar, multiplicar y dividir exactamente de la misma forma que en $\mathbb{R}$. Nada nos impide entonces calcular qué puntos cumplen la ecuación

$$y^2 \equiv x^3 + ax + b \pmod{n}$$

definiendo de esta forma el conjunto $E(GF(n))$.

A modo de ejemplo, vamos a estudiar la curva elíptica con $a = 7$ y $b = 4$ en $GF(17)$. En primer lugar, habremos de comprobar, igual que antes, que $4a^3 + 27b^2 \neq 0$:

$$4 \cdot 7^3 + 27 \cdot 4^2 = 2 \pmod{17}$$

Seguidamente, vamos a ver qué puntos pertenecen a la curva elíptica. Si hacemos los cálculos pertinentes, tenemos los siguientes:

(0,2) (0,15) (2,3) (2,14) (3,1) (3,16)
(11,1) (11,16) (15,4) (15,13) (16,8) (16,9)

Nótese que dado un punto de la curva (x, y), el valor $(x, -y)$ (mod n) también pertenece a ésta. Calculemos ahora la suma de dos puntos cualesquiera, por ejemplo $(2, 3)$ y $(3, 16)$, empleando la expresiones de (6.2):

$$d = (3 - 16)/(2 - 3) = 13 \pmod{17}$$
$$x = 13^2 - 2 - 3 = 11 \pmod{17}$$
$$y = -3 + 13 \cdot (2 - 11) = 16 \pmod{17}$$

luego $(2, 3) + (3, 16) = (11, 1)$. Como cabría esperar, nos da como resultado un punto que también pertenece a la curva.

6.3. Curvas elípticas en $GF(2^n)$

Vamos a dar un paso más. Como ya se vio en la sección 5.8.1, los elementos de $GF(p^n)$ —y las operaciones entre ellos— presentan unas

propiedades análogas a las de los elementos de $GF(n)$, con la característica añadida de que, cuando $p = 2$, la implementación de los algoritmos correspondientes es más sencilla y rápida. Definiremos entonces, de forma análoga a $E(GF(n))$, el conjunto $E(GF(2^n))$.

En $GF(2^n)$, debido a su especial estructura, la ecuación de curva elíptica que será útil para nuestros propósitos es ligeramente diferente a la dada en (6.1). Dado un polinomio irreducible $p(x)$, de grado n, las curvas elípticas asociadas vienen dadas por los puntos que cumplen la ecuación:

$$y^2 + xy \equiv x^3 + ax^2 + b \pmod{p(x)} \tag{6.4}$$

y la única condición necesaria para que genere un grupo es que $b \neq 0$.

Dentro de $GF(2^n)$, los puntos de nuestra curva van a ser pares de polinomios de grado $n-1$ con coeficientes binarios, por lo que podrán ser representados mediante cadenas de bits.

6.3.1. Suma en $E(GF(2^n))$

Sean los puntos $\mathbf{r} = (r_x, r_y)$, $\mathbf{s} = (s_x, s_y)$, $\mathbf{p} = (p_x, p_y)$, $\mathbf{t} = (t_x, t_y) \in E(GF(2^n))$, la operación suma se define de la siguiente forma:

- $\mathbf{r} + \mathcal{O} = \mathcal{O} + \mathbf{r} = \mathbf{r}$, sea cual sea el valor de $\mathbf{r}$.
- Si $r_x = s_x$ y $r_y = s_x + s_y$, decimos que $\mathbf{r}$ es el opuesto de $\mathbf{s}$, escribimos $\mathbf{r} = -\mathbf{s}$, y además $\mathbf{r} + \mathbf{s} = \mathbf{s} + \mathbf{r} = \mathcal{O}$ por definición.
- Si $\mathbf{r} \neq \mathbf{s}$ y $\mathbf{r} \neq -\mathbf{s}$, la suma $\mathbf{t} = \mathbf{r} + \mathbf{s}$ se calcula de la siguiente forma:

  $$d = \frac{s_y - r_y}{s_x - r_x}; \qquad t_x = d^2 + d + r_x + s_x + a; \qquad t_y = d(r_x + t_x) + t_x + r_y$$

- Para calcular la suma $\mathbf{t} = 2\mathbf{p}$, con $p_x \neq 0$, se emplea la siguiente fórmula:

  $$d = p_x + \frac{p_y}{p_x}; \qquad t_x = d^2 + d + a; \qquad t_y = p_x^2 + (d+1)t_x$$

- Finalmente, si $p_x = 0$, $2\mathbf{p} = \mathcal{O}$.

6.4. El problema de los logaritmos discretos en curvas elípticas

Tomando un punto $\mathbf{p}$ cualquiera de una curva elíptica, denominaremos $\langle \mathbf{p} \rangle$ al conjunto $\{\mathcal{O}, \mathbf{p}, 2\mathbf{p}, 3\mathbf{p}, \ldots\}$. En $E(GF(n))$ y $E(GF(2^m))$ los conjuntos de esta naturaleza deberán necesariamente ser finitos, ya que el número de puntos de la curva es finito. Por lo tanto, si disponemos de un punto $\mathbf{q} \in \langle \mathbf{p} \rangle$, debe existir un número entero k tal que $k\mathbf{p} = \mathbf{q}$.

El *Problema de los Logaritmos Discretos en Curvas Elípticas* consiste precisamente en hallar el número k a partir de $\mathbf{p}$ y $\mathbf{q}$. Hasta ahora no se ha encontrado ningún algoritmo eficiente (subexponencial) para calcular el valor de k. Al igual que los descritos en la sección 5.4, este problema puede ser empleado con éxito para el desarrollo de algoritmos criptográficos de llave pública.

6.5. Curvas elípticas usadas en Criptografía

Existen diversas recomendaciones de curvas elípticas concretas, con aplicaciones en Criptografía. Cada una de ellas consta de una curva específica dentro de uno de los conjuntos que hemos descrito en este capítulo, con sus correspondientes parámetros, más un punto base $\mathbf{p}$, que genera el conjunto $\langle \mathbf{p} \rangle$ que se usa en los cálculos. En algunos casos, las curvas propuestas presentan propiedades que las hacen más eficientes, o más resistentes a algunos tipos de ataques. Comentaremos en esta sección algunas de ellas.

6.5.1. Secp256k1

Definida en $\mathbb{Z}_n$, y propuesta originalmente por Certicom Research, esta curva es la que se emplea en la red Bitcoin. Sus parámetros son

$$n = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1,$$

y tiene la forma

$$y^2 = x^3 + 7.$$

Como punto base $\mathbf{p}$ se usan las coordenadas

$$p_x = 55066263022277343669578718895168534326250603453777594175500187360389116729240$$

$$p_y = 32670510020758816978083085130507043184471273380659243275938904335757337482424$$

Este valor de ı genera un conjunto $\langle \mathbf{p} \rangle$ con una cardinalidad conocida m. Los factores multiplicadores k que se pueden emplear en esta curva van de 1 a $m - 1$.

6.5.2. Curve25519

Propuesta por Daniel J. Bernstein en 2006, esta curva es en la actualidad una de las que da lugar a implementaciones más eficientes en términos computacionales. La ecuación que la define es la siguiente:

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

todo ello módulo $n = 2^{255} - 19$.

Esta curva elíptica pertenece a un tipo denominado *Curvas de Montgomery*. Entre otras cosas, permite definir el punto base únicamente a partir de su coordenada x, que en este caso es 9. También permite realizar las multiplicaciones de puntos mediante un método denominado *escalera de Montgomery* que, además de ser eficiente, se ejecuta en tiempo constante, lo cual dificulta los ataques basados en tiempo de computación o consumo energético.

6.6. Ejercicios resueltos

1. Se denomina raíz de un polinomio $p(x)$ a los valores de x tales que $p(x) = 0$. Las raíces r_i de un polinomio tienen la propiedad de que el polinomio $p_i(x) = x - r_i$ divide a $p(x)$. Una raíz es múltiple, y su multiplicidad es m, si el polinomio $(p_i(x))^m$ divide a $p(x)$. Por lo tanto, si el polinomio $e(x) = x^3 + ax + b$ tiene raíces múltiples, debe poder escribirse de la forma

$$x^3 + ax + b = (x - q)^2(x - r)$$

Demuestre, a partir de la expresión anterior, que $4a^3 + 27b^2 = 0$ es condición suficiente para que $e(x)$ tenga raíces múltiples.

Solución: Partiendo de la expresión

$$x^3 + ax + b = (x - q)^2(x - r)$$

desarrollaremos el segundo término:

$$\begin{aligned}(x - q)^2(x - r) = (x^2 - 2qx + q^2)(x - r) = \\ = x^3 - 2qx^2 - rx^2 + q^2x + 2qrx - q^2r\end{aligned}$$

Igualando los coeficientes del mismo grado tenemos las siguientes relaciones:

$$\begin{aligned}0 &= -2q - r \\ a &= q^2 + 2qr \\ b &= q^2r\end{aligned}$$

Despejando r en la primera igualdad y sustituyendo su valor en las dos restantes se obtiene

$$\begin{aligned}a &= -3q^2 \\ b &= -2q^3\end{aligned}$$

Elevando al cuadrado la primera expresi'on y al cubo la segunda, podemos despejar q^6 en ambas e igualar:

$$q^6 = \frac{a^3}{-27} = \frac{b^2}{4}$$

Para que el sistema de ecuaciones tenga solución, la igualdad anterior debe cumplirse; si la desarrollamos, finalmente nos queda

$$\frac{a^3}{-27} = \frac{b^2}{4} \quad \Longrightarrow \quad 4a^3 = -27b^2 \quad \Longrightarrow \quad 4a^3 + 27b^2 = 0$$

2. En el ejemplo de la sección 6.2, calcule el conjunto $\langle \mathbf{p} \rangle$ con $\mathbf{p} = (11, 16)$.

   Solución: Emplearemos la expresión (6.3) para calcular $2\mathbf{p}$ y (6.2) para el resto:

   - $2\mathbf{p} = \mathbf{p} + \mathbf{p}$:

     $$\begin{array}{rcccr} d & = & (3 \cdot 11^2 + 7)/(2 \cdot 16) & = & 2 \\ t_x & = & 4 - 22 & = & 16 \\ t_y & = & -16 + 2 \cdot (11 - 16) & = & 8 \end{array}$$

     Por lo tanto,

     $$2\mathbf{p} = (16, 8)$$

   - $3\mathbf{p} = 2\mathbf{p} + \mathbf{p}$:

     $$\begin{array}{rcccr} d & = & (16 - 8)/(11 - 16) & = & 12 \\ t_x & = & 8 - 11 - 16 & = & 15 \\ t_y & = & -16 + 12 \cdot (11 - 15) & = & 4 \end{array}$$

     Por lo tanto,

     $$3\mathbf{p} = (15, 4)$$

   - ...

Aplicando los cálculos de forma sucesiva, tenemos que

$$\begin{aligned}\langle \mathbf{p} \rangle &= \{\mathcal{O}, (11,16), (16,8), (15,4), (0,2), (2,14), (3,16), \\ &\quad (3,1), (2,3), (0,15), (15,13), (16,9), (11,1)\}\end{aligned}$$

Como se puede observar, en este caso $\langle \mathbf{p} \rangle$ contiene todos los puntos de la curva elíptica en cuestión.

6.7. Ejercicios propuestos

1. Sea el grupo $GF(2^3)$ generado por el polinomio x^3+x+1. Calcule los puntos que pertenecen a la curva elíptica con parámetros $a = 100$ y $b = 010$.

2. Compruebe geométricamente las propiedades asociativa y conmutativa de la suma en $E(\mathbb{R})$.

Capítulo 7

Aritmética entera de múltiple precisión

En este capítulo daremos una serie de nociones básicas y algoritmos sobre aritmética entera de múltiple precisión, disciplina que ha cobrado un gran interés debido al uso extensivo que hacen de ella sobre todo los algoritmos asimétricos de cifrado y autentificación.

7.1. Representación de enteros largos

Llamaremos número *largo* a aquel que posee gran cantidad de dígitos significativos, normalmente más de los que los tipos de dato convencionales de los lenguajes de programación clásicos pueden soportar. En este apartado vamos a indicar cómo representarlos y operar con ellos empleando tipos de dato de menor precisión.

Todos conocemos la representación tradicional en base 10 de los números reales, en la que cada cifra contiene únicamente valores de 0 a 9. Esta representación no es más que un caso particular ($B = 10$) de la siguiente expresión general:

$$n = (-)\sum_{-\infty}^{\infty} a_i B^i$$

donde los términos con índice negativo corresponden a la parte no entera (*decimal*) del número real n. Sabemos que, dado el valor de B, dicha representación es única, y que significa que n en base B se escribe:

$$(-)a_n a_{n-1} \ldots a_0 . a_{-1} \ldots$$

Donde cada a_i está comprendido entre 0 y $B-1$. Por ejemplo, en base 10, el número $3{,}1415926$ correspondería a la expresión:

$$3 \cdot 10^0 + 1 \cdot 10^{-1} + 4 \cdot 10^{-2} + 1 \cdot 10^{-3} + 5 \cdot 10^{-4} + 9 \cdot 10^{-5} + 2 \cdot 10^{-6} + 6 \cdot 10^{-7}$$

En cualquier caso, puesto que nuestro objetivo es representar únicamente números enteros positivos, prescindiremos del signo y de los términos con subíndice negativo.

Cualquier número vendrá representado por una serie única de coeficientes a_i (cifras), de las que importa tanto su valor como su posición dentro del número. Esta estructura corresponde claramente a la de un vector (*array*). Para representar de forma eficiente enteros largos emplearemos una base que sea potencia de dos (normalmente se escoge $B = 2^{16}$ ó $B = 2^{32}$ para que cada *cifra* de nuestro número se pueda almacenar en un dato del tipo `unsigned int` sin desperdiciar ningún bit). Para almacenar los resultados parciales de las operaciones aritméticas emplearemos un tipo de dato de doble precisión (`unsigned long int`, correspondiente a $B = 2^{32}$ ó $B = 2^{64}$) de forma que no se nos desborde al multiplicar dos cifras. Normalmente se escoge una longitud que pueda manejar directamente la ALU (Unidad Aritmético-Lógica) de la computadora, para que las operaciones elementales entre cifras sean rápidas.

Por todo esto, para nosotros un número entero largo será un vector de `unsigned int`. En cualquier caso, y a partir de ahora, nuestro

objetivo será estudiar algoritmos eficientes para efectuar operaciones aritméticas sobre este tipo de números, independientemente de la base en la que se encuentren representados.

7.2. Operaciones aritméticas sobre enteros largos

Vamos a describir en este apartado cómo realizar operaciones aritméticas (suma, resta, multiplicación y división) de enteros largos.

7.2.1. Suma

La suma de $a = (a_0, a_1 \dots a_{n-1})$ y $b = (b_0, b_1 \dots b_{n-1})$ se puede definir como:

$$(a+b)_i = \begin{cases} (a_i + b_i + c_i) \text{ mód } B & \text{para } i = 0 \dots n-1 \\ c_i & \text{para } i = n \end{cases}$$

siendo

$$c_i = \begin{cases} 0 & \text{para } i = 0 \\ (a_{i-1} + b_{i-1} + c_{i-1}) \text{ div } B & \text{para } i = 1 \dots n \end{cases}$$

c_i es el *acarreo* de la suma de los dígitos inmediatamente anteriores. Tenemos en cuenta el coeficiente n de la suma porque puede haber desbordamiento, en cuyo caso la suma tendría $n+1$ dígitos y su cifra más significativa sería precisamente c_n. Este no es otro que el algoritmo clásico que todos hemos empleado en la escuela cuando hemos aprendido a sumar.

El algoritmo para la suma quedaría, pues, como sigue:

```
suma (unsigned *a, unsigned *b, unsigned *s)
    { unsigned long sum;
      unsigned acarreo;

      n=max(num. de digitos de a, num. de digitos de b)
      acarreo=0;
      for (i=0;i<n;i++)
        {  sum=acarreo+a[i]+b[i];
           s[i]=sum%BASE;
           acarreo=sum/BASE;
        }
      s[n]=acarreo;
    }
```

El resultado se devuelve en `s`.

7.2.2. Resta

La resta es muy parecida a la suma, salvo que en este caso los acarreos se restan. Suponiendo que $a > b$:

$$(a-b)_i = (a_i - b_i - r_i) \text{ mód } B \qquad \text{para } i = 0 \ldots n-1$$

siendo

$$r_i = \begin{cases} 0 & \text{para } i = 0 \\ 1 - \Big((a_{i-1} - b_{i-1} - r_{i-1} + B) \text{ div } B \Big) & \text{para } i = 1 \ldots n \end{cases}$$

r_i representa el acarreo de la resta (*borrow*), que puede valer 0 ó 1 según la resta parcial salga positiva o negativa. Nótese que, como $a > b$, el último acarreo siempre ha de valer 0.

```
resta (unsigned *a, unsigned *b, unsigned *d)
    { unsigned long dif;
      unsigned acarreo;

      n=max(num. de digitos de a, num. de digitos de b)
      acarreo=0;
      for (i=0;i<n;i++)
        {  dif=a[i]-b[i]-acarreo+BASE;
           d[i]=dif%BASE;
           acarreo=1-dif/BASE;
        }
    }
```

El resultado se devuelve en `d`. La razón por la que sumamos la base a `dif` es para que la resta parcial salga siempre positiva y poder hacer el módulo correctamente. En ese caso, si el valor era positivo, al sumarle B y dividir por B de nuevo nos queda 1. Si fuera negativo, nos saldría 0. Por eso asignamos al nuevo acarreo el valor `1-dif/BASE`.

Nos queda comprobar cuál de los dos números es mayor para poder emplearlo como minuendo. Esta comprobación se puede realizar fácilmente definiendo una función que devuelva el número cuyo dígito más significativo tenga un número de orden mayor. En caso de igualdad iríamos comparando dígito a dígito, empezando por los más significativos hasta que encontremos alguno mayor o lleguemos al último dígito, situación que únicamente ocurrirá si los dos números son iguales.

7.2.3. Producto

Para obtener el algoritmo del producto emplearemos la expresión general de un número entero positivo en base B. Si desarrollamos el producto de dos números cualesquiera a y b de longitudes m y n respectivamente nos queda:

$$ab = \sum_{i=0}^{m-1} a_i B^i b$$

A la vista de esto podemos descomponer el producto como m llamadas a una función que multiplica un entero largo por $a_i B^i$ (es decir, un entero largo con un único dígito significativo) y después sumar todos los resultados parciales.

Para poder implementar esto primero definiremos una función (`summult`) que multiplique b por $a_i B^i$ y el resultado se lo sume al vector `s`, que no tiene necesariamente que estar a cero:

```
summult(unsigned ai, int i, unsigned *b,
                            int m, unsigned *s)
      { int k;
        unsigned acarreo;
        unsigned long prod,sum;

        acarreo=0;
        for (k=0; k<m; k++)
          { prod=ai*b[k]+s[i+k]+acarreo;
            s[i+k]=prod%BASE;
            acarreo=prod/BASE;
          }
        k=m+i;
        while (acarreo!=0)
          { sum=s[k]+acarreo;
            s[k]=sum%BASE;
            acarreo=sum/BASE;
            k++;
          }
      }
```

La segunda parte de la función se encarga de acumular los posibles acarreos en el vector `s`. A partir de la función que acabamos de definir,

queda entonces como sigue:

```
producto(unsigned *a,int m, unsigned *b,
                            int n, unsigned *p)
   { int k;

     for (k=0;k<=m+n;k++)
       p[k]=0;
     for (k=0;k<m;k++)
       summult(a[k],k,b,n,p);
   }
```

El resultado se devuelve en `p`.

Existe otra propiedad de la multiplicación de enteros que nos va a permitir efectuar estas operaciones de manera más eficiente. Tomemos un número entero cualquiera a de k dígitos en base B. Dicho número puede ser representado mediante la de la siguiente expresión:

$$a = a_l B^{\frac{k}{2}} + a_r$$

Es decir, *partimos* a en dos mitades. Por razones de comodidad, llamaremos B_k a $B^{\frac{k}{2}}$. Veamos ahora cómo queda el producto de dos números cualesquiera a y b, en función de sus respectivas *mitades*:

$$ab = a_l b_l B_k^2 + (a_l b_r + a_r b_l) B_k + a_r b_r$$

Hasta ahora no hemos aportado nada nuevo. El *truco* para que este desarrollo nos proporcione un aumento de eficiencia consiste en hacer uso de la siguiente propiedad:

$$a_l b_r + a_r b_l = a_l b_r + a_r b_l + a_l b_l - a_l b_l + a_r b_r - a_r b_r = (a_l + a_r)(b_l + b_r) - a_l b_l - a_r b_r$$

quedando finalmente, lo siguiente:

$$x = a_l b_l \qquad y = (a_l + a_r)(b_l + b_r) \qquad z = a_r b_r$$
$$ab = xB_k^2 + (y - x - z)B_k + z$$

Hemos reducido los cuatro productos y tres sumas del principio a tres productos y seis sumas. Como es lógico, esta técnica debe emplearse dentro de una estrategia *divide y vencerás*.

7.2.4. División

El algoritmo más simple para dividir dos números se consigue a partir de su representación binaria:

```
cociente_bin(unsigned *c, unsigned *d, unsigned *a, unsigned *b)
   { /*
     Calcular a= c div d
              b= c mod d

     Bits_Significativos(x) => Devuelve el numero de bits
                               significativos de x, es decir, el
                               numero total de bits menos el numero
                               de ceros a la derecha.
     pow(a,b)  => Calcula el valor de a elevado a b.
     Poner_bit_a_1(a,x)  => Pone a 1 el i-esimo bit de a.
     Poner_bit_a_0(a,x)  => Pone a 0 el i-esimo bit de a.
     */

     m=Bits_Significativos(c);
     n=Bits_Significativos(d);
     b=c;
     a=0;
     d1=d*pow(2,m-n);      /* Desplazamos a la izquierda d */
     for (i=m-n;i>=0;i--)
       { if (b>d1)
            { Poner_bit_a_1(a,i);
              b=b-d1;
            }
            else Poner_bit_a_0(a,i);
         d1=d1/2;
       }
   }
```

El funcionamiento del algoritmo es extremadamente simple: copiamos el dividendo en b y desplazamos a la izquierda el divisor hasta que su longitud coincida con la del dividendo. Si el valor resultante es menor que b, se lo restamos y ponemos a 1 el bit correspondiente de a. Repitiendo esta operación sucesivamente se obtiene el cociente en a y el resto en b. A modo de ejemplo, dividiremos 37 (100101) entre 7 (111):

1. $b = 100101; \quad a = \ \ - - - - -; \quad d1 = 111000; \quad b \not> d1 \longrightarrow a = 0 \ - - -$
2. $b = 100101; \quad a = 0 \ - - -; \quad d1 = 11100; \quad b > d1 \longrightarrow a = 01 - -$
3. $b = 001001; \quad a = 01 - -; \quad d1 = 1110; \quad b \not> d1 \longrightarrow a = 010 -$
4. $b = 001001; \quad a = 010 -; \quad d1 = 111; \quad b > d1 \longrightarrow a = 0101$
5. $b = 010$

Este algoritmo resulta muy lento, ya que opera a nivel de bit, por lo que intentaremos encontrar otro más rápido —aunque con el mismo orden de eficiencia—. Nos basaremos en el algoritmo tradicional de la división. Suponiendo que queremos dividir c por d, obteniendo su cociente (a) y resto (b), iremos calculando cada dígito del cociente en base B de un solo golpe. Nuestro objetivo será estimar a la baja el valor de cada uno de los dígitos a, e incrementarlo hasta alcanzar el valor correcto. Para que la estimación se quede lo más cerca posible del valor correcto efectuaremos una normalización de los números, de forma que el dígito más significativo d tenga su bit de mayor peso a 1. Esto se consigue multiplicando c y d por 2^k, siendo k el número de ceros a la izquierda del bit más significativo del divisor d. Posteriormente habremos de tener en cuenta lo siguiente:

$$c = ad + b \Longleftrightarrow 2^k c = a(2^k d) + 2^k b$$

luego el cociente será el mismo pero el resto habrá que dividirlo por el factor de normalización. Llamaremos $\bar{c}, \bar{d}$ a los valores de c y d normalizados.

Para hacer la estimación a la baja de los a_i, dividiremos $c_j B + c_{j-1}$ por $\bar{d}_m + 1$ (c_j es el dígito más significativo de c en el paso i, y $\bar{d}_m$ es el dígito más significativo de $\bar{d}$). Luego actualizamos $\bar{c}$ con $\bar{c} - \bar{d} a_i B^i$ y vamos incrementando a_i (y actualizando $\bar{c}$) mientras nos quedemos cortos. Finalmente, habremos calculado el valor del cociente (a) y el valor del resto será

$$b = \frac{\bar{b}}{2^k}$$

El algoritmo podría quedar como sigue:

```
cociente(unsigned *c, unsigned *d, unsigned *a, unsigned *b)
{
/* Calcular a= c div d
            b= c mod d


   Digito_Mas_Significativo(a) => Devuelve el valor del
                                  digito de mayor peso de a.
   Bits_Significativos(x) => Devuelve el numero de bits
                             significativos de x, es decir, el
                             numero total de bits menos el numero
                             de ceros a la derecha.
   pow(a,b)  => Calcula el valor de a elevado a b.

*/

   despl=Num_bits_digito -
                Bits_significativos(Digito_Mas_Significativo(d));

   factor=pow(2,despl);  /* Desplazamos d hasta que su digito
                            mas significativo tenga su bit de mayor
                            peso a 1 (di>=B/2)
                         */
   dd=d*factor;
   cc=c*factor;
   if (Digitos(cc)==Digitos(c))
      Poner_Un_Cero_A_La_Izquierda(cc);  /* Garantizar que cc
                                            tiene exactamente un
                                            digito mas que c
                                         */
```

```
   t=Digito_Mas_Significativo(dd);

   /* Ya hemos normalizado. El cociente que obtengamos seguira
      siendo valido, pero el resto habra luego que dividirlo por
      factor */

   Poner_a_cero(a);

   for (i=Digitos(c)-Digitos(dd); i>=0; i--)
     {
        /* Subestimar digito del cociente (ai) */

        if (t==B-1)  /* No podemos dividir por t+1 */

           ai=cc[i+Digitos(dd)];  /* La estimacion es el primer
                                     digito significativo de cc
                                  */

          else ai=(cc[i+Digitos(dd)]*B+cc[i+Digitos(dd)-1])/(t+1);
                                 /* La estimacion es el cociente
                                    entre los dos primeros digitos
                                    de cc y t+1
                                 */

        cc=cc-ai*dd*pow(B,i);                /* Restar a cc */


        while (cc[i+Digitos(dd)] ||  /* Si no se ha hecho cero el
                                        digito mas sign. de cc...
                                     */
               mayor(cc,dd*pow(B,i))) /* o si cc es mayor o igual
                                         que dd*B^i
                                      */

           { ai++;             /* Hemos de aumentar la estimacion */
             cc=cc-dd*pow(B,i);
           }
        a[i]=ai;
     }
  b=cc/factor; /* Lo que nos queda en cc es el resto
                  dividimos por factor para deshacer
                  la normalizacion
               */
}
```

Aunque a primera vista pueda parecer un algoritmo muy complejo, vamos a ver que no es tan complicado siguiendo su funcionamiento para un ejemplo concreto, con $B = 16$, $c = 3FBA2$, y $d = 47$:

1. Normalización: multiplicamos por 2 y nos queda $\bar{c} = 7F744$, $\bar{d} = 8E$

2. $a_2 = 7F$ div $9 = E$; $\bar{c} = \bar{c} - a_2\bar{d}B^2 = 7F744 - 7C400 = 3344$
   Puesto que $\bar{c} < \bar{d}B^2 = 8E00$, no hay que incrementar a_2.

3. $a_1 = 33$ div $9 = 5$; $\bar{c} = \bar{c} - a_1\bar{d}B = 3344 - 2C60 = 6E4$
   Puesto que $\bar{c} < \bar{d}B = 8E0$, no hay que incrementar a_1.

4. $a_0 = 6E$ div $9 = C$; $\bar{c} = \bar{c} - a_0\bar{d} = 6E4 - 6A8 = 3C$
   Puesto que $\bar{c} < \bar{d} = 8E$, tampoco hay que incrementar a_0

5. $a = E5C$; $b = \frac{\bar{c}}{2} = 1E$

7.3. Aritmética modular con enteros largos

Los algoritmos criptográficos de llave pública más extendidos se basan en operaciones modulares sobre enteros muy largos. Empleando los algoritmos del apartado 7.2 son inmediatas las operaciones de suma, resta y multiplicación módulo n. La división habremos de tratarla de manera diferente.

- Para sumar dos números módulo n basta con efectuar su suma entera y, si el resultado es mayor que n, restar el módulo.
- Para restar basta con comprobar que el minuendo es mayor que el sustraendo, en cuyo caso aplicamos directamente el algoritmo de la resta. Si, por el contrario, el sustraendo fuera mayor que el minuendo, sumamos a este último el valor de n antes de hacer la resta.
- El producto se lleva a cabo multiplicando los factores y tomando el resto de dividir el resultado por el módulo.

- La división habremos de implementarla multiplicando el dividendo por la inversa del divisor. Para calcular la inversa de un número módulo n basta con emplear el Algoritmo Extendido de Euclides, sustituyendo las operaciones elementales por llamadas a las operaciones con enteros largos descritas en la sección 7.2.

7.4. Ejercicios resueltos

1. Efectúe el trazado del algoritmo de la división con $B = 8$ para calcular el siguiente cociente: $c = 35240$, $d = 234$.

   Solución: A partir de los valores $c = 35240$ y $d = 234$, calculamos el factor de normalización, que será 2, por lo que $dd = 470$ y $cc = 72500$. Nótese que todas las operaciones están efectuadas en base octal. Los pasos del algoritmo arrojarán los siguientes valores:

$$
\begin{array}{rcll}
t & = & 4 & \\
a_2 & = & 07 \div 5 = 1 & \quad cc = 72500 - 1 \cdot 47000 = 23500 \\
a_1 & = & 25 \div 5 = 3 & \quad cc = 23500 - 3 \cdot 4700 = 5000 \\
a_1 & = & a_1 + 1 = 4 & \quad cc = 5000 - 4700 = 100 \\
a_0 & = & 1 \div 5 = 0 & \quad cc = 100 - 0 \cdot 470 = 100
\end{array}
$$

   Ahora deshacemos la normalización, con lo que nos queda un cociente $a = 140$ y un resto $b = 40$.

7.5. Ejercicios propuestos

1. La técnica *divide y vencerás* se basa en subdividir el problema y aplicar recursivamente el algoritmo en cuestión hasta llegar a un umbral mínimo, a partir del cual la técnica no recursiva es más eficiente. Implemente el algoritmo de la multiplicación mediante esta técnica y calcule el umbral correspondiente.

2. Elabore la especificación de una Estructura de Datos que permita almacenar números enteros *largos* y defina sus primitivas básicas.

3. Proponga una especificación para la estructura del ejercicio anterior y discuta su eficiencia.

Capítulo 8

Criptografía y números aleatorios

Los algoritmos asimétricos, debido a su mayor orden de complejidad, suelen ser empleados en conjunción con cifrados simétricos de la siguiente forma (ver capítulo 12): el mensaje primero se codifica empleando un algoritmo simétrico con una clave aleatoria única, que será diferente cada vez. Es únicamente la clave simétrica la que se cifra empleando criptografía asimétrica, produciendo un importante ahorro de coste computacional, además de otras ventajas. Sin embargo, si el proceso de generación de estas claves fuera predecible, o reproducible de alguna manera, un atacante malicioso podría llegar a adivinar la siguiente clave a partir de una o varias claves empleadas en el pasado, lo cual tendría resultados catastróficos. Un famoso ejemplo de este problema tuvo lugar en una de las primeras versiones del navegador Netscape, que resultaba insegura debido al uso de un generador de claves demasiado previsible. La única manera de protegerse frente a estos ataques es asegurarse de que no exista ningún tipo de dependencia entre una clave y la siguiente, esto es, que sean aleatorias. De aquí surge el interés por los números aleatorios en Criptografía.

Seguro que el lector conoce generadores pseudoaleatorios y dife-

rentes *tests* de aleatoriedad —como el denominado *test* ψ^2, que puede ser consultado en casi cualquier libro de Estadística—. En realidad, los generadores tradicionales no permiten calcular secuencias realmente aleatorias, puesto que conociendo un número obtenido con el generador podemos determinar cualquiera de los posteriores —recordemos que cada elemento de la secuencia se emplea como *semilla* para calcular el siguiente—. Si bien las series que producen superan los test estadísticos de aleatoriedad, son totalmente previsibles, y esa condición es inadmisible para aplicaciones criptográficas.

En este capítulo vamos a caracterizar diferentes tipos de secuencias que pueden funcionar como aleatorias —aunque en la práctica no lo sean—, así como su interés en Criptografía. También veremos cómo implementar un buen generador aleatorio útil desde el punto de vista criptográfico.

8.1. Tipos de secuencias *aleatorias*

En realidad es casi del todo imposible generar secuencias auténticamente aleatorias en una computadora, puesto que estas máquinas son —al menos en teoría— completamente deterministas. De hecho, cualquier generador que emplee únicamente métodos algorítmicos en su propósito producirá secuencias reproducibles, por lo que estaremos hablando en realidad de secuencias *pseudoaleatorias*. En general, todos los generadores pseudoaleatorios producen secuencias finitas y periódicas de números empleando exclusivamente operaciones aritméticas y/o lógicas. No obstante, si empleamos elementos externos a la computadora, podremos generar también secuencias realmente aleatorias.

En esta sección describiremos dos tipos de secuencias pseudoaleatorias, en función de sus propiedades, además de las secuencias auténticamente aleatorias.

8.1.1. Secuencias estadísticamente aleatorias

En principio, es relativamente fácil conseguir que una secuencia pseudoaleatoria sea lo más larga posible antes de comenzar a repetirse y que supere los tests estadísticos de aleatoriedad, que lo que buscan principalmente es que la secuencia presente, al menos en apariencia, entropía máxima (sección 3.2), lo cual significa que todos los posibles valores aparezcan con la misma probabilidad. En este sentido podemos hablar de:

- *Secuencias estadísticamente aleatorias*: Secuencias pseudoaleatorias que superan los tests estadísticos de aleatoriedad.

Los generadores *congruenciales lineales*[1] cumplen esta propiedad, y de hecho son muy utilizados en Informática, especialmente en entornos de simulación, pero en Criptografía resultan del todo inútiles, debido a que cada valor de la secuencia se emplea como semilla para calcular el siguiente, lo cual nos permite conocer *toda* la serie a partir de un único valor. Supongamos que tenemos un sistema que se basa en emplear claves aleatorias para cada sesión y usamos un generador de este tipo. Bastaría con que una de las claves quedara comprometida para que todas las comunicaciones —pasadas y futuras— pudieran ser descifradas sin problemas. Incluso se ha demostrado que conociendo únicamente un bit de cada valor de la secuencia, ésta puede ser recuperada completamente con una cantidad relativamente pequeña de valores.

8.1.2. Secuencias criptográficamente aleatorias

El problema de las secuencias estadísticamente aleatorias, y lo que las hace poco útiles en Criptografía, es que son completamente predecibles. Definiremos, por tanto:

[1] Un generador congruencial lineal opera según la expresión $a_{n+1} = (a_n b + c) \bmod m$, donde a_0 es la semilla pseudoaleatoria y b, c y m son los parámetros del generador.

- *Secuencias criptográficamente aleatorias*: Para que una secuencia pseudoaleatoria sea *criptográficamente aleatoria*, ha de cumplir la propiedad de ser impredecible. Esto quiere decir que debe ser computacionalmente intratable el problema de averiguar el siguiente número de la secuencia, teniendo total conocimiento acerca de todos los valores anteriores y del algoritmo de generación empleado.

Existen generadores pseudoaleatorios capaces de generar secuencias criptográficamente aleatorias, generalmente a través del uso en el algoritmo de información de inicialización —denominada *semilla*— o de estado que ha de mantenerse en secreto. Sin embargo, habrá situaciones en las que esto no sea suficiente para nuestros propósitos y en las que deseemos tener valores realmente impredecibles, de forma que nuestro adversario no pueda averiguarlos ni tratar de simular el proceso de generación que nosotros hemos llevado a cabo.

8.1.3. Secuencias totalmente aleatorias

Como ya se ha dicho antes, no existe la aleatoriedad cuando se habla de computadoras. Sin embargo, podemos hacer que el ordenador, a través de sus dispositivos de entrada/salida, obtenga de su entorno sucesos que pueden considerarse impredecibles. Consideraremos pues un tercer tipo de secuencias:

- *Secuencias aleatorias*: Diremos que una secuencia es totalmente aleatoria (o simplemente aleatoria) si no puede ser reproducida de manera fiable.

8.2. Utilidad de las secuencias aleatorias en Criptografía

Llegados a este punto parece claro que nuestro objetivo en la mayor parte de las ocasiones no va a consistir en generar secuencias aleatorias puras, sino más bien secuencias impredecibles e irreproducibles para un atacante. De hecho, habrá dos escenarios típicos en los que nos vamos a encontrar:

- Queremos generar una secuencia de números impredecible e irreproducible, por ejemplo, para generar claves de un solo uso. Para ello podemos utilizar indistintamente un generador totalmente aleatorio, o un generador pseudoaleatorio criptográficamente aleatorio. En este último caso, emplearemos como semilla la salida producida por un generador totalmente aleatorio.
- Queremos generar una secuencia de números que luego pueda reproducirse, por ejemplo, para construir un cifrado de flujo (ver capítulo 11). En ese caso emplearemos un generador criptográficamente aleatorio, cuya semilla hará las veces de clave, ya que permitirá al emisor generar una secuencia pseudoaleatoria —impredecible para un atacante— y combinarla con el mensaje para obtener el criptograma. El receptor usará la misma semilla para generar una secuencia idéntica y recuperar así el mensaje original.

8.3. Generación de secuencias aleatorias criptográficamente válidas

Dedicaremos esta sección a las técnicas de generación de secuencias totalmente aleatorias, ya que los generadores de secuencia crip-

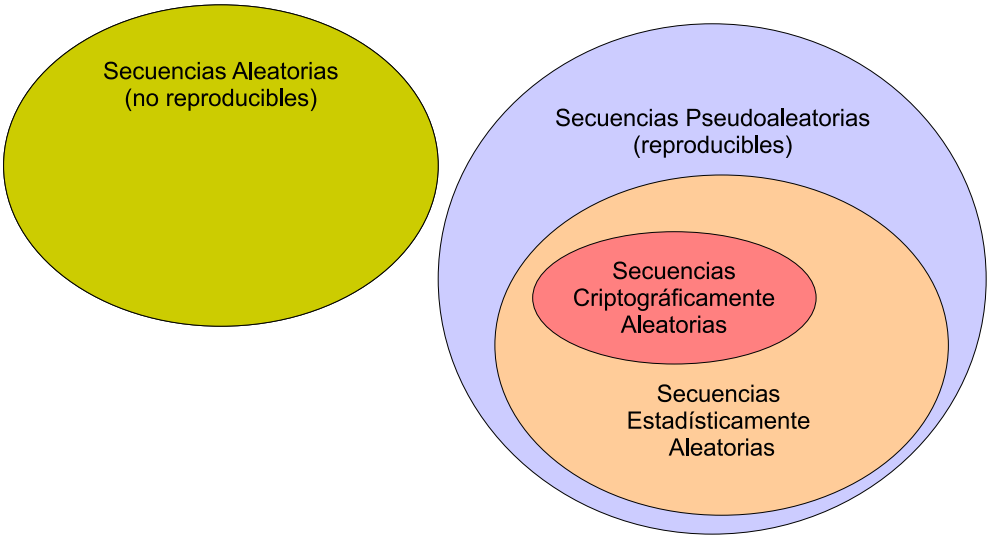

Figura 8.1: Clasificación de los distintos tipos de secuencias *aleatorias*.

tográficamente aleatorios serán estudiados con mayor detalle en el capítulo 11. Nuestro objetivo será, pues, la obtención de secuencias impredecibles e irreproducibles. Podemos obtener directamente dichas secuencias del exterior, y tratarlas para que no presenten ningún sesgo estadístico —o lo que es lo mismo, que tengan máxima entropía—, o emplearlas como semilla para alimentar algoritmos pseudoaleatorios, que también pueden hacer uso a su vez de valores difícilmente reproducibles, como el reloj del sistema.

8.3.1. Obtención de bits aleatorios

Como hemos dicho antes, las operaciones aritméticas y lógicas que realiza una computadora son completamente deterministas. Sin embargo, como veremos a continuación, los ordenadores suelen ir acompañados de dispositivos no tan predecibles que pueden resultar útiles

para nuestros propósitos. En cualquier caso, cada vez resulta más frecuente la inclusión de *hardware* específico en los ordenadores modernos para la obtención de información aleatoria.

Aunque lo ideal es disponer de elementos específicos, existen valores obtenidos del *hardware* convencional de una computadora que suelen proporcionar algunos bits de aleatoriedad. Parece razonable que leer en un momento dado el valor de un reloj interno de alta precisión proporcione un resultado más o menos impredecible, por lo que podríamos emplearlo para recolectar valores aleatorios. Diferentes pruebas han demostrado sin embargo que mecanismos de este tipo, que pueden ser útiles en ciertas arquitecturas y sistemas operativos, dejan de servir en otras versiones del mismo sistema o en arquitecturas muy similares, por lo que hemos de tener mucho cuidado con esto.

Algunas veces se ha propuesto el uso de los números de serie de los componentes físicos de un sistema, pero recordemos que estos números tienen una estructura muy rígida, y a veces conociendo simplemente el fabricante y la fecha aproximada de fabricación podemos *adivinar* casi todos sus dígitos, por lo que van a ser demasiado predecibles. Tampoco son útiles las fuentes *públicas* de información, como por ejemplo los bits de un CD de audio, puesto que nuestros atacantes pueden disponer de ellas, con lo que el único resto de *aleatoriedad* que nos va a quedar es la posición que escojamos dentro del CD para extraer los bits.

Fuentes adecuadas de obtención de bits aleatorios

Cuando no disponemos de un elemento físico en la computadora específicamente diseñado para producir datos aleatorios, podemos recurrir a algunos dispositivos relativamente comunes en los ordenadores actuales:

- *Conversores de señal analógica a digital*. Un dispositivo de este tipo sin ninguna entrada conectada, siempre que tenga ganancia su-

ficiente, capta esencialmente ruido térmico, con una distribución aleatoria, y por lo tanto puede ser apto para nuestros propósitos.

- *Unidades de disco*. Las unidades de disco presentan pequeñas fluctuaciones en su velocidad de giro debido a turbulencias en el aire. Disponiendo de un método para medir el tiempo de acceso de la unidad con suficiente precisión, se pueden obtener bits aleatorios de la calidad necesaria.

- *Pulsaciones de teclado*. Al igual que los accesos a disco, se puede considerar que valores de los bits menos significativos del reloj del sistema en los momentos en los que se detectan pulsaciones de teclas, se distribuyen de forma aleatoria.

Si no se dispone de una fuente fiable de bits aleatorios se puede efectuar la combinación de varias fuentes de información menos fiables. Por ejemplo, podríamos leer el reloj del sistema, algún identificador del *hardware*, la fecha y la hora locales, el estado de los registros de interrupciones del sistema, etc. Esto garantizará que en total se ha recogido una cantidad suficiente de bits realmente aleatorios.

La mezcla de todas esas fuentes puede proporcionarnos suficiente aleatoriedad para nuestros propósitos. Hemos de tener en cuenta que el número de bits realmente aleatorios que se obtendrán como resultado final del proceso ha de ser necesariamente menor que el número de bits recogido inicialmente. De hecho, tiene que ser igual a la entropía (sección 3.2) de la secuencia de bits de entrada. Para llevar a cabo esta combinación podemos emplear las denominadas *funciones de mezcla fuertes*.

Una función de mezcla fuerte es aquella que toma dos o más fuentes de información y produce una salida en la que cada bit es una función compleja y no lineal de todos los bits de la entrada. Por término medio, modificar un bit en la entrada debería alterar aproximadamente la mitad de los bits de salida. Podemos emplear diferentes algoritmos criptográficos para construir este tipo de funciones:

- *Algoritmos de cifrado por Bloques* (ver capítulo 10). Un algoritmo simétrico de cifrado por bloques puede ser útil como función de mezcla de la siguiente forma: supongamos que usa una clave de n bits, y que tanto su entrada como su salida son bloques de m bits. Si disponemos de $n+m$ bits inicialmente, podemos cifrar m bits usando como clave los n restantes, y así obtener como salida un bloque de m bits con mejor aleatoriedad.

- *Funciones Resumen* (ver capítulo 13) . Una función resumen puede ser empleada para obtener un número fijo de bits a partir de una cantidad arbitraria de bits de entrada.

Una vez aplicada la función de mezcla, debemos truncar su salida a una longitud igual o menor a la entropía de la secuencia inicial, ya que ese es precisamente el número máximo de bits realmente aleatorios que podemos obtener de la misma.

Veamos un ejemplo: sea una secuencia de 300 bits con una probabilidad $P(1) = 0{,}99$. La entropía de cada bit será

$$H = -0{,}99 \log_2(0{,}99) - 0{,}01 \log_2(0{,}01) = 0{,}08079 \; bits$$

Luego los 300 bits originales aportarán $300 \times 0{,}08079 \simeq 24 \; bits$ de información real. Podemos aplicar entonces una función de mezcla fuerte a dicha secuencia y considerar los 24 bits menos significativos del resultado como bits aleatorios válidos.

8.3.2. Eliminación del sesgo

En algunos casos no disponemos de forma explícita de la entropía de la secuencia proporcionada por la fuente aleatoria, ya que por ejemplo las frecuencias de sus valores pueden fluctuar a lo largo del tiempo. Existen métodos que a partir de la propia secuencia permiten obtener otra de menor longitud con máxima entropía, es decir, sin sesgo.

Bits de paridad

Si tenemos una secuencia de ceros y unos, con un sesgo arbitrario pero acotado, podemos emplear el bit de paridad[2] de la secuencia para obtener una distribución con una desviación tan pequeña como queramos. Para comprobarlo, supongamos que d es el sesgo, luego las probabilidades que tenemos para los bits de la secuencia son:

$$p = 0{,}5 + d \qquad q = 0{,}5 - d$$

donde p es la probabilidad para el 1 y q es la probabilidad para el 0. Se puede comprobar que las probabilidades para el bit de paridad de los n primeros bits valen

$$r = \frac{1}{2}\left((p+q)^n + (p-q)^n\right) \qquad s = \frac{1}{2}\left((p+q)^n - (p-q)^n\right)$$

donde r será la probabilidad de que el bit de paridad sea 0 ó 1 dependiendo de si n es par o impar. Puesto que $p + q = 1$ y $p - q = 2d$, tenemos

$$r = \frac{1}{2}(1 + (2d)^n) \qquad s = \frac{1}{2}(1 - (2d)^n)$$

Siempre que $n > \dfrac{\log_2(2\epsilon)}{\log_2(2d)}$ el sesgo de la paridad será menor que ϵ, por lo que bastará con coger esos n bits. Por ejemplo, si una secuencia de bits tiene $p = 0{,}01$ y $q = 0{,}99$, basta con coger la paridad de cada 308 bits para obtener un bit con sesgo inferior a $0{,}001$.

[2] El bit de paridad de una secuencia vale 1 si el número de unos de dicha secuencia es par (paridad impar) o impar (paridad par).

Método de Von Neumann

El método que propuso Von Neumann para eliminar el sesgo de una cadena de bits consiste simplemente en examinar la secuencia de dos en dos bits. Eliminamos los pares 00 y 11, e interpretamos 01 como 0 y 10 como 1. Por ejemplo, la serie 00.10.10.01.01.10.10.10.11 daría lugar a 1.1.0.0.1.1.1.

Es fácil comprobar que, siendo d el sesgo de la distribución inicial

$$P(01) = P(10) = (0{,}5 + d)(0{,}5 - d)$$

por lo que la cadena de bits resultantes presenta exactamente la misma probabilidad tanto para el 0 como para el 1. El problema de este método es que no sabemos a priori cuántos bits de información sesgada necesitamos para obtener cada bit de información no sesgada.

8.3.3. Generadores aleatorios criptográficamente seguros

Vamos a ver a continuación un par de generadores pseudoaleatorios que permiten obtener secuencias lo suficientemente seguras como para ser empleadas en aplicaciones criptográficas. Ambos emplean una semilla —que puede ser obtenida a partir de un generador totalmente aleatorio—, e incluso uno de ellos emplea internamente información de gran variabilidad, como es el reloj del sistema, para hacer más difícil de reproducir la secuencia resultante.

Generador X9.17

Propuesto por el Instituto Nacional de Estándares Norteamericano, permite, a partir de una *semilla* inicial s_0 de 64 bits, obtener secuencias de valores también de 64 bits. En sus cálculos emplea valores difíciles

de adivinar desde el exterior, como el tiempo del sistema en el momento de obtener cada elemento de la secuencia, para de esta forma aproximar más su comportamiento al de un generador totalmente aleatorio. El algoritmo para obtener cada uno de los valores g_n de la secuencia es el siguiente:

$$\begin{aligned} g_n &= \text{DES}_k(\text{DES}_k(t) \oplus s_n) \\ s_{n+1} &= \text{DES}_k(\text{DES}_k(t) \oplus g_n) \end{aligned}$$

donde k es una clave reservada para la generación de cada secuencia, y t es el tiempo en el que cada valor es generado —cuanta más resolución tenga (hasta 64 bits), mejor—. $\text{DES}_k(M)$ representa el cifrado de M mediante el algoritmo DES (capítulo 10), empleando la clave k, y $\oplus$ representa la función *or-exclusivo*. Nótese que el valor k ha de ser mantenido en secreto para que la seguridad de este generador sea máxima.

Generador Blum Blum Shub

Si bien se trata en realidad de un generador pseudoaleatorio, es uno de los algoritmos que más pruebas de resistencia ha superado, con la ventaja adicional de su gran simplicidad —aunque es computacionalmente mucho más costoso que el algoritmo *X9.17*—. Se basa en el problema de los residuos cuadráticos (sección 5.4.6), y consiste en escoger dos números primos *grandes*, p y q, que cumplan la siguiente propiedad:

$$p \equiv 3 \pmod 4 \qquad \text{y} \qquad q \equiv 3 \pmod 4$$

Sea entonces $n = pq$. Escogemos un número x aleatorio primo relativo con n, que será nuestra *semilla* inicial. Al contrario que x, que debe ser mantenido en secreto, n puede ser público. Calculamos los valores s_i de la serie de la siguiente forma:

$$\begin{aligned} s_0 &= (x^2) \pmod{n} \\ s_{i+1} &= (s_i^2) \pmod{n} \end{aligned}$$

Hay que tener cuidado de emplear únicamente como salida unos pocos de los bits menos significativos de cada s_i. De hecho, si cogemos no más que $\log_2(\log_2(s_i))$ bits en cada caso podemos asegurar que predecir el siguiente valor de la serie es al menos tan difícil como factorizar n.

Generador Fortuna

Fue propuesto por Bruce Schneier y Niels Ferguson en 2003. Está diseñado para ser resistente a multitud de ataques, integrando diferentes fuentes de entropía, con la característica de que sigue siendo seguro siempre y cuando una de las fuentes funcione correctamente.

Fortuna posee un estado general $\mathcal{G} = (K, C)$, donde K es un valor de clave de 256 bits, y C es un contador de 128 bits. Su funcionamiento se organiza en torno a tres elementos:

- *Generador:* A partir de K y C genera cadenas de *bytes* pseudoaleatorias de longitud arbitraria, hasta un límite máximo de 1 MB.
- *Acumulador:* Integra la entropía de diferentes fuentes, externas o no al sistema, y emplea el resultado para realimentar periódicamente el generador de secuencia o, lo que es lo mismo, cambiar el valor de K. No se permite el uso del generador hasta que el acumulador no haya actuado al menos una vez.
- *Fichero de semilla:* Este componente está pensado para dispositivos en los que sea necesario generar números aleatorios inmediatamente después de ser iniciados, antes de haber podido recolectar suficiente entropía de las fuentes externas. Su propósito es salvaguardar la entropía acumulada por el generador cuando el sistema se apague y así poder emplearla tras el arranque.

El generador se basa en el empleo de un algoritmo de cifrado simétrico E (capítulo 10) con tamaño de clave de 256 bits y tamaño de bloque de 128 bits en modo CTR (sección 11.4.2). Cada bloque de 16 *bytes* de la secuencia se calcula como $E_K(C)$, incrementando posteriormente el valor de C. En principio, este método por sí solo permite tener secuencias enormemente largas, pero tiene el problema de que nunca se repetiría un bloque, ya que los textos claros para las operaciones de cifrado son todos distintos, y en una secuencia verdaderamente aleatoria siempre existe la posibilidad de que aparezcan bloques repetidos. Por esta razón se limita el tamaño máximo de las secuencias que se le pueden solicitar al generador en una sola llamada a 2^{16} bloques (1 MB). Al finalizar el cálculo de cada secuencia, independientemente de su longitud, se generan dos bloques adicionales que serán el nuevo valor de K, todo ello sin reiniciar el valor de C.

El acumulador funciona combinando las fuentes de entropía mediante los denominados *depósitos*, de longitud arbitraria. En particular, se trabaja con treinta y dos de ellos, numerados de 0 a 31:

$$P_0, P_1, \ldots, P_{31}$$

Cada una de las fuentes de entropía distribuye la información que genera, de forma cíclica, añadiéndola al final de cada uno de los depósitos. Cada vez que P_0 alcanza un tamaño suficiente, se emplea la información procedente de los depósitos para alimentar el generador de la siguiente forma:

- Existe un valor r, entero de 128 bits, que actúa como contador y que al principio se inicializa a 0. Este valor se incrementa cada vez que se realimenta el generador.
- Se concatena K con los resúmenes de 256 bits (capítulo 13) de todos aquellos depósitos P_i que cumplan que 2^i sea un divisor de r. El nuevo valor para K será el *hash* de esta concatenación.
- Se incrementa el valor de C.

- Los depósitos empleados en el paso anterior se limpian, asignándoles la cadena vacía.

En cuanto al fichero de semilla, en él se almacena una cadena s de 64 *bytes* obtenidos a través del propio generador aleatorio. En el proceso de carga del fichero se llevan a cabo los siguientes pasos:

- Se calcula un nuevo valor para K como el *hash* de la concatenación entre el K actual y s.
- Se incrementa el valor de C.
- Se obtiene una nueva cadena de 64 *bytes* aleatorios y se graba en el fichero, sobrescribiendo los valores anteriores.

Parte III

Algoritmos criptográficos

Capítulo 9

Criptografía clásica

El ser humano siempre ha tenido secretos de muy diversa índole, y ha buscado mecanismos para mantenerlos fuera del alcance de *miradas indiscretas*. Julio César empleaba una sencilla técnica para evitar que sus comunicaciones militares fueran interceptadas. Leonardo Da Vinci escribía las anotaciones sobre sus trabajos de derecha a izquierda y con la mano zurda. Otros personajes, como Sir Francis Bacon o Edgar Allan Poe eran conocidos por su afición a los códigos criptográficos, que en muchas ocasiones constituían un apasionante divertimento y un reto para el ingenio.

En este capítulo haremos un breve repaso de los mecanismos criptográficos considerados *clásicos*. Podemos llamar así a todos los sistemas de cifrado anteriores a la II Guerra Mundial, o lo que es lo mismo, al nacimiento de las computadoras. Estas técnicas tienen en común que pueden ser empleadas usando simplemente lápiz y papel, y que pueden ser criptoanalizadas casi de la misma forma. De hecho, con la ayuda de las computadoras, los mensajes cifrados mediante el uso de estos códigos son fácilmente descifrables, por lo que cayeron rápidamente en desuso.

La transición desde la Criptografía clásica a la moderna se da precisamente durante la II Guerra Mundial, cuando el Servicio de Inte-

ligencia aliado *rompe* dos sistemas empleados por el ejército alemán, la máquina ENIGMA y el cifrado de Lorenz, considerados hasta ese momento absolutamente inexpugnables. Pero lo más importante de esa victoria es que se consigue a través de revolucionarios desarrollos matemáticos, combinados con el nacimiento de las computadoras modernas.

Todos los algoritmos criptográficos clásicos son de carácter simétrico, ya que hasta mediados de los años setenta no nació la Criptografía Asimétrica.

9.1. Algoritmos clásicos de cifrado

Estudiaremos en esta sección algunos criptosistemas que en la actualidad han perdido gran parte de su eficacia, debido a que son fácilmente criptoanalizables empleando una computadora doméstica, bien mediante análisis estadístico o directamente por la fuerza bruta, pero que fueron empleados con éxito hasta principios del siglo XX. Algunos se remontan incluso, como el algoritmo de César, a la Roma Imperial. Sin embargo aún conservan el interés teórico, ya que algunas de sus propiedades resultan muy útiles para entender mejor los algoritmos modernos.

9.1.1. Cifrados de sustitución

Los algoritmos de esta familia se basan en cambiar por otros los símbolos del mensaje, sin alterar su orden relativo. Cada uno de ellos vendrá definido por el mecanismo concreto empleado para efectuar dicho cambio, pudiendo ser independiente de la posición que ocupa el símbolo el el mensaje *(cifrados monoalfabéticos)*, o venir determinado por ésta *(cifrados polialfabéticos)*. Como vimos en la sección 3.8, esta transformación se corresponde con el concepto de *confusión*.

Cifrados monoalfabéticos

Se engloban dentro de este apartado todos los algoritmos criptográficos que, sin desordenar los símbolos dentro del mensaje, establecen una única función de sustitución para todos ellos a lo largo del texto. Es decir, si al símbolo A le corresponde el símbolo D, esta correspondencia se mantiene para todo el mensaje.

Algoritmo de César. El algoritmo de *César*, llamado así porque es el que empleaba Julio César para enviar mensajes secretos, es uno de los algoritmos criptográficos más simples. Consiste en sumar 3 al número de orden de cada letra. De esta forma a la A le corresponde la D, a la B la E, y así sucesivamente. Si asignamos a cada letra un número ($A = 0$,$B = 1\ldots$), y consideramos un alfabeto de 26 letras, la transformación criptográfica sería:

$$c = (m + 3) \text{ mód } 26$$

obsérvese que este algoritmo ni siquiera posee clave, puesto que la transformación siempre es la misma. Obviamente, para descifrar basta con restar 3 al número de orden de las letras del criptograma.

Sustitución Afín. Es el caso general del algoritmo de César. Su transformación sería:

$$E_{(a,b)}(m) = (a \cdot m + b) \text{ mód } N$$

siendo a y b dos números enteros menores que el cardinal N del alfabeto, y cumpliendo que $mcd(a, N) = 1$. La clave de cifrado k viene entonces dada por el par (a, b). El algoritmo de César será pues una transformación afín con $k = (1, 3)$.

Cifrado Monoalfabético General. Es el caso más general de cifrado monoalfabético. La sustitución ahora es arbitraria, siendo la clave k precisamente la tabla de sustitución de un símbolo por otro. En este caso tenemos $N!$ posibles claves.

Criptoanálisis de los Métodos de Cifrado Monoalfabéticos. El cifrado monoalfabético constituye la familia de métodos más simple de criptoanalizar, puesto que las propiedades estadísticas del texto claro se conservan en el criptograma. Supongamos que, por ejemplo, la letra que más aparece en Castellano es la A. Parece lógico que la letra más frecuente en el texto codificado sea aquella que corresponde con la A. Emparejando las frecuencias relativas de aparición de cada símbolo en el mensaje cifrado con el histograma de frecuencias del idioma en el que se supone está el texto claro, podremos averiguar fácilmente la clave.

En el peor de los casos, es decir, cuando tenemos un emparejamiento arbitrario, la Distancia de Unicidad de Shannon que obtenemos es:

$$S = \frac{H(K)}{D} = \frac{\log_2(N!)}{D} \tag{9.1}$$

donde D es la redundancia del lenguaje empleado en el mensaje original, y N es el número de símbolos de dicho lenguaje. Como es lógico, suponemos que las $N!$ claves diferentes son equiprobables en principio.

En casos más restringidos, como el afín, el criptoanálisis es aún más simple, puesto que el emparejamiento de todos los símbolos debe responder a alguna combinación de coeficientes (a, b).

Cifrados polialfabéticos

En los cifrados polialfabéticos la sustitución aplicada a cada carácter varía en función de la posición que ocupe éste dentro del tex-

to claro. En realidad corresponde a la aplicación cíclica de n cifrados monoalfabéticos.

Cifrado de Vigenère. Es un ejemplo típico de cifrado polialfabético que debe su nombre a Blaise de Vigenère, su creador, y que data del siglo XVI. La clave está constituida por una secuencia de símbolos $K = \{k_0, k_1, \ldots k_{d-1}\}$, y se emplea la siguiente función de cifrado:

$$E_k(m_i) = m_i + k_{(i \mod d)} \pmod n$$

siendo m_i el $i-$ésimo símbolo del texto claro y n el cardinal del alfabeto de entrada.

Criptoanálisis. Para criptoanalizar este tipo de claves basta con efectuar d análisis estadísticos independientes agrupando los símbolos según la k_i empleada para codificarlos. Necesitaremos al menos d veces más cantidad de texto que con los métodos monoalfabéticos.

En lo que respecta a la estimación del valor de d, podemos emplear el método propuesto por Friedrich Kasiski en 1863, que consiste en buscar subcadenas de tres o más letras repetidas dentro del texto cifrado, y anotar las distancias s_i que las separan. Lo más probable es que los patrones encontrados se correspondan con subcadenas repetidas también en el texto claro, separadas por un número de caracteres múltiplo de d. Podremos, por tanto, estimar d calculando el máximo común divisor de todos los s_i que hayamos localizado.

Cifrados por sustitución homofónica

Para paliar la sensibilidad frente a ataques basados en el estudio de las frecuencias de aparición de los símbolos, existe una familia de algoritmos monoalfabéticos que trata de ocultar las propiedades estadísticas del texto claro, empleando un alfabeto de salida con más

símbolos que el alfabeto de entrada.

Supongamos que nuestro alfabeto de entrada posee cuatro letras, $\{a, b, c, d\}$. Supongamos además que en nuestros textos la letra a aparece con una probabilidad 0.4, y el resto con probabilidad 0.2. Podríamos emplear el siguiente alfabeto de salida $\{\alpha, \beta, \gamma, \delta, \epsilon\}$ efectuando la siguiente asociación:

$$\begin{array}{rcc} E(a) & = & \left\{ \begin{array}{ll} \alpha & \text{con probabilidad } 1/2 \\ \beta & \text{con probabilidad } 1/2 \end{array} \right. \\ E(b) & = & \gamma \\ E(c) & = & \delta \\ E(d) & = & \epsilon \end{array}$$

En el texto cifrado ahora todos los símbolos aparecen con igual probabilidad, maximizando su entropía y eliminando su redundancia, lo que en principio imposibilita un ataque basado en frecuencias. A pesar de su gran potencial, este método necesita un alfabeto de salida mayor que el de entrada —y tanto más grande cuanto mejor queramos *aplanar* el histograma de frecuencias del criptograma—, lo cual representa un problema, especialmente en lo que a aplicaciones informáticas se refiere.

9.1.2. Cifrados de transposición

Este tipo de mecanismos de cifrado no sustituye unos símbolos por otros, sino que cambia su orden dentro del texto, siguiendo el concepto de *difusión* definido por Shannon. Quizás el más antiguo conocido sea el *escitalo*, empleado en la Antigua Grecia, especialmente en Esparta. Este dispositivo estaba formado por un bastón cilíndrico con un radio particular y una tira de piel que se enrollaba alrededor de aquél. El texto se escribía a lo largo del bastón y sólo podía ser leído si se disponía de otro bastón de dimensiones similares. Un mecanismo de transposición sencillo, que no precisa otra cosa que lápiz y papel, consiste en colocar el texto en una tabla de n columnas, y dar como texto cifrado

los símbolos de una columna —ordenados de arriba abajo— concatenados con los de otra, etc. La clave k se compone del número n junto con el orden en el que se deben leer las columnas.

Por ejemplo, supongamos que queremos cifrar el texto *"El perro de San Roque no tiene rabo"*, con $n = 5$ y la permutación $\{3, 2, 5, 1, 4\}$ como clave. Colocamos el texto en una tabla y obtenemos:

1	2	3	4	5
E	L		P	E
R	R	O		D
E		S	A	N
	R	O	Q	U
E		N	O	
T	I	E	N	E
	R	A	B	O

Tendríamos como texto cifrado la concatenación de las columnas 3,2,5,1 y 4 respectivamente: *" Osonealr r irednu eoere et p aqonb"*. Nótese que hemos de conservar el espacio al principio del texto cifrado para que el mecanismo surta efecto.

Criptoanálisis. Este tipo de mecanismos de cifrado se puede criptoanalizar efectuando un estudio estadístico sobre la frecuencia de aparición de pares y tripletas de símbolos en el lenguaje en que esté escrito el texto claro. Suponiendo que conocemos n, que en nuestro caso es igual a 5, tenemos $5! = 120$ posibles claves. Descifraríamos el texto empleando cada una de ellas y comprobaríamos si los pares y tripletas de símbolos consecutivos que vamos obteniendo se corresponden con los más frecuentes en Castellano. De esa forma podremos asignarle una probabilidad automáticamente a cada una de las posibles claves.

Si, por el contrario, desconocemos n, basta con ir probando con $n = 2$, $n = 3$ y así sucesivamente. Este método es bastante complejo de llevar a cabo manualmente, a no ser que se empleen ciertos tru-

cos, pero una computadora puede completarlo en un tiempo más que razonable sin demasiados problemas.

9.2. Máquinas de rotores. La máquina ENIGMA

En el año 1923, un ingeniero alemán llamado Arthur Scherbius patentó una máquina específicamente diseñada para facilitar las comunicaciones seguras. Se trataba de un instrumento de apariencia simple, parecido a una máquina de escribir. Quien deseara codificar un mensaje sólo tenía que teclearlo y las letras correspondientes al texto cifrado se irían iluminando en un panel. El destinatario copiaba dichas letras en su propia máquina y el mensaje original aparecía de nuevo. La *clave* la constituían las posiciones iniciales de tres tambores o *rotores* que el ingenio poseía en su parte frontal.

En la figura 9.1 podemos apreciar un esquema de esta máquina, llamada ENIGMA. Los rotores no son más que tambores con contactos en su superficie y cableados en su interior, de forma que con cada pulsación del teclado, la posición de éstos determina cuál es la letra que se ha de iluminar. Cada vez que se pulsa una tecla el primer rotor avanza una posición; el segundo avanza cuando el anterior ha dado una vuelta completa y así sucesivamente. El reflector no existía en los primeros modelos, se introdujo posteriormente para permitir que la misma máquina sirviera tanto para cifrar como para descifrar, como veremos más adelante.

9.2.1. Un poco de historia

ENIGMA pronto llamó la atención del ejército alemán, que la utilizó de forma intensiva a lo largo de la II Guerra Mundial. Además se le aplicaron varias mejoras, como incluir un pequeño sistema pre-

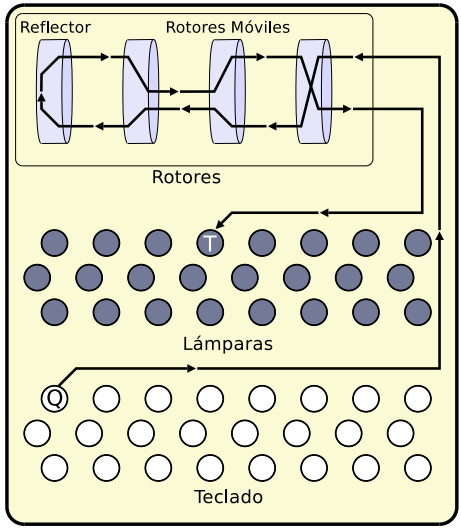

Figura 9.1: Esquema de la máquina Enigma.

vio de permutación, llamado *stecker* o clavijero —que permitía escoger seis pares de letras para ser intercambiadas previamente al cifrado—, hacer que los rotores fueran intercambiables —se podían elegir y colocar en cualquier orden tres de entre cinco disponibles—, e incluso ampliar a cuatro el número de rotores.

Aunque ENIGMA parecía virtualmente imposible de romper, presentaba una serie de debilidades, tanto en su diseño como en los mecanismos empleados para utilizarla, que fueron aprovechadas por el ejército aliado. El primero en conseguir avances significativos fue el servicio de inteligencia polaco, ya que en 1931 los franceses, en virtud de un acuerdo de cooperación firmado diez años antes, les facilitaron información detallada sobre la máquina[1], que ellos a su vez habían obtenido sobornando a un miembro de la oficina de cifras alemana. De hecho, los espías franceses consideraban esta información totalmente

[1]En anteriores ediciones de este libro se mencionaba el envío por error a Polonia de una máquina, pero parece que se trata simplemente de una leyenda.

inútil, ya que pensaban que ENIGMA era, sencillamente, indescifrable.

El conocimiento preciso de la máquina permitió a un equipo de tres matemáticos (Marian Rejewski, Jerzy Rozycki y Henryk Zygalski) elaborar un mecanismo para aprovechar una debilidad, no en la máquina en sí, sino en el protocolo empleado por el ejército alemán para colocar los rotores al principio de cada mensaje. Dicho protocolo consistía en escoger una posición de un libro de claves, y enviar tres letras cualesquiera *dos veces*, para evitar posibles errores. En realidad se estaba introduciendo una redundancia en el mensaje que permitía obtener, con un nivel de esfuerzo al alcance de los polacos, la clave empleada. Se construyó un aparato que permitía descifrar los mensajes, y se le bautizó como *Ciclómetro*.

En 1938 Alemania cambió el protocolo, lo cual obligó a los matemáticos polacos a refinar su sistema, aunque básicamente se seguían enviando tres letras repetidas. No vamos a entrar en detalles, pero el ataque se basaba en buscar ciertas configuraciones de la máquina, con propiedades específicas. Estas configuraciones especiales daban una información vital sobre la posición inicial de los rotores para un mensaje concreto. Se construyó entonces una versión mejorada del ciclómetro, llamada *Bomba*, que era capaz de encontrar estas configuraciones de forma automática. Sin embargo, a finales de ese mismo año se introdujeron dos rotores adicionales, lo cual obligaba a emplear sesenta *bombas* simultáneamente para romper el sistema. Polonia simplemente carecía de medios económicos para afrontar su construcción.

Los polacos entonces pusieron en conocimiento de los servicios secretos británico y francés sus progresos, esperando poder establecer una vía de colaboración para seguir descifrando los mensajes germanos, pero la invasión de Polonia era inminente. Tras destruir todas las pruebas que pudieran indicar al ejército alemán el éxito polaco frente a ENIGMA, el equipo de Rejewski huyó precipitadamente, transportando lo que pudieron salvar en varios camiones. Tras pasar por Rumanía e Italia, y tener que quemar todos los camiones por el camino excepto uno, llegaron a París, donde colaboraron con un equipo

de siete españoles expertos en criptografía, liderados por un tal Camazón. Cuando al año siguiente Alemania invadió Francia el nuevo equipo tuvo que huir a África, y posteriormente instalarse en Montpellier, donde reanudaron sus trabajos. En 1942, la entrada alemana en Vichy forzó a los matemáticos a escapar de nuevo, los polacos a España (donde murió Rozycki), y los españoles a África, donde se perdió definitivamente su pista.

Cuando el equipo de Rejewski llegó por fin a Inglaterra, ya no se le consideró seguro, al haber estado en contacto con el enemigo, y se le confiaron únicamente trabajos menores. Mientras tanto, en Bletchley Park, Alan Turing desarrollaba una segunda *Bomba* basándose en los estudios del polaco, más evolucionada y rápida que su antecesora, en el marco del proyecto *ULTRA* británico, que se encargaba de recoger información acerca de los sistemas de comunicaciones germanos. Este nuevo dispositivo aprovechaba una debilidad esencial en ENIGMA: un mensaje no puede codificarse en sí mismo, lo cual implica que ninguna de las letras del texto claro puede coincidir con ninguna del texto cifrado. La Bomba de Turing partía de una *palabra adivinada* —en contra de las normas de uso de ENIGMA, la mayoría de los mensajes que enviaba el ejército alemán comenzaban de igual forma, lo cual facilitó la tarea del equipo aliado enormemente—, y buscaba un emparejamiento con el mensaje cifrado tal que el supuesto texto claro y el fragmento de criptograma asociado no coincidieran en ninguna letra. A partir de ahí la Bomba realizaba una búsqueda exhaustiva de la configuración inicial de la máquina para decodificar el mensaje, mediante un ingenioso sistema que permitía ignorar la posición del *stecker*.

Un hecho bastante poco conocido es que Alemania regaló al régimen de Franco casi una veintena de máquinas ENIGMA, que fueron utilizadas para comunicaciones secretas hasta entrados los años cincuenta, suponemos que para regocijo de los servicios de espionaje británico y norteamericano.

9.2.2. Consideraciones teóricas sobre la máquina ENIGMA

Observemos que un rotor no es más que una permutación dentro del alfabeto de entrada. El cableado hace que cada una de las letras se haga corresponder con otra. Todas las letras tienen imagen y no hay dos letras con la misma imagen. Si notamos una permutación como π, podemos escribir que la permutación resultante de combinar todos los rotores en un instante dado es:

$$\pi_{total} = \langle \pi_0, \pi_1, \pi_2, \pi_3, \pi_2^{-1}, \pi_1^{-1}, \pi_0^{-1} \rangle$$

La permutación π_3 corresponde al reflector, y debe cumplir que $\pi_3 = \pi_3^{-1}$, es decir, que aplicada dos veces nos dé lo mismo que teníamos al principio. De esta forma se cumple la propiedad de que, para una misma posición de los rotores, la codificación y la decodificación son simétricas.

La fuerza de la máquina ENIGMA radica en que tras codificar cada letra se giran los rotores, lo cual hace que la permutación que se aplica a cada letra sea diferente. La máquina, por tanto, es un sistema de cifrado de sustitución polialfabética. Además, cada sustitución concreta no se repite hasta que los rotores recuperan su posición inicial, lo que da lugar a un tamaño de ciclo realmente grande. Tengamos en cuenta que hay 17576 posiciones iniciales de los rotores, y 60 combinaciones de tres rotores a partir de los cinco de entre los que se puede elegir. Puesto que el *stecker* presenta en torno a cien mil millones de combinaciones, existe una cantidad enorme de posibles disposiciones iniciales de la máquina —aproximadamente 10^{17}—. La potencia del método de criptoanálisis empleado radica en que se podía identificar un emparejamiento válido entre el criptograma y el texto claro, e ignorar la posición del *stecker*, de forma que sólo bastaba con rastrear dentro del espacio de posibles configuraciones para encontrar aquella que llevara a cabo la transformación esperada. No disponer de dicho emparejamiento hubiera complicado enormemente el criptoanálisis, tal vez

hasta el punto de hacerlo fracasar.

9.2.3. Otras máquinas de rotores

Además de la máquina alemana ENIGMA, existieron otros dispositivos criptográficos basados en rotores. En esta sección comentaremos SIGABA y PURPLE, ambos empleados durante la II Guerra Mundial.

La máquina SIGABA

Esta máquina de rotores, también conocida como ECM Mark II, Converter M-134 y CSP-889, fue empleada por el ejército de los EE.UU. durante la Segunda Guerra Mundial. A diferencia de la máquina Enigma, en la que los rotores avanzan una posición cada vez que se pulsa una tecla, SIGABA incorpora un segundo juego de rotores, que se encarga de *decidir* qué rotores principales avanzan cada vez que se pulsa una tecla. Esto aumenta considerablemente la longitud de ciclo de la máquina, y complica la localización de posibles patrones en los textos cifrados.

El principal inconveniente de esta máquina era su excesivo peso y tamaño, sin contar con su complejidad mecánica, dificultad de manejo y fragilidad. Esto supuso que, en la práctica, no pudiera ser utilizada en muchas situaciones a lo largo de la guerra, a diferencia de la máquina Enigma, mucho más ligera y resistente. En su lugar, se usaba, entre otros, el famoso *código* consistente en emplear indios navajos, que simplemente se comunicaban por radio en su propio idioma, demasiado desconocido y complejo como para ser comprendido por el enemigo.

La Máquina PURPLE

Esta máquina, bautizada como PURPLE por los EE.UU., fue empleada por el gobierno japonés desde poco antes de iniciarse la Se-

gunda Guerra Mundial, con fines diplomáticos. Se trata de la sucesora de otra máquina, denominada RED, y fue diseñada por un capitán de la armada japonesa. Criptoanalizada durante la II Guerra Mundial por un equipo del Servicio de Inteligencia de Señales de la Armada de EE.UU., dirigido por William Friedman, debió su caída más a una mala política de elección de claves, que a debilidades intrínsecas de la propia máquina.

9.3. El cifrado de Lorenz

El cifrado de Lorenz se llevaba a cabo mediante una máquina, denominada SZ40, que tenía un diseño considerablemente más complejo que el de la ENIGMA. Para su descifrado se desarrolló una máquina descifradora, denominada Colossus, que supuso el germen de las computadoras tal y como hoy las conocemos. Si bien la *Bomba* efectuaba en esencia una búsqueda sistemática de la clave, el cifrado de Lorenz necesitaba de análisis estadísticos más complejos, y para ello el matemático Max Newman[2], inspirándose en el concepto de Máquina Universal de Turing, dirigió el desarrollo de una máquina, bautizada con el nombre de Colossus, que podría muy bien ser considerada como la primera computadora moderna, aunque su existencia se mantuvo en secreto hasta mediados de los años 70. El ingenio, cuya construcción fue llevada a cabo por el ingeniero Tommy Flowers, constaba de unas 1.500 válvulas de vacío, tecnología mucho más moderna que los relés que constituían el corazón de las *Bombas*.

En el cifrado de Lorenz se empleaba para codificar teletipos, en los que los textos venían representados por una matriz formada por columnas de cinco puntos. Cada una de esas columnas correspondía a una letra, y en cada punto podía haber (o no) un agujero. En esencia, tenemos un sistema de codificación de cinco bits por letra. La máquina de Lorenz generaba una secuencia pseudoaleatoria (ver capítulo

[2]No confundir con John Von Neumann, que también hizo aportaciones cruciales en los inicios de la Informática, pero nunca estuvo en Bletchley Park.

8) binaria que era combinada con la matriz de puntos, mediante una operación *or–exclusivo* para producir el criptograma.

El protocolo de comunicaciones usado para la máquina de Lorenz obligaba a usar secuencias pseudoaleatorias distintas para mensajes distintos. Sin embargo, en agosto de 1941, un operador alemán cometió un terrible error: tenía que transmitir un mensaje de cerca de 4.000 caracteres, y tras enviarlo, recibió la siguiente respuesta: *"¿podrías repetirlo?"*. El operador comenzó a codificar de nuevo el mensaje a mano, y, probablemente molesto por tener que repetir la operación, comenzó a abreviar el texto claro, de tal forma que se enviaron dos mensajes diferentes combinados con la misma secuencia pseudoaleatoria. Esta información permitió a los espías del bando contrario extraer la secuencia y comenzar a extraer patrones de la misma —después de todo, resultó más *pseudo* que *aleatoria*—. A partir de aquí el único problema fue que para descifrar los mensajes, el método manual se mostraba demasiado lento. Precisamente por eso se desarrolló Colossus.

9.3.1. Consideraciones teóricas sobre el cifrado de Lorenz

La máquina SZ40 pretendía emular un sistema Seguro de Shannon (sección 3.5), pero para ello las secuencias generadas tendrían que ser totalmente aleatorias. Sin embargo, si las secuencias producidas por la máquina fueran de este tipo, sería imposible reproducirlas en los dos extremos de la comunicación, por lo que el sistema en realidad es una técnica de cifrado de Flujo (capítulo 11).

Si uno dispone de dos mensajes con sentido en un idioma determinado, cifrados con la misma secuencia pseudoaleatoria, bastará con buscar cadenas de bits que permitan descifrar simultáneamente ambos mensajes. Por ejemplo, supongamos la siguiente codificación binaria para cada letra:

a	b	c	d	e	f	g	h	i	j	k	l	m
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	1	1	1
0	0	0	0	1	1	1	1	0	0	0	0	1
0	0	1	1	0	0	1	1	0	0	1	1	0
0	1	0	1	0	1	0	1	0	1	0	1	0

n	o	p	q	r	s	t	u	v	w	x	y	z
0	0	0	1	1	1	1	1	1	1	1	1	1
1	1	1	0	0	0	0	0	0	0	0	1	1
1	1	1	0	0	0	0	1	1	1	1	0	0
0	1	1	0	0	1	1	0	0	1	1	0	0
1	0	1	0	1	0	1	0	1	0	1	0	1

Si nos encontramos los mensajes:

10100 11000 11000 10101
10000 11000 11001 11011

podemos generar las 1.024 combinaciones posibles de 10 bits, y tratar de descifrar las dos primeras letras de cada mensaje. Nos quedaremos únicamente con aquellas combinaciones de bits que den lugar a sílabas (o partes de sílabas) legales en castellano en ambos mensajes. Por ejemplo, la cadena 1010101010 da lugar a las letras BS para el primer mensaje, y FS para el segundo, ambas con muy poca probabilidad de aparecer al principio de un mensaje correcto en castellano.

Si, por el contrario, contáramos con un único mensaje cifrado, este análisis resultaría imposible, ya que para todos y cada uno de los mensajes posibles en castellano existirá una secuencia de bits que lo genera. La clave está en que existirán muy pocas secuencias —tal vez solo una— que den lugar, en ambos mensajes cifrados, a textos claros válidos.

La tarea es tediosa, pero da resultado, incluso si la secuencia empleada para cifrar es totalmente aleatoria. En consecuencia, los fallos

sobre los que se cimentó el éxito del criptoanálisis del cifrado de Lorenz fueron dos: en primer lugar, el cifrado accidental de dos mensajes distintos con la misma secuencia, y en segundo, el carácter poco aleatorio de la secuencia en cuestión.

9.4. Ejercicios propuestos

1. Descifre los mensajes comentados en la sección 9.3.1, teniendo en cuenta cada uno de ellos es una palabra completa y correcta en castellano.

Capítulo 10

Cifrados por bloques

10.1. Introducción

Una gran parte de los algoritmos de cifrado simétrico opera dividiendo el mensaje que se pretende codificar en bloques de tamaño fijo, y aplican sobre cada uno de ellos una combinación más o menos compleja de operaciones de confusión —sustituciones— y difusión —transposiciones— (ver sección 3.8). Estas técnicas se denominan, en general, *cifrados por bloques*.

Recordemos que la confusión consiste en tratar de ocultar la relación que existe entre el texto claro, el texto cifrado y la clave. Un buen mecanismo de confusión hará demasiado complicado extraer relaciones estadísticas entre las tres cosas. Por su parte la difusión trata de repartir la influencia de cada bit del mensaje original lo más posible entre el mensaje cifrado.

Un hecho digno de ser tenido en cuenta es que la confusión por sí sola resulta suficiente, ya que si establecemos una tabla de sustitución completamente diferente para cada clave con todos los textos claros posibles tendremos un sistema extremadamente seguro. Sin embargo, dichas tablas ocuparían cantidades astronómicas de memoria, por lo

que en la práctica resultan inviables. Por ejemplo, un algoritmo que codificara bloques de 128 bits empleando una clave de 80 bits necesitaría una tabla de sustitución con un tamaño del orden de 10^{64} bits.

Lo que en realidad se hace para conseguir algoritmos fuertes sin necesidad de almacenar tablas enormes es intercalar la confusión (sustituciones simples, con tablas pequeñas) y la difusión (permutaciones). Esta combinación se conoce como *cifrado de producto*. La mayoría de los algoritmos se basan en diferentes capas de sustituciones y permutaciones, estructura que denominaremos *Red de Sustitución-Permutación*. En muchos casos el criptosistema no es más que una operación combinada de sustituciones y permutaciones, repetida n veces, como ocurre con DES.

10.1.1. Redes de Feistel

Muchos algoritmos de cifrado tienen en común que dividen un bloque de longitud n en dos mitades, L y R. Se define entonces un cifrado de producto iterativo en el que la salida de cada ronda se usa como entrada para la siguiente según la relación (ver figura 10.1):

$$\left.\begin{array}{rcl} L_i & = & R_{i-1} \\ R_i & = & L_{i-1} \oplus f(R_{i-1}, K_i) \end{array}\right\} \quad \text{si } i < n.$$
$$\begin{array}{rcl} L_n & = & L_{n-1} \oplus f(R_{n-1}, K_n) \\ R_n & = & R_{n-1} \end{array} \tag{10.1}$$

Este tipo de estructura se denomina *Red de Feistel*, y es empleada en multitud de algoritmos, como DES, Lucifer, FEAL, CAST, Blowfish, etc. Tiene la interesante propiedad de que para invertir la función de cifrado —es decir, para descifrar— basta con aplicar el mismo algoritmo, pero con las K_i en orden inverso. Nótese, además, que esto ocurre independientemente de cómo sea la función f.

Podemos emplear la *inducción matemática* [1] para comprobar esta

[1] Este principio garantiza que si demostramos el caso correspondiente a $n = 1$,

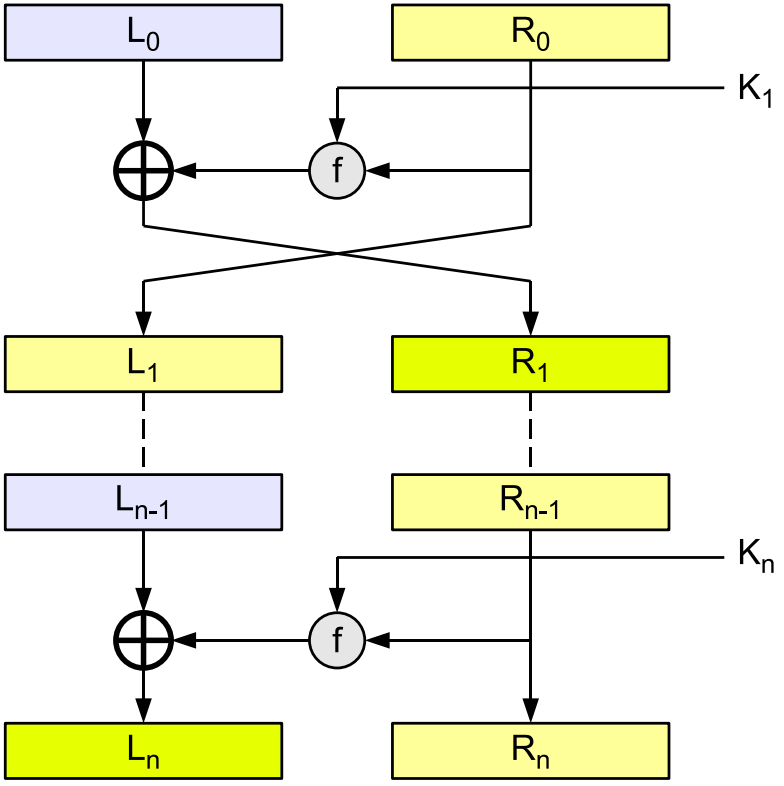

Figura 10.1: Estructura de una red de Feistel.

propiedad. Sea $E_n(L,R)$ la función de cifrado para una red de Feistel de n rondas y $D_n(L,R)$ la función de descifrado análoga. Desdoblaremos cada función en sus bloques izquierdo y derecho y los denotaremos con superíndices, $E_n^L(L,R)$, $E_n^R(L,R)$, $D_n^L(L,R)$ y $D_n^R(L,R)$. Hemos de demostrar que

$$D_n^L(E_n^L(L,R),E_n^R(L,R)) = L \qquad \text{y} \qquad D_n^R(E_n^L(L,R),E_n^R(L,R)) = R$$

para cualquier valor de n.

- Si $n = 1$ tenemos:

$$\begin{aligned} E_1^L(A,B) &= A \oplus f(B,K_1) \\ E_1^R(A,B) &= B \\ &\text{y} \\ D_1^L(A,B) &= A \oplus f(B,K_1) \\ D_1^R(A,B) &= B \end{aligned}$$

  luego

$$D_1^L\Big(E_1^L(L,R),E_1^R(L,R)\Big) = E_1^L(L,R) \oplus f\Big(E_1^R(L,R),K_1\Big) = \\ = \Big(L \oplus f(R,K_1)\Big) \oplus f(R,K_1) = L$$

$$\text{y}$$

$$D_1^R\Big(E_1^L(L,R),E_1^R(L,R)\Big) = E_1^R(L,R) = R$$

- Suponiendo que se cumple el caso n, demostrar el caso $n+1$:

  Nótese en primer lugar que cifrar con $n+1$ rondas equivale a hacerlo con n rondas, permutar el resultado y aplicar el paso $n+1$ de cifrado según la relación 10.1:

$$\begin{aligned} E_{n+1}^L(L,R) &= E_n^R(L,R) \oplus f\Big(E_n^L(L,R),K_{n+1}\Big) \\ E_{n+1}^R(L,R) &= E_n^L(L,R) \end{aligned}$$

---

y luego demostramos el caso $n+1$ suponiendo cierto el caso n, la propiedad en cuestión ha de cumplirse para cualquier valor entero de n igual o superior a 1

El descifrado con $n+1$ será igual a aplicar el primer paso del algoritmo con K_{n+1} y luego descifrar el resultado con n rondas:

$$D_{n+1}(A,B) = D_n\Big(B, A \oplus f(B, K_{n+1})\Big)$$

Haciendo que A y B sean ahora el resultado de cifrar con $n+1$ rondas tenemos:

$$D_{n+1}\Big(E^L_{n+1}(L,R), E^R_{n+1}(L,R)\Big) =$$
$$= D_n\Big(E^R_{n+1}(L,R), E^L_{n+1}(L,R) \oplus f(E^R_{n+1}(L,R), K_{n+1})\Big)$$

Sustituyendo $E^L_{n+1}(L,R)$ y $E^R_{n+1}(L,R)$ en la parte derecha de la anterior expresión nos queda:

$$D_{n+1}\Big(E^L_{n+1}(L,R), E^R_{n+1}(L,R)\Big) =$$
$$= D_n\Bigg(E^L_n(L,R), \Big(E^R_n(L,R) \oplus f(E^L_n(L,R), K_{n+1})\Big) \oplus$$
$$\oplus f(E^L_n(L,R), K_{n+1})\Bigg)$$

o sea,

$$D^L_{n+1}\Big(E^L_{n+1}(L,R), E^R_{n+1}(L,R)\Big) =$$
$$= D^L_n\Big(E^L_n(L,R), E^R_n(L,R)\Big) = L$$

$$D^L_{n+1}\Big(E^R_{n+1}(L,R), E^R_{n+1}(L,R)\Big) =$$
$$= D^R_n\Big(E^L_n(L,R), E^R_n(L,R)\Big) = R$$

con lo que finaliza nuestra demostración.

10.1.2. Cifrados con estructura de grupo

Otra de las cuestiones a tener en cuenta en los cifrados de producto es la posibilidad de que posean estructura de *grupo*. Se dice que un cifrado tiene estructura de grupo si se cumple la siguiente propiedad:

$$\forall\, k_1, k_2, M \quad \exists\, k_3 \quad \text{tal que} \quad E_{k_2}(E_{k_1}(M)) = E_{k_3}(M) \tag{10.2}$$

esto es, si hacemos dos cifrados encadenados con k_1 y k_2, existe una clave k_3 que realiza la transformación equivalente.

Resulta muy conveniente que un cifrado carezca de este tipo de estructura, ya que si ciframos un mensaje primero con la clave k_1 y el resultado con la clave k_2, es como si hubiéramos empleado una clave de longitud doble, aumentando la seguridad del sistema. Si, por el contrario, la transformación criptográfica presentara estructura de grupo, esto hubiera sido equivalente a cifrar el mensaje una única vez con una tercera clave, con lo que no habríamos ganado nada.

10.1.3. S-Cajas

Hemos dicho antes que para poder construir buenos cifrados de producto, intercalaremos sustituciones sencillas (confusión), empleando tablas relativamente pequeñas y compactas, y permutaciones (difusión). Estas tablas de sustitución se denominan de forma genérica S-Cajas.

Una s-caja de $m \times n$ bits (ver figura 10.2) es una tabla de sustitución que toma como entrada cadenas de m bits y da como salida cadenas de n bits. DES, por ejemplo, emplea ocho S-Cajas de 6×4 bits. La utilización de las s-cajas es sencilla: se divide el bloque original en trozos de m bits y cada uno de ellos se sustituye por otro de n bits, haciendo uso de la S-caja correspondiente. Normalmente, cuanto más grandes sean las s-cajas, más resistente será el algoritmo resultante, aunque la

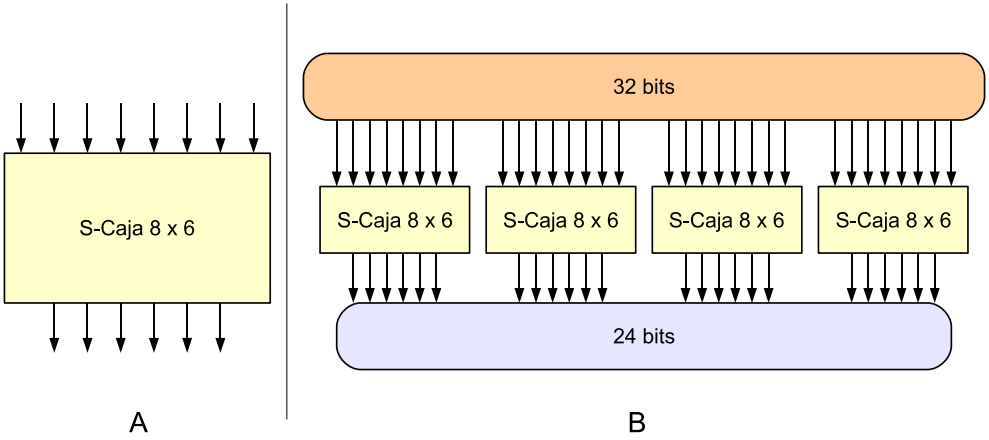

Figura 10.2: **A**: S-Caja individual. **B**: combinación de cuatro S-Cajas.

elección de los valores de salida para que den lugar a cifrado seguro no es en absoluto trivial.

Existe un algoritmo criptográfico, llamado CAST, que emplea seis s-cajas de 8×32 bits. CAST codifica bloques de 64 bits empleando claves de 64 bits, consta de ocho rondas y deposita prácticamente toda su fuerza en las S-Cajas. De hecho, existen muchas variedades de CAST, cada una con sus S-Cajas correspondientes —algunas de ellas secretas—. CAST se ha demostrado resistente a las técnicas habituales de criptoanálisis, y sólo se conoce la fuerza bruta como mecanismo para atacarlo.

10.2. DES

Durante mucho tiempo fue el algoritmo simétrico más extendido mundialmente. Se basa en LUCIFER, desarrollado por IBM a principios de los setenta, y fue adoptado como estándar por el Gobierno de

los EE.UU. para comunicaciones no clasificadas en 1976. Al parecer la NSA lo diseñó para ser implementado por *hardware*, creyendo que los detalles iban a ser mantenidos en secreto, pero la Oficina Nacional de Estandarización publicó su especificación con suficiente detalle como para que cualquiera pudiera implementarlo por *software*. No fue casualidad que el siguiente algoritmo adoptado (*Skipjack*) fuera mantenido en secreto.

A mediados de 1998, se demostró que un ataque por la fuerza bruta a DES era viable, debido a la escasa longitud que emplea en su clave. No obstante, el algoritmo aún no ha demostrado ninguna debilidad grave desde el punto de vista teórico, por lo que su estudio sigue siendo plenamente interesante.

DES codifica bloques de 64 bits empleando claves de 56 bits. Es una Red de Feistel de 16 rondas, más dos permutaciones, una que se aplica al principio (P_i) y otra que se aplica al final (P_f), tales que $P_i = P_f^{-1}$.

La función f (figura 10.3) se compone de una permutación de expansión (E), que convierte el bloque de 32 bits correspondiente en uno de 48. Después realiza un *or-exclusivo* con el valor K_i, también de 48 bits, aplica ocho S-Cajas de 6×4 bits, y efectúa una nueva permutación P.

Se calcula un total de 16 valores de K_i (figura 10.4), uno para cada ronda, efectuando primero una permutación inicial `EP1` sobre la clave de 64 bits, llevando a cabo desplazamientos a la izquierda de cada una de las dos mitades —de 28 bits— resultantes, y realizando finalmente una elección permutada (`EP2`) de 48 bits en cada ronda, que será la K_i. Los desplazamientos a la izquierda son de dos bits, salvo para las rondas 1, 2, 9 y 16, en las que se desplaza sólo un bit. Nótese que aunque la clave para el algoritmo DES tiene en principio 64 bits, se ignoran ocho de ellos —un bit de paridad por cada byte de la clave—, por lo que en la práctica se usan sólo 56 bits.

Para descifrar basta con usar el mismo algoritmo (ya que $P_i = P_f^{-1}$) empleando las K_i en orden inverso.

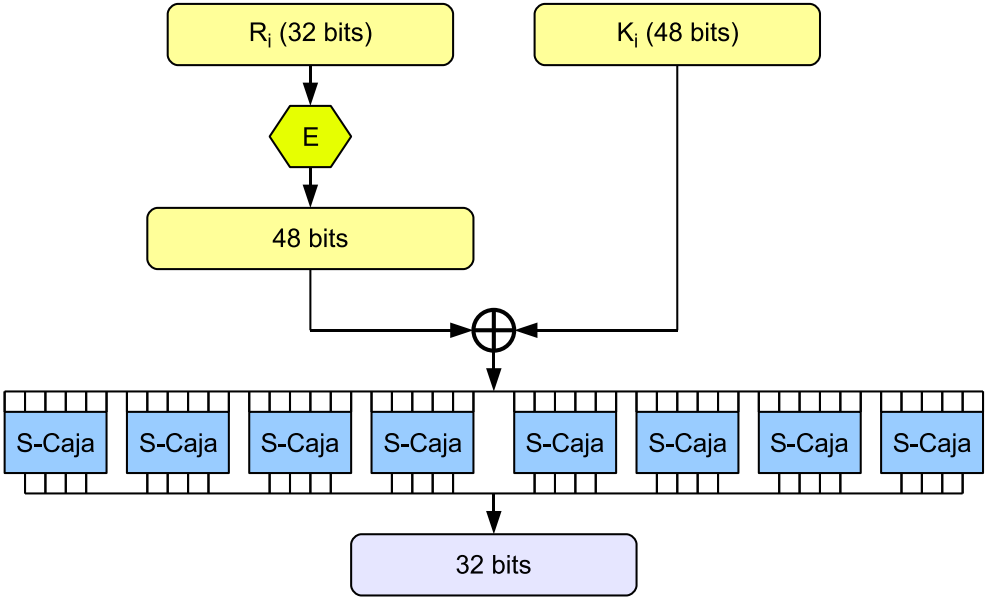

Figura 10.3: Esquema de la función f del algoritmo DES.

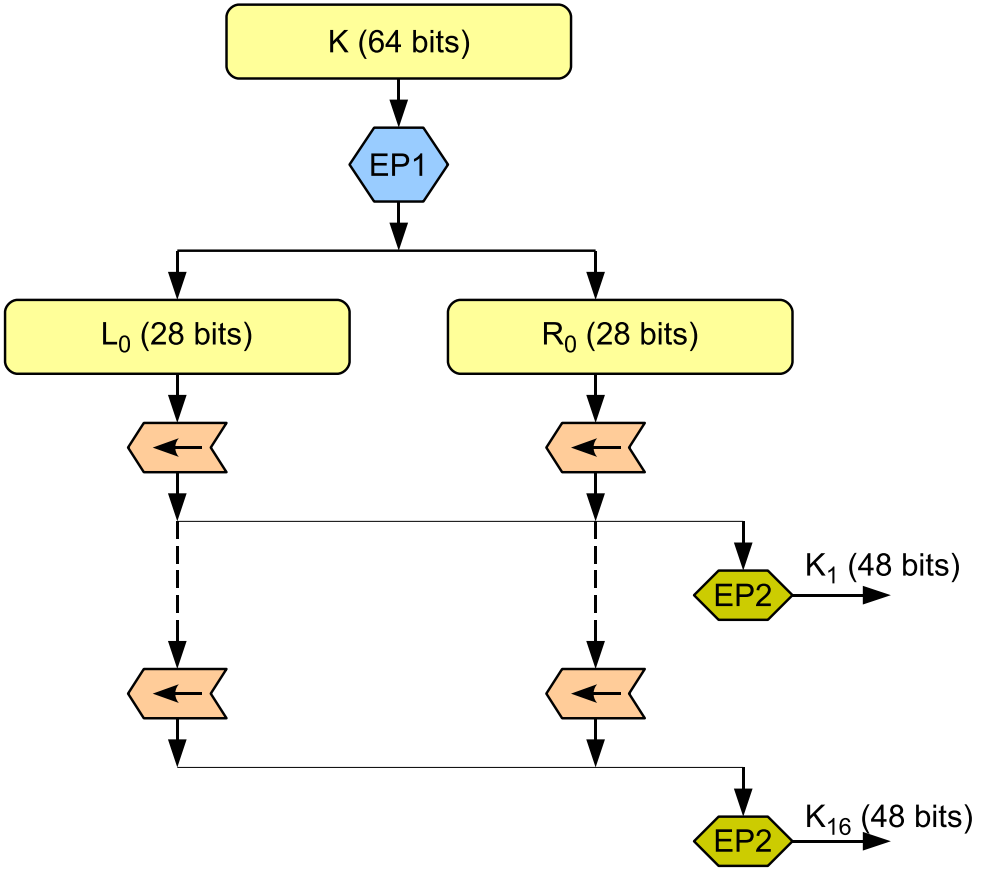

Figura 10.4: Cálculo de las K_i para el algoritmo DES. `EP1` representa la primera elección permutada, que sólo conserva 56 bits de los 64 de entrada, y `EP2` representa la segunda, que se queda con 48 bits.

Clave	Clave tras aplicar `EP1`
`0101010101010101`	`0000000 0000000`
`1F1F1F1F0E0E0E0E`	`0000000 FFFFFFF`
`E0E0E0E0F1F1F1F1`	`FFFFFFF 0000000`
`FEFEFEFEFEFEFEFE`	`FFFFFFF FFFFFFF`

Cuadro 10.1: Claves débiles para el algoritmo DES (64 bits), expresadas en hexadecimal.

10.2.1. Claves débiles en DES

El algoritmo DES presenta algunas claves débiles. En general, todos aquellos valores de la llave que conducen a una secuencia inadecuada de K_i serán poco recomendables. Distinguiremos entre claves *débiles* (cuadro 10.1), que son aquellas que generan un conjunto de dieciséis valores iguales de K_i —y que cumplen $E_k(E_k(M)) = M$—, y claves *semidébiles* (cuadro 10.2), que generan dos valores diferentes de K_i, cada uno de los cuales aparece ocho veces. En cualquier caso, el número de llaves de este tipo es tan pequeño en comparación con el número total de posibles claves, que no debe suponer un motivo de preocupación.

10.3. Variantes de DES

A mediados de julio de 1998 la EFF (Electronic Frontier Foundation), una organización sin ánimo de lucro, logró fabricar una máquina capaz de descifrar un mensaje DES en menos de tres días. Curiosamente, pocas semanas antes, un alto cargo del Departamento de Justicia de los EE.UU había declarado que dicho algoritmo seguía siendo seguro, y que descifrar un mensaje resultaba aún excesivamente costoso, incluso para organizaciones gubernamentales. *DES-Cracker* costó menos de 250.000 euros.

A pesar de su *caída*, DES siguió siendo ampliamente utilizado en

Clave	Clave tras aplicar `EP1`
`01FE01FE01FE01FE`	`AAAAAAA AAAAAAA`
`FE01FE01FE01FE01`	`5555555 5555555`
`1FE01FE00EF10EF1`	`AAAAAAA 5555555`
`E01FE01FF10EF10E`	`5555555 AAAAAAA`
`01E001E001F101F1`	`AAAAAAA 0000000`
`E001E001F101F101`	`5555555 0000000`
`1FFE1FFE0EFE0EFE`	`AAAAAAA FFFFFFF`
`FE1FFE1FFE0EFE0E`	`5555555 FFFFFFF`
`011F011F010E010E`	`0000000 AAAAAAA`
`1F011F010E010E01`	`0000000 5555555`
`E0FEE0FEF1FEF1FE`	`FFFFFFF AAAAAAA`
`FEE0FEE0FEF1FEF1`	`FFFFFFF 5555555`

Cuadro 10.2: Claves semi-débiles para el algoritmo DES (64 bits), expresadas en hexadecimal.

multitud de aplicaciones durante años, como por ejemplo las transacciones de los cajeros automáticos. De todas formas, el problema real de DES no radica en su diseño, sino en que emplea una clave demasiado corta (56 bits), lo cual hace que con el las computadoras actuales los ataques por la fuerza bruta sean una opción realista. Mucha gente se resistió a abandonar DES, precisamente porque fue capaz de *sobrevivir* durante veinte años sin mostrar ninguna debilidad en su diseño, y se alargó su vida útil proponiendo variantes que, de un lado evitaban el riesgo de tener que confiar en algoritmos nuevos, y de otro permitían aprovechar gran parte de las implementaciones por hardware existentes de DES.

10.3.1. DES múltiple

Consiste en aplicar varias veces el algoritmo DES con diferentes claves al mensaje original. Este método es genérico, y se puede apli-

car a cualquier cifrado que no presente estructura de *grupo* (ecuación 10.2). El más común de todos ellos es el Triple-DES, que responde a la siguiente estructura:

$$C = E_{k_1}(E_{k_2}^{-1}(E_{k_1}(M)))$$

es decir, codificamos con la subclave k_1, decodificamos con k_2 y volvemos a codificar con k_1. La clave resultante es la concatenación de k_1 y k_2, con una longitud de 112 bits.

10.3.2. DES con subclaves independientes

Consiste en emplear subclaves diferentes para cada una de las 16 rondas de DES. Puesto que estas subclaves son de 48 bits, la clave resultante tendría 768 bits en total. No es nuestro objetivo entrar en detalles, pero empleando criptoanálisis diferencial, esta variante podría ser rota con 2^{61} textos claros escogidos, por lo que en la práctica no presenta un avance sustancial sobre DES estándar.

10.3.3. DES generalizado

Esta variante emplea n trozos de 32 bits en cada ronda en lugar de dos, por lo que aumentamos tanto la longitud de la clave como el tamaño de mensaje que se puede codificar, manteniendo sin embargo el orden de complejidad del algoritmo. Se ha demostrado sin embargo que no sólo se gana poco en seguridad, sino que en muchos casos incluso se pierde.

10.3.4. DES con S-Cajas alternativas

Consiste en utilizar S-Cajas diferentes a las de la versión original de DES. En la práctica no se han encontrado S-Cajas mejores que las pro-

pias de DES. De hecho, algunos estudios han revelado que las S-Cajas originales presentan propiedades que las hacen resistentes a técnicas de criptoanálisis que no fueron conocidas fuera de la NSA hasta muchos años después de la aparición del algoritmo.

10.4. IDEA

El algoritmo IDEA (International Data Encryption Algorithm) es bastante más joven que DES, pues data de 1992. Trabaja con bloques de 64 bits de longitud y emplea una clave de 128 bits. Como en el caso de DES, se usa el mismo procedimiento tanto para cifrar como para descifrar, aunque en lugar de cambiar simplemente el orden de las subclaves, lo hace empleando otras distintas.

Aunque IDEA ya ha sido sustituido por otros cifrados más versátiles y con menos limitaciones de uso, ya que estaba patentado, es un algoritmo bastante seguro, y hasta ahora se ha mostrado resistente a multitud de ataques, entre ellos el criptoanálisis diferencial. No presenta claves débiles[2].

Como ocurre con todos los algoritmos simétricos de cifrado por bloques, IDEA se basa en los conceptos de confusión y difusión, haciendo uso de las siguientes operaciones elementales (todas ellas fáciles de implementar):

- XOR.
- Suma módulo 2^{16}.
- Producto módulo $2^{16} + 1$.

El algoritmo IDEA consta de ocho rondas. Dividiremos el bloque X a codificar, de 64 bits, en cuatro partes X_1, X_2, X_3 y X_4 de 16 bits.

[2]En realidad, IDEA tiene un pequeñísimo subconjunto de claves que pueden dar ciertas ventajas a un criptoanalista, pero la probabilidad de encontrarnos con una de ellas es de 1 entre 2^{96}, por lo que no representan un peligro real.

Para la interpretación entera de dichos registros se empleará el criterio *big endian*, lo cual significa que el primer byte es el más significativo. Denominaremos Z_i a cada una de las 52 subclaves de 16 bits que vamos a necesitar. Las operaciones que llevaremos a cabo en cada ronda son las siguientes:

1. Multiplicar X_1 por Z_1.
2. Sumar X_2 con Z_2.
3. Sumar X_3 con Z_3.
4. Multiplicar X_4 por Z_4.
5. Hacer un XOR entre los resultados del paso 1 y el paso 3.
6. Hacer un XOR entre los resultados del paso 2 y el paso 4.
7. Multiplicar el resultado del paso 5 por Z_5.
8. Sumar los resultados de los pasos 6 y 7.
9. Multiplicar el resultado del paso 8 por Z_6.
10. Sumar los resultados de los pasos 7 y 9.
11. Hacer un XOR entre los resultados de los pasos 1 y 9.
12. Hacer un XOR entre los resultados de los pasos 3 y 9.
13. Hacer un XOR entre los resultados de los pasos 2 y 10.
14. Hacer un XOR entre los resultados de los pasos 4 y 10.

La salida de cada iteración serán los cuatro sub-bloques obtenidos en los pasos 11, 12, 13 y 14, que serán la entrada del siguiente ciclo, en el que emplearemos las siguientes seis subclaves, hasta un total de 48. Al final de todo intercambiaremos los dos bloques centrales (en

realidad con eso *deshacemos* el intercambio que llevamos a cabo en los pasos 12 y 13).

Después de la octava iteración, se realiza la siguiente transformación:

1. Multiplicar X_1 por Z_{49}.
2. Sumar X_2 con Z_{50}.
3. Sumar X_3 con Z_{51}.
4. Multiplicar X_4 por Z_{52}.

Las primeras ocho subclaves se calculan dividiendo la clave de entrada en bloques de 16 bits. Las siguientes ocho se calculan rotando la clave de entrada 25 bits a la izquierda y volviendo a dividirla, y así sucesivamente.

Las subclaves necesarias para descifrar se obtienen cambiando de orden las Z_i y calculando sus inversas para la suma o la multiplicación, según el cuadro 10.3. Puesto que $2^{16} + 1$ es un número primo, nunca podremos obtener cero como producto de dos números, por lo que no necesitamos representar dicho valor. Cuando estemos calculando productos, utilizaremos el cero para expresar el número 2^{16} —un uno seguido de 16 ceros—. Esta representación es coherente puesto que los registros que se emplean internamente en el algoritmo poseen únicamente 16 bits.

10.5. El algoritmo Rijndael (AES)

En octubre de 2000 el NIST (*National Institute for Standards and Technology*) anunciaba oficialmente la adopción del algoritmo *Rijndael* (pronunciado más o menos como *reindal*[3]) como nuevo *Estándar Avanzado*

[3]Gracias a Sven Magnus por la aclaración.

Ronda	Subclaves de Cifrado						Subclaves de Descifrado					
1	Z_1	Z_2	Z_3	Z_4	Z_5	Z_6	Z_{49}^{-1}	$-Z_{50}$	$-Z_{51}$	Z_{52}^{-1}	Z_{47}	Z_{48}
2	Z_7	Z_8	Z_9	Z_{10}	Z_{11}	Z_{12}	Z_{43}^{-1}	$-Z_{45}$	$-Z_{44}$	Z_{46}^{-1}	Z_{41}	Z_{42}
3	Z_{13}	Z_{14}	Z_{15}	Z_{16}	Z_{17}	Z_{18}	Z_{37}^{-1}	$-Z_{39}$	$-Z_{38}$	Z_{40}^{-1}	Z_{35}	Z_{36}
4	Z_{19}	Z_{20}	Z_{21}	Z_{22}	Z_{23}	Z_{24}	Z_{31}^{-1}	$-Z_{33}$	$-Z_{32}$	Z_{34}^{-1}	Z_{29}	Z_{30}
5	Z_{25}	Z_{26}	Z_{27}	Z_{28}	Z_{29}	Z_{30}	Z_{25}^{-1}	$-Z_{27}$	$-Z_{26}$	Z_{28}^{-1}	Z_{23}	Z_{24}
6	Z_{31}	Z_{32}	Z_{33}	Z_{34}	Z_{35}	Z_{36}	Z_{19}^{-1}	$-Z_{21}$	$-Z_{20}$	Z_{22}^{-1}	Z_{17}	Z_{18}
7	Z_{37}	Z_{38}	Z_{39}	Z_{40}	Z_{41}	Z_{42}	Z_{13}^{-1}	$-Z_{15}$	$-Z_{14}$	Z_{16}^{-1}	Z_{11}	Z_{12}
8	Z_{43}	Z_{44}	Z_{45}	Z_{46}	Z_{47}	Z_{48}	Z_7^{-1}	$-Z_9$	$-Z_8$	Z_{10}^{-1}	Z_5	Z_6
Final	Z_{49}	Z_{50}	Z_{51}	Z_{52}			Z_1^{-1}	$-Z_2$	$-Z_3$	Z_4^{-1}		

Cuadro 10.3: Subclaves empleadas en el algoritmo IDEA

de Cifrado (AES) para su empleo en aplicaciones criptográficas no militares, culminando así un proceso de más de tres años, encaminado a proporcionar a la comunidad internacional un nuevo algoritmo de cifrado potente, eficiente, y fácil de implementar. DES tenía por fin un sucesor.

La palabra *Rijndael* —en adelante, para referirnos a este algoritmo, emplearemos la denominación AES— es un acrónimo formado por los nombres de sus dos autores, los belgas Joan Daemen y Vincent Rijmen. Su interés radica en que todo el proceso de selección, revisión y estudio tanto de este algoritmo como de los restantes candidatos, se efectuó de forma pública y abierta, por lo que, prácticamente por primera vez, toda la comunidad criptográfica mundial había participado en su análisis, lo cual convirtió a Rijndael en un cifrado perfectamente digno de la confianza de todos, y animó a la comunidad a adoptar este método para proponer y escoger nuevos métodos de cifrado.

AES es un sistema de cifrado por bloques, diseñado para manejar longitudes de clave y de bloque variables, ambas comprendidas entre los 128 y los 256 bits. Realiza varias de sus operaciones internas a nivel de byte, interpretando éstos como elementos de un cuerpo de Galois $GF(2^8)$ (ver sección 5.8.1). El resto de operaciones se efectúan en términos de registros de 32 bits. Sin embargo, en algunos casos, una secuencia de 32 bits se toma como un polinomio de grado inferior a 4, cuyos coeficientes son a su vez polinomios en $GF(2^8)$.

Si bien, como ya se ha dicho, este algoritmo soporta diferentes tamaños de bloque y clave, en el estándar adoptado por el Gobierno Estadounidense en noviembre de 2001 (FIPS PUB 197), se especificaba una longitud fija de bloque de 128 bits ($N_b = 4$, como se verá más adelante), y la longitud de clave a escoger entre 128, 192 y 256 bits.

10.5.1. Estructura de AES

AES *no* posee estructura de red de Feistel, lo cual era una característica innovadora en su día. En su lugar se ha definido cada ronda como una composición de cuatro funciones invertibles diferentes, formando tres *capas*, diseñadas para proporcionar resistencia frente a criptoanálisis lineal y diferencial (sección 10.7). Cada una de las funciones tiene un propósito preciso:

- La *capa de mezcla lineal* —funciones *DesplazarFila* y *MezclarColumnas*— permite obtener un alto nivel de difusión a lo largo de varias rondas.
- La *capa no lineal* —función *ByteSub*— consiste en la aplicación paralela de s-cajas con propiedades óptimas de no linealidad.
- La *capa de adición de clave* es un simple *or-exclusivo* entre el estado intermedio y la subclave correspondiente a cada ronda.

10.5.2. Elementos de AES

AES es un algoritmo que se basa en aplicar un número determinado de rondas a un valor intermedio que se denomina *estado*. Dicho estado puede representarse mediante una matriz rectangular de bytes, que posee cuatro filas, y N_b columnas. Así, por ejemplo, si nuestro bloque tiene 160 bits (cuadro 10.4), N_b será igual a 5.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$

Cuadro 10.4: Ejemplo de matriz de estado con N_b=5 (160 bits).

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Cuadro 10.5: Ejemplo de clave con N_k=4 (128 bits).

La clave tiene una estructura análoga a la del estado, y se representará mediante una tabla con cuatro filas y N_k columnas. Si nuestra clave tiene, por ejemplo, 128 bits, N_k será igual a 4 (cuadro 10.5).

En algunos casos, tanto el estado como la clave se consideran como vectores de registros de 32 bits, estando cada registro constituido por los bytes de la columna correspondiente, ordenados de arriba a abajo.

El bloque que se pretende cifrar o descifrar se traslada directamente byte a byte sobre la matriz de estado, siguiendo la secuencia $a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0}, a_{0,1} \ldots$, y análogamente, los bytes de la clave se copian sobre la matriz de clave en el mismo orden, a saber, $k_{0,0}, k_{1,0}, k_{2,0}, k_{3,0}, k_{0,1} \ldots$.

Siendo B el bloque que queremos cifrar, y S la matriz de estado, el algoritmo AES con n rondas queda como sigue:

1. Calcular $K_0, K_1, \ldots K_n$ subclaves a partir de la clave K.
2. $S \longleftarrow B \oplus K_0$
3. Para $i = 1$ hasta n hacer

	$N_b = 4$ (128bits)	$N_b = 6$ (192bits)	$N_b = 8$ (256bits)
$N_k = 4$ (128bits)	10	12	14
$N_k = 6$ (192bits)	12	12	14
$N_k = 8$ (256bits)	14	14	14

Cuadro 10.6: Número de rondas para AES en función de los tamaños de clave y bloque.

4. Aplicar ronda i–ésima del algoritmo con la subclave K_i.

Puesto que cada ronda es una sucesión de funciones invertibles, el algoritmo de descifrado consistirá en aplicar las inversas de cada una de las funciones en el orden contrario, y utilizar los mismos K_i que en el cifrado, sólo que comenzando por el último.

10.5.3. Las rondas de AES

Puesto que AES permite emplear diferentes longitudes tanto de bloque como de clave, el número de rondas requerido en cada caso es variable. En el cuadro 10.6 se especifica cuántas rondas son necesarias en función de N_b y N_k.

Siendo S la matriz de estado, y K_i la subclave correspondiente a la ronda i–ésima, cada una de las rondas posee la siguiente estructura:

1. $S \longleftarrow \mathit{ByteSub}(S)$
2. $S \longleftarrow \mathit{DesplazarFila}(S)$
3. $S \longleftarrow \mathit{MezclarColumnas}(S)$
4. $S \longleftarrow K_i \oplus S$

La última ronda es igual a las anteriores, pero eliminando el paso 3.

Función *ByteSub*

La transformación *ByteSub* es una sustitución no lineal que se aplica a cada byte de la matriz de estado, mediante una s-caja 8×8 invertible, que se obtiene componiendo dos transformaciones:

1. Cada byte es considerado como un elemento del $GF(2^8)$ que genera el polinomio irreducible $m(x) = x^8+x^4+x^3+x+1$, y sustituido por su inversa multiplicativa. El valor cero queda inalterado.

2. El siguiente paso consiste en aplicar la siguiente transformación afín en $GF(2)$, siendo $x_0, x_1, \ldots, x_7$ los bits del byte correspondiente, e $y_0, y_1, \ldots, y_7$ los del resultado:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

La función inversa de *ByteSub* sería la aplicación de la inversa de la s-caja correspondiente a cada byte de la matriz de estado.

Función *DesplazarFila*

Esta transformación (figura 10.5) consiste en desplazar a la izquierda cíclicamente las filas de la matriz de estado. Cada fila f_i se desplaza un número de posiciones c_i diferente. Mientras que c_0 siempre es igual a cero (esta fila siempre permanece inalterada), el resto de valores viene en función de N_b y se refleja en el cuadro 10.7.

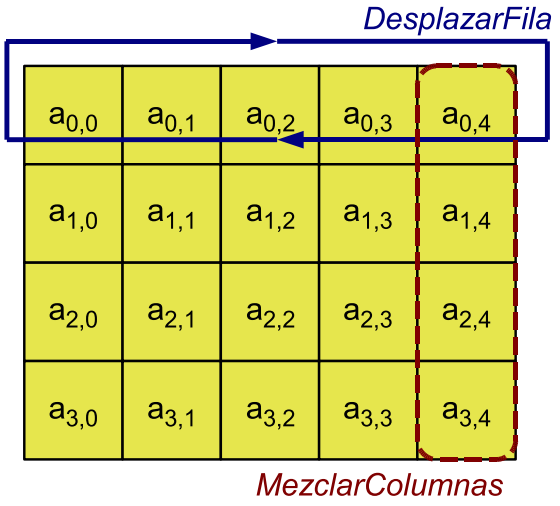

Figura 10.5: Esquema de las funciones *DesplazarFila* y *MezclarColumnas* de AES.

N_b	c_1	c_2	c_3
4	1	2	3
6	1	2	3
8	1	3	4

Cuadro 10.7: Valores de c_i según el tamaño de bloque N_b

La función inversa de *DesplazarFila* será, obviamente, un desplazamiento de las filas de la matriz de estado el mismo número de posiciones que en el cuadro 10.7, pero a la derecha.

Función *MezclarColumnas*

Para aplicar esta función (ver figura 10.5), cada columna del vector de estado se considera un polinomio cuyos coeficientes pertenecen a $GF(2^8)$ —es decir, son también polinomios– y se multiplica módulo $x^4 + 1$ por:

$$c(x) = \mathsf{03}x^3 + \mathsf{01}x^2 + \mathsf{01}x + \mathsf{02}$$

donde 03 es el valor hexadecimal que se obtiene concatenando los coeficientes binarios del polinomio correspondiente en $GF(2^8)$, en este caso 00000011, o sea, $x + 1$, y así sucesivamente.

La inversa de *MezclarColumnas* se obtiene multiplicando cada columna de la matriz de estado por el polinomio:

$$d(x) = \mathsf{0B}x^3 + \mathsf{0D}x^2 + \mathsf{09}x + \mathsf{0E}$$

10.5.4. Cálculo de las subclaves

Las diferentes subclaves K_i se derivan de la clave principal K mediante el uso de dos funciones: una de expansión y otra de selección. Siendo n el número de rondas que se van a aplicar, la función de expansión permite obtener, a partir del valor de K, una secuencia de $4 \cdot (n+1) \cdot N_b$ bytes. La selección simplemente toma consecutivamente de la secuencia obtenida bloques del mismo tamaño que la matriz de estado, y los va asignando a cada K_i.

Sea $K(i)$ un vector de bytes de tamaño $4 \cdot N_k$, conteniendo la clave, y sea $W(i)$ un vector de $N_b \cdot (n + 1)$ registros de 4 bytes, siendo n el

número de rondas. La función de expansión tiene dos versiones, según el valor de N_k:

a) Si $N_k \leq 6$:

1. Para i desde 0 hasta $N_k - 1$ hacer
2. $\quad W(i) \leftarrow (K(4 \cdot i), K(4 \cdot i + 1), K(4 \cdot i + 2), K(4 \cdot i + 3))$
3. Para i desde N_k hasta $N_b \cdot (n + 1)$ hacer
4. $\quad tmp \leftarrow W(i - 1)$
5. $\quad$ Si $i \bmod N_k = 0$
6. $\quad\quad tmp \leftarrow Sub(Rot(tmp)) \oplus Rc(i/N_k)$
7. $\quad W(i) \leftarrow W(i - N_k) \oplus tmp$

b) Si $N_k > 6$:

1. Para i desde 0 hasta $N_k - 1$ hacer
2. $\quad W(i) \leftarrow (K(4 \cdot i), K(4 \cdot i + 1), K(4 \cdot i + 2), K(4 \cdot i + 3))$
3. Para i desde N_k hasta $N_b \cdot (n + 1)$ hacer
4. $\quad tmp \leftarrow W(i - 1)$
5. $\quad$ Si $i \bmod N_k = 0$
6. $\quad\quad tmp \leftarrow Sub(Rot(tmp)) \oplus Rc(i/N_k)$
7. $\quad$ Si $i \bmod N_k = 4$
8. $\quad\quad tmp \leftarrow Sub(tmp)$
9. $\quad W(i) \leftarrow W(i - N_k) \oplus tmp$

En los algoritmos anteriores, la función Sub devuelve el resultado de aplicar la s-caja de AES a cada uno de los bytes del registro de cuatro que se le pasa como parámetro. La función Rot desplaza a la izquierda una posición los bytes del registro, de tal forma que si le pasamos como parámetro el valor (a, b, c, d) nos devuelve (b, c, d, a). Finalmente, $Rc(j)$ es una constante definida de la siguiente forma:

- $Rc(j) = (R(j), 0, 0, 0)$
- Cada $R(i)$ es el elemento de $GF(2^8)$ correspondiente al valor $x^{(i-1)}$, módulo $x^8 + x^4 + x^3 + x + 1$.

10.5.5. Seguridad de AES

Según sus autores, es altamente improbable que existan claves débiles o semidébiles en AES, debido a la estructura de su diseño, que busca eliminar la simetría en las subclaves. También se ha comprobado que es resistente a criptoanálisis tanto lineal como diferencial (ver sección 10.7). En efecto, el método más eficiente conocido hasta la fecha para recuperar la clave a partir de un par texto cifrado–texto claro es la búsqueda exhaustiva, por lo que podemos considerar este algoritmo como uno de los más seguros en la actualidad. Otro hecho que viene a corroborar la fortaleza de AES es que en junio de 2003 fue aprobado por la NSA para cifrar información clasificada como alto secreto.

10.6. Modos de operación para algoritmos de cifrado por bloques

En esta sección comentaremos algunos métodos para aplicar cifrados por bloques a mensajes de gran longitud. Cuando hemos definido estos algoritmos solo nos hemos centrado en cifrar un bloque individual, sin tener en cuenta que pueda formar parte de un mensaje mayor, pero de alguna manera tenemos que generar un mensaje cifrado completo. Esto se puede hacer de diversas formas, desde procesar de forma completamente independiente cada bloque para luego concatenar los resultados, hasta combinar de diversas maneras cada bloque con la entrada o salida de los bloques anteriores o posteriores. El mecanismo concreto que se emplee para hacer todo esto es lo que se conoce

como *modo de operación*, y condiciona las propiedades, desde el punto de vista de eficiencia y la seguridad, que se obtienen finalmente.

10.6.1. Relleno *(padding)*

En general, habrá que subdividir el mensaje original en trozos del mismo tamaño pero, ¿qué ocurre cuando la longitud de la cadena que queremos cifrar no es un múltiplo exacto del tamaño de bloque? En ese caso será necesario añadir información de relleno *(padding)* al final de la misma para que sí lo sea, de forma que el mensaje pueda recuperar su longitud original, una vez descifrado.

Uno de los mecanismos más sencillos, especificado en la norma ANSI X.923, consiste en rellenar con ceros el último bloque que se codifica, salvo el último *byte*, que contiene el número total de *bytes* que se han añadido (ver figura 10.6). Esto tiene el inconveniente de que si el tamaño original es múltiplo del bloque, hay que alargarlo con otro bloque entero. Por ejemplo, si el tamaño de bloque fuera 64 bits (8 *bytes*), y nos sobraran cinco bytes al final, añadiríamos dos ceros y un tres, para completar los ocho *bytes* necesarios en el último bloque. Si por contra no sobrara nada, tendríamos que añadir siete ceros y un ocho.

Otras variantes de este método de relleno son la norma ISO 10126, que rellena con valores aleatorios en lugar de ceros, o PKCS#7, que repite en todo el relleno el *byte* con la longitud del mismo.

10.6.2. Modo ECB

El modo ECB (*Electronic Codebook*) es el método más sencillo y directo de aplicar un algoritmo de cifrado por bloques. Simplemente se subdivide el mensaje en bloques del tamaño adecuado y se cifran todos ellos empleando la misma clave.

A favor de este método podemos decir que permite codificar los bloques independientemente de su orden, lo cual es adecuado para ci-

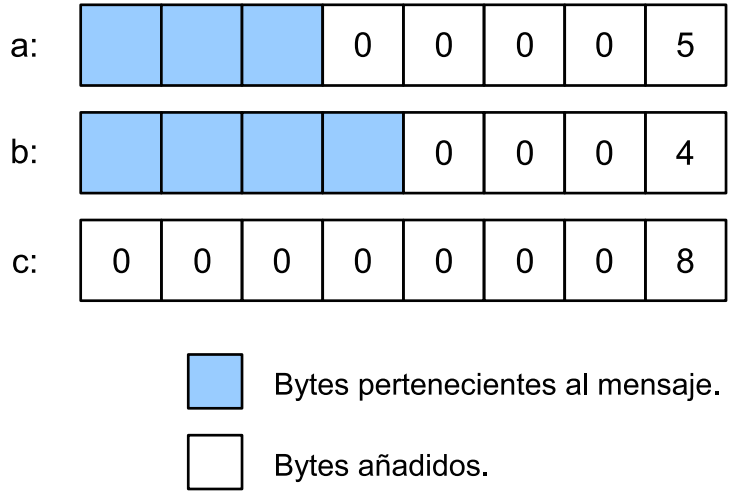

Figura 10.6: Ejemplos de relleno (*padding*) al emplear un algoritmo de cifrado por bloques de 64 bits (8 bytes), en los que el último bloque tiene: a) 3 bytes, b) 4 bytes, c) *8 bytes*. Obsérvese cómo en el caso c) se añade un bloque completo al mensaje.

frar bases de datos o ficheros en los que se requiera un acceso aleatorio. También es resistente a errores, pues si uno de los bloques sufriera una alteración, el resto quedaría intacto.

Por contra, si el mensaje presenta patrones repetitivos, el texto cifrado también los presentará, y eso es peligroso, sobre todo cuando se codifica información muy redundante (como ficheros de texto), o con patrones comunes al inicio y final (como el correo electrónico). Un atacante puede en estos casos efectuar un ataque estadístico y extraer bastante información.

Otro riesgo bastante importante que presenta el modo ECB es el de la *sustitución de bloques*. El atacante puede cambiar un bloque sin mayores problemas, y alterar los mensajes incluso desconociendo la clave y el algoritmo empleados. Simplemente se *escucha* una comunicación de la que se conozca el contenido, como por ejemplo una transacción bancaria a nuestra cuenta corriente. Luego se escuchan otras comunicaciones y se sustituyen los bloques correspondientes al número de cuenta del beneficiario de la transacción por la versión codificada de nuestro número (que ni siquiera nos habremos molestado en descifrar). En cuestión de horas nos habremos hecho ricos.

10.6.3. Modo CBC

El modo CBC (*Cipher Book Chaining Mode*) incorpora un mecanismo de retroalimentación en el cifrado por bloques. Esto significa que el cifrado de bloques anteriores condiciona la codificación del actual, por lo que será imposible sustituir un bloque individual en el mensaje cifrado, ya que esta sustitución afectará a dos bloques en el mensaje descifrado resultante. Esto se consigue efectuando una operación XOR entre el bloque del mensaje que queremos codificar y el último criptograma obtenido (ver figura 10.7). Para cifrar el primer bloque, se emplea el denominado *vector de inicalización* (V.I.), que deberá ser conocido por ambos interlocutores.

Este método evita que un atacante inserte, elimine o reordene blo-

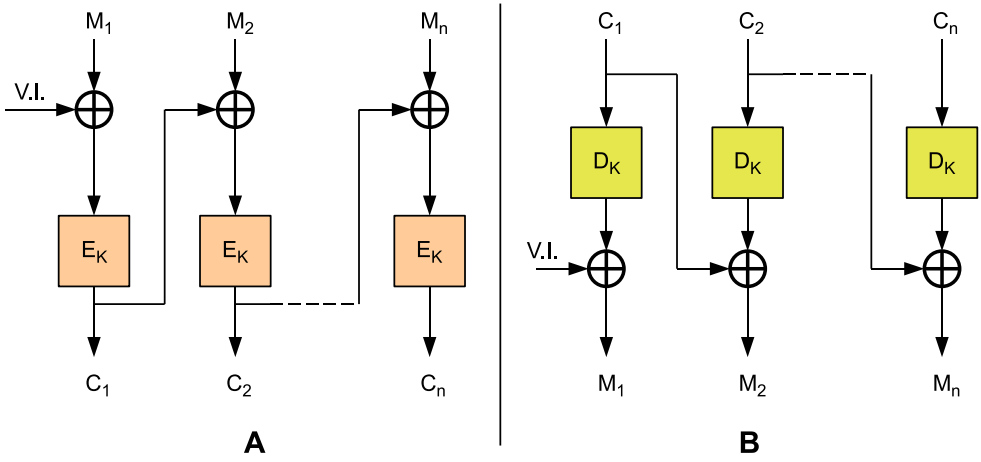

Figura 10.7: Modo de operación CBC. **A**: cifrado, **B**: descifrado. **V.I.**: Vector de Iniciacización.

ques del mensaje cifrado. También puede comprobarse que los textos cifrados correspondientes a dos cadenas que difieran en un único bit, serán idénticos hasta el bloque que contenga ese bit, momento a partir del cual serán totalmente distintos. Esto permitiría a un atacante identificar mensajes con inicios comunes. Para evitar este problema, se puede usar un vector de inicialización diferente para cada mensaje.

10.6.4. Modo CFB

El modo de operación CFB (*Cipher-Feedback*), cada bloque es cifrado y luego combinado, mediante la operación XOR, con el siguiente bloque del mensaje original. Al igual que en el modo CBC, se emplea un vector de inicialización a la hora de codificar el primer bloque. Con este método, si se produce un error en la transmisión, ésta se vuelve a sincronizar de forma automática, a partir del segundo bloque consecutivo que llegue de forma correcta.

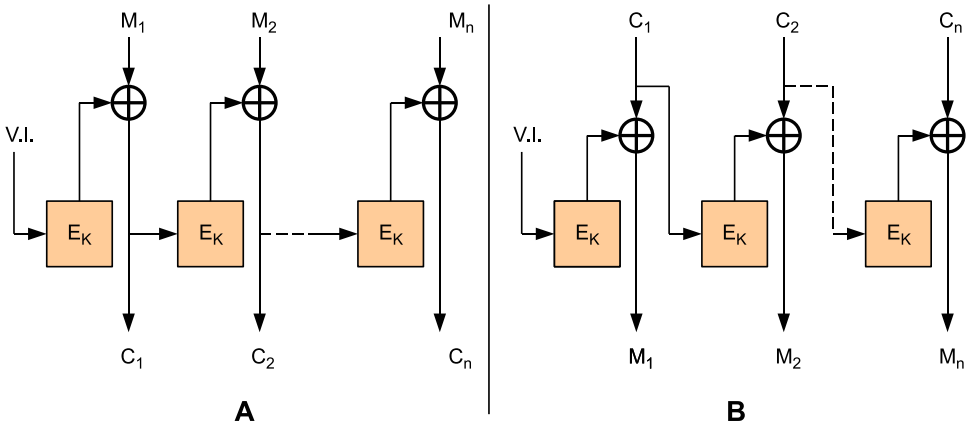

Figura 10.8: Esquema del modo de operación CFB. **A**: cifrado, **B**: descifrado. **V.I.**: Vector de Iniciacización.

Una ventaja importante de este modo de operación radica en el hecho de que la longitud de los bloques del mensaje puede ser menor que la longitud de bloque que acepta el algoritmo de cifrado. Esto ocurre porque el paso final para calcular cada C_i es un or-exclusivo entre el último bloque cifrado y el valor de M_i, operación que puede llevarse a cabo a trozos. El modo de operación CFB es, por tanto, apto para ser usado en situaciones en las que se requiere enviar la información en *paquetes* más pequeños que la longitud de bloque del algoritmo de cifrado subyacente, como por ejemplo cuando se quiere cifrar la comunicación de una computadora con un terminal remoto —teclado y monitor—. También es destacable que, tanto para codificar como a la hora de descifrar, se usa únicamente la operación de cifrado del algoritmo por bloques correspondiente.

10.6.5. Otros modos

Existen otros modos de operación que presentan propiedades adicionales. Algunos, como los modos OFB (sección 11.4.1) y CTR (sección 11.4.2), permiten ser usados como base para los denominados cifrados de flujo, y serán discutidos en el capítulo 11.

Otros modos de operación permiten proporcionar propiedades adicionales al cifrado, como puede ser la autentificación, dando lugar a los denominados *cifrados autentificados* y *cifrados autentificados con datos asociados*, que además de proporcionar confidencialidad, permiten verificar el origen y la integridad de los datos. Algunos de ellos serán comentados en la sección 13.6.

10.7. Criptoanálisis de algoritmos de cifrado por bloques

Se podría decir que el criptoanálisis se comenzó a estudiar seriamente con la aparición de DES. El interés por buscar posibles debilidades en él no solo ha llevado a desarrollar técnicas que posteriormente han tenido éxito con otros algoritmos, sino a descubrir el porqué de algunas de sus inicialmente secretas directrices de diseño, elaboradas con todos los conocimientos que, por entonces, poseía la NSA.

Ni que decir tiene que estos métodos no han conseguido doblegar a DES —de hecho, hoy sabemos que DES fue diseñado para maximizar su resistencia frente a alguno de ellos, casi veinte años antes de ser conocidos por la comunidad científica—, pero sí pueden proporcionar mecanismos significativamente más eficientes que la fuerza bruta. En esta sección veremos algunos de ellos, aplicables a cualquier método de cifrado por bloques, y que son tenidos en cuenta en la actualidad a la hora de diseñar nuevos algoritmos.

10.7.1. Criptoanálisis diferencial

Descubierto por Biham y Shamir en 1990, aunque ya conocido en 1974 por IBM y la NSA, permite efectuar un ataque de texto claro escogido que resulta más eficiente que la fuerza bruta para DES. Para ello se generan pares de mensajes idénticos, que difieren en una serie de bits fijados de antemano. Después se calcula la diferencia entre los dos criptogramas asociados, con la esperanza de detectar patrones estadísticos. Por ejemplo, podemos definir ΔM como la diferencia entre dos textos claros M_1 y M_2, de manera que $\Delta M = M_1 \oplus M_2$. Cifrando entonces M_1 y M_2 obtendríamos la diferencia en bits entre los textos cifrados,

$$\Delta C = E_k(M_1) \oplus E_k(M_2)$$

Denominaremos *diferencial* al par $(\Delta M, \Delta C)$ que acabamos de calcular. Un algoritmo resistente a este análisis debería presentar una distribución aparentemente aleatoria para un conjunto de diferenciales lo suficientemente grande. Si, por el contrario, hay patrones que aparezcan con una frecuencia significativamente mayor que otros, este sesgo podrá explotarse para ganar conocimiento sobre la clave de cifrado k empleada.

Este ataque puede emplearse tanto si se conocen los detalles del algoritmo de cifrado en cuestión como si se carece de ellos. En el primer caso, el análisis puede ser más preciso, ya que puede estudiarse cómo se *propagan* las diferencias a lo largo de las distintas fases del mismo.

10.7.2. Criptoanálisis lineal

El criptoanálisis lineal, descubierto por Mitsuru Matsui en 1992, basa su funcionamiento en tomar algunos bits del texto claro y efectuar una operación XOR entre ellos, tomar algunos del texto cifrado y hacerles lo mismo, y finalmente hacer un XOR de los dos resultados anteriores, obteniendo un único bit. Efectuando esa operación a una gran cantidad de pares de texto claro y criptograma diferentes $(M, E_k(M))$,

el resultado debería tener una apariencia aleatoria, por lo que la probabilidad de obtener un 1 o un 0 debería estar próxima a $1/2$.

Si el algoritmo criptográfico en cuestión es vulnerable a este tipo de ataque, existirán combinaciones de bits que, bien escogidas, den lugar a un sesgo significativo en la medida anteriormente definida, es decir, que el número de ceros (o unos) es apreciablemente superior. Esta propiedad nos va a permitir poder asignar mayor probabilidad a unas claves sobre otras y de esta forma descubrir la clave que buscamos.

10.7.3. Criptoanálisis *imposible*

Propuesta por Eli Biham en 1998, esta estrategia de criptoanálisis permitió atacar con éxito una versión del algoritmo Skipjack, reducida a 31 rondas de las 32 originales. Se basa en buscar diferenciales (ver sección 10.7.1) que, para una clave k dada, nunca puedan darse, o lo que es lo mismo, que resulten imposibles. De esta manera, si desciframos un par de mensajes empleando una clave tentativa k_t y obtenemos uno de esos diferenciales, podremos afirmar que k_t no es la clave que buscamos. Aplicando esta técnica a modo de criba, podemos reducir considerablemente la incertidumbre sobre la clave buscada, lo cual puede reducir el esfuerzo computacional necesario hasta mejorar a la fuerza bruta.

Capítulo 11

Cifrados de flujo

En 1917, J. Mauborgne y G. Vernam inventaron un criptosistema perfecto según el criterio de Shannon (ver sección 3.8). Dicho sistema consistía en emplear una secuencia aleatoria de igual longitud que el mensaje, que se usaría una única vez —lo que se conoce en inglés como *one–time pad*, o cuaderno de un solo uso —, combinándola mediante alguna función simple y reversible —usualmente el *or exclusivo*— con el texto en claro carácter a carácter. Este método presenta el grave inconveniente de que la clave es tan larga como el propio mensaje, y si disponemos de un canal seguro para enviar la clave, ¿por qué no emplearlo para transmitir el mensaje directamente?

Evidentemente, un sistema de Vernam carece de utilidad práctica en la mayoría de los casos, pero supongamos que disponemos de un generador pseudoaleatorio capaz de generar secuencias *criptográficamente aleatorias*, de forma que la longitud de los posibles ciclos sea extremadamente grande. En tal caso podríamos, empleando la semilla del generador como clave, obtener cadenas de bits de *usar y tirar*, y emplearlas para cifrar mensajes simplemente aplicando la función *xor* entre el texto en claro y la secuencia generada. Todo aquel que conozca la semilla podrá reconstruir la secuencia pseudoaleatoria y de esta forma descifrar el mensaje.

En este capítulo analizaremos algunos criptosistemas simétricos que explotan esta idea. Dichos algoritmos no son más que la especificación de un generador pseudoaleatorio, y permiten cifrar mensajes de longitud arbitraria, combinando el mensaje con la secuencia mediante la operación *or exclusivo byte* a *byte*, en lugar de dividirlos en bloques para codificarlos por separado. Como cabría esperar, estos criptosistemas no proporcionan seguridad perfecta, ya que mientras en el cifrado de Vernam el número de posibles claves es tan grande como el de posibles mensajes, cuando empleamos un generador tenemos como mucho tantas secuencias distintas como posibles valores iniciales de la semilla.

Una condición esencial para garantizar la seguridad de un cifrado de flujo es que nunca deben cifrarse dos mensajes diferentes con la misma secuencia. Supongamos que ciframos dos mensajes distintos m_1 y m_2 con una misma clave k, que da lugar a la secuencia o_k. Tendríamos los siguientes criptogramas:

$$\begin{aligned} c_1 &= m_1 \oplus o_k \\ c_2 &= m_2 \oplus o_k \end{aligned} \tag{11.1}$$

Si calculamos $x = c_1 \oplus c_2 = m_1 \oplus m_2$, es fácil comprobar que, si tenemos un fragmento de m_1, basta con calcular su *or exlusivo* con x para obtener el fragmento correspondiente de m_2. Eso permitiría validar posibles *suposiciones* acerca de m_1 y m_2 para un atacante, y deducir los fragmentos de secuencia empleados.

Por esta circunstancia es conveniente incorporar algún tipo de información que nunca se repita en la semilla de la secuencia. Esto es lo que se conoce en Criptografía como *nonce*, y suele formar parte de forma explícita en muchos de los algoritmos de cifrado de flujo.

11.1. Secuencias pseudoaleatorias

Como veíamos en el capítulo 8, los generadores *criptográficamente aleatorios* tienen la propiedad de que, a partir de una porción de la secuencia arbitrariamente grande, resulta computacionalmente intratable el problema de predecir el siguiente bit de la secuencia. Adicionalmente, dijimos que podrían no ser buenos como generadores aleatorios debido a que el conocimiento de la semilla nos permite regenerar la secuencia por completo. Evidentemente, en el caso que nos ocupa, esta característica se convertirá en una ventaja, ya que es precisamente lo que necesitamos: que por un lado no pueda calcularse la secuencia completa a partir de una porción de ésta, y que a la vez pueda reconstruirse completamente conociendo una pieza de información como la semilla del generador.

11.2. Tipos de generadores de secuencia

Los generadores que se emplean como cifrado de flujo pueden dividirse en dos grandes grupos, dependiendo de que se empleen o no fragmentos anteriores del mensaje cifrado a la hora de calcular los valores de la secuencia. Comentaremos brevemente en esta sección sus características básicas.

11.2.1. Generadores síncronos

Un generador *síncrono* es aquel en el que la secuencia es calculada de forma independiente tanto del texto en claro como del texto cifrado. En el caso general, ilustrado en la figura 11.1.a, viene dado por las siguientes ecuaciones:

$$\begin{aligned} s_{i+1} &= g(s_i, k) \\ o_i &= h(s_i, k) \\ c_i &= w(m_i, o_i) \end{aligned} \tag{11.2}$$

Donde k es la clave, s_i es el estado interno del generador, s_0 es el estado inicial, o_i es la salida en el instante i, m_i y c_i son la i–ésima porción del texto claro y cifrado respectivamente, y w es una función reversible, usualmente *or exclusivo*. En muchos casos, la función h depende únicamente de s_i, siendo $k = s_0$.

Cuando empleamos un generador de estas características, necesitamos que tanto el emisor como el receptor estén sincronizados para que el texto pueda descifrarse. Si durante la transmisión se pierde o inserta algún bit, ya no se estará aplicando en el receptor un *xor* con la misma secuencia, por lo que el resto del mensaje será imposible de descifrar. Esto nos obliga a emplear tanto técnicas de verificación como de restablecimiento de la sincronía.

Otro problema muy común con este tipo de técnicas es que si algún bit del criptograma es alterado, la sincronización no se pierde, pero el texto claro se verá modificado en la misma posición. Esta característica podría permitir a un atacante introducir cambios en nuestros mensajes, simplemente conociendo qué bits debe alterar. Para evitar esto, deben emplearse mecanismos de verificación que garanticen la integridad del mensaje recibido, como las funciones resumen (ver capítulo 13).

Existe también una debilidad intrínseca a los métodos de cifrado de flujo basados en generadores síncronos que vale la pena destacar: si un atacante conoce parte del texto claro, podrá sustituirlo por otro sin que lo advierta el legítimo destinatario. Supongamos que m_i es una porción del mensaje original conocida por el atacante, y c_i el trozo de mensaje cifrado correspondiente a él. Sabemos que

$$c_i = w(m_i, o_i)$$

siendo o_i el trozo de secuencia pseudoaleatoria que fue combinado con el texto en claro. Puesto que w es una función reversible, podemos recuperar los o_i asociados al fragmento conocido m_i. Calculamos entonces:

$$c'_i = w(m'_i, o_i)$$

siendo m'_i un mensaje falso de nuestra elección. Seguidamente sustituimos los c_i originales por los c'_i que acabamos de obtener. Cuando el destinatario descifre el mensaje alterado, obtendrá la porción de mensaje m'_i, en lugar del original, de forma totalmente inadvertida. Esta circunstancia aconseja, de nuevo, emplear estos métodos de cifrado en combinación con técnicas que garanticen la integridad del mensaje (ver capítulo 13).

11.2.2. Generadores asíncronos

Un generador de secuencia *asíncrono* o *auto-sincronizado* es aquel en el que la secuencia generada es función de una semilla, más una cantidad fija de los bits anteriores del mensaje cifrado, como puede verse en la figura 11.1.b. Formalmente:

$$\begin{aligned} o_i &= h(k, c_{i-t}, c_{i-t+1}, \ldots, c_{i-1}) \\ c_i &= w(o_i, m_i) \end{aligned} \tag{11.3}$$

Donde k es la clave, m_i y c_i son la i–ésima porción del texto claro y cifrado respectivamente y w es una función reversible. Los valores $c_{-t}, c_{-t+1}, \ldots, c_{-1}$ constituyen el estado inicial del generador.

Esta familia de generadores es resistente a la pérdida o inserción de información, ya que acaba por volver a sincronizarse automáticamente, en cuanto llegan t bloques correctos de forma consecutiva. También será sensible a la alteración de un mensaje, puesto que que si se modifica la unidad de información c_i, el receptor tendrá valores erróneos de

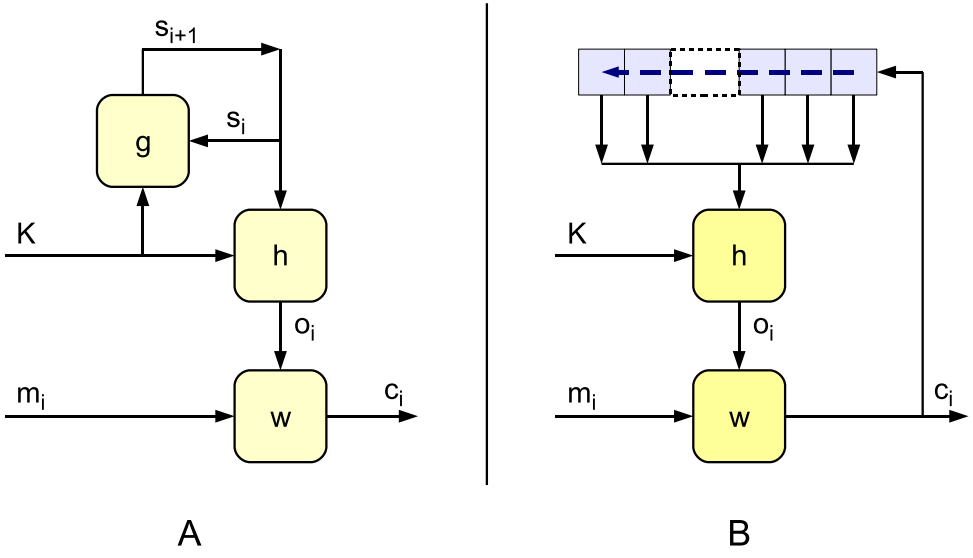

Figura 11.1: Esquema de generadores de secuencia: **A**: generador síncrono. **B**: generador asíncrono.

entrada en su función h hasta que se alcance el bloque c_{i+t}, momento a partir del cual la transmisión habrá recuperado la sincronización. En cualquier caso, al igual que con los generadores síncronos, habrá que introducir mecanismos de verificación.

Se puede construir un generador asíncrono a partir de cifrados por bloques. De hecho, si empleamos como función h un algoritmo de esta familia, obtenemos el modo de operación CFB (sección 10.6.4).

Una propiedad interesante de estos generadores es la dispersión de las propiedades estadísticas del texto claro a lo largo de todo el mensaje cifrado, ya que cada dígito del mensaje influye en todo el criptograma. Esto hace que los generadores asíncronos se consideren en general más resistentes frente a ataques basados en la redundancia del texto en claro.

11.3. Registros de desplazamiento retroalimentados

Los registros de desplazamiento retroalimentados (*feedback shift registers*, o FSR en inglés) son la base de muchos generadores de secuencia síncronos para cifrados de flujo, especialmente aquellos que están basados en *hardware*. Aunque en la actualidad no se aconseja su uso, ya que disponemos de algoritmos mucho más seguros y de gran eficiencia, dedicaremos esta sección a analizar su estructura básica y algunas de sus propiedades.

11.3.1. Registros de desplazamiento retroalimentados lineales

Estos registros, debido a que permiten generar secuencias con períodos muy grandes y con buenas propiedades estadísticas, además de su bien conocida estructura algebraica, se encuentran presentes en muchos de los generadores de secuencia propuestos en la literatura.

Un *registro de desplazamiento retroalimentado lineal* $\mathcal{L}$ es un conjunto de L *variables de estado*, $\{S_0, S_1, \ldots, S_{L-1}\}$, capaces de almacenar un bit cada un (fig 11.2.a). Esta estructura viene controlada por un reloj que coordina los flujos de información entre los estados. Durante cada unidad de tiempo se efectúan las siguientes operaciones:

1. El contenido de S_0 es la salida del registro.
2. El contenido de S_i es desplazado al estado S_{i-1}, para $1 \leq i \leq L-1$.
3. El contenido de S_{L-1} se calcula como la suma módulo 2 de los valores de un subconjunto prefijado de $\mathcal{L}$.

Un generador de estas características devolverá, en función de los

valores iniciales de los estados, y del subconjunto concreto de $\mathcal{L}$ empleado en el paso 3, una secuencia de salidas de carácter periódico —en algunos casos, la secuencia será periódica si ignoramos una cierta cantidad de bits al principio—.

11.3.2. Registros de desplazamiento retroalimentados no lineales

Un *registro de desplazamiento retroalimentado general* (o *no lineal*) $\mathcal{L}$ es un conjunto de L *variables de estado*, $\{S_0, S_1, \ldots, S_{L-1}\}$, capaces de almacenar un bit cada uno (fig 11.2.b). Durante cada unidad de tiempo se efectúan las siguientes operaciones:

1. El contenido de S_0 es la salida del registro.
2. El contenido de S_i es desplazado al estado S_{i-1}, para $1 \leq i \leq L-1$.
3. El contenido de S_{L-1} se calcula como una función booleana

$$f(S_{j-1}, S_{j-2}, \ldots, S_{j-L}),$$

donde S_{j-i} es el contenido del registro S_{L-i} en el estado anterior.

Obsérvese que si sustituimos la función f en un registro de esta naturaleza por la suma módulo 2 de un subconjunto de $\mathcal{L}$, obtenemos un registro de desplazamiento lineal.

11.3.3. Combinación de registros de desplazamiento

En la mayoría de los casos, los registros de desplazamiento retroalimentados no lineales presentan unas mejores condiciones como generadores de secuencia que los generadores de tipo lineal. Sin embargo,

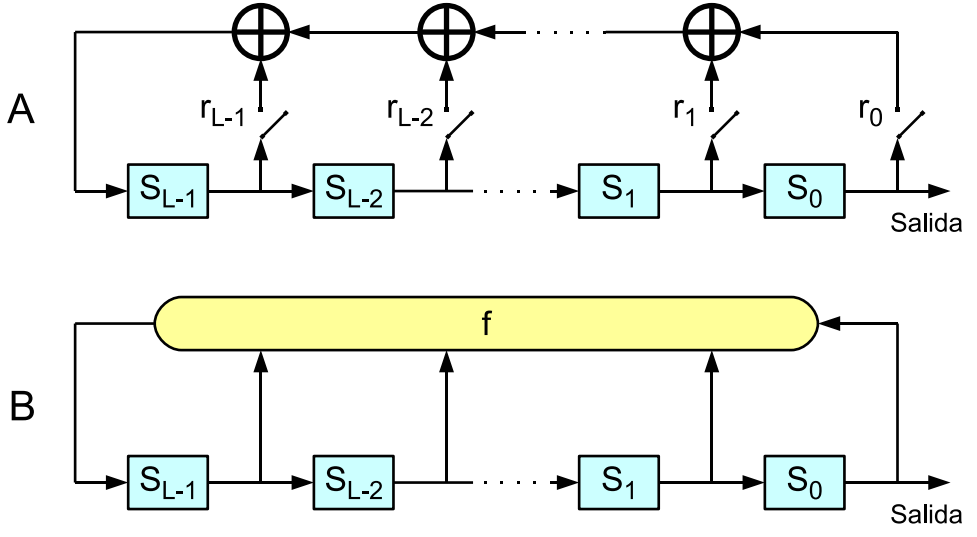

Figura 11.2: Registros de Desplazamiento Retroalimentados: **A**: Registro lineal, en el que cerrando el circuito en los puntos r_0 a r_{L-1} se puede seleccionar qué estados se emplearán para calcular el nuevo valor de S_{L-1}. **B**: Registro no lineal, donde se emplea una función f genérica.

la extrema facilidad de implementación por *hardware* de estos últimos ha llevado a los diseñadores a estudiar diferentes combinaciones de registros lineales, de tal forma que se puedan obtener secuencias mejores.

En general, se emplearían n generadores lineales y una función f no lineal para combinar sus salidas, de tal forma que cada bit de la secuencia se obtendría mediante la expresión

$$f(R_1, R_2, \ldots, R_n) \tag{11.4}$$

siendo R_i la salida del i–ésimo registro de desplazamiento lineal.

11.4. Generadores de secuencia basados en cifrados por bloques

??

Se pueden construir generadores de secuencia a partir de cifrados por bloques definiendo los modos de operación (ver sección 10.6) adecuados. Ya hemos comentado que el modo CFB (sección 10.6.4) produce un generador de secuencia asíncrono. En esta sección describiremos otros modos de operación para algoritmos de cifrado por bloques que dan lugar a generadores de secuencia síncronos.

11.4.1. Modo de operación OFB

Este modo de operación funciona como generador síncrono, ya que produce, de forma totalmente independiente del mensaje, una secuencia pseudoaleatoria basada en una clave. En la figura 11.3 podemos ver cómo a partir de una clave K y de un vector de inicialización (V.I.), estos algoritmos permiten generar una secuencia o_i de bloques perfec-

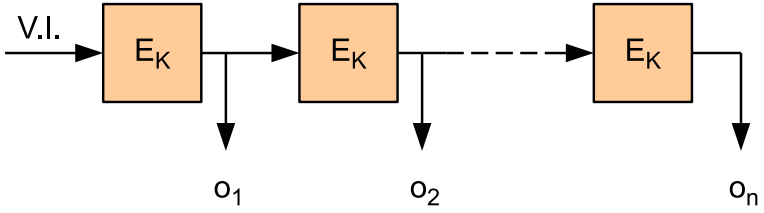

Figura 11.3: Esquema del modo de operación OFB, para emplear algoritmos de cifrado por bloques como generadores de secuencia síncronos para cifrados de flujo.

tamente válida para ser empleada dentro de un esquema de cifrado de flujo.

11.4.2. Modo de operación CTR

Existe otro modo de operación para cifrados por bloques, denominado *modo contador* (o CTR), que también puede ser usado como generador de secuencia. En este caso, se emplea un valor de inicialización (que actúa como *nonce*), y se combina con un contador para generar un bloque que será cifrado, mediante una algoritmo simétrico, con una clave determinada (figura 11.3). El mencionado contador se incrementa y se vuelve a combinar con el *nonce* para generar el siguiente bloque. Esto permite generar una secuencia criptográficamente aleatoria, con la ventaja de que es posible obtener cualquier fragmento de la misma de forma independiente, únicamente proporcionando los valores adecuados del contador.

El contador puede ser incrementado de manera secuencial, o mediante cualquier función que modifique su valor de manera determinista recorriendo todos los valores posibles. Adicionalmente, puede ser combinado con el *nonce* mediante simple concatenación, *or-exclusivo*, etc.

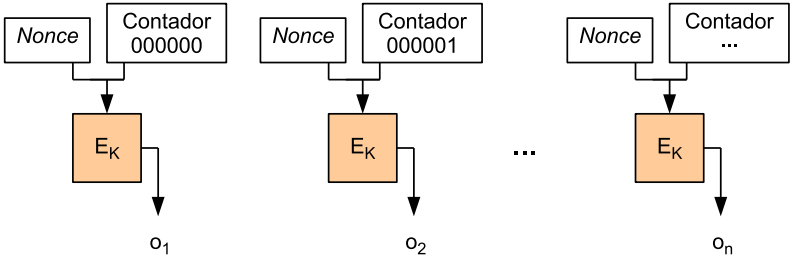

Figura 11.4: Esquema del modo de operación CTR (contador). Con este modo de operación es posible generar secuencias para cifrados de flujo, en bloques independientes.

11.5. Algoritmos de generación de secuencia

Además de emplear como base cifrados de bloques en la construcción de generadores de secuencia, la comunidad ha propuesto diversos algoritmos especialmente diseñados para producir secuencias pseudoaleatorias, válidas para llevar a cabo cifrados de flujo.

11.5.1. Algoritmo RC4

El algoritmo RC4 fue diseñado por Ron Rivest en 1987 para la compañía RSA Data Security. Su implementación es extremadamente sencilla y rápida, y está orientado a generar secuencias en unidades de un *byte*, además de permitir claves de diferentes longitudes. Durante mucho tiempo fue formalmente un algoritmo *propietario*, ya que RSA Data Security nunca lo liberó oficialmente, y no fue hasta 2014 que el propio Rivest confirmó que lo que se conocía desde 1994 como ARC4 *(alleged RC4)*, era efectivamente el algoritmo creado por él décadas atrás. En la actualidad se desaconseja su uso, debido a que se le han encontrado diferentes debilidades, pero lo incluimos aquí debido a su importancia histórica y a su simplicidad.

RC4 consta de una s-caja 8×8, que almacenará una permutación del conjunto $\{0, \ldots, 255\}$. Dos contadores i y j se ponen a cero. Luego, cada *byte* O_r de la secuencia se calcula como sigue:

1. $i = (i + 1) \text{ mód } 256$
2. $j = (j + S_i) \text{ mód } 256$
3. Intercambiar los valores de S_i y S_j
4. $t = (S_i + S_j) \text{ mód } 256$
5. $O_r = S_t$

Para calcular los valores iniciales de la s-caja, se hace lo siguiente:

1. $S_i = i, \quad \forall 0 \leq i \leq 255$
2. Rellenar el *array* K_0 a K_{255} repitiendo la clave tantas veces como sea necesario.
3. $j = 0$
4. Para $i = 0$ hasta 255 hacer:
   $j = (j + S_i + K_i) \text{ mód } 256$
   Intercambiar S_i y S_j.

El algoritmo RC4 genera secuencias en las que los ciclos son bastante grandes, y es inmune a los criptoanálisis diferencial y lineal, si bien algunos estudios indican que puede poseer claves débiles, y que es sensible a estudios analíticos del contenido de la s-caja. De hecho, algunos afirman que en una de cada 256 claves posibles, los *bytes* que se generan tienen una fuerte correlación con un subconjunto de los bytes de la clave, lo cual es un comportamiento muy poco recomendable.

11.5.2. Algoritmo SEAL

SEAL es un generador de secuencia diseñado en 1993 para IBM por Phil Rogaway y Don Coppersmith, cuya estructura está especialmente pensada para funcionar de manera eficiente en computadores con una longitud de palabra de 32 bits. Su funcionamiento se basa en un proceso inicial en el que se calculan los valores para unas tablas a partir de la clave, de forma que el cifrado propiamente dicho puede llevarse

a cabo de una manera realmente rápida. Por desgracia, también es un algoritmo sujeto a patentes.

Una característica muy útil de este algoritmo es que no se basa en un sistema lineal de generación, sino que define una *familia de funciones pseudoaleatorias*, de tal forma que se puede calcular cualquier porción de la secuencia suministrando únicamente un número entero n de 32 bits. La idea es que, dado ese número, junto con la clave k de 160 bits, el algoritmo genera un bloque $k(n)$ de L bits de longitud. De esa forma, cada valor de k da lugar a una secuencia total de $L \cdot 2^{32}$ bits, compuesta por la yuxtaposición de los bloques $k(0), k(1), \ldots, k(2^{32} - 1)$.

SEAL se basa en el empleo del algoritmo SHA (ver sección 13.3.2) para generar las tablas que usa internamente. De hecho, existen dos versiones del algoritmo, la 1.0 y la 2.0, que se diferencian precisamente en que la primera emplea SHA y la segunda su versión revisada, SHA-1.

11.5.3. Algoritmo Salsa20

Salsa20 es un algoritmo de generación de secuencia creado por Daniel J. Bernstein en 2005, que incorpora una serie de características interesantes. Por un lado, emplea únicamente funciones sencillas de llevar a cabo por la ALU de cualquier computadora moderna (suma, rotación y *or-exclusivo* con *palabras* de 32 bits), lo cual le proporciona una gran eficiencia. Por otro lado, incluye de forma explícita un índice dentro de la clave, lo que le permite generar cualquier bloque de la secuencia de manera independiente del resto, a diferencia de otros algoritmos, que necesitan generar toda la secuencia desde el principio para llegar a un punto dado de la misma.

La longitud de clave de este algoritmo es de 256 bits, que se complementan con otros 64 bits que actúan como índice, más 64 bits para un *nonce*, que funciona como identificador de la secuencia, y facilita no reutilizar claves, aunque en la práctica se comporta como parte del propio índice. Como resultado se obtiene la porción de 512 bits de la

secuencia correspondiente al índice proporcionado.

Elementos de Salsa20

El núcleo de Salsa20 es una función *hash* (ver capítulo 13), que transforma una entrada de 512 bits en una salida de la misma longitud, empleando un vector de estado compuesto por 16 *palabras* de 32 bits. Este vector se carga inicialmente con la clave, el *nonce*, el índice, y los 128 bits restantes se rellenan con valores constantes. Estas 16 *palabras* se organizan en una matriz 4x4, de la siguiente manera:

$$\begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} \tag{11.5}$$

La función *cuarto de ronda* toma un vector $y = (y_0, y_1, y_2, y_3)$ de cuatro *palabras* y devuelve otro vector cuartoderonda$(y) = (z_0, z_1, z_2, z_3)$:

$$\begin{aligned} z_1 &= y_1 \oplus ((y_0 + y_3) <<< 7) \\ z_2 &= y_2 \oplus ((z_1 + y_0) <<< 9) \\ z_3 &= y_3 \oplus ((z_2 + z_1) <<< 13) \\ z_0 &= y_0 \oplus ((z_3 + z_2) <<< 18) \end{aligned} \tag{11.6}$$

donde $a <<< b$ representa la operación de desplazar a a la izquierda b bits.

La función *ronda de filas* toma el vector de estado completo $(x_0, x_1, \ldots, x_{15})$ y devuelve otro vector rondafilas$(x) = (z_0, z_1, \ldots, z_{15})$ de la siguiente forma:

$$\begin{aligned}(z_0, z_1, z_2, z_3) &= \text{cuartoderonda}(x_0, x_1, x_2, x_3)\\ (z_5, z_6, z_7, z_4) &= \text{cuartoderonda}(x_5, x_6, x_7, x_4)\\ (z_{10}, z_{11}, z_8, z_9) &= \text{cuartoderonda}(x_{10}, x_{11}, x_8, x_9)\\ (z_{15}, z_{12}, z_{13}, z_{14}) &= \text{cuartoderonda}(x_{15}, x_{12}, x_{13}, x_{14})\end{aligned} \tag{11.7}$$

Análogamente, se define la función rondacolumnas(x):

$$\begin{aligned}(z_0, z_4, z_8, z_{12}) &= \text{cuartoderonda}(x_0, x_4, x_8, x_{12})\\ (z_5, z_9, z_{13}, z_1) &= \text{cuartoderonda}(x_5, x_9, x_{13}, x_4)\\ (z_{10}, z_{14}, z_2, z_6) &= \text{cuartoderonda}(x_{10}, x_{14}, x_2, x_6)\\ (z_{15}, z_3, z_7, z_{11}) &= \text{cuartoderonda}(x_{15}, x_3, x_7, x_{11})\end{aligned} \tag{11.8}$$

Como puede verse, la primera función procesa las filas de la matriz, y la segunda procesa sus columnas. La función *dobleronda* se define como la combinación de ambas:

$$\text{dobleronda}(x) = \text{rondafilas}(\text{rondacolumnas}(x)) \tag{11.9}$$

Finalmente, la función *hash* de Salsa20 queda definida de la siguiente manera:

$$\text{Salsa20}(x) = x + \text{dobleronda}^{10}(x). \tag{11.10}$$

Esta suma se realiza subdividiendo x en 16 fragmentos de 32 bits, y sumándolos uno a uno módulo 32 (interpretándolos según el criterio *little endian)*.

Una vez descritos todos estos elementos, estamos en condiciones de definir la función de generación de secuencia de Salsa20. Siendo k una clave de 256 bits (dividida en dos mitades de 128 bits, k_0 y k_1), y n la concatenación del *nonce* y el índice, de 128 bits, un bloque de la secuencia se genera de la siguiente forma:

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

donde

$$\begin{aligned}
\sigma_0 &= (101, 120, 112, 97), \\
\sigma_1 &= (110, 100, 32, 51), \\
\sigma_2 &= (50, 45, 98, 121), \\
\sigma_3 &= (116, 101, 32, 107).
\end{aligned} \tag{11.11}$$

ChaCha

ChaCha es una variante de Salsa20, en la que la operación realizada dentro de las rondas es ligeramente distinta, lo cual incrementa la difusión de cada ronda, y lo hace algo más eficiente.

Si Salsa20 se basaba en la siguiente operación:

$$b = b \oplus ((a + c) <<< k) \tag{11.12}$$

ChaCha se basa en esta otra:

$$\begin{aligned}
b &= b + c \\
c &= a \oplus b \\
a &= a <<< k
\end{aligned} \tag{11.13}$$

La cantidad de bits que se desplaza en cada operación también es diferente.

Capítulo 12

Cifrados asimétricos

Los algoritmos asimétricos o de clave pública han demostrado su interés para ser empleados en redes de comunicación inseguras (Internet). Introducidos por Whitfield Diffie y Martin Hellman a mediados de los años 70, su novedad fundamental con respecto a la criptografía simétrica es que las claves no son únicas, sino que forman pares. Hasta la fecha han aparecido diversos algoritmos asimétricos, la mayoría de los cuales son inseguros; otros son poco prácticos, bien sea porque el criptograma es considerablemente mayor que el mensaje original, bien sea porque la longitud de la clave es enorme. Se basan en general en plantear al atacante problemas matemáticos difíciles de resolver (ver capítulo 5). En la práctica muy pocos algoritmos son realmente útiles. El más popular por su sencillez es RSA, que ha sobrevivido a multitud de intentos de rotura. Otros algoritmos son los de ElGamal y Rabin.

La criptografía asimétrica basada en aritmética modular emplea generalmente longitudes de clave mucho mayores que la simétrica. Por ejemplo, mientras que para algoritmos simétricos se empieza a considerar segura una clave a partir de 128 bits, para los asimétricos basados en aritmética modular se recomiendan claves de al menos 2048 bits. Para los cifrados basados en curvas elípticas las longitudes de clave son sensiblemente menores, aunque siguen siendo superiores

Clave simétrica	RSA y Diffie-Hellman	Curva elíptica
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

Cuadro 12.1: Longitudes de clave en bits equivalentes para cifrados simétricos y asimétricos.

que las de los algoritmos simétricos, como puede verse en la tabla 12. Además, la complejidad de cálculo que requieren los cifrados asimétricos los hace considerablemente más lentos que los simétricos. En la práctica los métodos asimétricos se emplean únicamente para cifrar una clave simétrica, única para cada mensaje o transacción particular, o un *hash* en el caso de las firmas digitales.

12.1. Aplicaciones de los algoritmos asimétricos

Los algoritmos asimétricos poseen dos claves diferentes en lugar de una, K_p y K_P, denominadas *clave privada* y *clave pública* respectivamente. Una de ellas se emplea para cifrar, mientras que la otra se usa para descifrar. Dependiendo de la aplicación que le demos al algoritmo, la clave pública será la de cifrado o viceversa. Para que estos criptosistemas sean seguros también ha de cumplirse que a partir de una de las claves resulte extremadamente difícil calcular la otra.

12.1.1. Protección de la confidencialidad

Una de las aplicaciones inmediatas de los algoritmos asimétricos es el cifrado de la información sin tener que transmitir la clave de descifrado, lo cual permite su uso en canales inseguros. Supongamos que A quiere enviar un mensaje a B (figura 12.1). Para ello solicita a B su clave pública K_P. A genera entonces el mensaje cifrado $E_{K_P}(m)$ [1]. Una vez hecho esto únicamente quien posea la clave K_p —en nuestro ejemplo, B— podrá recuperar el mensaje original m.

Nótese que para este tipo de aplicación, la clave que se hace pública es aquella que permite cifrar los mensajes, mientras que la clave privada es aquella que permite descifrarlos.

12.1.2. Autentificación

La segunda aplicación de los algoritmos asimétricos es la autentificación de mensajes, con ayuda de funciones MDC (ver capítulo 13), que nos permiten obtener una *signatura* o resumen a partir de un mensaje. Dicha signatura es mucho más pequeña que el mensaje original, y es muy difícil encontrar otro mensaje diferente que dé lugar al mismo resumen. Supongamos que A recibe un mensaje m de B y quiere comprobar su autenticidad. Para ello B genera un resumen del mensaje $r(m)$ (ver figura 12.2) y lo codifica empleando la clave de cifrado, que en este caso será privada. La clave de descifrado se habrá hecho pública previamente, y debe estar en poder de A. B envía entonces a A el criptograma correspondiente a $r(m)$. A puede ahora generar su propia $r'(m)$ y compararla con el valor $r(m)$ obtenido del criptograma enviado por B. Si coinciden, el mensaje será auténtico, puesto que el único que posee la clave para codificar es precisamente B.

Nótese que en este caso la clave que se emplea para cifrar es la clave privada, justo al revés que para la simple codificación de mensajes.

[1] En realidad, se genera una clave única k, se cifra m usando un algoritmo simétrico, obteniendo $E_k(m)$, y lo que se envía es $E_{K_P}(E_k(m))$.

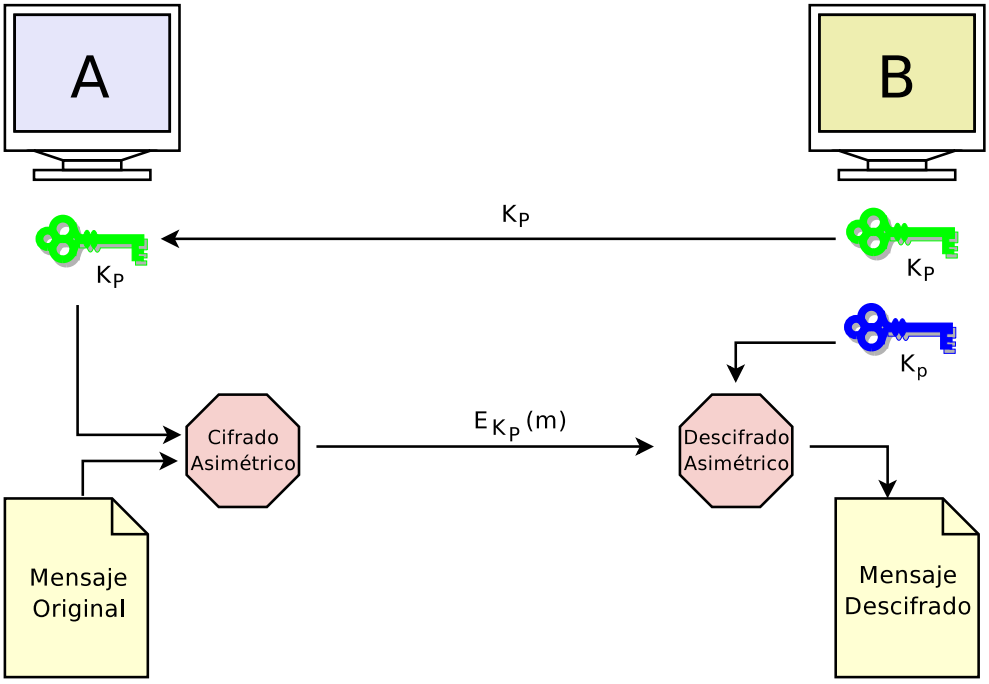

Figura 12.1: Transmisión de información empleando algoritmos asimétricos. 1. **B** envía a **A** su clave pública, K_P; 2. **A** codifica el mensaje y envía a **B** el criptograma $E_{K_P}(m)$; 3. **B** decodifica el criptograma empleando la clave privada K_p.

En muchos algoritmos asimétricos ambas claves sirven tanto para cifrar como para descifrar, de manera que si empleamos una para codificar, la otra permitirá decodificar y viceversa. Esto ocurre con el algoritmo RSA, en el que un único par de claves es suficiente tanto para cifrar información como para autentificarla.

12.2. Ataques de intermediario

El ataque de intermediario, *man in the middle* en inglés, (figura 12.3) puede darse con cualquier algoritmo asimétrico, dando lugar a un grave peligro del que hay que ser consciente, y tratar de evitar a toda costa. Supongamos que A quiere establecer una comunicación con B, y que C quiere espiarla. Cuando A le solicite a B su clave pública K_B, C se interpone, obteniendo la clave de B y enviando a A una clave falsa K_C creada por él. A partir de ese momento puede pasar lo siguiente:

- Cualquier documento firmado digitalmente por C será interpretado por A como procedente de B.
- Si A cifra un mensaje para B, en realidad estará generando un mensaje cifrado para C, que podrá interceptarlo, descifrarlo con su propia clave privada, volverlo a cifrar con la clave K_B correcta, y reenviárselo a B. De esta forma C tendrá acceso a toda la información cifrada que viaje de A hasta B sin que ninguna de sus víctimas advierta el engaño.

La única manera de evitar esto consiste en buscar mecanismos para poder garantizar que la clave pública que recibe A pertenece realmente a B. Para ello la solución más obvia consiste en que K_B esté *firmada digitalmente* por un amigo común, que certifique la autenticidad de la clave. Si A y B carecen de amigos comunes, pueden recurrir a las llamados *redes de confianza*, que permiten certificar la autenticidad de las claves a través de redes sociales, en las que cada usuario está relacionado con unos cuantos y decide en quiénes confía, sin necesidad de

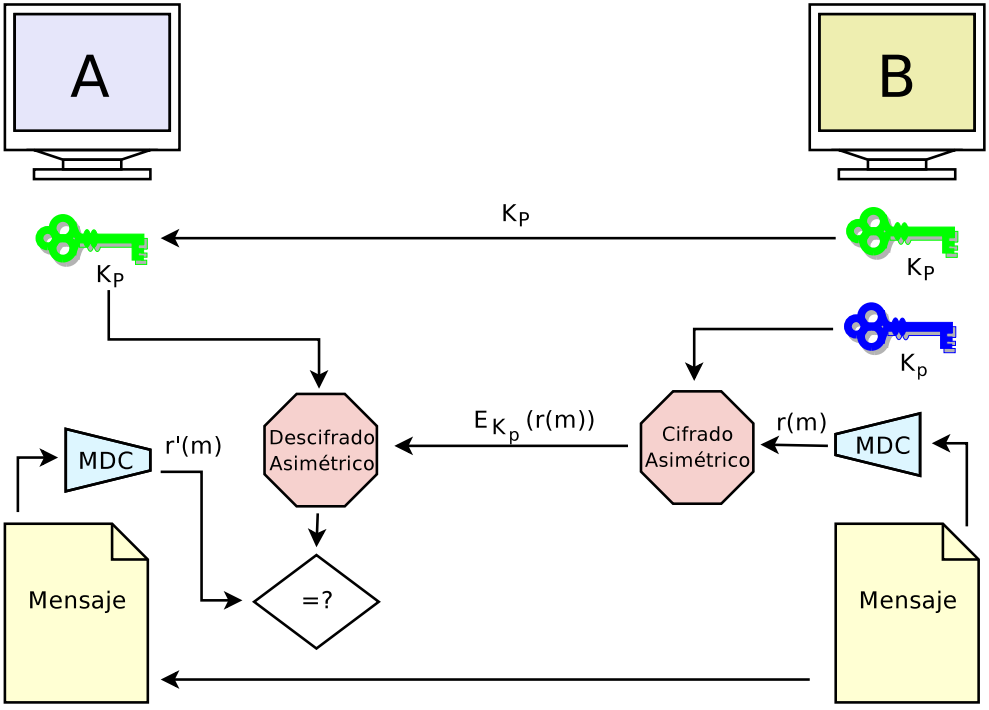

Figura 12.2: Autentificación de información empleando algoritmos asimétricos. 1. **A**, que posee la clave pública K_P de **B**, recibe un mensaje y quiere autentificarlo; 2. **B** genera el resumen $r(m)$ envía a **A** el criptograma asociado $E_{K_p}(r(m))$; 3. **A** genera por su cuenta $r'(m)$ y decodifica el criptograma recibido usando la clave K_P; 4. **A** compara $r(m)$ y $r'(m)$ para comprobar la autenticidad del mensaje m.

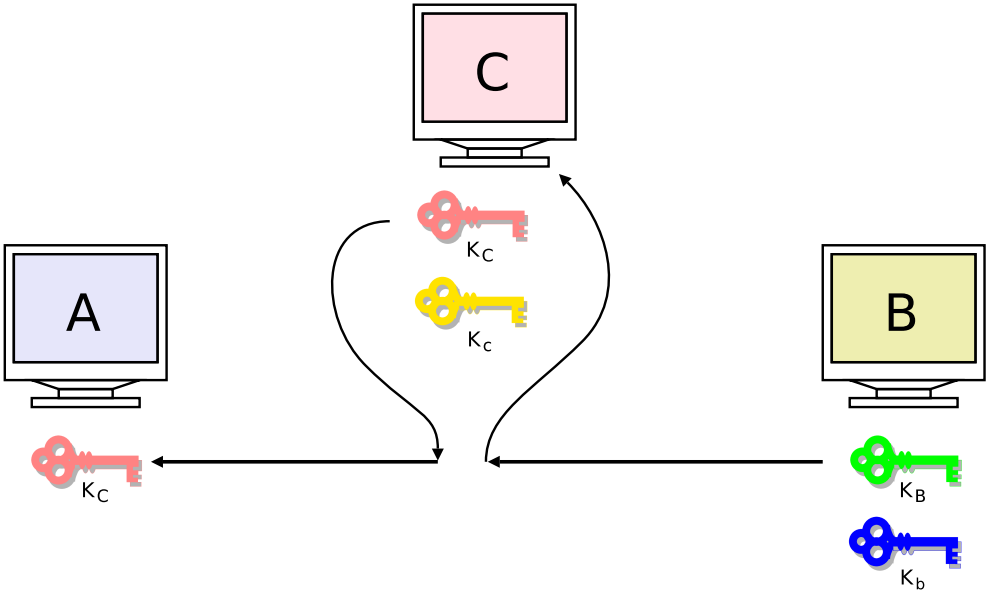

Figura 12.3: Ataque de intermediario para un algoritmo asimétrico.

centralizar el proceso. Por eso se nos suele recomendar, cuando instalamos paquetes de cifrado asimétrico, como por ejemplo PGP (capítulo 18), que firmemos todas las claves sobre las que tengamos certeza de su autenticidad, y únicamente esas.

12.3. Algoritmo RSA

De entre todos los algoritmos asimétricos, quizá RSA sea el más sencillo de comprender e implementar. Como ya se ha dicho, sus claves sirven indistintamente tanto para cifrar como para autentificar. Debe su nombre a sus tres inventores: Ronald Rivest, Adi Shamir y Leonard Adleman, que lo publicaron en 1977. La patente del mismo permaneció vigente hasta el 20 de septiembre de 2000, por lo que su uso comercial estuvo restringido hasta esa fecha. De hecho, las primeras versiones de PGP (ver capítulo 18) lo incorporaban como método de cifrado y firma digital, pero se desaconsejó su uso a partir de la versión 5 en favor de otros algoritmos, que por entonces sí eran totalmente libres. Sujeto a múltiples controversias, desde su nacimiento nadie ha conseguido probar o rebatir su seguridad, pero se le tiene como uno de los algoritmos asimétricos más seguros, al menos desde el punto de vista teórico, ya que suele ser bastante difícil de implementar de manera completamente segura.

RSA se basa en la dificultad para factorizar grandes números. Las claves pública y privada se calculan a partir de un número que se obtiene como producto de dos *primos grandes*. El atacante se enfrentará, si quiere recuperar un texto claro a partir del criptograma y la clave pública, a un problema de factorización (ver sección 5.6) o tendrá que resolver un logaritmo discreto (ver sección 5.4.4).

Para generar un par de claves (K_P, K_p), en primer lugar se eligen aleatoriamente dos números primos grandes, p y q. Después se calcula el producto de ambos $n = pq$, que será el módulo que emplearemos en nuestros cálculos.

Nuestro objetivo será encontrar dos exponentes que, aplicados de forma sucesiva, devuelvan el valor original módulo n. En la sección 5.2.2 vimos que $m^{\phi(n)} \equiv 1 \pmod{n}$, si m y n eran primos relativos, así que necesitaremos dos números, e y d, que sean inversos módulo $\phi(n)$, ya que

$$(m^e)^d = m^{ed} = m^{k\phi(n)+1} \equiv m \pmod{n}.$$

Cuando n es primo, es primo relativo con cualquier m, pero cuando es compuesto la cosa cambia ligeramente. Recordemos buscamos el orden de m, es decir, el mínimo valor de a que verifique

$$m^a \equiv 1 \pmod{n}.$$

Tiene que cumplirse que $m^a \equiv 1 \pmod{p}$ y $m^a \equiv 1 \pmod{q}$, ya que p y q dividen a n. Aplicando el Teorema de Fermat (expresión 5.6), tenemos que a debe ser múltiplo de $(p-1)$ y de $(q-1)$, es decir $a = \text{mcm}(p-1, q-1)$. Ya que el mínimo común múltiplo de $(p-1)$ y $(q-1)$ divide a $(p-1)(q-1)$, lo que tenemos es que el orden de m no solo divide a $\phi(n)$ (sección 5.4.1), sino también a $\text{mcm}(p-1, q-1)$.

Escogeremos por lo tanto un número e primo relativo con $t = \text{mcm}(p-1, q-1)$. El par (e, n) será la clave pública. Puesto que e tiene inversa módulo t, existirá un número d tal que

$$de \equiv 1 \pmod{t}.$$

Esta inversa puede calcularse fácilmente empleando el Algoritmo Extendido de Euclides. El par (d, n) será la clave privada. Nótese que si desconocemos los factores de n, este cálculo resulta prácticamente imposible.

La operación de cifrado se lleva a cabo según la expresión:

$$c = m^e \pmod{n} \tag{12.1}$$

mientras que el descifrado se hará de la siguiente forma:

$$m = c^d \pmod{n} \tag{12.2}$$

ya que

$$c^d = (m^e)^d = m^{ed} = m^{kt+1} = (m^k)^t m$$

Recordemos que t es el orden de m (o un divisor de este valor), por lo que

$$(m^k)^t = 1,$$

lo cual nos lleva de nuevo a m.

En muchos casos, se suele utilizar el Teorema Chino del Resto (sección 5.3) para facilitar los cálculos a la hora de descifrar un mensaje. Para ello se incluyen p y q en la clave privada, se calcula $p_1 = p^{-1} \pmod{q}$, y cuando se desea descifrar un mensaje c, se plantea el siguiente sistema de congruencias:

$$\begin{aligned} [c \bmod p]^{[d \bmod (p-1)]} &\equiv m_1 \pmod{p} \\ [c \bmod q]^{[d \bmod (q-1)]} &\equiv m_2 \pmod{q} \end{aligned}$$

Como ya se vio, estas ecuaciones tienen una solución única m módulo n. Para recuperar el mensaje original m, en lugar de usar la fórmula que dimos en la demostración del teorema, emplearemos otra ligeramente distinta:

$$m = m_1 + p[(m_2 - m_1)p_1 \pmod{q}].$$

Es inmediato comprobar que esta expresión es igual a m_1 módulo p. Si por el contrario tomamos módulo q, vemos que el segundo sumando es igual a $m_2 - m_1$, por lo que nos quedará m_2. Con ello conseguimos

que el módulo a la hora de hacer las exponenciaciones sea sensiblemente menor y, en consecuencia, los cálculos más rápidos. Nótese que los valores $[d \mod (p-1)]$, $[d \mod (q-1)]$ y p_1 pueden tenerse calculados de antemano —y, de hecho, se suelen incluir en la clave privada—.

Un posible atacante, si quiere recuperar la clave privada a partir de la pública, debe conocer los factores p y q de n, y esto representa un problema computacionalmente intratable, siempre que p y q —y, por lo tanto, n— sean lo suficientemente grandes.

12.3.1. Seguridad del algoritmo RSA

Técnicamente no es del todo cierto que el algoritmo RSA deposite su fuerza en el problema de la factorización. En realidad el hecho de tener que factorizar un número para descifrar un mensaje sin la clave privada es una mera *conjetura*. Nadie ha demostrado que no pueda surgir un método en el futuro que permita descifrar un mensaje sin usar la clave privada y sin factorizar el módulo n. De todas formas, este método podría ser empleado como una nueva técnica para factorizar números enteros, por lo que la anterior afirmación se considera en la práctica cierta. De hecho, existen estudios que demuestran que incluso recuperar sólo algunos bits del mensaje original resulta tan difícil como descifrar el mensaje entero.

Aparte de factorizar n, podríamos intentar calcular $\phi(n)$ directamente, o probar por la fuerza bruta tratando de encontrar la clave privada. Ambos ataques son más costosos computacionalmente que la propia factorización de n, afortunadamente.

Otro punto que cabría preguntarse es qué pasaría si los primos p y q que escogemos realmente fueran compuestos. Recordemos que los algoritmos de prueba de primos que conocemos son probabilísticos, por lo que jamás tendremos la absoluta seguridad de que p y q son realmente primos. Pero obsérvese que si aplicamos, por ejemplo, treinta pasadas del algoritmo de Rabin-Miller (sección 5.7), las probabilidades de que el número escogido pase el test y siga siendo primo son de

una contra 2^{60}: resulta más fácil que nos toque la lotería primitiva el mismo día que seamos alcanzados por un rayo (cuadro 1.1). Por otra parte, si p o q fueran compuestos, el algoritmo RSA simplemente no funcionaría correctamente.

12.3.2. Vulnerabilidades de RSA

Aunque el algoritmo RSA es bastante seguro conceptualmente, existen algunos puntos débiles en la forma de utilizarlo que pueden ser aprovechados por un atacante. En esta sección comentaremos estas posibles vulnerabilidades, así como la forma de evitar que surjan.

Claves débiles en RSA

Al emplear operaciones de exponenciación, RSA se basa en la generación de elementos de los conjuntos $\langle m \rangle$, dentro de $\mathbb{Z}_n^*$. Sabemos que el tamaño de $\langle m \rangle$ divide a mcm$(p-1, q-1)$ y, puesto que este número es par, ya que tanto $(p-1)$ como $(q-1)$ lo son, habrá valores de m cuyo orden puede ser muy pequeño, en concreto 1 o 2.

En el caso de que $\langle m \rangle$ tenga tamaño 1, RSA dejaría el mensaje original tal cual, es decir

$$m^e \equiv m \pmod{n} \tag{12.3}$$

En realidad, siempre hay mensajes que quedan inalterados al ser codificados mediante RSA, sea cual sea el valor de n, un ejemplo trivial es cuando $m = 1$. Nuestro objetivo será escoger los parámetros para reducir al mínimo el número de valores de m de orden 1. Se puede comprobar que, siendo $n = pq$ y e el exponente para cifrar,

$$\sigma_n = [1 + \text{mcd}(e-1, p-1)] \cdot [1 + \text{mcd}(e-1, q-1)]$$

es el número de valores de m que quedan igual al ser codificados. Si hacemos que $p = 1 + 2p'$ y $q = 1 + 2q'$, con p' y q' primos, entonces $\mathrm{mcd}(e-1, p-1)$ puede valer 1, 2 ó p' —análogamente ocurre con q'—. Los valores posibles de σ_n serán entonces 4, 6, 9, $2(p'+1)$, $2(q'+1)$, $3(p'+1)$, $3(p'+1)$, y $(p'+1)(q'+1)$. Afortunadamente, los cinco últimos son extremadamente improbables, por lo que en la inmensa mayoría de los casos, el número de mensajes que quedarían inalterados al cifrarlos mediante RSA es muy pequeño. En cualquier caso, es muy fácil detectar esta posibilidad, y siempre se puede modificar m aplicando distintos valores de relleno hasta completar la longitud en bits del módulo n. No obstante, como medida de precaución, se puede calcular σ_n a la hora de generar las claves pública y privada.

Claves demasiado cortas

En los primeros años de vida de RSA se recomendaba una longitud mínima de 512 para las claves. Esta recomendación subió a 768 en los 90 del siglo XX, y luego subió a 1024 bits. Actualmente no se considera segura una clave de menos de 2048 bits. Teniendo en cuenta los avances de la tecnología, y suponiendo que el algoritmo RSA no sea roto analíticamente, deberemos escoger la longitud de clave en función del tiempo que queramos que nuestra información permanezca en secreto. Una clave de 2048 bits parece a todas luces demasiado corta como para proteger información por más de unos pocos años.

Ataques de texto claro escogido

Supongamos que queremos falsificar una firma sobre un mensaje m. Se pueden calcular dos mensajes individuales m_1 y m_2, aparentemente inofensivos, tales que $m_1 m_2 = m$, y enviárselos a la *víctima* para que los firme. Entonces obtendríamos un m_1^d y m_2^d. Aunque desconozcamos d, si calculamos

$$m_1^d m_2^d = m^d \pmod{n}$$

obtendremos el mensaje m firmado. En realidad, este tipo de ataque no es peligroso si la firma la llevamos a cabo cifrando el *hash* del mensaje, en lugar del mensaje en sí. En este caso, tendríamos que encontrar tres mensajes, tales que la signatura de uno de ellos sea igual al producto de las signaturas de los otros dos, lo cual es extremadamente difícil.

Ataques de módulo común

Podría pensarse que, una vez generados p y q, será más rápido generar tantos pares de claves como queramos, en lugar de tener que emplear dos números primos diferentes en cada caso. Sin embargo, si lo hacemos así, un atacante podrá descifrar nuestros mensajes sin necesidad de la clave privada. Sea m el texto claro desconocido, que ha sido cifrado empleando dos claves de cifrado diferentes e_1 y e_2. Los criptogramas que tenemos son los siguientes:

$$\begin{aligned} c_1 &= m^{e_1} \pmod{n} \\ c_2 &= m^{e_2} \pmod{n} \end{aligned}$$

El atacante conoce pues n, e_1, e_2, c_1 y c_2. Si e_1 y e_2 son primos relativos, el Algoritmo Extendido de Euclides nos permitirá encontrar r y s tales que

$$re_1 + se_2 = 1$$

Ahora podemos hacer el siguiente cálculo

$$c_1^r c_2^s = m^{e_1 r} m^{e_2 s} = m^{e_1 r + e_2 s} \equiv m^1 \pmod{n}$$

Recordemos que esto sólo se cumple si e_1 y e_2 son números primos relativos, pero precisamente eso es lo que suele ocurrir en la gran ma-

yoría de los casos. Por lo tanto, se deben generar p y q diferentes para cada par de claves.

Ataques de exponente bajo

Si el exponente de cifrado e es demasiado bajo —hay implementaciones de RSA que, por razones de eficiencia emplean valores pequeños, como $e = 3$— existe la posibilidad de que un atacante pueda romper el sistema.

En primer lugar, puede que $m^e < n$, es decir, que el número que representa el texto claro elevado al exponente de cifrado resulte inferior que el módulo. En ese caso, bastaría con aplicar un logaritmo tradicional para recuperar el mensaje original a partir del criptograma. Esto se soluciona fácilmente aplicando un relleno a la representación binaria de m con bits aleatorios por la izquierda. Puesto que la longitud de m suele ser la de una clave simétrica o un *hash*, este valor suele ser mucho más corto que n, por lo que ese relleno se puede hacer sin problemas, y eliminarlo posteriormente a la operación de descifrado.

Existe otro ataque basado en valores de e bajos. Si, por ejemplo, tenemos tres claves públicas con el mismo valor $e = 3$ y diferentes módulos, n_1, n_2 y n_3, y alguien cifra un mismo mensaje m con las tres, tendremos tres valores c_1, c_2 y c_3. Como lo más probable es que los n_i sean primos relativos, y se cumple que $m^3 < n_1 n_2 n_3$, podemos aplicar el Teorema Chino del Resto (sección 5.3) para encontrar un valor x tal que $x = m^3$. Calculando la raíz cúbica de dicho valor, habremos recuperado el valor de m. Para protegerse de este ataque basta con emplear la misma estrategia descrita en el anterior párrafo, es decir, aplicar un relleno a m. De esta forma será extremadamente improbable que se emplee exactamente el mismo valor de m en los tres casos.

También existen ataques que se aprovechan de valores bajos en el exponente de descifrado d, por lo que conviene comprobar que d tenga aproximadamente el mismo número de bits que n.

Firmar y cifrar

Este problema no solo afecta a RSA, sino a cualquier método de cifrado asimétrico, y consiste en que nunca se debe firmar un mensaje después de cifrarlo. Por el contrario, debe firmarse primero y cifrar después el mensaje junto con la firma. Si ciframos un mensaje y lo firmamos después, un atacante podría eliminar nuestra firma y cambiarla por otra, ya que esta se ha hecho sobre el mensaje ya cifrado, que está disponible. Esto permitiría llevar a cabo ataques que podrían hacer creer al destinatario que el mensaje proviene de un autor diferente del auténtico.

12.4. Algoritmo de Diffie-Hellman

Es un algoritmo asimétrico, basado en el problema de Diffie-Hellman (sección 5.4.5), que se emplea fundamentalmente para acordar una clave común entre dos interlocutores, a través de un canal de comunicación inseguro. La ventaja de este sistema es que no son necesarias claves públicas en el sentido estricto, sino una información compartida por ambas partes.

Sean A y B los interlocutores en cuestión. En primer lugar, se calcula un número primo p y un generador $\alpha \in \mathbb{Z}_p^*$, con $2 \leq \alpha \leq p-2$. Esta información es pública y conocida por ambos. El algoritmo queda como sigue:

1. A escoge un número aleatorio x, comprendido entre 1 y $p-2$ y envía a B el valor
$$\alpha^x \pmod{p}$$
2. B escoge un número aleatorio y, análogamente al paso anterior, y envía a A el valor
$$\alpha^y \pmod{p}$$
3. B recoge α^x y calcula $k = (\alpha^x)^y \pmod{p}$.

4. A recoge α^y y calcula $k = (\alpha^y)^x \pmod p$.

Puesto que x e y no viajan por la red, al final A y B acaban compartiendo el valor de k, sin que nadie que capture los mensajes transmitidos pueda repetir el cálculo. El valor acordado k pertenece al conjunto $\langle\alpha\rangle$, que es un subconjunto de $\mathbb{Z}_p^*$, lo cual implica que hay valores de este conjunto que nunca aparecerán, por lo que conviene emplear una función KDF de blanqueamiento (sección 13.7) para derivar un valor que pueda ser usado como clave.

12.4.1. Uso fuera de línea del algoritmo de Diffie-Hellman

El algoritmo de Diffie-Hellman puede ser empleado fuera de línea, de forma que no haga falta establecer un diálogo entre emisor y receptor, usando como clave privada el valor x, y como clave pública el valor $\alpha^x \pmod p$.

Supongamos que B quiere enviar un mensaje m a A, cuya clave pública K_p es la tupla $[p, \alpha, \alpha^x \pmod p]$, y cuya clave privada K_P es x. Siendo $E_k(\cdot)$, $D_k(\cdot)$ las funciones de cifrado y descifrado de un algoritmo simétrico, el protocolo queda como sigue:

1. B genera un número y aleatorio, comprendido entre 1 y $p-2$.
2. B calcula $k = \text{KDF}\,[(\alpha^x)^y \pmod p]$.
3. B envía a A los valores $\alpha^y \pmod p$ y $c = E_k(m)$.
4. A calcula $k = \text{KDF}\,[(\alpha^y)^x \pmod p]$, y obtiene $m = D_k(c)$.

Este algoritmo es válido para cifrar información, pero no sirve para generar firmas digitales.

12.4.2. Parámetros adecuados para el algoritmo de Diffie-Hellman

Dado un número primo p, las potencias del valor α generan un subgrupo cuyo tamaño divide a $\phi(p)$. En el mejor de los casos, α será un generador de $\mathbb{Z}_p^*$ y su tamaño será exactamente igual a $\phi(p)$. Sin embargo, existe un algoritmo para calcular logaritmos discretos (denominado de Pohlig-Hellman) de manera relativamente eficiente si $\phi(p)$ no contiene ningún factor primo *grande*. Para evitar esto se ha propuesto que p tenga la forma $p = 2q + 1$, siendo q un número primo grande.

En este caso, dado un valor α, su orden puede ser $2, q$ o $2q$. En principio, podría parecer que nos interesa un valor de α cuyo orden sea $2q$ (que será un generador de $\mathbb{Z}_p^*$), pero en ese conjunto todos los elementos pares son residuos cuadráticos, mientras que los impares no, por lo que un atacante podría deducir, calculando el símbolo de Legendre de un elemento en $\mathbb{Z}_p^*$ (sección 5.4.2), el bit menos significativo de su exponente. Por lo tanto, lo que nos interesa es un valor α que genere un subgrupo multiplicativo de grado q, cuyos elementos son todos residuos cuadráticos. El algoritmo de elección de los parámetros p y α quedaría entonces de la siguiente forma:

1. Escoger un número primo q.
2. Comprobar que $2q + 1$ es primo. En caso contrario volver al paso 1.
3. $p = 2q + 1$.
4. Escoger un valor α aleatorio.
5. Comprobar que $\alpha^q \equiv 1 \pmod{p}$. En caso contrario volver al paso 4.

12.5. Otros algoritmos asimétricos

12.5.1. Algoritmo de ElGamal

Fue diseñado en un principio para producir firmas digitales, pero posteriormente se extendió también para cifrar mensajes. Se basa en el problema de los logaritmos discretos y en el de Diffie-Hellman.

Para generar un par de claves, se escoge un número primo n y dos números aleatorios p y x menores que n. Se calcula entonces

$$y = p^x \pmod{n}$$

La clave pública es (p, y, n), mientras que la clave privada es x.

Escogiendo n primo, garantizamos que sea cual sea el valor de p, el conjunto $\{p, p^2, p^3 \ldots\}$ es una permutación del conjunto $\{1, 2, \cdots, n-1\}$. Nótese que esto no es necesario para que el algoritmo funcione, por lo que podemos emplear realmente un n no primo, siempre que el conjunto generado por las potencias de p sea lo suficientemente grande.

Firmas digitales de ElGamal

Para *firmar* un mensaje m basta con escoger un número k aleatorio, que sea primo relativo con $n-1$, y calcular

$$\begin{array}{rcl} a & = & p^k \pmod{n} \\ b & = & (m - xa)k^{-1} \pmod{(n-1)} \end{array} \tag{12.4}$$

La firma la constituye el par (a, b). En cuanto al valor k, debe mantenerse en secreto y ser diferente cada vez. La firma se verifica comprobando que

$$y^a a^b = p^m \pmod{n} \tag{12.5}$$

Cifrado de ElGamal

Para cifrar el mensaje m se escoge primero un número aleatorio k primo relativo con $(n-1)$, que también será mantenido en secreto. Calculamos entonces las siguientes expresiones

$$\begin{array}{rcl} a & = & p^k \pmod{n} \\ b & = & y^k m \pmod{n} \end{array} \tag{12.6}$$

El par (a, b) es el texto cifrado, de doble longitud que el texto original. Para decodificar se calcula

$$m = b \cdot a^{-x} \pmod{n} \tag{12.7}$$

12.5.2. Algoritmo de Rabin

El sistema de clave asimétrica de Rabin se basa en el problema de calcular raíces cuadradas módulo un número compuesto. Este problema se ha demostrado que es equivalente al de la factorización de dicho número.

En primer lugar escogemos dos números primos, p y q, ambos congruentes con 3 módulo 4 (los dos últimos bits a 1). Estos primos son la clave privada. La clave pública es su producto, $n = pq$.

Para cifrar un mensaje m, simplemente se calcula

$$c = m^2 \pmod{n} \tag{12.8}$$

El descifrado del mensaje se hace calculando lo siguiente:

$$\begin{aligned} m_1 &= c^{(p+1)/4} \pmod{p} \\ m_2 &= (p - c^{(p+1)/4}) \pmod{p} \\ m_3 &= c^{(q+1)/4} \pmod{q} \\ m_4 &= (q - c^{(q+1)/4}) \pmod{q} \end{aligned}$$

Luego se escogen a y b tales que $a = q(q^{-1} \pmod{p})$ y $b = p(p^{-1} \pmod{q})$. Los cuatro posibles mensajes originales son

$$\begin{aligned} m_a &= (am_1 + bm_3) \pmod{n} \\ m_b &= (am_1 + bm_4) \pmod{n} \\ m_c &= (am_2 + bm_3) \pmod{n} \\ m_d &= (am_2 + bm_4) \pmod{n} \end{aligned} \tag{12.9}$$

Desgraciadamente, no existe ningún mecanismo para decidir cuál de los cuatro es el auténtico, por lo que el mensaje deberá incluir algún tipo de información para que el receptor pueda distinguirlo de los otros.

12.5.3. Algoritmo DSA

El algoritmo DSA (*Digital Signature Algorithm*) es una parte el estándar de firma digital DSS (*Digital Signature Standard*). Este algoritmo, propuesto por el NIST, data de 1991, es una variante del método asimétrico de ElGamal.

Creación del par clave pública-clave privada

El algoritmo de generación de claves es el siguiente:

1. Seleccionar un número primo q de 160 bits.
2. Escoger un número primo p, tal que $p-1$ sea múltiplo de q, y cuya longitud en bits sea múltiplo de 64 y esté comprendida entre 512 y 1024.

3. Seleccionar un elemento $g \in \mathbb{Z}_p^*$ y calcular $\alpha = g^{(p-1)/q} \pmod{p}$.
4. Si $\alpha = 1$ volver al paso 3.
5. Seleccionar un número entero aleatorio a, tal que $1 < a < q - 1$
6. Calcular $y = \alpha^a \pmod{p}$.
7. La clave pública es (p, q, α, y). La clave privada es a.

Generación y verificación de la firma

Siendo h la salida de una función MDC (sección 13.3) sobre el mensaje m, la generación de una firma se hace mediante el siguiente algoritmo:

1. Seleccionar un número aleatorio k tal que $1 < k < q$.
2. Calcular $r = [\alpha^k \pmod{p}] \pmod{q}$.
3. Calcular $s = k^{-1}(h + ar) \pmod{q}$.
4. Si r o s valen 0, volver al paso 1.
5. La firma del mensaje m es el par (r, s).

Es muy importante que el valor k aleatorio empleado en el cálculo de la firma sea impredecible y secreto, ya que la clave privada podría ser deducida a partir del mismo. Adicionalmente, k también ha de ser único para cada firma, ya que en caso contrario se puede deducir fácilmente su valor, comprometiendo por tanto la clave privada.

El destinatario efectuará las siguientes operaciones, suponiendo que conoce la clave pública (p, q, α, y), para verificar la autenticidad de la firma:

1. Verificar que $0 < r < q$ y $0 < s < q$. En caso contrario, rechazar la firma.

2. Calcular $\omega = s^{-1} \pmod{q}$.

3. Calcular $u_1 = h\omega \pmod{q}$.

4. Calcular $u_2 = r\omega \pmod{q}$.

5. Calcular $v = [\alpha^{u_1} y^{u_2} \pmod{p}] \pmod{q}$.

6. Aceptar la firma si y sólo si $v = r$.

12.6. Criptografía de curva elíptica

Como vimos en la sección 6.4, para curvas elípticas existe un problema análogo al de los logaritmos discretos en grupos finitos de enteros. Esto nos va a permitir trasladar cualquier algoritmo criptográfico definido sobre enteros, y que se apoye en este problema, al ámbito de las curvas elípticas. La ventaja que se obtiene es que, con claves más pequeñas, se alcanza un nivel de seguridad equiparable.

Debido a la relación existente entre ambos, muchos algoritmos que se apoyan en el problema de la factorización pueden ser replanteados para descansar sobre los logaritmos discretos. De hecho, existen versiones de curva elíptica de algunos de los algoritmos asimétricos más populares. A modo de ejemplo, en esta sección veremos cómo se redefinen los algoritmos de ElGamal y Diffie-Hellman.

12.6.1. Cifrado de ElGamal sobre curvas elípticas

Sea un grupo de curva elíptica, definido en $GF(n)$ ó $GF(2^n)$. Sea $\mathbf{p}$ un punto de la curva. Sea el conjunto $\langle \mathbf{p} \rangle$, de cardinal n. Escogemos entonces un valor entero x comprendido entre 1 y $n-1$, y calculamos

$$\mathbf{y} = x\mathbf{p} \tag{12.10}$$

La clave pública vendrá dada por $(\mathbf{p}, \mathbf{y}, n)$, y la clave privada será x.

El cifrado se hará escogiendo un número aleatorio k primo relativo con n. Seguidamente calculamos las expresiones

$$\begin{array}{rcl} \mathbf{a} & = & k\mathbf{p} \\ \mathbf{b} & = & \mathbf{m} + k\mathbf{y} \end{array} \tag{12.11}$$

siendo $\mathbf{m}$ el mensaje original representado como un punto de la curva. El criptograma será el par $(\mathbf{a}, \mathbf{b})$. Para descifrar, será suficiente con calcular

$$\mathbf{m} = -(x\mathbf{a}) + \mathbf{b} \tag{12.12}$$

12.6.2. Cifrado de Diffie-Hellman sobre curvas elípticas

Este algoritmo (cuya nomenclatura en inglés es ECDH, *Elliptic Curve Diffie-Hellman)* es en la actualidad uno de los más utilizados. De hecho, la adaptación del algoritmo es trivial, basta con escoger una curva elíptica definida por un polinomio irreducible y un punto $\mathbf{p}$ de esa curva, que serán los parámetros públicos.

1. A escoge un número aleatorio x, y envía a B el valor $x\mathbf{p}$.
2. B escoge un número aleatorio y, y envía a A el valor $y\mathbf{p}$.
3. B recoge $x\mathbf{p}$ y calcula $\mathbf{k} = y(x\mathbf{p})$.
4. A recoge $y\mathbf{p}$ y calcula $\mathbf{k} = x(y\mathbf{p})$.

Finamente, la clave negociada se suele calcular empleando una función resumen sobre el valor $\mathbf{k}$ obtenido.

12.7. Ejercicios resueltos

1. Suponga un sistema RSA con los siguientes parámetros:

$$\begin{aligned} N &= 44173 \\ K_P &= 25277 \\ C &= 8767, 18584, 7557, 4510, 40818, 39760, \\ &\quad 4510, 39760, 6813, 7557, 14747 \end{aligned}$$

   a) Factorizar el módulo N.

   b) Calcular la clave privada K_p.

   c) Descifrar el mensaje C.

   Solución:

   a) Para factorizar N, basta con emplear el método de tanteo (*prueba y error*) a partir de la raíz cuadrada de 44173, obteniéndose que $N = 271 \cdot 163$.

   b) K_p debe ser la inversa de K_P módulo $\phi(N) = 270 \cdot 162 = 43740$. Empleando el Algoritmo Extendido de Euclides, llegamos a

$$K_p = K_P^{-1} = 25277^{-1} \pmod{43740} = 26633$$

   c) El descifrado podemos llevarlo a cabo empleando el Algoritmo Rápido de Exponenciación:

$$\begin{array}{rclrcl} c_0 &=& 8767 & m_0 &=& 8767^{K_p} \pmod{44173} = 75 \\ c_1 &=& 18584 & m_1 &=& 18584^{K_p} \pmod{44173} = 114 \\ c_2 &=& 7557 & m_2 &=& 7557^{K_p} \pmod{44173} = 105 \\ c_3 &=& 4510 & m_3 &=& 4510^{K_p} \pmod{44173} = 112 \\ c_4 &=& 40818 & m_4 &=& 40818^{K_p} \pmod{44173} = 116 \\ c_5 &=& 39760 & m_5 &=& 39760^{K_p} \pmod{44173} = 111 \\ c_6 &=& 4510 & m_6 &=& 4510^{K_p} \pmod{44173} = 112 \\ c_7 &=& 39760 & m_7 &=& 39760^{K_p} \pmod{44173} = 111 \\ c_8 &=& 6813 & m_8 &=& 6813^{K_p} \pmod{44173} = 108 \\ c_9 &=& 7557 & m_9 &=& 7557^{K_p} \pmod{44173} = 105 \\ c_{10} &=& 14747 & m_{10} &=& 14747^{K_p} \pmod{44173} = 115 \end{array}$$

Capítulo 13

Funciones resumen

Las funciones resumen (*hash*, en inglés) proporcionan, a partir de un mensaje de longitud arbitraria, una secuencia de bits de longitud fija que va asociada al propio mensaje, actuando como una especie de *huella dactilar* del mismo, y que resulta muy difícil de falsificar.

La utilidad principal de estas funciones es la de detectar pérdidas de integridad de la información. De esta forma, si se transmite un mensaje junto con su *hash* correspondiente, es fácil comprobar que éste no ha sufrido ninguna alteración calculando su resumen de forma independiente y comparando el resultado con el que se nos envió. También se usan cuando necesitamos comprobar que alguien posee un secreto —por ejemplo, una contraseña— sin tener que guardarlo *en claro* en nuestro sistema. En este caso, basta con tener almacenado el resumen del mismo y compararlo con el *hash* del valor que haya introducido el usuario.

Existen fundamentalmente dos tipos de funciones resumen: aquellas que se calculan directamente sobre el mensaje con el propósito es garantizar que no ha sufrido modificaciones, denominadas MDC (*modification detection codes*), y las que emplean en sus cálculos una clave adicional, denominadas MAC (*message authentication codes*), que garantizan además el origen del mensaje, ya que sólo pueden calcularas (y

verificarlas) aquellos individuos que estén en posesión de la clave.

13.1. Propiedades

Una función resumen debe actuar como si fuera una *huella dactilar* del mensaje, ligando de manera unívoca el mensaje con su *hash* (o *signatura)*. Llamaremos *colisión* a un par de mensajes (m, m') con el mismo *hash*. Como veremos más adelante, el espacio de posibles valores *hash* es mucho menor que el de mensajes, por lo que todas las funciones resumen presentan colisiones. Por lo tanto, debemos imponer las siguientes condiciones a cualquier función resumen $r(x)$:

1. $r(m)$ es de longitud fija, independientemente de la longitud de m.
2. Dado m, es fácil calcular $r(m)$.
3. Dado $r(m)$, es computacionalmente intratable recuperar m.
4. Dado m, es computacionalmente intratable obtener un m' tal que $r(m) = r(m')$.

Obsérvese que la condición número 3 obliga a que una función resumen sea indistinguible, para un observador externo, de un mapeado aleatorio entre el conjunto de mensajes y el de signaturas. Las propiedades que acabamos de enunciar son válidas tanto para los MDC como para los MAC, con la dificultad añadida para estos últimos de que el atacante deberá averiguar además la clave correspondiente. De hecho, conocida la clave, un MAC se comporta exactamente igual que un MDC.

13.2. Longitud adecuada para una signatura

Para decidir cuál debe ser la longitud apropiada de una signatura, veamos primero el siguiente ejemplo: ¿Cuál es la cantidad n de personas que hay que poner en una habitación para que la probabilidad P de que el cumpleaños de una de ellas sea el mismo día que el mío supere el 50 %? Sabemos que cuando $n = 1$, $P = \frac{1}{365}$. Cuando $n = 2$, la probabilidad de que *ningún cumpleaños* coincida con el nuestro es el producto de la probabilidad de que no coincida el primero, por la probabilidad de que no coincida el segundo, luego:

$$P = 1 - \frac{364}{365} \cdot \frac{364}{365}$$

En el caso general,

$$P = 1 - \left(\frac{364}{365}\right)^n$$

Para que $P > 0{,}5$, n debe ser al menos igual a 253. Sin embargo, ¿cuál sería la cantidad de gente necesaria para que la probabilidad Q de que dos personas cualesquiera tengan el mismo cumpleaños supere el 50 %? Las dos primeras personas (o sea, cuando $n = 2$) tienen una probabilidad $\frac{364}{365}$ de *no compartir* el cumpleaños; una tercera, supuesto que las dos primeras no lo comparten, tiene una probabilidad $\frac{363}{365}$ de no compartirlo con las otras dos, por lo que tenemos $\frac{364 \cdot 363}{365 \cdot 365}$, y así sucesivamente. En el caso general nos queda

$$Q = 1 - \left(\frac{364 \cdot 363 \ldots (365 - n + 1)}{365^{(n-1)}}\right) \qquad \text{con } n \geq 2$$

Si hacemos los cálculos, veremos que $Q > 0{,}5$ si $n > 22$, una cantidad *sorprendentemente* mucho menor que 253.

La consecuencia de este ejemplo, conocido como la *paradoja del cumpleaños*, es que aunque resulte muy difícil dado m calcular un m' tal

que $r(m) = r(m')$, es considerablemente menos costoso generar muchos valores aleatoriamente, y posteriormente buscar entre ellos una pareja cualquiera (m, m'), tal que $r(m) = r(m')$.

En el caso de una signatura de 64 bits, necesitaríamos explorar del orden de 2^{64} mensajes, dado un m, para obtener un m' que produzca una colisión con m, pero bastaría con generar aproximadamente 2^{32} mensajes aleatorios para que aparecieran dos con la misma signatura —en general, si la primera cantidad es muy grande, la segunda cantidad es aproximadamente su raíz cuadrada—. El primer ataque nos llevaría 600.000 años si empleamos una computadora que generara un millón de mensajes por segundo, mientras que el segundo necesitaría apenas una hora.

Hemos de añadir por tanto a nuestra lista de condiciones la siguiente:

- Debe ser difícil encontrar dos mensajes aleatorios, m y m', tales que $r(m) = r(m')$.

Cabe resaltar que este tipo de ataques solo tienen sentido contra funciones MDC. En el caso de las funciones MAC, un atacante que quisiera hacer esto tendría que haber encontrado antes la clave, lo cual ya representa una complejidad computacional enorme que habría que añadir a todo el proceso. En consecuencia, hoy por hoy se recomienda emplear signaturas de al menos 256 bits para funciones MDC, y de un mínimo de 128 bits para funciones MAC.

13.3. Funciones MDC

En general, los MDC se basan en la idea de *funciones de compresión*, que dan como resultado bloques de longitud fija a a partir de bloques de longitud fija b, con $a < b$. Estas funciones se encadenan de forma iterativa, haciendo que la entrada en el paso i sea función del i-ésimo bloque del mensaje (m_i) y de la salida del paso $i-1$ (ver figura 13.1). En

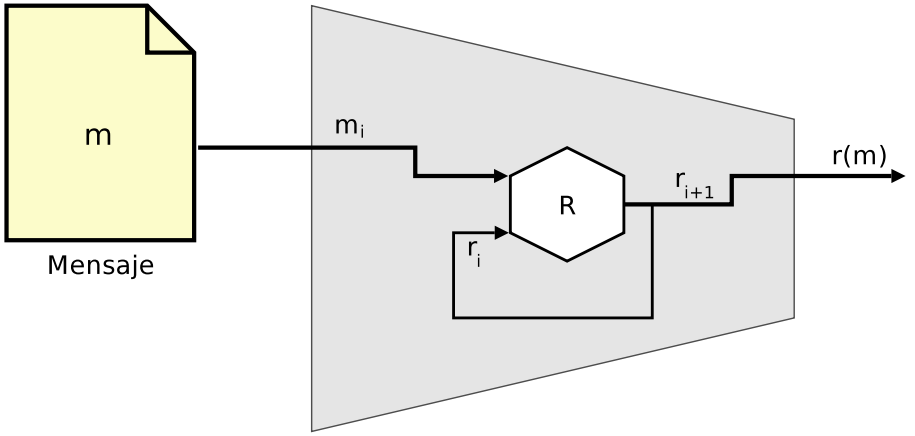

Figura 13.1: Estructura iterativa de una *función resumen*. R representa la función de compresión, m es el mensaje completo, m_i el i-ésimo trozo de m, y r_i la salida de la función en el paso i.

general, se suele incluir en alguno de los bloques del mensaje m —al principio o al final—, información sobre la longitud total del mensaje.

13.3.1. Algoritmo MD5

Diseñado por Ron Rivest, se trata de uno de los más populares algoritmos de generación de signaturas, debido en gran parte a su inclusión en las primeras versiones de PGP. Resultado de una serie de mejoras sobre el algoritmo MD4, propuesto anteriormente por el mismo autor, procesa los mensajes de entrada en bloques de 512 bits, y produce una salida de 128 bits.

Siendo m un mensaje de b bits de longitud, en primer lugar se alarga m hasta que su longitud sea exactamente 64 bits inferior a un múltiplo de 512. El alargamiento (o *padding)* se lleva a cabo añadiendo un 1 seguido de tantos ceros como sea necesario. En segundo lugar, se

añaden 64 bits representando el valor de b, empezando por el byte menos significativo (criterio *little-endian)*. De esta forma tenemos el mensaje como un número entero de bloques de 512 bits, y además le hemos incorporado información sobre su longitud.

Antes de procesar el primer bloque del mensaje, se inicializan cuatro registros de 32 bits con los siguientes valores hexadecimales, según el criterio *little endian* —el byte menos significativo queda en la dirección de memoria más baja—:

$$\begin{aligned} A &= 67452301 \\ B &= EFCDAB89 \\ C &= 98BADCFE \\ D &= 10325476 \end{aligned}$$

Posteriormente comienza el *ciclo principal* del algoritmo, que se repetirá para cada bloque de 512 bits del mensaje. En primer lugar copiaremos los valores de A,B,C y D en otras cuatro variables, a,b,c y d. Luego definiremos las siguientes cuatro funciones:

$$\begin{aligned} F(X,Y,Z) &= (X \wedge Y) \vee ((\neg X) \wedge Z) \\ G(X,Y,Z) &= (X \wedge Z) \vee ((Y \wedge (\neg Z)) \\ H(X,Y,Z) &= X \oplus Y \oplus Z \\ I(X,Y,Z) &= Y \oplus (X \vee (\neg Z)) \end{aligned}$$

Ahora representaremos por m_j el j-ésimo bloque de 32 bits del mensaje m (de 0 a 15), y definiremos otras cuatro funciones:

$$\begin{aligned} FF(a,b,c,d,m_j,s,t_i) &\Rightarrow a = b + ((a + F(b,c,d) + m_j + t_i) \lhd s) \\ GG(a,b,c,d,m_j,s,t_i) &\Rightarrow a = b + ((a + G(b,c,d) + m_j + t_i) \lhd s) \\ HH(a,b,c,d,m_j,s,t_i) &\Rightarrow a = b + ((a + H(b,c,d) + m_j + t_i) \lhd s) \\ II(a,b,c,d,m_j,s,t_i) &\Rightarrow a = b + ((a + I(b,c,d) + m_j + t_i) \lhd s) \end{aligned}$$

donde la función $a \lhd s$ representa desplazar circularmente la representación binaria del valor a s bits a la izquierda, con reentrada.

Las 64 operaciones que se realizan en total quedan agrupadas en cuatro rondas.

- Primera Ronda:
  $FF(a, b, c, d, m_0, 7, D76AA478)$
  $FF(d, a, b, c, m_1, 12, E8C7B756)$
  $FF(c, d, a, b, m_2, 17, 242070DB)$
  $FF(b, c, d, a, m_3, 22, C1BDCEEE)$
  $FF(a, b, c, d, m_4, 7, F57C0FAF)$
  $FF(d, a, b, c, m_5, 12, 4787C62A)$
  $FF(c, d, a, b, m_6, 17, A8304613)$
  $FF(b, c, d, a, m_7, 22, FD469501)$
  $FF(a, b, c, d, m_8, 7, 698098D8)$
  $FF(d, a, b, c, m_9, 12, 8B44F7AF)$
  $FF(c, d, a, b, m_{10}, 17, FFFF5BB1)$
  $FF(b, c, d, a, m_{11}, 22, 895CD7BE)$
  $FF(a, b, c, d, m_{12}, 7, 6B901122)$
  $FF(d, a, b, c, m_{13}, 12, FD987193)$
  $FF(c, d, a, b, m_{14}, 17, A679438E)$
  $FF(b, c, d, a, m_{15}, 22, 49B40821)$

- Segunda Ronda:
  $GG(a, b, c, d, m_1, 5, F61E2562)$
  $GG(d, a, b, c, m_6, 9, C040B340)$
  $GG(c, d, a, b, m_{11}, 14, 265E5A51)$
  $GG(b, c, d, a, m_0, 20, E9B6C7AA)$
  $GG(a, b, c, d, m_5, 5, D62F105D)$
  $GG(d, a, b, c, m_{10}, 9, 02441453)$
  $GG(c, d, a, b, m_{15}, 14, D8A1E681)$
  $GG(b, c, d, a, m_4, 20, E7D3FBC8)$
  $GG(a, b, c, d, m_9, 5, 21E1CDE6)$
  $GG(d, a, b, c, m_{14}, 9, C33707D6)$
  $GG(c, d, a, b, m_3, 14, F4D50D87)$
  $GG(b, c, d, a, m_8, 20, 455A14ED)$
  $GG(a, b, c, d, m_{13}, 5, A9E3E905)$

$GG(d, a, b, c, m_2, 9, FCEFA3F8)$
$GG(c, d, a, b, m_7, 14, 676F02D9)$
$GG(b, c, d, a, m_{12}, 20, 8D2A4C8A)$

- Tercera Ronda:
  $HH(a, b, c, d, m_5, 4, FFFA3942)$
  $HH(d, a, b, c, m_8, 11, 8771F681)$
  $HH(c, d, a, b, m_{11}, 16, 6D9D6122)$
  $HH(b, c, d, a, m_{14}, 23, FDE5380C)$
  $HH(a, b, c, d, m_1, 4, A4BEEA44)$
  $HH(d, a, b, c, m_4, 11, 4BDECFA9)$
  $HH(c, d, a, b, m_7, 16, F6BB4B60)$
  $HH(b, c, d, a, m_{10}, 23, BEBFBC70)$
  $HH(a, b, c, d, m_{13}, 4, 289B7EC6)$
  $HH(d, a, b, c, m_0, 11, EAA127FA)$
  $HH(c, d, a, b, m_3, 16, D4EF3085)$
  $HH(b, c, d, a, m_6, 23, 04881D05)$
  $HH(a, b, c, d, m_9, 4, D9D4D039)$
  $HH(d, a, b, c, m_{12}, 11, E6DB99E5)$
  $HH(c, d, a, b, m_{15}, 16, 1FA27CF8)$
  $HH(b, c, d, a, m_2, 23, C4AC5665)$

- Cuarta Ronda:
  $II(a, b, c, d, m_0, 6, F4292244)$
  $II(d, a, b, c, m_7, 10, 432AFF97)$
  $II(c, d, a, b, m_{14}, 15, AB9423A7)$
  $II(b, c, d, a, m_5, 21, FC93A039)$
  $II(a, b, c, d, m_{12}, 6, 655B59C3)$
  $II(d, a, b, c, m_3, 10, 8F0CCC92)$
  $II(c, d, a, b, m_{10}, 15, FFEFF47D)$
  $II(b, c, d, a, m_1, 21, 85845DD1)$
  $II(a, b, c, d, m_8, 6, 6FA87E4F)$
  $II(d, a, b, c, m_{15}, 10, FE2CE6E0)$
  $II(c, d, a, b, m_6, 15, A3014314)$
  $II(b, c, d, a, m_{13}, 21, 4E0811A1)$
  $II(a, b, c, d, m_4, 6, F7537E82)$

$II(d, a, b, c, m_{11}, 10, BD3AF235)$
$II(c, d, a, b, m_2, 15, 2AD7D2BB)$
$II(b, c, d, a, m_9, 21, EB86D391)$

Finalmente, los valores resultantes de a,b,c y d son sumados con A,B,C y D, quedando listos para procesar el siguiente bloque de datos. El resultado final del algoritmo es la concatenación de A,B,C y D.

A modo de curiosidad, diremos que las constantes t_i empleadas en cada paso son la parte entera del resultado de la operación $2^{32} \cdot \text{abs}(\sin(i))$, estando i representado en radianes.

13.3.2. Algoritmo SHA-1

El algoritmo SHA-1 fue desarrollado por la NSA, para ser incluido en el estándar DSS *(Digital Signature Standard)*. Al contrario que el otros algoritmos de cifrado propuestos por esta organización, SHA-1 se consideraba seguro[1] y libre de *puertas traseras*, ya que el hecho de que el algoritmo fuera realmente seguro favorecía a los propios intereses de la NSA. Produce firmas de 160 bits, a partir de bloques de 512 bits del mensaje original.

SHA-1 es similar a MD5, con la diferencia de que usa la ordenación *big endian* (los números enteros se representan empezando por el *byte* más significativo). Se inicializa de igual manera, es decir, añadiendo al final del mensaje un uno seguido de tantos ceros como sea necesario hasta completar 448 bits en el último bloque, para luego yuxtaponer la longitud en bits del propio mensaje —en este caso, el primer *byte* de la secuencia será el más significativo—. A diferencia de MD5, SHA-1 emplea cinco registros de 32 bits en lugar de cuatro, que deben ser inicializados antes de procesar el primer bloque con los siguientes valores:

[1] Desafortunadamente, la seguridad de SHA-1 ha quedado puesta en entredicho debido a los avances conseguidos en 2004 y 2005 por un equipo de criptólogos chinos, liderado por Xiaoyun Wang.

$$
\begin{aligned}
A &= 67452301 \\
B &= EFCDAB89 \\
C &= 98BADCFE \\
D &= 10325476 \\
E &= C3D2E1F0
\end{aligned}
$$

Una vez que los cinco valores están inicializados, se copian en cinco variables, a, b, c, d y e. El ciclo principal tiene cuatro rondas con 20 operaciones cada una:

$$
\begin{aligned}
F(X,Y,Z) &= (X \wedge Y) \vee ((\neg X) \wedge Z) \\
G(X,Y,Z) &= X \oplus Y \oplus Z \\
H(X,Y,Z) &= (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)
\end{aligned}
$$

La operación F se emplea en la primera ronda (t comprendido entre 0 y 19), la G en la segunda (t entre 20 y 39) y en la cuarta (t entre 60 y 79), y la H en la tercera (t entre 40 y 59). Además se emplean cuatro constantes, una para cada ronda:

$$
\begin{aligned}
K_0 &= 5A827999 \\
K_1 &= 6ED9EBA1 \\
K_2 &= 8F1BBCDC \\
K_3 &= CA62C1D6
\end{aligned}
$$

El bloque de mensaje m se trocea en 16 partes de 32 bits m_0 a m_{15} y se convierte en 80 trozos de 32 bits w_0 a w_{79} usando el siguiente algoritmo:

$$
\begin{aligned}
w_t &= m_t && \text{para } t = 0 \ldots 15 \\
w_t &= (w_{t-3} \oplus w_{t-8} \oplus w_{t-14} \oplus w_{t-16}) \lhd 1 && \text{para } t = 16 \ldots 79
\end{aligned}
$$

Todos los m_i obtenidos se interpretan como enteros en las operaciones del algoritmo empleando la ordenación *big endian*. Como curiosidad, diremos que la NSA introdujo en 1995 el desplazamiento a la

izquierda para corregir una debilidad del algoritmo, que no fue descubierta por los criptógrafos civiles hasta 2004, lo cual supuso modificar el nombre del mismo para llamar a partir de entonces SHA-0 a la versión original, y SHA-1 a la modificada.

El ciclo principal del algoritmo es entonces el siguiente:

$$
\begin{array}{ll}
\text{FOR} & t = 0 \text{ TO } 79 \\
& i = t \text{ div } 20 \\
& Tmp = (a \lhd 5) + A(b, c, d) + e + w_t + K_i \\
& e = d \\
& d = c \\
& c = b \lhd 30 \\
& b = a \\
& a = Tmp
\end{array}
$$

siendo A la función F, G o H según el valor de t (F para $t \in [0, 19]$, G para $t \in [20, 39]$ y $[60, 79]$, H para $t \in [40, 59]$). Después los valores de a a e son sumados a los registros A a E y el algoritmo continúa con el siguiente bloque de datos. Finalmente, el valor de la función resumen será la concatenación de los contenidos de los registros A a E resultantes de procesar el último bloque del mensaje.

13.3.3. Algoritmos SHA-2

La familia de cifrados SHA-2 fue propuesta en 2001 por el NIST, y comprende dos algoritmos diferentes, denominados SHA-256 y SHA-512, que devuelven *hashes* de 256 y 512 bits respectivamente. Estas dos funciones se pueden emplear según seis modalidades diferentes:

- SHA-224: Se calcula SHA-256 y se trunca a 224 bits.
- SHA-256.
- SHA-384: Se calcula SHA-512 y se trunca a 384 bits.

- SHA-512.
- SHA-512/224: Se calcula SHA-512 y se trunca a 224 bits.
- SHA-512/256: Se calcula SHA-512 y se trunca a 256 bits.

La estructura y funcionamiento de SHA-256 y SHA-512 es parecida. Se diferencian en que SHA-256 emplea un vector de estado interno de 256 bits, organizado como 8 *palabras* de 32 bits, mientras que SHA-512 emplea 8 *palabras* de 64 bits. SHA-256 trocea el mensaje en bloques de 512 bits, y SHA-512 lo hace en bloques de 1024 bits. Finalmente, SHA-256 realiza 64 rondas, mientras que SHA-512 lleva a cabo 80.

El proceso de relleno *(padding)* es el mismo que en MD5 y SHA-1: se añade un bit 1 al final del mensaje, seguido de una serie de ceros hasta dejar exactamente 64 bits libres (128 en el caso de SHA-512) al final del último bloque, y se codifica en estos la longitud del mensaje, según el criterio *big-endian* (primero el *byte* más significativo).

Ambos algoritmos emplean operaciones lógicas *and, or y or-exclusivo*, rotaciones, y sumas modulares.

Si bien hace unos años se sospechaba que estas funciones podían ser rotas de manera inminente, debido a la caída de SHA-0 y SHA-1, en la actualidad (año 2018) no se les conocen debilidades significativas, por lo que siguen siendo consideradas válidas.

13.3.4. Algoritmo Keccak (SHA-3)

En octubre de 2012 culminó el proceso de selección de un nuevo algoritmo de *hashing*, iniciado por el NIST en 2007. El ganador fue Keccak, diseñado por Guido Bertoni, Joan Daemen (uno de los autores de Rijndael), Michaël Peeters y Gilles Van Assche. Si bien puede trabajar con cualquier tamaño de palabra que sea potencia de 2, para el estándar SHA-3 se escogió un tamaño de pablabra de $w = 2^6 = 64$ bits.

r	n
1152	224
1088	256
832	384
576	512

Cuadro 13.1: Número de bits del mensaje que *absorbe* el algoritmo Keccak en cada bloque, en función del tamaño de resumen deseado. Obsérvese que, en todos los casos, $1600 - r = 2n$.

Keccak posee un vector de estado de 1600 bits, organizado en una matriz de 5×5 palabras. Cada operación de actualización de dicha matriz consta de 24 iteraciones, en las que se aplican cinco funciones sencillas de manera secuencial. Estas operaciones se basan en el cálculo de bits de paridad, rotaciones, permutaciones, y *or-exclusivos*.

Para procesar un mensaje, este algoritmo emplea una construcción denominada *de esponja*, lo cual permite modular su resistencia a colisiones. El mensaje, después de ser ajustada su longitud mediante bits de relleno, se divide en trozos de tamaño r. El valor de r varía en función de la longitud final de resumen deseada (tabla 13.1). Cada bloque es combinado *(absorbido)* mediante or exclusivo con los r primeros bits del vector de estado. Después de aplicar la función completa de actualización, se extraen *(exprimen)* n bits. Se denomina *capacidad* de la función resumen al número c de bits del vector de estado que quedan sin combinar con los bits de r. En el caso de SHA-3, c es exactamente el doble de la longitud n del resumen.

13.4. Seguridad de las funciones MDC

Puesto que el conjunto de posibles mensajes de longitud arbitraria es infinito, y el conjunto de posibles valores de una función resumen es finito, inevitablemente habrá valores para la función que se corres-

pondan con más de un mensaje. De hecho, puede demostrarse que al menos una signatura se corresponde necesariamente con infinitos mensajes, y es razonable sospechar que, en general, cada uno de los posibles valores va a corresponder con infinitos mensajes. Aunque el estudio de este tipo de coincidencias, que permitirían *falsificar* mensajes, puede tener interés para funciones MAC, centraremos esta sección en las funciones MDC, ya que para las MAC habría primero que encontrar la clave, y si ésta quedara comprometida la seguridad del sistema habría sido rota por completo.

De lo argumentado en el párrafo anterior, podemos deducir que todas las funciones MDC presentan colisiones. Distinguiremos no obstante dos tipos de estrategias para hallarlas, con objeto de delimitar el grado de compromiso que pueden provocar en un algoritmo concreto:

- De *preimagen*: El atacante busca un mensaje m' para que proporcione un valor h concreto de *hash*, es decir $r(m') = h$. Si en lugar de usar h como punto de partida tomamos como entrada un m y su *hash* asociado, el procedimiento se denomina de *segunda preimagen*.
- De *colisión* propiamente dicha: El atacante se limita a buscar dos valores m y m' que colisionen, pero desconoce inicialmente tanto sus valores como el que tomará la función resumen.

Si nuestra función resumen fuera realmente un mapeado aleatorio con una longitud de n bits, siempre podríamos plantear una búsqueda de *preimagen* generando aproximadamente una media de 2^{n-1} mensajes aleatorios, y una búsqueda de *colisión* generando $2^{\frac{n}{2}-1}$ mensajes. Por lo tanto, solo consideraremos válidos aquellos ataques que mejoren en eficiencia a alguna de estas dos estrategias.

Un ataque de *preimagen* satisfactorio comprometería la función resumen de manera grave, ya que bastará con sustituir un m con el m' que hayamos calculado para falsificar un mensaje. De todas formas, es bastante difícil que el m' tenga un *aspecto* válido. Piénsese por ejemplo en un mensaje m de texto: nuestra técnica de preimagen tendría

que ser capaz de generar otro mensaje m', también de texto, y con un significado concreto, para que la falsificación tuviera interés. Lo más habitual es que el m' obtenido tenga un aspecto más o menos aleatorio, lo cual le conferiría una utilidad mucho más limitada, como por ejemplo la intoxicación de redes de comunicación con datos erróneos que puedan pasar por auténticos.

En cuanto a los ataques de *colisión*, la situación sería menos grave. Puesto que no sabemos *a priori* los valores m y m' que vamos a encontrar, difícilmente podremos emplear una técnica de este tipo para falsificar una firma digital —a no ser que logremos que la víctima firme un mensaje sin sentido y luego lo sustituyamos por otro, también sin sentido—.

En la actualidad se han encontrado colisiones para los algoritmos HAVAL-128, MD4, MD5, RIPEMD, SHA-0 y SHA-1, por lo que, teniendo en cuenta que contamos con numerosas alternativas que sí se ajustan a los estándares de seguridad actuales, no podemos aconsejar el uso de ninguno de ellos.

13.4.1. Ataques de extensión de mensaje

Muchas funciones resumen dan como salida la totalidad del estado interno r_i una vez procesado el último bloque del mensaje (ver figura 13.1). En este caso, podríamos plantear el siguiente ataque: si tenemos un mensaje m_1 y conocemos su signatura $r(m_1)$, en realidad tenemos el estado interno del algoritmo tras procesar m_1, por lo que, usando este estado como punto de partida, podemos añadir un segundo mensaje m_2, obteniendo el resumen de la concatenación de m_1 y m_2.

A modo de ejemplo, supongamos que diseñamos un protocolo que, para añadir propiedades de autentificación a una función MDC, emplee una clave secreta k, y calcule lo siguiente para un mensaje m:

$$r(m) = MDC(k||m)$$

donde $a||b$ representa la concatenación de a y b. Un atacante podría tomar $r(m)$, reconstruir el estado final del proceso de cálculo de este *hash*, e inicializando el algoritmo con estos valores, calcular una signatura para un mensaje m'. Como resultado obtendría una signatura válida para:

$$(k||m||m')$$

sin necesidad de conocer la clave k. Únicamente habría que tener en cuenta el proceso de relleno, e incluir la información asociada correspondiente entre m y m'.

13.5. Funciones MAC

Los MAC se caracterizan por el empleo de una clave secreta para poder calcular la integridad del mensaje. Puesto que dicha clave sólo es conocida por el emisor y el receptor, el receptor es el único que puede, mediante el cálculo de la función correspondiente, comprobar tanto la integridad como la procedencia del mensaje. Aparte de los algoritmos MAC específicos, existen varias técnicas para construir funciones MAC a partir de otros elementos:

- *Basados en cifrados por bloques*: Existen modos de operación de cifrado por bloques (sección 10.6), que pueden funcionar como funciones MAC. Por ejemplo, cifrando el mensaje en modo CBC con un vector de inicialización igual a 0, podemos tomar el último bloque cifrado como valor MAC. Este método tiene el problema de que es totalmente secuencial, y no puede ser paralelizado.
- *HMAC*: Se basan en el uso de cualquier función MDC existente. Siendo H una función MDC con tamaño de bloque de entrada b, K una clave secreta y m un mensaje, la función HMAC correspondiente queda definida como:

1. Si la longitud de K es menor que b, $K' = K$.
2. Si la longitud de K es mayor que b, $K' = H(K)$.
3. K'' se obtiene alargando K' por la derecha con ceros hasta que su longitud sea b.
4. $opad$ es el valor hexadecimal 5C repetido hasta que su longitud sea b.
5. $ipad$ es el valor hexadecimal 36 repetido hasta que su longitud sea b.
6. $\text{HMAC}(m) = H(K'' \oplus opad \,||\, H(K'' \oplus ipad \,||\, m))$.

En esta definición el símbolo $\oplus$ representa la operación *or-exclusivo* y $||$ representa la concatenación.

- *Basados en generadores de secuencia*: Empleando un generador de secuencia pseudoaleatorio el mensaje se *parte* en dos subcadenas —correspondientes al mensaje combinado con la secuencia y a la propia secuencia—, cada una de las cuales alimenta un Registro de Desplazamiento Retroalimentado (sección 11.3). El valor del MAC se obtiene a partir de los estados finales de ambos registros.

13.5.1. Algoritmo Poly1305

Poly1305 es un algoritmo MAC, propuesto por Daniel J. Bernstein en 2005, que genera *hashes* de 128 bits, y se apoya en cualquier algoritmo de cifrado simétrico capaz de cifrar un bloque de 128 bits con una clave de la misma longitud. En concreto, este algoritmo define la siguiente función:

$$Poly1305_r(m, E_k(n))$$

donde:

- m es el mensaje a autentificar, que debe tener una longitud en bits múltiplo de 8. En otras palabras, debe ser una secuencia de *bytes*.
- r es una clave de 128 bits (aunque su longitud efectiva es de 106 bits).
- E_k es un algoritmo de cifrado simétrico, con una clave k de 128 bits.
- n es un *nonce* de 128 bits.

El funcionamiento del algoritmo es relativamente simple. En primer lugar se trocea el mensaje m en bloques de 17 *bytes* según el siguiente criterio:

- Se toma cada bloque m_i de 16 *bytes* del mensaje y se le añade un *byte* al final con valor 1.
- Si el último bloque tiene menos de 16 *bytes*, se le añade un *byte* al final con valor 1, y el resto se rellena con ceros.

Seguidamente, cada uno de esos bloques $m_i = \{m_{i,0}, m_{i,1}, \ldots, m_{i,16}\}$, donde cada $m_{i,j}$ es un *byte*, se convierte a número entero según el criterio *little-endian* (el primer *byte* es el menos significativo):

$$c_i = m_{i,0} + 2^8 m_{i,1} + 2^{16} m_{i,2} + \cdots + 2^{128} m_{i,16}$$

La clave r es una secuencia de 16 *bytes*, $r = \{r_0, r_1, r_2, \ldots, r_{15}\}$, con las siguientes restricciones:

- r_3, r_7, r_{11} y r_{15} tienen que estar entre 0 y 15 o, lo que es lo mismo, tener los cuatro bits más significativos a cero.
- r_4, r_8 y r_{12} tienen que ser múltiplos de 4, es decir, tener los dos bits menos significativos a cero.

r, al igual que el valor resultante de $E_k(n)$, también se interpreta como un número entero *little-endian*.

Finalmente, siendo q el número de bloques en los que hemos troceado el mensaje m, el valor del *hash* se calcula como:

$$(((c_1 r^q + c_2 r^{q-1} + \ldots c_q r^1) \bmod 2^{130} - 5) + E_k(n)) \bmod 2^{128}.$$

Este algoritmo proporciona un nivel de seguridad muy próximo al del algoritmo simétrico en el que se apoye.

13.6. Cifrados autentificados

Los cifrados autentificados (AE, del inglés *authenticated encryption)* y los cifrados autentificados con datos asociados (AEAD, del inglés *authenticated encryption with associated data)* son aquellos cifrados simétricos que, además de proporcionar confidencialidad, permiten garantizar la integridad y el origen de los datos, ya que proporcionan un valor de verificación que se calcula a partir de la clave, y funciona como una MAC. Los AE se diferencian de los AEAD en que estos últimos autentifican, junto con el mensaje cifrado, una información que se envía sin cifrar, y que representa el contexto del mensaje. Por ejemplo, si ciframos un paquete de red, la cabecera estaría sin cifrar, y el resto cifrado, de modo que el AEAD cifraría sólo la carga útil, y autentificaría la totalidad del paquete.

Este tipo de cifrados puede conseguirse añadiendo una función MAC al algoritmo de cifrado simétrico, que se aplica al criptograma y se envía en claro *(Encrypt and then MAC)*, o que se aplica al texto en claro y se envía en claro *(Encrypt and MAC)* o cifrada *(MAC then Encrypt)* (figura 13.2). Existen modos de operación específicos para cifrados por bloques que incorporan esta característica. Comentaremos brevemente algunos de ellos:

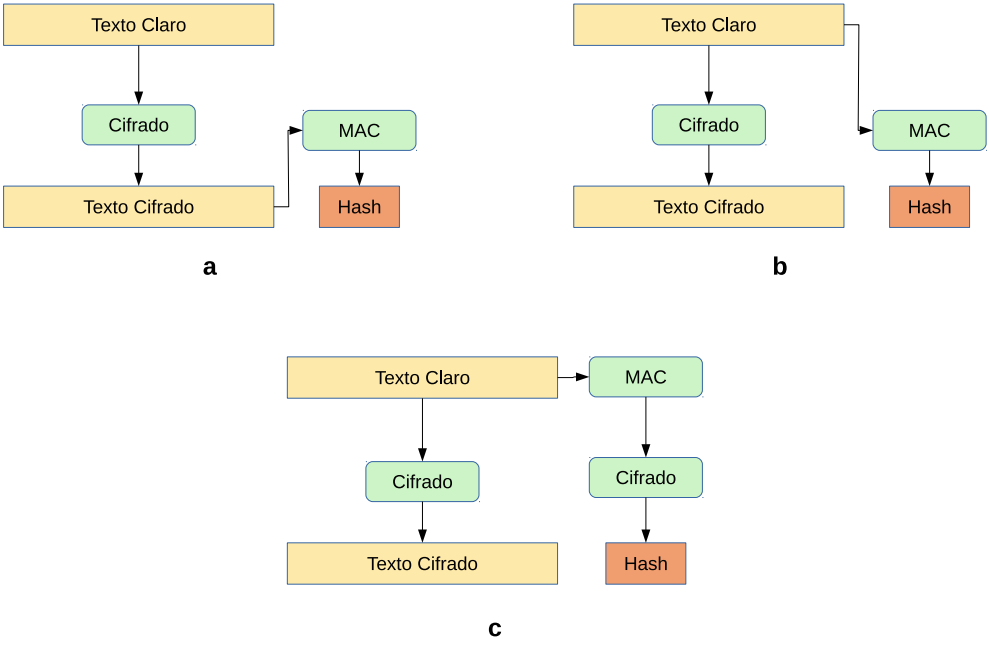

Figura 13.2: Formas de combinar un cifrado y una MAC para obtener un cifrado autentificado. **a**: *Encrypt and then MAC*, **b**: *Encrypt and MAC*, **c**: *MAC then Encrypt*.

- Modo de operación GCM: Es una extensión del modo CTR (sección 11.4.2) que, empleando campos de Galois, permite generar una *etiqueta de autentificación* que garantiza la autenticidad tanto del texto claro original, como de la información de contexto. Tiene la ventaja de que los bloques del criptograma pueden ser procesados en paralelo para calcular la etiqueta de autentificación.

- Modo de operación CCM: Se trata combinar los modos de operación CTR junto con una MAC generada mediante el modo de operación CBC. En concreto, se calcula la MAC sobre el texto claro, se añade a este último y se cifra todo empleando el método CTR, según el esquema *MAC then Encrypt*.

13.7. Funciones KDF

En determinadas circunstancias, puede ser necesario obtener claves de cifrado a partir de fuentes con poca entropía o, lo que es lo mismo, con patrones detectables, como puede ser una contraseña o un secreto compartido mediante el algoritmo de Diffie-Hellman (sección 12.4). Las funciones de derivación de claves (*key derivation functions*, o KDF en inglés) están muy relacionadas con las funciones resumen —de hecho, en algunos casos una función resumen puede actuar directamente como KDF—, y básicamente permiten derivar un valor aparentemente aleatorio a partir de otro valor de entrada, más un valor aleatorio y usualmente único, denominado *sal*. Son de aplicación en diferentes escenarios:

- *Separación de claves:* A partir de una clave maestra, el sistema necesita generar múltiples claves aparentemente independientes, por ejemplo para comunicarse con diferentes servidores. Esto se conseguiría, por ejemplo, aplicando la función KDF a esa clave maestra con diferentes valores de *sal*.

- *Blanqueamiento de clave:* La mayoría de los métodos de cifrado necesitan como clave un bloque de bits que sea indistinguible de un bloque realmente aleatorio (ruido blanco). Si partimos de información no aleatoria en el sentido de que no todas las combinaciones de bits son equiprobables, aunque tenga suficiente entropía, necesitaremos mapearla en un valor aparentemente aleatorio, y posiblemente más corto. En este caso podemos usar como KDF una función *hash* aplicada directamente al valor de entrada.

- *Estiramiento de clave:* Este caso es parecido al anterior, con la diferencia de que partimos de un valor con poca entropía, como una contraseña (ver sección 17.5.1). La función KDF se suele usar entonces para generar un valor que se almacena, junto con la *sal*, y que permitirá verificar la contraseña en el futuro sin guardarla explícitamente. Puesto que los posibles valores para la contraseña serían relativamente pocos, un atacante podría hacer una búsqueda exhaustiva, con un coste computacional dado por la entropía inicial del conjunto. Para aumentar la dificultad de este ataque por la fuerza bruta, haciendo que en la práctica sea equivalente a una búsqueda dentro un espacio de valores mucho mayor, se emplean funciones que sean deliberadamente exigentes desde el punto de vista computacional.

Centrándonos en el problema del estiramiento de clave, una alternativa es emplear como base una función resumen tradicional, y seguir una de las dos estrategias siguientes:

- Aplicar la función *hash* a la contraseña junto con la *sal* en una primera iteración, y luego aplicar la misma función *hash* al resultado de la anterior iteración un número determinado de veces.

- Almacenar solo una parte de la *sal*, de forma que haya que emplear la fuerza bruta a la hora verificar el *hash*. En general, si dejamos s bits sin guardar, tendremos que calcular una media de 2^{s-1} *hashes* hasta dar con el correcto.

En ambos casos, solo hemos aumentado el coste en tiempo de la función, no su consumo de memoria, lo cual permitiría paralelizar los ataques por la fuerza bruta, e incluso desarrollar *hardware* específico para paralelizar y acelerar el proceso a un bajo coste. Para dificultar esto se han diseñado funciones específicas, que demandan un alto coste computacional, tanto en tiempo como en memoria.

El algoritmo bcrypt, reado por Niels Provos y David Mazières en 1999, está basado a su vez en el algoritmo de cifrado simétrico Blowfish, de Bruce Schneier. Este último funciona como una red de Feistel con 18 subclaves y 4 s-cajas de 8×32 (ver secciones 10.1.1 y 10.1.3). Bcrypt usa una versión modificada, mucho más costosa, del algoritmo de derivación de subclaves de Blowfish denominada `EksBlowfishSetup`, que toma como entrada un parámetro de coste c, la *sal* y la contraseña en cuestión. Posteriormente se cifra en modo ECB, con las subclaves y s-cajas resultantes, la cadena `OrpheanBeholderScryDoubt` 64 veces usando el algoritmo Blowfish.

En la función `EksBlowfishSetup` se realiza una expansión de clave que emplea tanto la sal como la contraseña. Luego se repite 2^c veces una expansión doble de clave, que emplea primero la contraseña y el valor 0 como parámetro, y después la *sal* y el 0.

En cualquier caso, este algoritmo solo aumenta el esfuerzo computacional desde el punto de vista del número de cálculos necesarios, pero no de la memoria necesaria, por lo que se puede paralelizar con relativa facilidad.

El algoritmo scrypt, creado por Colin Percival en 2009, requiere además de un alto número de opearciones, una gran cantidad de memoria para su ejecución. Esto se logra creando tablas de acceso pseudoaleatorias, que luego son accedidas en un orden también pseudoaleatorio, por lo que lo más eficiente es almacenarlo todo en memoria.

No obstante, de forma general, suele ser posible establecer un compromiso entre tiempo y memoria, de forma que si reducimos el consumo de memoria aumentamos el coste en tiempo de cálculo, y vicever-

sa. Esto complica significativamente el diseño de funciones que sean verdaderamente intensivas tanto en tiempo como en memoria.

13.7.1. Algoritmo Argon2

En 2015 Alex Biryukov y su equipo ganaron, con su algoritmo Argon2, la competición propuesta por la comunidad criptográfica para encontrar una buena función de derivación de *hashes* de contraseñas, siguiendo la estela de competiciones anteriores organizadas por el NIST, como las que dieron lugar a AES o SHA-3.

Argon2 presenta un esquema similar al de scrypt, creando vectores grandes a los que se accede posteriormente, e intenta dar respuesta a algunas de las problemas que presenta el diseño de algoritmos como este:

- Si realizamos los accesos a los vectores de forma predecible e independiente de la clave, es posible optimizar el proceso y reducir su consumo de memoria.
- Si realizamos los accesos de forma dependiente de la clave, es posible emplear canales *laterales*, analizando el acceso a memoria, para deducir información de la clave. En este sentido hay que tener en cuenta las arquitecturas de los propios procesadores, y de sus mecanismos de optimización del acceso a memoria, como las cachés.
- Si rellenamos mucha memoria, pero accedemos a ella poco, es más fácil realizar optimizaciones en las que se calculen los valores en el momento de usarlos. Por el contrario si rellenamos menos memoria, y accedemos a ella en múltiples ocasiones, resulta más difícil esa optimización.

La solución propuesta por Argon2 consiste en establecer un compromiso entre el acceso a memoria, teniendo en cuenta las peculiaridades de las arquitecturas de Intel y AMD, el uso de múltiples núcleos en

paralelo, y la posibilidad de construir ataques basados en la optimización de tiempo por memoria —sacrificar tiempo a cambio de ahorrar memoria—. Esto se hace a través de dos variantes del algoritmo, con diferentes propiedades:

- *Argon2d*: Es más rápida y usa accesos a memoria dependientes de los datos de entrada. Está orientada a entornos en los que no se consideran una amenaza los ataques de canal lateral, como por ejemplo el minado de criptomonedas.
- *Argon2i*: Usa un acceso a memoria independiente de los datos, y es más adecuada para el procesamiento de contraseñas. Esta versión es más lenta que la anterior, debido a que hace más pasadas sobre la memoria, con objeto de dificultar optimizaciones de tiempo por memoria.

Internamente, Argon2 hace uso de la función resumen Blake2, evolución del algoritmo Blake, uno de los cinco finalistas de la competición de la que salió SHA-3, y que a su vez se basa en el algoritmo de flujo ChaCha (sección 11.5.3).

Capítulo 14

Esteganografía

La palabra *esteganografía* proviene de los térmimos griegos $\sigma\tau\acute{\epsilon}\gamma\omega$ (cubierto) y $\gamma\rho\acute{\alpha}\varphi\epsilon\iota\nu$ (escritura), por lo que literalmente significa *escritura encubierta*. A diferencia del concepto de criptografía, en el que se habla de escritura oculta, aquí se hace mención explícita de la existencia de una capa superior que cubre el mensaje, de forma que éste pueda pasar inadvertido. A lo largo de la Historia podemos encontrar múltiples ejemplos de técnicas esteganográficas: desde la escritura en la cabeza afeitada de un mensajero, cuyo pelo se dejaba crecer antes de enviarlo, hasta los micropuntos empleados en la II Guerra Mundial, que camuflaban páginas enteras de texto microfilmadas en un simple signo de puntuación, dentro de una aparentemente inofensiva carta. Quizás el caso más conocido de esteganografía en lengua castellana sea el poema que aparece al principio de *La Celestina*, en el que el bachiller Fernando de Rojas codificó su nombre y lugar de nacimiento empleando las letras iniciales de cada verso.

Podemos decir que criptografía y esteganografía son técnicas diferentes, e incluso complementarias. Mientras que la primera se encarga de hacer el mensaje ilegible frente a agentes no autorizados, sin preocuparse de que éste pueda tener un aspecto claramente reconocible, la segunda provee mecanismos para hacer que el texto resulte indetecta-

ble, independientemente de que se encuentre cifrado o no. De hecho, en canales de comunicación inseguros, la simple destrucción del mensaje por parte de un atacante puede ser suficiente para sus propósitos, independientemente de que pueda o no acceder a sus contenidos. En este sentido, puede afirmarse que la esteganografía ha sido la técnica más ampliamente utilizada para escapar de la censura prácticamente desde que existe la propia escritura.

Desde un punto de vista más formal, la esteganografía toma un *mensaje anfitrión*, y lo modifica hasta encontrar otro mensaje diferente con el mismo *significado*[1]. Ese proceso de modificación se hace a partir del *mensaje huésped* que queremos ocultar, de forma que únicamente aquellos que conozcan el proceso seguido para su ocultación puedan recuperarlo de manera satisfactoria. En función de la naturaleza del mensaje anfitrión (un texto ASCII, una imagen JPEG, un fragmento de sonido MP3, etc.), cambiará radicalmente el concepto de *significado*, y por lo tanto los procesos de modificación que permitirán alojar el huésped sin despertar sospechas.

Una combinación adecuada de criptografía y esteganografía puede permitir que, aunque el atacante conozca por completo el mecanismo de ocultación del huésped en el anfitrión, únicamente recupere algo cuyas propiedades estadísticas son iguales a las del ruido blanco, por lo que el autor del mensaje podrá repudiarlo (ver sección 2.6), evitando que se le obligue a facilitar sus claves criptográficas o se le someta a represalias, ya que resultará matemáticamente imposible demostrar la propia existencia del mensaje.

[1] En algunos casos, el mensaje anfitrión puede ser generado a partir del mensaje que se pretende ocultar. En esas situaciones, la única condición que el mensaje generado debe cumplir es que posea algún sentido, al menos aparentemente.

14.1. Métodos esteganográficos

Dedicaremos esta sección a comentar brevemente y de forma general algunas técnicas esteganográficas. Dependiendo de la naturaleza del mensaje anfitrión, los bits que lo componen serán interpretados de una u otra forma, y por lo tanto tendremos que actuar de manera diferente para poder ocultar información en él. Distinguiremos entre archivos de texto, en los que el mensaje es una secuencia de letras, y archivos que representan algún tipo de señal, ya sea unidimensional —sonido—, bidimensional —imagen—, o tridimensional —vídeo—, que denominaremos genéricamente *archivos multimedia*.

14.1.1. En archivos de texto

En un archivo de texto, en general, cada *byte* viene asociado a una letra, un número, un símbolo o un carácter de control. Por ejemplo, si empleamos la codificación ASCII, la letra *"a"* se corresponde con el valor numérico 97. Es posible por tanto considerar el mensaje anfitrión como una secuencia de caracteres, en lugar de tomarlo colmo una secuencia de bits. Podemos entonces actuar de dos maneras: modificar un texto existente en función del mensaje que queremos ocultar, sin alterar su significado, o bien generar un texto aparentemente inocuo a partir del mensaje huésped.

En el primer caso podemos jugar con los caracteres de control, introduciendo espacios o retornos de carro superfluos que no alteren el significado del mensaje anfitrión. Por ejemplo, si queremos codificar un 1, podríamos introducir un espacio doble entre dos palabras consecutivas, y un espacio simple si queremos representar un 0. De esta forma será relativamente fácil introducir un mensaje huésped de tantos bits de longitud como huecos entre palabras contenga el mensaje anfitrión.

En el segundo caso haremos uso un generador de frases, programado con una serie de reglas gramaticales y un vocabulario más o menos

extenso, que empleará el mensaje huésped como guía para generar oraciones correctas gramaticalmente. El destinatario utilizaría un analizador léxico–sintáctico para deshacer la operación. Existen aplicaciones que generan, a través de este método, mensajes con apariencia de correos basura (*spam*), capaces de pasar desapercibidos entre los cientos de millones de correos de este tipo que cada día viajan por Internet.

14.1.2. En archivos multimedia

Un archivo *multimedia* representa usualmente la digitalización de algún tipo de señal analógica. Dicha señal analógica, de carácter *continuo* —con infinitos posibles valores en cada instante infinitesimal de tiempo— se convierte en una serie de números *discretos* —que sólo pueden tomar un número finito de valores en un número finito de instantes en el tiempo—. En el caso del sonido (figura 14.1), los niveles de presión en el aire se miden un número fijo de veces por segundo (frecuencia de muestreo), y se aproximan con números enteros (precisión). Por ejemplo, en un CD de audio la frecuencia de muestreo es de 44.100Hz, y se emplea un número entero de 16 bits para representar el nivel de señal en cada canal, lo cual permite representar 65.536 niveles distintos. Eso nos deja un total de 176.400 *bytes* por cada segundo de sonido.

Cuando se trata de representar imágenes (figura 14.2), éstas se subdividen en una matriz de $m \times n$ puntos —píxeles—, y para cada uno de ellos se almacena un valor entero, que representará el nivel de gris si es una imagen monocromática, o un vector de tres valores si es una imagen en color, que representará usualmente los niveles de rojo (R), verde (G) y azul (B) del píxel en cuestión. En el caso de un vídeo, añadiríamos una tercera dimensión, correspondiente al tiempo.

En general, un archivo multimedia acaba convirtiéndose en una secuencia de números que viene a representar una imagen estática o en movimiento, un sonido, o una combinación de todo lo anterior. Como el lector ya habrá advertido, almacenar directamente los valores reco-

gidos en el proceso de digitalización dará lugar a ficheros extremadamente grandes. Por ejemplo, una hora de sonido con calidad CD ocupa unos 600 MB, mientras que una imagen en color de 2000×2000 píxeles tiene un tamaño aproximado de 12MB. Por esta razón la inmensa mayoría de los formatos de almacenamiento de imagen y sonido emplean técnicas de compresión, que a su vez pueden dividirse en dos tipos:

- *Sin pérdida*. El algoritmo de compresión permite recuperar exactamente los mismos datos que fueron obtenidos para cada una de las muestras durante en el proceso de digitalización. Estas técnicas trabajan eliminando la redundancia de la cadena de bits correspondiente (ver sección 3.6), de forma que al descomprimirla obtenemos una copia idéntica a la original.
- *Con pérdida*. El algoritmo no permite recuperar con exactitud los valores de cada muestra, y por lo tanto da lugar a una imagen o sonido distintos del original, aunque para el ojo u oído humanos la diferencia es apenas perceptible. Estas técnicas permiten obtener grados de compresión mucho mayores que las anteriores, e incluso establecer un nivel de compromiso entre el tamaño del archivo resultante y el grado de degradación de la imagen o sonido originales.

Archivos multimedia sin pérdida

En este caso, después del proceso de digitalización se aplica un algoritmo de compresión, que da lugar al fichero de imagen o sonido. Este fichero será nuestro mensaje anfitrión. Cuando se desee representar el contenido del archivo, se usará un algoritmo de descompresión, que permitirá recuperar exactamente los valores originales obtenidos durante el proceso de muestreo de la señal correspondiente, y se enviará el resultado al dispositivo que corresponda. A la hora de camuflar nuestro mensaje huésped, actuaremos directamente sobre los valores de muestreo, antes del proceso de compresión, ya que éstos son

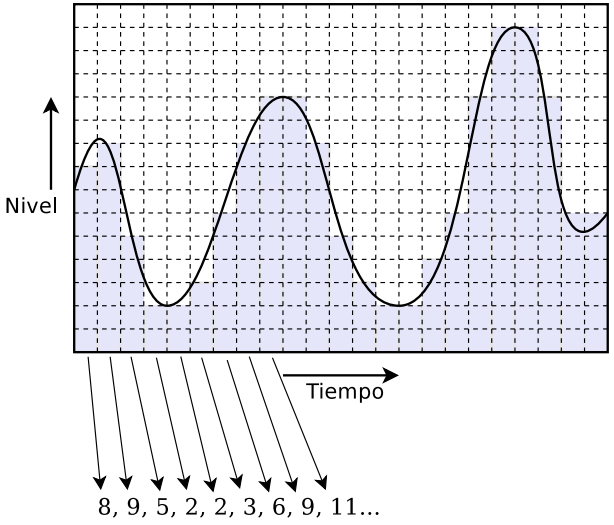

Figura 14.1: Proceso de muestreo de una señal de audio.

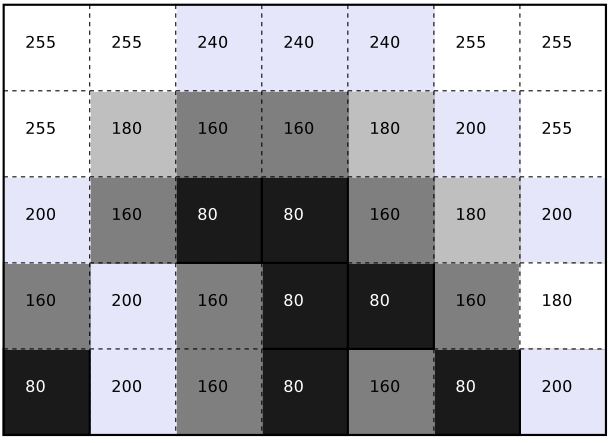

Figura 14.2: Proceso de muestreo de una imagen monocromática. En el caso de una imagen RGB, en cada píxel se recogerían tres valores: el nivel de rojo (R), el nivel de verde (G), y el nivel de azul (B).

únicamente una secuencia de datos numéricos. Obsérvese que todo lo que digamos en este apartado será también válido para archivos multimedia sin comprimir.

En el caso de un archivo de sonido, una técnica bastante efectiva consiste en manipular los bits menos significativos de cada muestra, sustituyéndolos por los bits de nuestro mensaje huésped. Eso introducirá una distorsión en el sonido resultante prácticamente imperceptible para el oído humano. Por ejemplo, si usamos los dos bits menos significativos de cada muestra en un archivo con calidad CD, podríamos llegar a introducir unos 75 MB de datos por cada de sonido, con una distorsión en la señal manipulada inferior al 0.01 %.

En lo que respecta a los archivos de imagen podemos actuar de la misma forma, si bien en este caso existen técnicas que, explotando la organización espacial de los píxeles de la imagen, permiten detectar zonas en las que la distorsión provocada por el mensaje huésped resulta menos perceptible para el ojo humano, con lo que se consigue ocultar mayor cantidad de información de forma satisfactoria en una misma imagen.

Archivos multimedia con pérdida

Los formatos de almacenamiento de imagen y sonido con pérdida dan lugar a secuencias de valores numéricos distintas de las obtenidas en el proceso de digitalización, por lo que si manipulamos directamente los bits de la secuencia, el proceso de compresión destruirá la información que hayamos ocultado. Describiremos brevemente el funcionamiento básico de estos algoritmos de compresión, para poder entender cómo deben tratarse si queremos usarlos a modo de huésped.

Las llamadas *transformadas* son herramientas matemáticas que permiten representar cualquier función —continua o discreta— a través de una serie de coeficientes. En general, necesitaremos un número infinito de coeficientes en el caso continuo, y tantos coeficientes como valores en el caso discreto, para poder representar de manera absolu-

tamente precisa la señal. La ventaja que tiene trabajar con transformadas es que la mayor parte de la información suele estar *concentrada* en un número relativamente pequeño de coeficientes, por lo que podemos obtener buenas aproximaciones de la señal original a partir de un subconjunto de la totalidad de sus coeficientes, que serán más o menos precisas en función del número de coeficientes que conservemos.

Los formatos de compresión de archivos multimedia más comunes emplean distintos tipos de transformada, como la Transformada de Fourier, o la Transformada Discreta del Coseno. En el caso de los archivos JPEG, la imagen se divide en regiones de 8×8 píxeles de tamaño, y en el de los archivos MP3 en *marcos* de un número determinado de muestras, asociados a un intervalo determinado de tiempo. A cada trozo se le aplica una transformada, y los coeficientes resultantes se truncan —cuantizan—, siguiendo unas pautas perceptuales[2]. Finalmente se aplica un algoritmo de compresión sin pérdida al resultado, y se obtiene el fichero final.

Si queremos ocultar un mensaje dentro de uno de estos archivos, habrá que manipular directamente los coeficientes cuantizados, e introducir en ellos los bits del mensaje. Como es lógico, ya que los bits que se preservan en los coeficientes son los que mayor cantidad de información transportan, también serán más sensibles a alteraciones, por lo que podremos ocultar muchos menos bits del mensaje huésped si queremos mantener un nivel de distorsión en el resultado final que sea realmente imperceptible.

Una segunda aproximación para ocultar información en archivos multimedia consiste en manipular la imagen o el sonido originales, antes de pasar por el algoritmo de compresión, introduciendo modificaciones sutiles que puedan sobrevivir a diversos tipos de transformaciones, como recortes, escalados, rotaciones, diferentes compresiones y recompresiones con pérdida, etc. Estas técnicas resultan más complejas y se suelen basar en superponer a la señal original alguna fun-

[2]Está demostrado que los sentidos del ser humano son más sensibles a determinadas características, por lo que a la hora de truncar los coeficientes, se les da prioridad a los que mejor percibimos, para que las distorsiones sean poco perceptibles.

ción global cuyo espectro[3] esté en frecuencias muy bajas, tal que el ser humano no la perciba y las degradaciones surgidas del proceso de compresión no la destruyan. Usualmente, las técnicas de *marcas de agua* (*watermarking*) sobre contenidos multimedia digitales se basan en estos principios.

14.2. Detección de mensajes esteganografiados

Hasta ahora hemos comentado una serie de métodos más o menos sutiles para ocultar información en diferentes tipos de archivo, pero la cuestión realmente importante es la siguiente: ¿es posible detectar si un mensaje cualquiera alberga información esteganografiada? Esta pregunta resulta crucial, ya que a la postre el objetivo de las técnicas esteganográficas es impedir –o al menos dificultar– esa detección.

Para decidir si un mensaje cualquiera es en realidad el anfitrión de otro, tendremos que ponernos en el lugar de quien lo generó, y seleccionar en él los bits que consideremos más adecuados para ocultar información. Una vez aislados, habrá que realizar un estudio acerca de su distribución estadística típica en un mensaje normal, y comparar los resultados con los valores extraídos del archivo sospechoso. Podremos considerar que hemos encontrado un indicio de la presencia de un mensaje oculto si los resultados obtenidos difieren de los que presenta un mensaje *limpio*. Esto nos obliga a conocer, al menos a grandes rasgos, qué métodos esteganográficos ha podido usar nuestro adversario, y a diseñar pruebas específicas para detectar cada uno de ellos.

Evidentemente, la recuperación completa del mensaje huésped es una prueba irrefutable de que éste existe, que siempre puede llegar a conseguirse. No obstante, si el mensaje huésped se encuentra cifrado, podemos considerar esa posibilidad fuera del alcance de quien ana-

[3] El espectro de una señal es el rango de los coeficientes no nulos de su transformada.

liza el mensaje anfitrión. En ese caso, la *prueba* de la presencia de un mensaje oculto dentro de otro tendrá que basarse en las propiedades estadísticas de la información analizada. En general, se puede conseguir que resulte imposible, desde un punto de vista matemático, demostrar que un mensaje alberga información esteganografiada. Para ello han de seguirse las siguientes pautas:

1. Analizar estadísticamente los bits del mensaje anfitrión que van a ser alterados durante el proceso de esteganografiado.
2. Cifrar el mensaje huésped antes de introducirlo en el anfitrión.
3. Manipular el mensaje huésped, una vez cifrado, para que presente una distribución estadística similar a la de los bits analizados previamente.

El caso ideal consistiría en seleccionar un subconjunto de bits del mensaje anfitrión que posea una distribución estadísticamente aleatoria, y cuya alteración no resulte perceptible en el fichero resultante, para luego sustituirlos por una versión comprimida y cifrada del mensaje huésped, que también deberá presentar una distribución estadísticamente aleatoria.

Capítulo 15

Pruebas de conocimiento cero

15.1. Introducción

Cuando un agente A pretende convencer a otro B de que posee una cierta información X, la estrategia más simple e intuitiva consiste en que A proporcione a B el valor de X. Sin embargo, a partir de ese momento, B conocerá el secreto de A y podrá contárselo a todo el mundo. O lo que es peor: un atacante C puede espiar la comunicación entre A y B y robar el secreto.

Si bien este problema puede ser resuelto a partir de criptografía asimétrica (capítulo 12) o de funciones resumen (capítulo 13), como veremos en el capítulo 17, existe un mecanismo que, a través de un proceso interactivo, permite a A probar a B que posee el secreto en cuestión, sin revelar ningún tipo de información acerca de X en el proceso. En general, estas técnicas, conocidas como *Pruebas de Conocimiento Cero*, suelen tener modestos requerimientos computacionales en comparación con otros protocolos, de ahí su interés.

Una prueba de conocimiento cero se basa en la formulación por parte de B de una serie de preguntas. Si A conoce el valor de X, podrá responder correctamente a todas ellas; en caso contrario, tendrá una

probabilidad determinada de *acertar* la respuesta en cada caso —usualmente del 50 %—. De esta forma, la probabilidad de que un impostor logre superar una prueba de n preguntas es de $1/2^n$. Otra característica importante es que ninguno de los mensajes que intercambian A y B a lo largo del proceso aporta información a un eventual espía C sobre el valor de X. Finalmente, es necesario recalcar que la secuencia de preguntas debe ser diferente y aleatoria en cada caso, de forma que C no pueda *memorizar* la secuencia concreta de respuestas y así engañar a B.

15.2. Elementos

A la hora de describir el funcionamiento de una prueba de conocimiento cero, es interesante definir los distintos elementos que la componen, con objeto de facilitar su comprensión:

- Un *Secreto* X, que puede ser una contraseña, una clave privada, o cualquier otra información cuya posesión se quiera demostrar.
- Un *Demostrador* (al que llamaremos David a partir de ahora), que es quien pretende demostrar que posee el secreto X.
- Un *Verificador* (al que llamaremos Víctor), que es quien debe asegurarse de que David está en posesión de X.
- Un *Problema* —usualmente algún problema matemático computacionalmente costoso— sobre el que basar cada una de las preguntas que se formularán a lo largo del proceso.

Adicionalmente, habremos de tener en cuenta la posibilidad de que existan agentes que *escuchen* los mensajes que intercambian Víctor y David, e incluso que los eliminen, los manipulen, o añadan mensajes falsos, para poder garantizar la efectividad de estas pruebas.

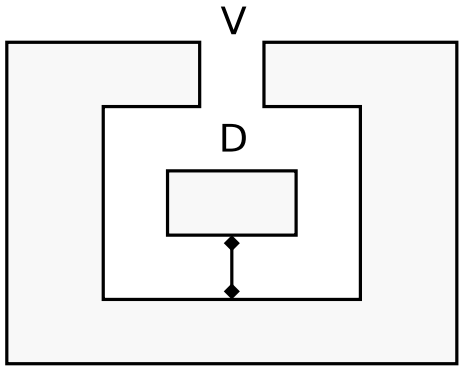

Figura 15.1: La *cueva de conocimiento cero*.

15.3. Desarrollo

Un ejemplo típico en la literatura para explicar las pruebas de conocimiento cero se basa en una variante circular de la cueva de Alí Babá (figura 15.1), tal que la entrada se bifurca y las dos ramas resultantes quedan comunicadas por una puerta. Supongamos que David conoce la contraseña que abre la puerta, y quiere convencer a Víctor de ello sin decírsela. David se introduciría por una de las ramas de la cueva sin que Víctor supiera cuál, seguidamente Víctor entraría y pediría a David que saliera por una de las ramas, escogida de forma aleatoria. Si David conoce la contraseña podrá abrir la puerta cuando lo necesite para salir por el lugar solicitado, pero si la ignora sólo podrá salir por el lugar correcto la mitad de las veces. Tras repetir un número razonable de veces este proceso, Víctor quedará convencido de que David posee la contraseña correcta, sin conocerla él mismo. Además, la observación de todo el proceso por parte de un tercero no servirá a nadie para poder hacerse pasar por David.

Este ejemplo no es más que una analogía para entender mejor una prueba de conocimiento cero. De hecho, a David le hubiera bastado con entrar por una rama y salir por la otra para convencer a Víctor.

Veamos ahora el proceso desde un punto de vista más completo y formal. David, para demostrar que se encuentra en posesión del secreto X, construye un problema matemático M, computacionalmente difícil de resolver, de forma que X constituya una solución para M —nótese que David parte de la solución para elaborar el problema—. Se produce entonces el siguiente diálogo:

1. David transforma M para construir un nuevo problema matemático M', cuya solución X' se podrá calcular fácilmente a partir de X, y lo envía a Víctor.
2. Víctor genera un bit aleatorio B y lo remite a David.
3. Si:
   - $B = 0$, David demuestra la relación entre M y M', sin dar la solución a M'.
   - $B = 1$, David proporciona la solución X' del problema M', sin revelar la relación entre M y M'.

Observando el protocolo con atención, puede verse que la única forma de que David pueda responder correctamente a ambas preguntas es que posea la solución X. David únicamente revela, o bien la relación entre M y M', o bien el valor de X', y que cada una de estas cosas por separado resulta inútil para calcular X.

15.4. Modos de operación

Fundamentalmente, una prueba de conocimiento cero puede ser planteada de tres formas diferentes:

- *Interactiva*, de forma que David y Víctor generan e intercambian cada mensaje en línea, realizando los cálculos para el siguiente sólo cuando han recibido el anterior.

- *Paralela*, en la que Víctor genera un *paquete* de preguntas, las envía a David, y éste las contesta todas a la vez.

- *Fuera de línea*, en la que David genera una serie de problemas $\mathcal{M}$, usa una función resumen (capítulo 13) aplicada sobre un mensaje concreto m concatenado con $\mathcal{M}$, emplea los bits del resultado como los diferentes valores de B, y añade a $(m, \mathcal{M})$ el conjunto $\mathcal{S}$ de soluciones correspondientes a los problemas. De esta manera cualquiera podrá verificar que el único que pudo generar el mensaje final $(m, \mathcal{M}, \mathcal{S})$ fue David, lo cual puede funcionar como firma digital de David sobre el mensaje m (ver capítulo 17).

15.5. Conocimiento cero sobre grafos

Existen muchos problemas matemáticos susceptibles de ser empleados como base para un protocolo de conocimiento cero. Uno de ellos es el del homomorfismo de grafos[1]. Dados dos grafos con el mismo número de nodos, averiguar si reordenando los nodos del primero se puede obtener una copia exacta del segundo —son homomorfos—, es un problema computacionalmente intratable. Para construir una prueba de conocimiento cero basada en grafos, David partiría de un grafo G_1, y reordenando los nodos en función del valor de X calcularía G_2. De esta forma, el secreto X queda asociado a la correspondencia entre G_1 y G_2. El protocolo quedaría como sigue:

1. David reordena los nodos de G_1, y obtiene un grafo H, que será homomorfo a G_1 y G_2.

2. Víctor genera el bit aleatorio B.

3. David envía a Víctor la correspondencia entre G_1 y H si $B = 1$, o la correspondencia entre G_2 y H si $B = 0$.

[1] Un grafo es un conjunto de puntos o nodos conectados entre sí por arcos

Obsérvese que para conocer la correspondencia entre G_1 y G_2 —o sea, el valor del secreto X— son necesarias simultáneamente las correspondencias entre G_1 y H, y entre H y G_2. Puesto que David únicamente revela una de las dos, el protocolo funciona correctamente.

15.6. Ataques de intermediario

Al igual que cualquier otra técnica, las pruebas de conocimiento cero también presentan puntos débiles, que deben ser conocidos para su correcta utilización. En este caso la mayoría de los ataques pueden producirse a través de intermediario.

Supongamos que Andrés quiere suplantar a David frente a Víctor. Para conseguir su objetivo actuaría de la siguiente forma:

1. Andrés, haciéndose pasar por David, solicita a Víctor realizar la prueba de conocimiento cero.
2. Andrés, haciéndose pasar por Víctor, informa a David de que debe realizar la prueba.
3. David genera el problema correspondiente, Andrés lo recoge y lo envía a Víctor como si lo hubiera generado él.
4. Víctor genera el bit aleatorio B, lo envía Andrés, y Andrés se lo pasa a David.
5. David responde, y Andrés reenvía la respuesta a Víctor.
6. El proceso se repite tantas veces como requiera Víctor.

Como resultado tenemos que Andrés ha logrado convencer a Víctor de que posee el mismo secreto que David, —aunque no lo conce, ni ha ganado información sobre él—. Este ataque resulta indetectable, y puede dar lugar a importantes brechas de seguridad, especialmente cuando la prueba de conocimiento cero se usa para comprobar la identidad de alguien.

Parte IV

Aplicaciones criptográficas

Capítulo 16

Protocolos de Comunicación Segura

16.1. Introducción

Quizás la aplicación más antigua de la Criptografía sea precisamente la de establecer canales de comunicaciones seguros entre dos puntos. Desde un soldado galopando a través de territorio enemigo hasta un haz láser, pasando por un hilo telegráfico, el ser humano ha empleado infinidad de medios para poder enviar sus mensajes, cada uno de ellos con sus propias peculiaridades. Pero si hay una característica que podemos considerar común a todos los canales de comunicaciones, es la ausencia de control que sobre el mismo poseen ambos interlocutores. En el caso del jinete, sería muy interesante poder crear un pasillo de territorio *amigo* a lo largo de todo su trayecto, pero en ese caso su propia tarea carecería prácticamente de sentido. En general, hemos de considerar que nuestros mensajes son depositados en un medio ajeno a nosotros —y usualmente hostil—, y que los medios que apliquemos para su protección deben ser válidos en los casos más desfavorables.

Un mensaje liberado en un medio hostil se enfrenta principalmente a dos peligros:

- Acceso por agentes no autorizados. En un medio sobre el que no podemos ejercer ningún control, esta posibilidad debe tomarse muy en serio. Tanto que en lugar de suponer que el *enemigo* puede acceder al mensaje, hemos de dar por hecho que va a hacerlo. Por lo tanto, nuestros sistemas de protección deben centrarse en garantizar que el mensaje resulte ininteligible a nuestro atacante.
- Alteraciones en el mensaje. Este problema puede llegar a ser mucho peor que el anterior, ya que si recibimos un mensaje que ha sido modificado y lo damos por bueno, las consecuencias para la comunicación pueden ser catastróficas. En este sentido, las alteraciones pueden aplicarse tanto sobre el mensaje propiamente dicho, como sobre la información acerca de su verdadera procedencia.

La Criptografía, como ya hemos visto en anteriores capítulos, proporciona mecanismos fiables para evitar los dos peligros que acabamos de mencionar. En general, cada una de las aplicaciones concretas que necesiten de estas técnicas poseerá unas características específicas, por lo que en cada caso habrá una combinación de algoritmos criptográficos que permitirá proporcionar al sistema el nivel de seguridad necesario. Estas combinaciones de algoritmos se estructurarán finalmente en forma de protocolos, para proporcionar métodos de comunicación segura normalizados.

16.2. Protocolos TCP/IP

El conjunto básico de protocolos sobre los que se construye la red Internet se conoce popularmente como TCP/IP, agrupación de los nombres de dos de los elementos más importantes, que no los únicos, de la familia: TCP *(Transmission Control Protocol)* e IP *(Internet Protocol)*.

El modelo de comunicaciones sobre el que se basa Internet se estructura en forma de *capas* apiladas, de manera que cada una de ellas se comunica con las capas inmediatamente superior e inferior, logrando diversos niveles de abstracción, que permiten intercambiar información de forma transparente entre ordenadores. La consecuencia más importante de este enfoque es que dos dispositivos cualesquiera, que pueden estar conectados a Internet por medios totalmente distintos —fibra óptica, cable de cobre, láser, ondas electromagnéticas...—, y separados por multitud de enlaces diferentes —satélite, cables submarinos, redes inalámbricas...—, pueden conectarse entre ellos simplemente con que dispongan de una implementación de TCP/IP.

A diferencia del modelo OSI, que consta de siete capas, denominadas *aplicación*, *presentación*, *sesión*, *transporte*, *red*, *enlace* y *física*, los protocolos TCP/IP se organizan únicamente en cinco (figura 16.1). Aunque la correspondencia no es exacta, podemos decir que, básicamente, los tres niveles superiores del modelo OSI se agrupan en el nivel de aplicación de TCP/IP. Comentaremos brevemente cada uno de ellos:

- *Capa física*. Describe las características físicas de la comunicación, como son el medio empleado, los voltajes necesarios, la modulación empleada, etc.
- *Capa de enlace*. Indica cómo los paquetes de información viajan a través del medio físico, indicando qué campos de bits se añaden a éstos para que puedan ser reconocidos satisfactoriamente en destino. Ejemplos de protocolos de enlace: Ethernet, 802.11 WiFi, Token Ring, etc.
- *Capa de red*. En ella se ubica el protocolo IP, cuyo propósito consiste en hacer llegar los paquetes a su destino a través de una única red. Existen algunos protocolos de mayor nivel, como ICMP o IGMP, que aunque se construyen sobre IP, también pertenecen a la capa de red, a diferencia de lo que ocurre en el modelo OSI.
- *Capa de transporte*. Su propósito es garantizar que los paquetes han llegado a su destino, y en el orden correcto. El protocolo más

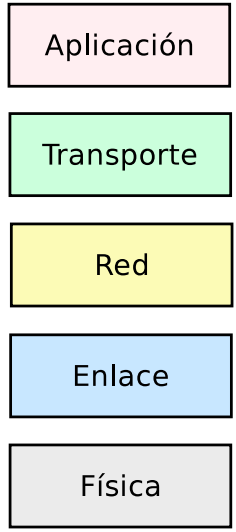

Figura 16.1: Esquema del conjunto de protocolos TCP/IP, en los que se basa la red Internet.

importante en este nivel es TCP, pero existen otros como UDP, DCCP o RTP.

- *Capa de aplicación*. Esta es la capa a la que acceden de forma directa la mayoría de las aplicaciones que usan Internet. En ella se reciben los datos, que son pasados a las capas inferiores para que sean enviados a su destino. A este nivel pertenecen prococolos tales como HTTP, FTP, SSH, HTTPS, IMAP, DNS, SMTP, IRC, etc.

En la práctica, podemos encontrar protocolos encaminados a obtener comunicaciones seguras en prácticamente todos los niveles de este esquema. En las próximas secciones comentaremos brevemente algunos de ellos.

Los distintos protocolos de comunicación segura pueden ser utilizados para construir las denominadas redes privadas virtuales. Una

red privada virtual, en inglés VPN *(Virtual Private Network)* es una red de comunicaciones privada construida sobre una red pública. Hacia los usuarios se comporta como si de una red interna se tratase, ofreciendo acceso únicamente a aquellos que estén autorizados, y resultando inaccesible para los demás, cuando en realidad todas las conexiones se realizan a través de Internet.

16.3. Protocolo SSL

El protocolo SSL (*Secure Sockets Layer*), desarrollado originalmente por la empresa Netscape, permite establecer conexiones seguras a través de Internet, de forma sencilla y transparente. Se sitúa en la capa de aplicación (figura 16.1), directamente sobre el protocolo TCP, y aunque puede proporcionar seguridad a cualquier aplicación que corra sobre TCP, se usa principalmente para proporcionar seguridad a los protocolos HTTP *(web)*, SMTP *(email)* y NNTP *(news)*, dando lugar en el primero de los casos a los servidores *web* seguros, cuya URL comienza por el prefijo `https://`. Su fundamento consiste en interponer una fase de codificación de los mensajes antes de enviarlos a través de la red. Una vez que se ha establecido la comunicación, cuando una aplicación quiere enviar información a otra computadora, la capa SSL la recoge y la codifica, para luego enviarla a su destino a través de la red. Análogamente, el módulo SSL del otro ordenador se encarga de decodificar los mensajes y se los pasa como texto claro a la aplicación destinataria.

SSL también incorpora un mecanismo de autentificación que permite garantizar la identidad de los interlocutores. Típicamente, ya que este protocolo se diseñó originalmente para establecer comunicaciones *web*, el único que suele autentificarse es el servidor, aunque también puede realizarse una autentificación mutua.

Una comunicación a través de SSL implica tres fases fundamentalmente:

- Establecimiento de la conexión y negociación de los algoritmos criptográficos que van a usarse en la comunicación, a partir del conjunto de algoritmos soportados por cada uno de los interlocutores.
- Intercambio de claves, empleando algún mecanismo de clave pública (ver sección 12.4), y autentificación de los interlocutores a partir de sus certificados digitales (ver capítulo 17).
- Cifrado simétrico del tráfico.

Una de las ventajas de emplear un protocolo de comunicaciones en lugar de un algoritmo o algoritmos concretos, es que ninguna de las fases del protocolo queda atada a ningún algoritmo, por lo que si en el futuro aparecen algoritmos mejores, o alguno de los que se emplean en un momento dado quedara comprometido, el cambio se puede hacer sin modificar el protocolo. Las implementaciones típicas de SSL soportaban algoritmos como RSA, Diffie-Hellman o DSA para la parte asimétrica (capítulo 12); RC2, RC4, IDEA, DES, TripleDES o AES para la simétrica (capítulos 10 y 11), y como funciones resumen (capítulo 13) SHA-1 o MD5.

Las ventajas de protocolos como SSL son evidentes, ya que liberan a las aplicaciones de llevar a cabo las operaciones criptográficas antes de enviar la información, y su transparencia permite usarlo de manera inmediata sin modificar apenas los programas ya existentes.

Hasta diciembre de 1999, debido a las restricciones de exportación de material criptográfico que había en los EE.UU., la mayoría de los navegadores incorporaban SSL con un nivel de seguridad bastante pobre (claves simétricas de 40 bits), lo cual era a todas luces insuficiente, incluso para los estándares de aquella época.

Existen implementaciones de SSL que permiten construir los denominados *túneles SSL*, que permiten dirigir cualquier conexión a un puerto TCP a través de una conexión SSL previa, de forma transparente para las aplicaciones que se conectan.

16.4. Protocolo TLS

TLS (descrito inicialmente en el documento *RFC 2246*) es un protocolo desarrollado en 1999 por la IETF *(Internet Engineering Task Force)*, y basado en la versión 3.0 de SSL, si bien con una serie de mejoras que lo hacen incompatible con este último. Una de las ventajas que proporciona sobre SSL es que puede ser *iniciado* a partir de una conexión TCP ya existente, lo cual permite seguir trabajando con los mismos puertos que los protocolos no cifrados. Mientras que SSL es un protocolo incompatible con TCP, lo cual significa que no podemos establecer una conexión de un cliente TCP a un servidor SSL ni al revés, y por tanto es necesario diferenciarlos utilizando distintos números de puerto (80 para un servidor *web* normal y 443 para un servidor *web* sobre SSL), con TLS puede establecerse la conexión normalmente a través de TCP y el puerto 80, y luego activar sobre el mismo el protocolo TLS.

En este protocolo se emplea una serie de medidas de seguridad adicionales, encaminadas a protegerlo de distintos tipos de ataque, en especial de los de intermediario (sección 12.2):

- Uso de funciones MAC en lugar de funciones MDC únicamente (ver capítulo 13).
- Numeración secuencial de todos los campos que componen la comunicación, e incorporación de esta información al cálculo de los MAC.
- Protección frente a ataques que intentan forzar el empleo de versiones antiguas —menos seguras— del protocolo o cifrados más débiles.
- El mensaje que finaliza la fase de establecimiento de la conexión incorpora una signatura *(hash)* de todos los datos intercambiados por ambos interlocutores.

En agosto de 2018 se definió la versión 1.3 de TLS, a través del RFC 8446, que incorpora diferencias significativas con versiones anteriores,

fundamentalmente encaminadas tanto simplificar el protocolo, como a eliminar componentes que, como la compresión, han revelado ser fuente de problemas. En este sentido, cabe destacar la eliminación del protocolo de intercambio de Diffie-Hellman y de todos los sistemas de cifrado que no sean AEAD (sección 13.6).

Un hecho bastante relevante es que, en sus inicios, el uso de TLS (y antes de SSL) era opcional, siendo la ausencia de cifrado la opción por defecto para comunicaciones a través de Internet. Esta situación fue cambiando paulatinamente, hasta el punto de que entre 2015 y 2018 los principales navegadores comenzaron a marcar como inseguros aquellos sitios que no cifraban las comunicaciones.

16.5. Protocolos IPsec

IPsec es un estándar que proporciona cifrado y autentificación a los paquetes IP, trabajando en la capa de red (figura 16.1). En lugar de tratarse de un único protocolo, IPsec es en realidad un conjunto de protocolos, definidos en diversos *RFC*s (principalmente en el 2401), encaminados a proporcionar autentificación, confidencialidad e integridad a las comunicaciones IP. Su carácter obligatorio dentro del estándar IPv6 —recordemos que en IPv4, la versión más empleada en la actualidad de este protocolo, es opcional— hará con seguridad que la popularidad de IPsec crezca al mismo ritmo que la implantación de la nueva versión del protocolo IP.

IPsec puede ser utilizado para proteger una o más rutas entre un par de ordenadores, un par de *pasarelas de seguridad* —ordenadores que hacen de intermediarios entre otros, y que implementan los protocolos IPsec— o una pasarela y un ordenador. En función del tipo de ruta que se proteja, se distinguen dos modos de operación:

- *Modo túnel*: Se realiza entre dos pasarelas de seguridad, de forma que éstas se encargan de crear una ruta segura entre dos or-

denadores conectados a ellas, a través de la cual viajan los paquetes. De este modo se puede disponer dentro de una red local de un ordenador que desempeñe las labores de pasarela, al que las computadoras de la propia red envíen los paquetes, para que éste les aplique los protocolos IPsec antes de remitirlos al destinatario —o a su pasarela de seguridad asociada—. Este modo permite interconectar de forma segura ordenadores que no incorporen IPsec, con la única condición de que existan pasarelas de seguridad en las redes locales de cada uno de ellos.

- *Modo transporte*: En este caso los cálculos criptográficos relativos a los protocolos IPsec se realizan en cada extremo de la comunicación.

Básicamente, IPsec se compone a su vez de dos protocolos, cada uno de los cuales añade una serie de campos, o modifica los ya existentes, a los paquetes IP:

- Cabecera de autentificación IP, abreviado como AH *(IP Authenticacion Header)*, diseñado para proporcionar integridad, autentificación del origen de los paquetes, y un mecanismo opcional para evitar ataques por repetición de paquetes.
- Protocolo de encapsulamiento de carga de seguridad, o ESP *(Encapsulating Security Payload)* que, además de proveer integridad, autentificación y protección contra repeticiones, permite cifrar el contenido de los paquetes.

Debido a que algunos de los servicios que IPsec proporciona necesitan de la distribución e intercambio de las claves necesarias para cifrar, autentificar y verificar la integridad de los paquetes, es necesario que éste trabaje en consonancia con un conjunto externo de mecanismos que permita llevar a cabo esta tarea, tales como IKE, SKIP o Kerberos.

Capítulo 17

Autentificación, certificados y firmas digitales

17.1. Introducción

Cuando se establece una comunicación de cualquier tipo es necesario poder asegurar que los mensajes no han sufrido alteraciones, es decir, que la información recibida coincide exactamente con la enviada. En muchos casos, existe el requerimiento adicional de conocer la identidad de nuestro interlocutor —sea éste una persona o algún tipo de dispositivo—, para evitar que sea suplantado por un impostor. Denominaremos en general *autentificación* (o autenticación) a las operaciones consistentes en verificar tanto la identidad de nuestro interlocutor como la integridad de los mensajes que de él recibimos.

Independientemente de que la operación de autentificación se lleve a cabo sobre el contenido de una comunicación o sobre los propios interlocutores, ésta puede realizarse en el mismo momento, de forma interactiva —como cuando se introduce una contraseña para acceder a un sistema—, o dejarse pospuesta para ser realizada posteriormente *fuera de línea* —como cuando se firma digitalmente un mensaje, en cuyo

caso la firma puede ser verificada tantas veces como se desee, una vez finalizada la comunicación—.

17.2. Firmas digitales

Una *firma digital* es una secuencia de bits que se añade a una pieza de información cualquiera, y que permite garantizar su autenticidad de forma independiente del proceso de transmisión, tantas veces como se desee. Presenta una analogía directa con la firma manuscrita, y para que sea equiparable a esta última debe cumplir las siguientes propiedades:

- Va ligada indisolublemente al mensaje. Una firma digital válida para un documento no puede ser válida para otro distinto.
- Sólo puede ser generada por su legítimo titular. Al igual que cada persona tiene una forma diferente de escribir, y que la escritura de dos personas diferentes puede ser distinguida mediante análisis caligráficos, una firma digital sólo puede ser construida por la persona o personas a quienes legalmente corresponde.
- Es públicamente verificable. Cualquiera puede comprobar su autenticidad en cualquier momento, de forma sencilla.

La forma más extendida de calcular firmas digitales consiste en emplear una combinación de cifrado asimétrico (capítulo 12) y funciones resumen (capítulo 13). El esquema de funcionamiento queda ilustrado en la figura 17.1.

17.3. Certificados digitales

Un certificado digital es esencialmente una clave pública y un identificador, firmados digitalmente por una *autoridad de certificación*, y su

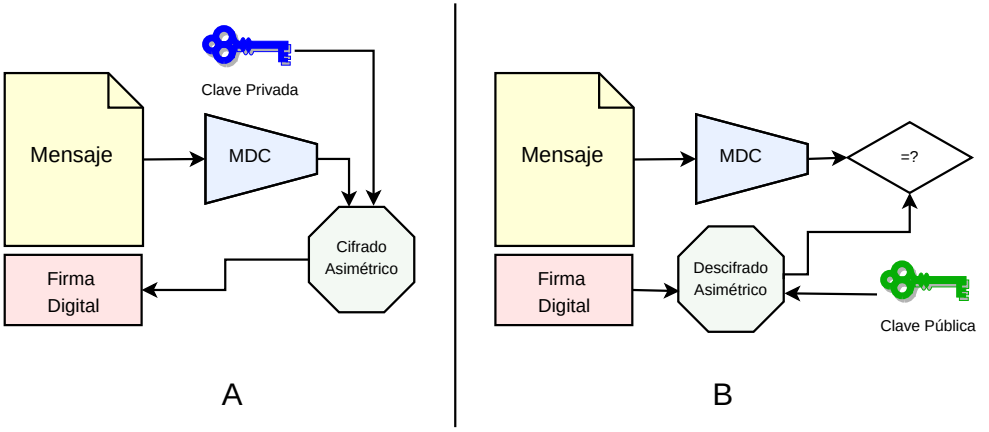

Figura 17.1: Esquema de una firma digital basada en funciones resumen y algoritmos de cifrado asimétricos. A: generación de la firma; B: verificación.

utilidad es demostrar que una clave pública pertenece a un usuario concreto. Evidentemente, la citada autoridad de certificación debe encargarse de verificar previamente que la clave pública es auténtica. En España, por ejemplo, la *Fábrica Nacional de Moneda y Timbre* actúa como autoridad certificadora, firmando las claves públicas de los ciudadanos y generando los certificados digitales correspondientes. Cualquier entidad que disponga de la clave pública de la FNMT estará en condiciones de verificar sus certificados digitales, otorgando la confianza correspondiente a las claves públicas asociadas a los mismos.

17.3.1. Certificados X.509

El formato de certificados X.509 (Recomendación X.509 de CCITT:*"The Directory - Autentication Framework"*. 1988) es uno de los más comunes y extendidos en la actualidad.

El estándar X.509 sólo define la sintaxis de los certificados, por lo

que no está atado a ningún algoritmo en particular, y contempla los siguientes campos:

- Versión.
- Número de serie.
- Identificador del algoritmo empleado para la firma digital.
- Nombre del certificador.
- Periodo de validez.
- Nombre del sujeto.
- Clave pública del sujeto.
- Identificador único del certificador.
- Identificador único del sujeto.
- Extensiones.
- Firma digital de todo lo anterior generada por el certificador.

17.3.2. Certificados de revocación

Cuando una clave pública pierde su validez —por destrucción o robo de la clave privada correspondiente, por ejemplo—, es necesario anularla. Para ello se emplean los denominados *certificados de revocación* que no son más que un mensaje que identifica a la clave pública que se desea anular, firmada por la clave privada correspondiente. De esta forma se garantiza que una clave pública únicamente puede ser revocada por su legítimo propietario —si la clave privada resulta comprometida, al atacante no le interesará revocarla, ya que entonces el material robado perdería su valor—. Como puede verse, para revocar una clave pública es necesario estar en posesión de la privada, por

lo que si perdemos esta última, jamás podremos hacer la revocación. Para evitar estos problemas, conviene seguir una serie de pautas:

- Generar los pares de claves con un período limitado de validez. De esta forma, si no podemos revocar una clave, expirará por sí misma.
- Generar el certificado de revocación junto con el propio par de claves, y almacenarlo en lugar seguro.
- Algunos protocolos permiten nombrar *revocadores* para nuestra clave pública, que podrán generar un certificado de revocación empleando únicamente sus propias claves privadas.

Si una clave ha sido anulada por alguna causa, las autoridades que la hubieran certificado deben cancelar todos sus certificados asociados. Esto hace que todas las autoridades dispongan de *listas de revocación de certificados* (CRL), que actualizan periódicamente, además de un servicio de consulta de las mismas. Dicho servicio suele permitir tanto la consulta de la validez de un certificado concreto, como la descarga total o parcial de la CRL correspondiente.

17.4. Verificación de certificados digitales

Una autoridad de certificación suele tener un ámbito relativamente local, como puede ser una empresa, un campus universitario o un país entero. Si fuera necesario verificar un certificado digital de un certificador ajeno, del cual desconocemos su fiabilidad, existe la posibilidad de que la clave pública del propio certificador esté a su vez firmada por otra entidad de la que sí nos fiemos, y de esta forma *propagar* nuestra confianza hacia la entidad certificadora en cuestión. Esta circunstancia puede ser aprovechada de forma jerárquica —como en las PKI (Infraestructuras de Clave Pública o, en inglés *Public Key Infrastructure*)— o distribuida —como hace PGP (capítulo 18)—.

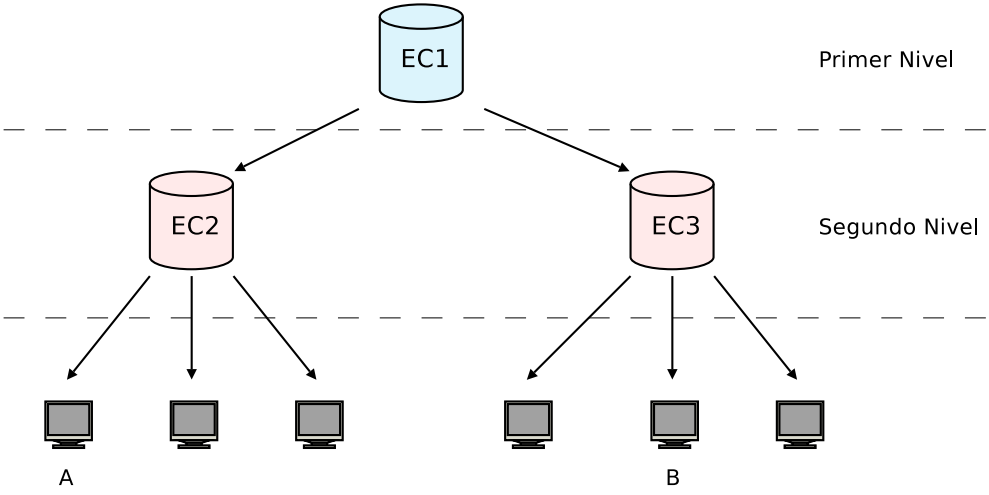

Figura 17.2: Esquema jerárquico de certificación. Si A quiere comprobar la identidad de B, empleará la clave pública de $EC1$ para verificar el certificado digital de $EC3$. Una vez hecha esta comprobación, podrá confiar en $EC3$ como certificador de la clave pública de B.

17.4.1. Infraestructuras jerárquicas

En esta modalidad, las entidades certificadoras se organizan en forma de árbol por *niveles* (ver figura 17.2), de tal manera que las entidades certificadoras de un nivel poseen certificados digitales emitidos por autoridades de niveles superiores. Podremos verificar satisfactoriamente un certificado digital cualquiera, siempre que poseamos la clave pública de un certificador de primer nivel —que son muy pocos e internacionalmente reconocidos—.

Como es natural, las entidades certificadoras que generen certificados finales —correspondientes a las hojas del árbol— tendrán la única responsabilidad de comprobar de manera fehaciente que cada clave pública pertenece a su propietario. Sin embargo, aquellas entidades que certifiquen a otras entidades, deberán garantizar además que estas

últimas emplean mecanismos adecuados para comprobar las identidades de sus clientes. De lo contrario, alguien podría crear una autoridad de certificación, obtener el correspondiente certificado digital de niveles superiores, y luego emitir certificados falsos.

El esquema jerárquico es realmente simple y efectivo, pero presenta un problema importante: si uno de los certificadores resulta comprometido, todos sus descendientes en el árbol quedan invalidados. Esto obliga, por un lado, a que las autoridades de certificación sean lo más transparentes posible, y por otro a que se mantengan siempre al día las listas de revocación de certificados.

17.4.2. Infraestructuras distribuidas

Frente a la estructura jerárquica, se puede construir un esquema distribuido de certificación de claves, también conocido como *anillo de confianza —ring of trust*, en inglés—, en el que todos los usuarios actúan como autoridades de certificación. Este sistema presenta la ventaja de ser muy resistente, ya que no depende de un pequeño grupo de entidades certificadoras, pero tiene el inconveniente de ser considerablemente más complejo de manejar para los propios usuarios.

Puesto que no existen autoridades de certificación centralizadas, cada usuario tiene que responsabilizarse de lo siguiente:

- Verificar la autenticidad de todas aquellas claves públicas que le sea posible.
- Certificar aquellas claves sobre las que tenga absoluta certeza de que pertenecen a sus propietarios.
- Elegir en qué condiciones confiará en los certificados de otro usuario.

Según el grado de confianza que presente un usuario, uno puede elegir *creerse* todos sus certificados, no aceptar ninguno —piénsese en

un usuario que certifica todo lo que cae en sus manos, sin hacer ninguna comprobación—, o aceptar aquellos que, además, posean firmas de otros usuarios.

Como puede comprobarse, en este esquema la confianza en una entidad certificadora puede tomar muchos valores, frente a los dos —confiable y no confiable— que puede tomar en un esquema jerárquico. Se establece, de hecho, una gradación de niveles de confianza que, como resultado, proporciona a su vez grados de confianza sobre las claves públicas que nos encontremos, variando desde desconfianza total hasta confianza absoluta.

17.5. Autentificación mediante funciones resumen

En muchos casos, la autentificación se lleva a cabo a partir de una pieza de información, que puede ser una clave criptográfica o una palabra secreta o contraseña. Este elemento se encuentra en posesión del agente que pretende autentificarse. En esencia, el proceso de autentificación consiste en demostrar que se está en posesión de la información secreta, sin revelar nada en el proceso que un tercero pueda usar para deducirla. Describiremos en esta sección dos ejemplos de este tipo: la autentificación de un usuario mediante una contraseña, y la de un dispositivo —tarjeta inteligente, por ejemplo— que posee una clave secreta embebida.

17.5.1. Autentificación por contraseñas

Existe un caso muy especial de autentificación, cuyo propósito es la identificación de una persona frente a un sistema informático, y que consiste en la introducción, usualmente mediante un teclado, de una palabra o frase secreta —*password* o *passphrase* en inglés— que única-

mente él conoce.

La primera opción que podríamos escoger consiste simplemente en que el sistema informático almacene las contraseñas en claro, y que cada vez que un usuario pretenda acceder, se le pida la palabra clave y después se la compare con el valor almacenado. El problema es que si alguien logra acceder a la base de datos de contraseñas, cosa que ocurre con bastante frecuencia, podrá suplantar fácilmente a todos y cada uno de los usuarios. Teniendo en cuenta que un sistema informático moderno suele permitir el acceso de múltiples usuarios, con diversos niveles de privilegios sobre el mismo, e incluso en muchos casos existe un usuario genérico *invitado*, podemos concluir que almacenar las contraseñas en claro representa un riesgo demasiado elevado.

Si en lugar de las contraseñas en claro, guardamos en nuestro sistema los valores resultantes de aplicar alguna función resumen (capítulo 13) sobre las mismas, podremos verificar las contraseñas introducidas por los usuarios simplemente calculando la signatura asociada a la información introducida por el usuario y comparando con lo que tengamos almacenado. De esta forma, debido a las propiedades de las funciones resumen, resultará extremadamente complejo para un atacante encontrar una contraseña a partir de una signatura dada. Este sistema, si bien es considerablemente más seguro que el almacenamiento en claro de las contraseñas, puede ser objeto de un ataque, denominado *de diccionario*, relativamente sencillo y que describiremos a continuación.

Ataque de diccionario

Es un hecho que las contraseñas, en general, suelen presentar poca entropía. Esto significa que, dentro del conjunto de todas las combinaciones aleatorias de letras y símbolos, solo unas pocas se emplean en la práctica como contraseñas. Por lo tanto, un usuario malicioso puede construir una base de datos con millones de contraseñas —un diccionario—, obtenidas a partir de palabras, nombres, fechas, combinacio-

nes numéricas más o menos habituales, etc., y calcular la signatura de cada una de ellas. Una vez obtenido el archivo que contiene los *hashes* de las contraseñas de cada usuario, bastará con buscar en la base de datos el valor correcto. Este diccionario puede construirse de forma previa e independiente al sistema informático que se pretenda atacar, y ser reutilizado tantas veces como se desee.

Para protegerse de los ataques de diccionario existen varias estrategias:

- Añadir información verdaderamente aleatoria a las contraseñas antes de calcular sus resúmenes. Esta información, denominada *sal*, se almacena en el ordenador junto con la signatura correspondiente, de forma que obliga a los posibles adversarios a recalcular todos los *hashes* del diccionario una vez conocido el valor de la *sal*.

- Emplear funciones resumen nuy costosas, como las KDF de *estiramiento* de clave (sección 13.7). Esto aumenta considerablemente el esfuerzo computacional necesario para construir un posible diccionario, compensando de alguna manera la falta de entropía de las propias contraseñas.

- Maximizar el espacio de búsqueda para un atacante, escogiendo valores difíciles de *adivinar*. Es fundamental, a la hora de seleccionar las contraseñas de los usuarios, que éstas sean lo suficientemente complejas como para no aparecer en un diccionario. Para ello es conveniente emplear métodos aleatorios específicos de generación de contraseñas, en lugar de confiar en nuestro propio ingenio, ya que está demostrado que el ser humano es bastante predecible. Otra medida bastante recomendable, y que ya incorporan bastantes sistemas informáticos, es emplear rutinas de medición de la *calidad* de las contraseñas e impedir a los usuarios seleccionar combinaciones de caracteres demasiado débiles.

Existe no obstante una serie de posibles problemas para un sistema

basado en contraseñas, independientes de su implementación desde el punto de vista lógico, que conviene tener en cuenta:

- Si el acceso se lleva a cabo desde un terminal remoto, la contraseña debe enviarse a través de un canal de comunicaciones, por lo que debería emplearse una conexión previamente cifrada.
- El empleo de un teclado para introducir la palabra secreta puede hacer el sistema susceptible de escuchas. Existen estudios que demuestran que a través de las radiaciones electromagnéticas de un teclado cualquiera, e incluso del simple sonido de cada tecla, es posible conocer lo que introduce el usuario en la consola.
- El modo más seguro de custodiar la contraseña es la memoria humana, pero si aquella es demasiado compleja, y especialmente si se cambia con frecuencia, puede resultar difícil de recordar. En este sentido, hay opiniones para todos los gustos: desde los que emplean reglas nemotécnicas, hasta aquellos que recomiendan tener anotada la contraseña y custodiarla cuidadosamente.

Además de estar bien salvaguardadas, cabría seguir las siguientes directrices para que nuestras palabras clave puedan considerarse *seguras*:

1. *Deben permanecer a salvo de terceros*. Una contraseña jamás debe ser conocida por un extraño, lo cual desaconseja llevarla escrita en algún sitio, o bien obliga a custodiar cuidadosamente el lugar donde esté anotada. Existen autores que abogan por llevar las contraseñas escritas en una tarjeta, y cuidar de ella como de nuestro dinero o las llaves de nuestra casa.
2. *No utilizar la misma clave en diferentes lugares*, ya que si uno resulta comprometido, el resto también caerá. Hay aplicaciones que, a partir de una *contraseña maestra*, permiten almacenar todas nuestras contraseñas de uso diario, lo cual facilitará sin duda nuestro día a día, permitiéndonos gestionar todas las palabras clave que necesitemos.

3. *Ser lo suficientemente complejas*. Una buena contraseña debe constar de al menos ocho letras. Pensemos que si empleamos únicamente seis caracteres alfanuméricos (números y letras), tenemos solo unos dos mil millones de posibilidades. Teniendo en cuenta que hay programas para PC capaces de probar más de cuarenta mil claves en un segundo, una clave de estas características podría ser descubierta en menos de quince horas.

4. *Carecer de significado*. Una contraseña jamás debe significar nada, puesto que entonces aumentará la probabilidad de que aparezca en algún diccionario. Evitemos los nombres propios, en especial aquellos que pertenezcan a lugares o personajes de ficción.

5. *Ser fáciles de recordar*. Si pretendemos memorizar nuestras claves, carecerá de sentido emplear contraseñas difíciles de recordar. Para esto podemos seguir reglas como que la palabra se pueda pronunciar en voz alta, o que responda a algún acrónimo más o menos complejo. En este punto no debemos olvidar que hay que evitar a toda costa palabras que *signifiquen algo*.

6. *Cambiarlas con frecuencia*. Hemos de partir de la premisa de que toda palabra clave será comprometida tarde o temprano, por lo que será muy recomendable que nuestras contraseñas sean cambiadas periódicamente. La frecuencia con la que se produzca el cambio dependerá de la complejidad de las claves y del nivel de seguridad que se desee alcanzar. Y lo más importante: ante cualquier sospecha, cambiar todas las contraseñas.

17.5.2. Autentificación por desafío

Existen muchas aplicaciones en las que un dispositivo electrónico —una tarjeta inteligente, por ejemplo— necesita identificarse frente a un sistema informático. Esto se puede hacer a través de una clave secreta almacenada en el dispositivo, en alguna zona de memoria que no pueda ser leída desde el exterior. Como es obvio, en ningún caso la

clave puede ser enviada en claro en el proceso de autentificación, porque si no un atacante que intercepte la comunicación podrá suplantar al dispositivo. También es necesario que cada proceso de autentificación involucre mensajes totalmente distintos, ya que si no el impostor podría *memorizar* las respuestas dadas por el dispositivo y replicarlas posteriormente. De hecho, debe construirse el protocolo de tal forma que la información que pudiese *escuchar* un atacante resulte completamente inútil.

Las funciones MAC permiten llevar a cabo de manera sencilla este tipo de autentificaciones, denominada *autentificación por desafío*, y se desarrolla de forma interactiva entre el sistema anfitrión y el dispositivo que se autentifica. Consiste en generar una clave K, de la cual habrá una copia tanto en el servidor como en el dispositivo (figura 17.3). Cuando el dispositivo solicita ser identificado, el servidor genera un valor aleatorio X y calcula su MAC empleando la clave K. Posteriormente envía el valor de X, que será distinto en cada realización del protocolo, al dispositivo, que realizará internamente los mismos cálculos y devolverá el resultado al servidor. Si la respuesta recibida por el dispositivo coincide con el valor calculado en el servidor, el proceso de autentificación habrá tenido éxito.

Un ejemplo de este mecanismo lo tenemos en las tarjetas SIM que emplean los teléfonos celulares GSM. Dichas tarjetas llevan implementado un algoritmo MAC, denominado COMP128, que en principio era secreto, fue reconstruido por la comunidad *cypherpunk* a partir de documentos filtrados e ingeniería inversa, y *roto* en pocos días por criptoanalistas de la Universidad de Berkeley.

Para que este tipo de protocolos funcione correctamente es necesario que el algoritmo MAC sea lo suficientemente bueno, es decir, que a partir de una cantidad arbitraria de pares $\{X, MAC_K(X)\}$ resulte computacionalmente imposible recuperar el valor de K. Este fue precisamente el problema de COMP128, ya que se descubrió una forma de recuperar K a partir de unos 100.000 pares $\{X, MAC_K(X)\}$. El ataque consistía básicamente en suplantar al servidor y enviar 100.000 valores de X escogidos cuidadosamente al dispositivo, que devolvía

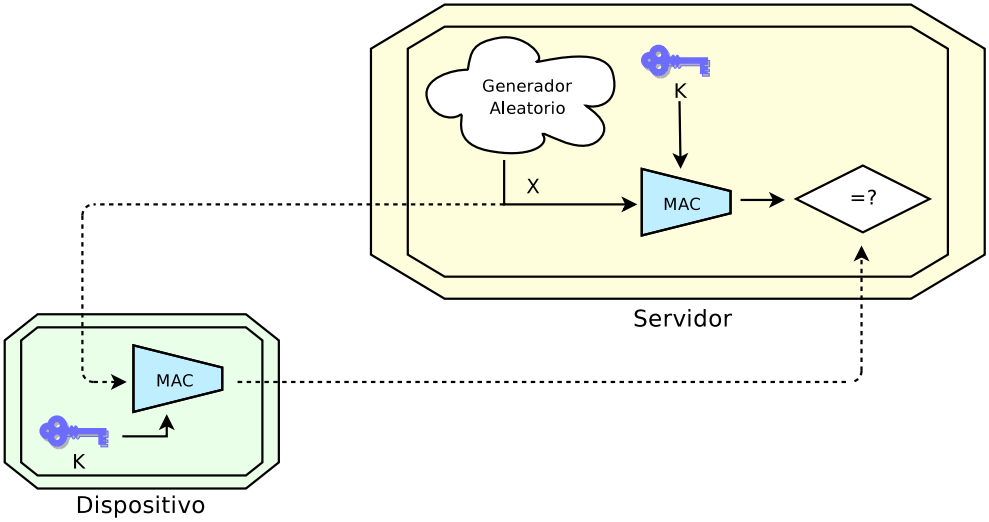

Figura 17.3: Esquema de autentificación por desafío.

sistemáticamente los valores de la función MAC correspondiente para cada uno de ellos. Posteriormente se realizanban los cálculos *fuera de línea*, obteniéndose como resultado el valor de K, suficiente para *clonar* la tarjeta SIM en cuestión.

También es necesario que los valores de X que se generen en cada realización del algoritmo sean criptográficamente aleatorios, ya que en caso contrario un atacante podría predecir el valor de X para la siguiente realización del protocolo, luego suplantaría al servidor usando ese valor, se pondría en contacto con el dispositivo para que éste le devolviera el valor de $MAC_K(X)$. Con esta información en su poder, podría solicitar al servidor ser autentificado, y lograría suplantar con éxito al dispositivo.

Capítulo 18

PGP

El nombre PGP responde a las siglas *pretty good privacy* (privacidad bastante buena), y se trata de un proyecto iniciado a principios de los 90 por Phil Zimmermann. La total ausencia por aquel entonces de herramientas sencillas, potentes y baratas que acercaran la criptografía *seria* al usuario movió a su autor a desarrollar una aplicación que llenara este hueco.

Con el paso de los años, PGP se ha convertido en uno de los mecanismos más populares y fiables para mantener la seguridad y privacidad en las comunicaciones, especialmente a través del correo electrónico, tanto para pequeños usuarios como para grandes empresas.

Actualmente PGP se ha convertido en un estándar internacional (*RFC 2440*), lo cual está dando lugar a la aparición de múltiples productos PGP, que permiten desde cifrar correo electrónico hasta codificar particiones enteras del disco duro (PGPDisk), pasando por la codificación automática y transparente de todo el tráfico TCP/IP (PGPnet).

18.1. Fundamentos e historia de PGP

PGP trabaja con criptografía asimétrica, y por ello tal vez su punto más fuerte sea precisamente la gran facilidad que ofrece al usuario a la hora de gestionar sus claves públicas y privadas. Si uno emplea algoritmos asimétricos, debe poseer las claves públicas de todos sus interlocutores, además de la clave privada propia. Con PGP surge el concepto de *anillo de claves* (o llavero), que no es ni más ni menos que el *lugar* que este programa proporciona para que el usuario guarde todas las claves que posee. El anillo de claves es un único fichero en el que se pueden efectuar operaciones de extracción e inserción de claves de manera sencilla, y que además proporciona un mecanismo de identificación y autentificación de llaves completo y simple de utilizar. Esta facilidad en la gestión de claves es una de las causas fundamentales que han hecho a PGP tan popular.

La historia de PGP se remonta a comienzos de los años 90. La primera versión era completamente diferente a los PGP posteriores, además de ser incompatible con éstos. La familia de versiones 2.x.x fue la que alcanzó una mayor popularidad, y sigue siendo utilizada por mucha gente en la actualidad. Los PGP 2.x.x emplean únicamente los algoritmos IDEA, RSA y MD5.

En algún momento una versión de PGP atravesó las fronteras de EE.UU. y nació la primera versión internacional de PGP, denominada PGPi, lo que le supuso a Phil Zimmermann una investigación de más de tres años por parte del FBI, ya que supuestamente se habían violado las restrictivas leyes de exportación de material criptográfico que poseen los Estados Unidos. Para la versión 5 de PGP se subsanó este problema exportando una versión impresa del código fuente, que luego era reconstruida y compilada en Europa (más información en `http://www.pgpi.com`).

Hasta principios de 2001 la política de distribución de PGP consistió en permitir su uso gratuito para usos no comerciales y en publicar el código fuente en su integridad, con el objetivo de satisfacer a los

desconfiados y a los curiosos. Sin embargo, con el abandono de la empresa por parte de Zimmermann, en febrero de 2001, el código fuente dejó de publicarse. En la actualidad (finales de 2004), la empresa que gestiona los productos PGP (PGP Corporation), ha vuelto a publicar el código fuente de los mismos.

Paralelamente a la azarosa existencia empresarial de PGP, el proyecto GNU ha estado desarrollando su propia aplicación de código abierto compatible con el *RFC 2440*, denominada GnuPG, y que solo emplea algoritmos libres de patentes.

18.2. Estructura de PGP

18.2.1. Codificación de mensajes

Como el lector ya sabe, los algoritmos simétricos de cifrado son considerablemente más rápidos que los asimétricos. Por esta razón PGP cifra primero el mensaje empleando un algoritmo simétrico (ver figura 18.1) con una clave generada aleatoriamente (*clave de sesión*) y posteriormente codifica la clave haciendo uso de la llave pública del destinatario. Dicha clave es extraída convenientemente del anillo de claves públicas a partir del identificador suministrado por el usuario, todo ello de forma transparente, por lo que únicamente debemos preocuparnos de indicar el mensaje a codificar y la lista de identificadores de los destinatarios. Nótese que para que el mensaje pueda ser leído por múltiples destinatarios basta con que se incluya en la cabecera la clave de sesión codificada con cada una de las claves públicas correspondientes.

Cuando se trata de decodificar el mensaje, PGP simplemente busca en la cabecera las claves públicas con las que está codificado y nos pide una contraseña. La contraseña servirá para que PGP abra nuestro anillo de claves privadas y compruebe si tenemos una clave que permita decodificar el mensaje. En caso afirmativo, PGP descifrará el mensa-

je. Nótese que siempre que queramos hacer uso de una clave privada, habremos de suministrar a PGP la contraseña correspondiente, por lo que si el anillo de claves privadas quedara comprometido, un atacante aún tendría que averiguar nuestra contraseña para descifrar nuestros mensajes. No obstante, si nuestro archivo de claves privadas cayera en malas manos, lo mejor será *revocar* todas las claves que tuviera almacenadas y generar otras nuevas.

Como puede comprenderse, gran parte de la seguridad de PGP reside en la calidad del generador aleatorio que se emplea para calcular las claves de sesión, puesto que si alguien logra predecir la secuencia de claves que estamos usando, podrá descifrar todos nuestros mensajes independientemente de los destinatarios a los que vayan dirigidos. Afortunadamente, PGP utiliza un método de generación de números pseudoaleatorios muy seguro —una secuencia aleatoria pura es imposible de conseguir, como se dijo en el capítulo 8—, y protege criptográficamente la semilla que necesita[1]. No obstante, consideraremos *sensible* al fichero que contiene dicha semilla —normalmente `RANDSEED.BIN`—, y por lo tanto habremos de evitar que quede expuesto.

18.2.2. Firma digital

En lo que se refiere a la firma digital, las primeras versiones de PGP obtienen en primer lugar la signatura MD5 (ver sección 13.3.1), que posteriormente se codifica empleando la clave privada RSA correspondiente. Versiones más modernas implementan el algoritmo DSS, que emplea la función resumen SHA-1 y el algoritmo asimétrico DSA (secciones 12.5.3 y 13.3.2).

La firma digital o signatura puede ser añadida al fichero u obtenida en otro fichero aparte. Esta opción es muy útil si queremos *firmar* un fichero ejecutable, por ejemplo.

[1] Algunas implementaciones de PGP emplean otras fuentes de aleatoriedad, como ocurre con GnuPG, por lo que no necesitan almacenar una semilla aleatoria.

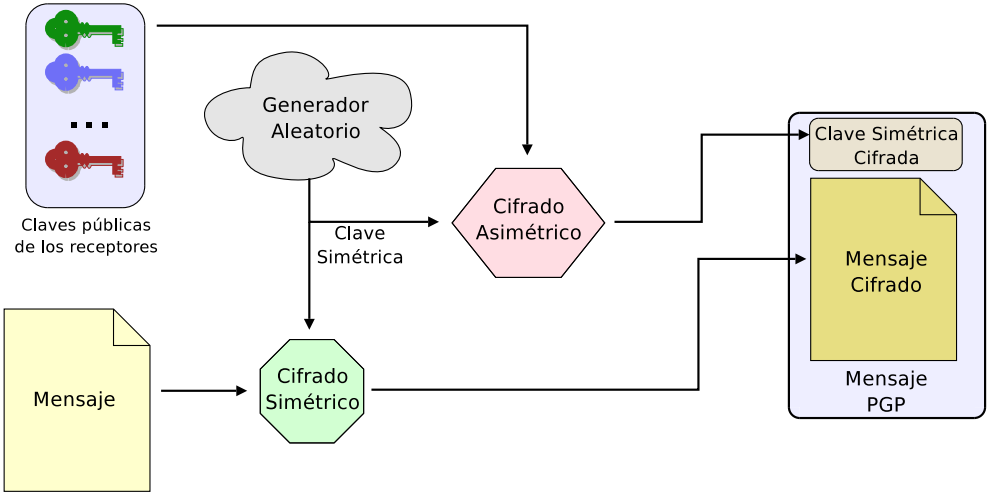

Figura 18.1: Codificación de un mensaje PGP

18.2.3. Armaduras ASCII

Una de las funcionalidades más útiles de PGP consiste en la posibilidad de generar una *armadura ASCII* para cualquiera de sus salidas. Obviamente, todas las salidas de PGP (mensajes codificados, claves públicas extraídas de algún anillo, firmas digitales, etc.) consisten en secuencias binarias, que pueden ser almacenadas en archivos. Sin embargo, en la mayoría de los casos puede interesarnos enviar la información mediante correo electrónico, o almacenarla en archivos de texto.

Recordemos que el código ASCII original emplea 7 bits para codificar cada letra, lo cual quiere decir que los caracteres situados por encima del valor ASCII 127 no están definidos, y de hecho diferentes computadoras y sistemas operativos los interpretan de manera distinta. También hay que tener en cuenta que entre los 128 caracteres ASCII se encuentran muchos que representan códigos de control, como el retorno de carro, el fin de fichero, el tabulador, etc. La idea es elegir 64

caracteres *imprimibles* (que no sean de control) dentro de esos 128. Con este conjunto de códigos ASCII podremos representar exactamente 6 bits, por lo que una secuencia de tres bytes (24 bits) podrá codificarse mediante cuatro de estos caracteres. Esta cadena de símbolos resultante se *trocea* colocando en cada línea un número razonable de símbolos, por ejemplo 72. El resultado es una secuencia de caracteres que pueden ser tratados como texto estándar, además de ser manipulados en cualquier editor. Existe la ventaja adicional de que esta representación es apropiada para ser enviada por correo electrónico, ya que muchas pasarelas de correo no admiten caracteres por encima de 127, y además truncan las líneas demasiado largas, por lo que podrían alterar los mensajes si *viajaran* en otro formato.

Como ejemplo se incluye un mensaje cifrado en formato ASCII:

```
-----BEGIN PGP MESSAGE-----
Version: GnuPG v1.4.11 (GNU/Linux)

hQIMA/iYbNm/6Fo7AQ/7BpbdYSPG2QLWSPIZFYLIBILjkVYxl7emcK8JCZt+furv
dE3AsvWWelkAYaR9UFttAQW7v3R29//vbXfXYQ1IDykouw/26LQ/tcyB7SvfDpZQ
7T3eUsHkS50YghntE/XLa524Hpxg+tlsyVT9/SRDcwh1yhavO7+bszlA6DEmTObd
quq8+GwR9CV0UQY/pTaUy1Pq/mFxsfMCDRT0/3NfuJIMCFo/+qslxRe6CYwpdGUY
qFFTlPxChpKIEvOaHDFzc7qJJoSTf89DrmE8TitRC/lMC2WTZJ33JBBjJIUEwuOO
Y3NGyLthNnH/N0iexTPLxb2E99NBR2ypykFezxAlX80YFxymL4r+zcvVKM75XEn1
Yns5QIZFztJhxyUbeL6mqeS+M9Bibm+5VN4vny46TekDQTlLxo1Lzo8+wXsn++88
XgzbxIao7U3oS5qayNIvaGLYQW1IclrFW0hEDyMyNfXAuAAVRejiHdbeZgyRIYuA
+hsTfyCEsW1dkgzspnV6SQeqL9V57QuC46ZuJIDO2YIOrvnWbPQ+VU5n3nuQn0S/
bVWIz8WJEC9R6QfPygFAsp6sFVX9pwyhsJKJN4T6efeTHJ1sLqiakOsYSlvB1mAs
Z+EG3AYCuVTQFcWElH6CTb2L9KaBTEBJ3zX1rb9q1KaSSTIyXopC/x04elX0/IjS
QAFiS1IELzsQTAJfF+Z4BUUF/pWHUTcwvTgx/Eky4Six78fq0mqMTVu/u8ZuqrQl
ansYQ2vNgzejjtY8ff9APkE=
=Xrpd
-----END PGP MESSAGE-----
```

Como puede verse, los únicos símbolos empleados son las letras mayúsculas y minúsculas, los números, y los signos '/' y '+'; el resto de símbolos y caracteres de control simplemente será ignorado. Cualquiera podría copiar esta clave pública a mano (¡!) o emplear una aplicación OCR[2] para introducirla en su anillo de claves correspondiente, aunque es mejor descargarla a través de Internet.

[2]OCR: *Optical Character Recognition*, reconocimiento óptico de caracteres. Permite convertir texto escrito a formato electrónico.

18.2.4. Gestión de claves

PGP, como ya se ha dicho, almacena las claves en unas estructuras denominadas *anillos*. Un anillo no es más que una colección de claves, almacenadas en un fichero. Cada usuario tendrá dos anillos, uno para las claves públicas (`PUBRING.PKR`) y otro para las privadas (`SECRING.SKR`).

Cada una de las claves, además de los valores relativos al algoritmo correspondiente y sus parámetros asociados, posee una serie de datos, como son el identificador hexadecimal de la clave (generalmente de 64 o 128 bits de longitud), la dirección de e-mail del usuario que la emitió, la fecha de expiración, la versión de PGP con que fue generada, y la denominada huella digital (*fingerprint*). Este último campo es bastante útil, pues se trata de un *hash* de la propia clave, lo cual hace que sea único, a diferencia del identificador hexadecimal, que puede presentar colisiones. Por ejemplo, la huella digital de la clave pública del autor de este libro es la siguiente:

```
FEE8 C959 6399 2668 9E16  38E7 9740 4C7A 6627 F766
```

La huella digital se emplea para asegurar la autenticidad de una clave. Si alguien quisiera asegurarse de que posee la clave antes citada y no una falsificación, bastaría con que llamara por teléfono a su dueño, y le pidiera que le leyera su huella digital. Afortunadamente, las últimas implementaciones de PGP permiten convertir esta cadena hexadecimal en una secuencia de palabras fácilmente legibles en voz alta.

18.2.5. Distribución de claves y redes de confianza

PGP, como cualquier sistema basado en clave pública, es susceptible a ataques de intermediario (sección 12.2). Esto nos obliga a establecer mecanismos para asegurarnos de que una clave procede realmente

de quien nosotros creemos. Una de las cosas que permite esto, aunque no la única, es la *huella digital*.

PGP permite a un usuario firmar claves, y de esta forma podremos confiar en la autenticidad de una clave siempre que ésta venga firmada por una persona de confianza. Hay que distinguir entonces dos tipos de confianza: aquella que nos permite creer en la validez de una clave, y aquella que nos permite fiarnos de una persona como certificador de claves. La primera se puede calcular automáticamente, en función de que las firmas que contenga una clave pertenezcan a personas de confianza, pero la segunda ha de ser establecida manualmente. No olvidemos que el hecho de que una clave sea auténtica no nos dice nada acerca de la persona que la emitió. Por ejemplo, yo puedo tener la seguridad de que una clave pertenece a una persona, pero esa persona puede dedicarse a firmar todas las claves que le llegan, sin asegurarse de su autenticidad, por lo que en ningún caso merecerá nuestra confianza.

Cuando una clave queda comprometida, puede ser revocada por su autor. Para ello basta con generar y distribuir un *certificado de revocación* que informará a todos los usuarios de que esa clave ya no es válida. Para generarlo es necesaria la clave privada, por lo que en muchos casos se recomienda generar con cada clave su certificado de revocación y guardarlo en lugar seguro, de forma que si perdemos la clave privada podamos revocarla de todas formas. Afortunadamente, las últimas versiones de PGP permiten nombrar revocadores de claves, que son usuarios capaces de invalidar nuestra propia clave, sin hacer uso de la llave privada.

18.2.6. *Otros* PGP

La rápida popularización de PGP entre ciertos sectores de la comunidad de Internet, y el desarrollo del estándar público *Open PGP*, han hecho posible la proliferación de variantes más o menos complejas del programa de Zimmermann. Muchas de ellas son desarrolladas por

los propios usuarios, para mejorar alguna característica, como manejar claves de mayor longitud (PGPg), y otras corresponden a aplicaciones de tipo comercial.

Especial mención merece la implementación de *Open PGP* que está llevando a cabo el proyecto GNU: GnuPG (*GNU Privacy Guard*), que funciona en múltiples plataformas, y emplea únicamente algoritmos de libre distribución —entre ellos AES—, aunque presenta una estructura que la hace fácilmente extensible. De hecho, hoy por hoy, podríamos decir que es la implementación de PGP más completa, segura y útil para cualquier usuario.

18.3. Vulnerabilidades de PGP

Según todo lo dicho hasta ahora, parece claro que PGP proporciona un nivel de seguridad que nada tiene que envidiar a cualquier otro sistema criptográfico jamás desarrollado. ¿Qué sentido tiene, pues, hablar de sus *vulnerabilidades*, si éstas parecen no existir?

Como cualquier herramienta, PGP proporcionará un gran rendimiento si se emplea correctamente, pero su uso inadecuado podría convertirlo en una protección totalmente inútil. Es por ello que parece interesante llevar a cabo una pequeña recapitulación acerca de las *buenas costumbres* que harán de PGP nuestro mejor aliado.

- *Escoger contraseñas adecuadas*. Todo lo comentado en la sección 17.5.1 es válido para PGP.
- *Proteger adecuadamente los archivos sensibles*. Estos archivos serán, lógicamente, nuestros llaveros (anillos de claves) y el fichero que alberga la semilla aleatoria. Esta protección debe llevarse a cabo tanto frente al acceso de posibles curiosos, como frente a una posible pérdida de los datos (¡recuerde que si pierde el archivo con su clave privada no podrá descifrar jamás ningún mensaje!).

- *Emitir revocaciones de nuestras claves al generarlas* y guardarlas en lugar seguro. Serán el único mecanismo válido para revocar una clave en caso de pérdida del anillo privado. Afortunadamente, la versión 6 de PGP permite nombrar *revocadores* para nuestras claves, de forma que éstos podrán invalidarla en cualquier momento sin necesidad de nuestra clave privada.

- *Firmar sólo las claves de cuya autenticidad estemos seguros*. Es la única manera de que las redes de confianza puedan funcionar, ya que si todos firmáramos las claves alegremente, podríamos estar certificando claves falsas.

Al margen de un uso correcto, que es fundamental, debemos mencionar que últimamente han sido detectados algunos fallos en las diversas implementaciones de PGP. Clasificaremos dichas vulnerabilidades en dos grupos claramente diferenciados:

- *Debidas a la implementación*: Estos agujeros de seguridad son provocados por una implementación defectuosa de PGP, y corresponden a versiones concretas del programa. Por ejemplo, el fallo descubierto en la versión 5.0 de PGP para UNIX, que hacía que las claves de sesión no fueran completamente aleatorias, o el encontrado en todas las versiones para Windows, desde la 5.0 a la 7.0.4, en la que un inadecuado procesamiento de las armaduras ASCII permitía a un atacante introducir ficheros en la computadora de la víctima.

- *Intrínsecas al protocolo*: En este apartado habría que reseñar aquellos agujeros de seguridad que son inherentes a la definición del estándar *Open PGP*. En este sentido, a principios de 2001 se hizo pública una técnica que permitía a un atacante falsificar firmas digitales. En cualquier caso, se necesita acceso físico a la computadora de la víctima para manipular su clave privada, por lo que el fallo carece de interés práctico.

Parte V

Apéndices

Apéndice A

Criptografía cuántica

La Física Cuántica estudia el comportamiento de la materia a escalas muy pequeñas, del orden de los átomos. En el mundo cuántico las reglas que rigen la Mecánica Clásica dejan de tener validez, y se producen fenómenos tan sorprendentes como interesantes, que abren las puertas a posibilidades de aplicación casi increíbles en muchos campos, entre los que se encuentra, por supuesto, la Criptografía.

Cabe recordar que hoy por hoy ya existen algunas aplicaciones prácticas de la Mecánica Cuántica en Criptografía, mientras que otras, como las basadas en los computadores cuánticos, siguen perteneciendo al ámbito de la especulación, ya que la tecnología que podría permitirnos desarrollar dispositivos de este tipo aún no existe.

A.1. Mecánica cuántica y Criptografía

Una de las aplicaciones directas de los fenómenos cuánticos en Criptografía viene de un principio básico de esta teoría: un objeto no puede interaccionar con otro sin experimentar alguna modificación. Esto está permitiendo fabricar canales de comunicación en los que los datos *viajan* en forma de fotones individuales con diferentes ca-

racterísticas. El hecho aquí es que si un atacante intentara interceptar la comunicación no tendría más remedio que interactuar con esos fotones, modificándolos de manera detectable por el receptor.

Este tipo de propiedades permite construir líneas de comunicación totalmente imposibles de interceptar sin ser descubierto, y de hecho ya se han llevado a cabo algunos experimentos en los que se ha logrado transmitir información a distancias y velocidades respetables. Evidentemente, estos canales ultraseguros difícilmente serán tan rápidos o tan baratos como las líneas eléctricas y ópticas actuales, pero en un futuro próximo constituirán medios idóneos para transmitir información de carácter sensible.

A.2. Computación cuántica

Existe un fenómeno en Mecánica cuántica realmente difícil de entender para nuestras *clásicas* mentes. Obsérvese la figura A.1. En ella se ilustra un conocido y sorprendente experimento. A es una fuente capaz de emitir fotones, 1 y 4 dos espejos completamente reflectantes, y 2 y 3 espejos semirreflectantes, que reflejan la mitad de la luz y dejan pasar la otra mitad. Si situamos en B y C detectores de fotones, la intuición —y la Mecánica Clásica— nos dirían que cada fotón acabará excitando B o C con un 50 % de probabilidades. Pues bien, lo que en realidad ocurre es que C se excita siempre y B no lo hace nunca. Esto demuestra que, a nivel subatómico, cualquier partícula puede *estar en dos sitios simultáneamente*, o más propiamente, en una *superposición cuántica de dos estados*, lo cual significa que está *realmente* en esos dos estados, en lugar de estar en uno u otro con determinada probabilidad.

Supongamos ahora que logramos construir un dispositivo capaz de representar bits mediante estados cuánticos de una o muy pocas partículas. Si colocamos dichas partículas en una combinación de los dos estados básicos, tendríamos un *bit cuántico* (o *qubit*), capaz de representar un 1 y un 0… ¡al mismo tiempo!.

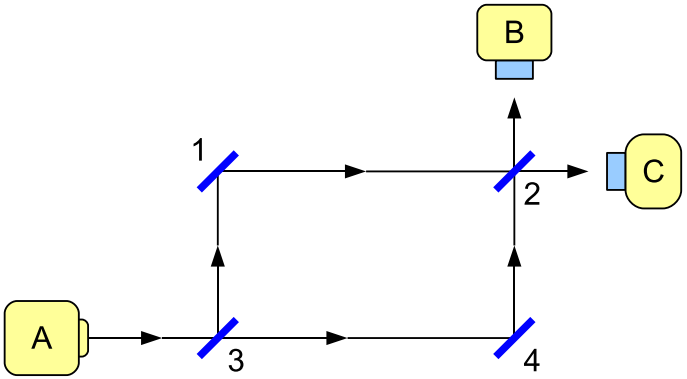

Figura A.1: Experimento con espejos para comprobar la superposición cuántica de estados en un fotón. A es una fuente emisora de fotones, B y C son receptores, 1 y 4 son espejos totalmente reflectantes, y 2 y 3 son espejos que reflejan exactamente la mitad de la luz y dejan pasar la otra mitad. Contrariamente a lo que diría la intuición, en B no se detecta nada.

Estas ideas, que datan de los años 80, se han barajado más bien como simples entretenimientos para mentes inquietas, hasta que a mediados de los 90 se propuso el primer algoritmo capaz de ser ejecutado en una computadora cuántica. Dicho algoritmo podría, de forma eficiente, factorizar números enteros muy grandes. Imagínense las implicaciones que esto tiene para la Criptografía moderna, ya que supondría la caída de la gran mayoría de los algoritmos asimétricos, que basan su funcionamiento en el problema de la factorización de grandes enteros, y la necesidad inmediata de alargar considerablemente las longitudes de claves para algoritmos simétricos. Evidentemente, estos resultados han provocado que mucha gente tome muy en serio este tipo de computadoras, y que en la actualidad haya importantes grupos dedicados a la investigación en este campo.

A.3. Expectativas de futuro

Por fortuna —o por desgracia, según se mire—, los modelos cuánticos de computación hoy por hoy no pasan de meras promesas, ya que la tecnología actual no permite confinar partículas individuales de forma que preserven su estado cuántico. Los más optimistas aseguran que en pocos años tendremos los primeros microprocesadores cuánticos en funcionamiento, mientras que la gran mayoría opina que todavía transcurrirán décadas antes de poder disponer del primer dispositivo realmente operativo —si es que lo conseguimos algún día—.

Lo que sí podemos afirmar con rotundidad es que los modelos criptográficos actuales seguirán siendo válidos durante algunos años más. En cualquier caso, no conviene perder de vista estas promesas tecnológicas, ya que cuando se conviertan en realidades, obligarán a replantear muchas cuestiones, y no sólo en el ámbito de la Criptografía.

Apéndice B

Ayudas a la implementación

Incluiremos en este apéndice información útil para facilitar al lector la implementación de diferentes algoritmos criptográficos. Aquellos que no sepan programar, o que simplemente no deseen escribir sus propias versiones de los criptosistemas que aparecen en este libro, pueden prescindir de esta sección.

B.1. DES

En el capítulo dedicado a algoritmos simétricos por bloques se ha hecho una descripción completa del algoritmo DES, pero se han omitido deliberadamente algunos detalles que sólo son útiles de cara a la implementación, como pueden ser los valores concretos de las s-cajas y de las permutaciones que se emplean en este algoritmo.

B.1.1. S-cajas

La tabla B.1 representa las ocho s-cajas 6*4 que posee DES. Para aplicarlas basta con coger el número de seis bits de entrada $b_0b_1b_2b_3b_4b_5$,

y buscar la entrada correspondiente a la fila b_0b_5, columna $b_1b_2b_3b_4$. Por ejemplo, el valor de la tercera s-caja para 110010 corresponde a la fila 2 (10), columna 9 (1001), es decir, 1 (0001).

B.1.2. Permutaciones

DES lleva a cabo permutaciones a nivel de bit en diferentes momentos. Las tablas que aquí se incluyen deben leerse por filas de arriba a abajo, y sus entradas corresponden al número de bit del valor inicial (empezando por el 1) que debe aparecer en la posición correspondiente. Por ejemplo, la primera tabla de B.2 lleva el valor $b_1b_2b_3 \ldots b_{64}$ en $b_{58}b_{50}b_{42} \ldots b_7$.

Permutaciones inicial y final

La tabla B.2 contiene las permutaciones inicial y final P_i y P_f del algoritmo DES. La primera de ellas se lleva a cabo justo al principio, antes de la primera ronda, y la segunda se aplica justo al final. Nótese que cada una de estas permutaciones es la inversa de la otra.

Función f

En el cálculo de la función f se emplean dos permutaciones, E y P (ver figura 10.3). Dichas permutaciones se detallan en la tabla B.3. E es una permutación de expansión, por lo que da como salida 48 bits a partir de los 32 de entrada.

Generación de las K_i

En la figura 10.4 podemos observar el proceso de generación de los 16 valores de K_i, en el que se emplean dos nuevas permutaciones (`EP1` y `EP2`), detalladas en la tabla B.4. La primera toma como entrada

Fila	Columna																S-Caja
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7	S_1
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8	
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0	
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13	
0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10	S_2
1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5	
2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15	
3	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9	
0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8	S_3
1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1	
2	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7	
3	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12	
0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15	S_4
1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9	
2	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4	
3	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14	
0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9	S_5
1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6	
2	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14	
3	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3	
0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11	S_6
1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8	
2	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6	
3	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13	
0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1	S_7
1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6	
2	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2	
3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12	
0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7	S_8
1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2	
2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8	
3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11	

Cuadro B.1: S-cajas de DES.

Permutación Inicial P_i															
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
Permutación Final P_f															
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

Cuadro B.2: Permutaciones Inicial (P_i) y Final (P_f) del algoritmo DES.

Permutación E															
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
Permutación 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

Cuadro B.3: Permutaciones E y P para la función f de DES.

Permutación EP1														
	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

Permutación EP2														
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

Cuadro B.4: Permutaciones EP1 y EP2 para DES.

64 bits, de los que conserva sólo 56, mientras que la segunda toma 56, y devuelve 48.

B.1.3. Valores de prueba

Una vez que tengamos implementado nuestro algoritmo DES, conviene asegurarse de que funciona adecuadamente. Se incluyen en esta sección algunos valores de prueba, que contienen todos los datos intermedios que se emplean en el algoritmo, para que el lector pueda compararlos y asegurarse de que su programa es correcto. Los datos están representados en hexadecimal, de izquierda a derecha.

Subclaves

```
Clave : 0123456789ABCDEF

Eleccion permutada :F0CCAA0AACCF00 -> L=F0CCAA0 R=AACCF00
Llaves Intermedias (Ki):
K01=0B02679B49A5 K02=69A659256A26 K03=45D48AB428D2 K04=7289D2A58257
K05=3CE80317A6C2 K06=23251E3C8545 K07=6C04950AE4C6 K08=5788386CE581
K09=C0C9E926B839 K10=91E307631D72 K11=211F830D893A K12=7130E5455C54
K13=91C4D04980FC K14=5443B681DC8D K15=B691050A16B5 K16=CA3D03B87032

-----
```

```
Clave : 23FE536344578A49

Eleccion permutada :42BE0B26F32C26 -> L=42BE0B2 R=6F32C26
Llaves Intermedias (Ki):
K01=A85AC6026ADB K02=253612F02DC3 K03=661CD4AE821F K04=5EE0505777C2
K05=0EC53A3C8169 K06=EE010FC2FC46 K07=2B8A096CA7B8 K08=0938BAB95C4B
K09=11C2CC6B1F64 K10=10599698C9BA K11=342965455E15 K12=836425DB20F8
K13=C907B4A1DB0D K14=D492A91236B6 K15=939262FD09A5 K16=B0AA1B27E2A4
```

Codificación

```
Codificando con  Clave : 0123456789ABCDEF

Texto Claro      :0000000000000000
Bloque permutado :0000000000000000
Paso01 : L=00000000 R=2F52D0BD   Paso02 : L=2F52D0BD R=0CB9A16F
Paso03 : L=0CB9A16F R=15C84A76   Paso04 : L=15C84A76 R=8E857E15
Paso05 : L=8E857E15 R=20AC7F5A   Paso06 : L=20AC7F5A R=526671A7
Paso07 : L=526671A7 R=D1AE9EE9   Paso08 : L=D1AE9EE9 R=6C4BBB2C
Paso09 : L=6C4BBB2C R=92882868   Paso10 : L=92882868 R=694A6072
Paso11 : L=694A6072 R=A0A3F716   Paso12 : L=A0A3F716 R=0A0D3F66
Paso13 : L=0A0D3F66 R=E672C20E   Paso14 : L=E672C20E R=C0DBACF2
Paso15 : L=C0DBACF2 R=0B78E40C   Paso16 : L=0B78E40C R=2F4BCFCD
Resultado sin permutar:2F4BCFCD0B78E40C
Resultado final       :D5D44FF720683D0D

-----

Codificando con  Clave : 0000000000000000

Texto Claro      :123456789ABCDEF0
Bloque permutado :CCFF6600F0AA7855
Paso01 : L=F0AA7855 R=E0D40658   Paso02 : L=E0D40658 R=BA8920BC
Paso03 : L=BA8920BC R=90264C4F   Paso04 : L=90264C4F R=2E3FA1F4
Paso05 : L=2E3FA1F4 R=8D42B315   Paso06 : L=8D42B315 R=8769003E
Paso07 : L=8769003E R=9F14B42F   Paso08 : L=9F14B42F R=E48646E9
Paso09 : L=E48646E9 R=6B185CDC   Paso10 : L=6B185CDC R=4E789B16
Paso11 : L=4E789B16 R=F3AA9FA8   Paso12 : L=F3AA9FA8 R=56397838
Paso13 : L=56397838 R=541678B2   Paso14 : L=541678B2 R=A4C1CE1A
Paso15 : L=A4C1CE1A R=191E936E   Paso16 : L=191E936E R=8C0D6935
Resultado sin permutar:8C0D6935191E936E
Resultado final       :9D2A73F6A9070648

-----

Codificando con  Clave : 23FE536344578A49

Texto Claro      :123456789ABCDEF0
Bloque permutado :CCFF6600F0AA7855
Paso01 : L=F0AA7855 R=A8AEA01C   Paso02 : L=A8AEA01C R=71F914D1
Paso03 : L=71F914D1 R=BC196339   Paso04 : L=BC196339 R=6893EC61
```

```
Paso05 : L=6893EC61 R=D5C2706F   Paso06 : L=D5C2706F R=ABD6DDAC
Paso07 : L=ABD6DDAC R=017151AF   Paso08 : L=017151AF R=3FB9D8DA
Paso09 : L=3FB9D8DA R=3AAAC260   Paso10 : L=3AAAC260 R=283E370C
Paso11 : L=283E370C R=FBA98CD4   Paso12 : L=FBA98CD4 R=65FBC266
Paso13 : L=65FBC266 R=FCA1C494   Paso14 : L=FCA1C494 R=F7A90537
Paso15 : L=F7A90537 R=745EBD6A   Paso16 : L=745EBD6A R=86810420
Resultado sin permutar:86810420745EBD6A
Resultado final       :1862EC2AA88BA258
```

Decodificación

```
Decodificando con Clave : 0123456789ABCDEF

Texto Cifrado    :0000000000000000
Bloque permutado :0000000000000000
Paso01 : L=00000000 R=01BA8064   Paso02 : L=01BA8064 R=A657157E
Paso03 : L=A657157E R=C4DEA13D   Paso04 : L=C4DEA13D R=0C766133
Paso05 : L=0C766133 R=95AD3310   Paso06 : L=95AD3310 R=C5C12518
Paso07 : L=C5C12518 R=1FFFFF76   Paso08 : L=1FFFFF76 R=33571627
Paso09 : L=33571627 R=CA47EDD9   Paso10 : L=CA47EDD9 R=5B462EE4
Paso11 : L=5B462EE4 R=DB9C4677   Paso12 : L=DB9C4677 R=E0B23FE6
Paso13 : L=E0B23FE6 R=8A5D943F   Paso14 : L=8A5D943F R=3ABFFA37
Paso15 : L=3ABFFA37 R=FE6A1216   Paso16 : L=FE6A1216 R=5CBDAD14
Resultado sin permutar:5CBDAD14FE6A1216
Resultado final       :14AAD7F4DBB4E094

-----

Decodificando con Clave : 0000000000000000

Texto Cifrado    :123456789ABCDEF0
Bloque permutado :CCFF6600F0AA7855
Paso01 : L=F0AA7855 R=E0D40658   Paso02 : L=E0D40658 R=BA8920BC
Paso03 : L=BA8920BC R=90264C4F   Paso04 : L=90264C4F R=2E3FA1F4
Paso05 : L=2E3FA1F4 R=8D42B315   Paso06 : L=8D42B315 R=8769003E
Paso07 : L=8769003E R=9F14B42F   Paso08 : L=9F14B42F R=E48646E9
Paso09 : L=E48646E9 R=6B185CDC   Paso10 : L=6B185CDC R=4E789B16
Paso11 : L=4E789B16 R=F3AA9FA8   Paso12 : L=F3AA9FA8 R=56397838
Paso13 : L=56397838 R=541678B2   Paso14 : L=541678B2 R=A4C1CE1A
Paso15 : L=A4C1CE1A R=191E936E   Paso16 : L=191E936E R=8C0D6935
Resultado sin permutar:8C0D6935191E936E
Resultado final       :9D2A73F6A9070648

-----

Decodificando con Clave : 23FE536344578A49

Texto Cifrado    :123456789ABCDEF0
Bloque permutado :CCFF6600F0AA7855
Paso01 : L=F0AA7855 R=3C272434   Paso02 : L=3C272434 R=0349A079
Paso03 : L=0349A079 R=57DB85A0   Paso04 : L=57DB85A0 R=2456EB13
```

```
Paso05 : L=2456EB13 R=0664691A   Paso06 : L=0664691A R=A7E17FC4
Paso07 : L=A7E17FC4 R=5C492B70   Paso08 : L=5C492B70 R=5DA12B1E
Paso09 : L=5DA12B1E R=A8F499FD   Paso10 : L=A8F499FD R=3556E6F4
Paso11 : L=3556E6F4 R=DA8A4F75   Paso12 : L=DA8A4F75 R=D544F4AE
Paso13 : L=D544F4AE R=6A25EFF3   Paso14 : L=6A25EFF3 R=30E29C71
Paso15 : L=30E29C71 R=5F3B58B8   Paso16 : L=5F3B58B8 R=AF054FAE
Resultado sin permutar:AF054FAE5F3B58B8
Resultado final       :F4E5D5EFAA638C43
```

B.2. IDEA

Incluimos ahora valores de prueba para el algoritmo IDEA, tanto para las claves intermedias Z_i de codificación y decodificación, como para los valores de las X_i en cada ronda. Los datos, al igual que en el caso de DES, están representados en hexadecimal. Nótese que la interpretación numérica de cada registro de 16 bits es, a diferencia de algoritmos como MD5, de tipo *big endian*. Esto significa que el primer *byte* en la memoria es el más significativo.

Subclaves

```
Clave: 0123 4567 89AB CDEF 0123 4567 89AB CDEF

Claves Intermedias Zi (Codificacion):
Ronda 1 : 0123 4567 89AB CDEF 0123 4567
Ronda 2 : 89AB CDEF CF13 579B DE02 468A
Ronda 3 : CF13 579B DE02 468A 37BC 048D
Ronda 4 : 159E 26AF 37BC 048D 159E 26AF
Ronda 5 : 1A2B 3C4D 5E6F 7809 1A2B 3C4D
Ronda 6 : 5E6F 7809 9ABC DEF0 1234 5678
Ronda 7 : 9ABC DEF0 1234 5678 E024 68AC
Ronda 8 : F135 79BD E024 68AC F135 79BD
Ronda 9 : 59E2 6AF3 7BC0 48D1

Claves Intermedias Zi (Decodificacion):
Ronda 1 : 74E6 950D 8440 BBF8 F135 79BD
```

```
Ronda 2 : AC8A 1FDC 8643 8794 E024 68AC
Ronda 3 : 6378 EDCC 2110 2CAD 1234 5678
Ronda 4 : 743E 6544 87F7 77DA 1A2B 3C4D
Ronda 5 : 1E4E A191 C3B3 E01F 159E 26AF
Ronda 6 : B2B4 C844 D951 7A66 37BC 048D
Ronda 7 : 963D 21FE A865 A086 DE02 468A
Ronda 8 : 3F93 30ED 3211 4F6A 0123 4567
Ronda 9 : 35AA BA99 7655 153B

-----

Clave: 6382 6F7E 8AB1 0453 BFED 93DC D810 9472

Claves Intermedias Zi (Codificacion):
Ronda 1 : 6382 6F7E 8AB1 0453 BFED 93DC
Ronda 2 : D810 9472 FD15 6208 A77F DB27
Ronda 3 : B9B0 2128 E4C7 04DE 114E FFB6
Ronda 4 : 4F73 6042 51C9 8E09 BDFA 2AC4
Ronda 5 : 6C9E E6C0 84A3 931C 137B F455
Ronda 6 : 8822 9DFF 8109 4726 3826 F7E8
Ronda 7 : AB10 453B FED9 3DCD 4C70 4DEF
Ronda 8 : D156 208A 77FD B27B 9B02 128E
Ronda 9 : DFA2 AC41 14EF FB64

Claves Intermedias Zi (Decodificacion):
Ronda 1 : 77BD 53BF EB11 C3BE 9B02 128E
Ronda 2 : CB03 8803 DF76 063B 4C70 4DEF
Ronda 3 : FF28 0127 BAC5 A8F7 3826 F7E8
Ronda 4 : 3921 7EF7 6201 B97D 137B F455
Ronda 5 : 6334 7B5D 1940 8F7B BDFA 2AC4
Ronda 6 : 7FF2 AE37 9FBE 470C 114E FFB6
Ronda 7 : DBFB 1B39 DED8 B150 A77F DB27
Ronda 8 : 3989 02EB 6B8E FB04 BFED 93DC
Ronda 9 : 2E3D 9082 754F B125

-----
```

```
Clave: 1111 2222 3333 4444 5555 6666 7777 8888

Claves Intermedias Zi (Codificacion):
Ronda 1 : 1111 2222 3333 4444 5555 6666
Ronda 2 : 7777 8888 4466 6688 88AA AACC
Ronda 3 : CCEE EF11 1022 2244 1111 5555
Ronda 4 : 9999 DDDE 2220 4444 8888 CCCD
Ronda 5 : AB33 33BB BC44 4088 8911 1199
Ronda 6 : 9A22 22AA 7778 8881 1112 2223
Ronda 7 : 3334 4445 5556 6667 0222 2444
Ronda 8 : 4666 6888 8AAA ACCC CEEE F111
Ronda 9 : 888C CCD1 1115 5559

Claves Intermedias Zi (Decodificacion):
Ronda 1 : D747 332F EEEB 199A CEEE F111
Ronda 2 : 2F67 7556 9778 9C34 0222 2444
Ronda 3 : AAAD AAAA BBBB 0005 1112 2223
Ronda 4 : 9791 8888 DD56 54A1 8911 1199
Ronda 5 : E637 43BC CC45 6BF7 8888 CCCD
Ronda 6 : 2AAA DDE0 2222 DFFF 1111 5555
Ronda 7 : CF04 EFDE 10EF 3F3E 88AA AACC
Ronda 8 : 5B6D BB9A 7778 D973 5555 6666
Ronda 9 : 7FF9 DDDE CCCD DFFF
```

Codificación

```
Codificando con Clave: 0123 4567 89AB CDEF
                       0123 4567 89AB CDEF

              X1   X2   X3   X4
Texto Claro: 0000 0000 0000 0000
Ronda  1   : 101C 6769 FD5D 8A28
Ronda  2   : 5F13 2568 288F 1326
Ronda  3   : BA0B A218 1F43 D376
Ronda  4   : 700D 8CE7 C7EE 4315
Ronda  5   : 7EC9 402F 8593 58EE
```

```
Ronda  6   : 478C FFA0 EBFF 2668
Ronda  7   : 348A 5D2B DFD1 E289
Ronda  8   : 5500 73E7 FAD6 5353
Resultado  : EC29 65C9 EFA7 4710

-----

Codificando con Clave: 6382 6F7E 8AB1 0453
                       BFED 93DC D810 9472

              X1   X2   X3   X4
Texto Claro: 0123 4567 89AB CDEF
Ronda  1   : 14E6 1CEF 9EE7 5701
Ronda  2   : E7A7 30E6 FFE5 B63C
Ronda  3   : 79A2 D4C4 EDCA 4B56
Ronda  4   : 095B 4ACF B0B8 B584
Ronda  5   : C6B0 D5D9 CCF4 C359
Ronda  6   : 4FB9 7BFD BF7A BB4E
Ronda  7   : 8219 6501 11EB B6EC
Ronda  8   : F2A5 C848 9746 6910
Resultado  : 7374 4387 DD37 5315

-----

Codificando con Clave: 1111 2222 3333 4444
                       5555 6666 7777 8888

              X1   X2   X3   X4
Texto Claro: 6E63 7F8A 8B8C 8394
Ronda  1   : B370 EDF7 C835 49A3
Ronda  2   : E798 CE57 118E 94EA
Ronda  3   : 6A74 FE29 618B 52D9
Ronda  4   : 8C64 BCB9 5E6C 0DE6
Ronda  5   : 1DE0 615A FB09 D5CD
Ronda  6   : 1872 CF37 E332 557B
Ronda  7   : A47C 34B1 F343 A473
Ronda  8   : C87D F1BD 131B 6E87
```

```
Resultado  : A16D DFEC 02D2 1B16
```

Decodificación

```
Decodificando con Clave: 0123 4567 89AB CDEF
                         0123 4567 89AB CDEF

                 X1   X2   X3   X4
Texto Cifrado: 0000 0000 0000 0000
Ronda  1     : 39EB 36B0 E85D 3959
Ronda  2     : 9FDD 04DB B915 178F
Ronda  3     : C190 33CE 5D6F D44F
Ronda  4     : 3AB1 172A CDBE 744D
Ronda  5     : B874 B1F9 2D7B 9A42
Ronda  6     : 4A76 9475 6BA5 B114
Ronda  7     : BFB0 1DD6 83A0 F4A3
Ronda  8     : 02DE 8519 C980 CBD8
Resultado    : DCD3 8419 FB6E A1E1

-----

Decodificando con Clave: 6382 6F7E 8AB1 0453
                         BFED 93DC D810 9472

                 X1   X2   X3   X4
Texto Cifrado: 0123 4567 89AB CDEF
Ronda  1     : 4490 2B63 85DB 5A10
Ronda  2     : 61D8 C3DB 881D 2404
Ronda  3     : C7DB 9502 4CE9 C1FC
Ronda  4     : AFB0 58F8 1920 4DA6
Ronda  5     : E988 A044 DCCC D5A7
Ronda  6     : 0C98 B5C8 CD67 9A95
Ronda  7     : A38B 5982 EA9C D31D
Ronda  8     : 5D35 58BD FD37 4D2F
Resultado    : AACC 8DB9 CE0C 7163
```

```
-----

Decodificando con Clave: 1111 2222 3333 4444
                         5555 6666 7777 8888

                  X1   X2   X3   X4
Texto Cifrado: 6E63 7F8A 8B8C 8394
Ronda  1     : F4C7 EB12 C708 F851
Ronda  2     : 19DF 90E0 E5F2 B16B
Ronda  3     : 6C8A 4D53 8F75 C3EB
Ronda  4     : 497E BA5D E167 26BB
Ronda  5     : C558 D308 3327 BA26
Ronda  6     : 9114 9FD0 784A 2A59
Ronda  7     : 8C36 FE0F D3B9 420F
Ronda  8     : E658 1F85 E165 736D
Resultado    : 4073 BF43 EC52 8795
```

B.3. AES

Para el algoritmo AES vamos a representar, en primer lugar, los conjuntos de subclaves K_i para ejemplos de claves de cifrado de 128, 192 y 256 bits respectivamente. Cada subclave se ha representado como un conjunto de números hexadecimales de ocho dígitos, cada uno de los cuales correspondería a una columna de la matriz de clave (ver cuadro 10.5, en la página 193), de forma que los dos primeros dígitos del primer número corresponden al valor $k_{0,0}$, los dos siguientes a $k_{1,0}$, y así sucesivamente

```
Clave  : 0123456789ABCDEF0123456789ABCDEF (128 bits)
Total rondas : 10

Subclaves de cifrado:
K00 : 67452301 EFCDAB89 67452301 EFCDAB89
K01 : C09A9E62 2F5735EB 481216EA A7DFBD63
```

```
K02 : 3BC6001A 149135F1 5C83231B FB5C9E78
K03 : 87C94A15 93587FE4 CFDB5CFF 3487C287
K04 : 90D15D38 038922DC CC527E23 F8D5BCA4
K05 : D9905E4D DA197C91 164B02B2 EE9EBE16
K06 : 9EB855C3 44A12952 52EA2BE0 BC7495F6
K07 : DCDDC7A9 987CEEFB CA96C51B 76E250ED
K08 : 89E55F7A 1199B181 DB0F749A ADED2477
K09 : 7C700A57 6DE9BBD6 B6E6CF4C 1B0BEB3B
K10 : 9EDF2188 F3369A5E 45D05512 5EDBBE29


Clave  : 8765F4765A8594E74635D869
         50B78432C756365A15326D0E (192 bits)
Total rondas : 12

Subclaves de cifrado:
K00 : 76F46587 E794855A 69D83546 3284B750
K01 : 5A3656C7 0E6D3215 2F5F59A5 C8CBDCFF
K02 : A113E9B9 93975EE9 C9A1082E C7CC3A3B
K03 : CD991227 0552CED8 A4412761 37D67988
K04 : FE7771A6 39BB4B9D 938BF890 96D93648
K05 : 32981129 054E68A1 FB391907 C282529A
K06 : 2BAEEB98 BD77DDD0 8FEFCCF9 8AA1A458
K07 : 7198BD5F B31AEFC5 8DC34957 30B49487
K08 : BF5B587E 35FAFC26 44624179 F778AEBC
K09 : E8ABF593 D81F6114 6744396A 52BEC54C
K10 : 16DC8435 E1A42A89 4F53BC36 974CDD22
K11 : F008E448 A2B62104 B46AA531 55CE8FB8
K12 : 23AF37C5 B4E3EAE7 44EB0EAF E65D2FAB


Clave  : 8765F4765A8594E74635D86950B78432
         C756365A15326DE012345678E214320A (256 bits)
Total rondas : 14

Subclaves de cifrado:
K00 : 76F46587 E794855A 69D83546 3284B750
```

```
K01 : 5A3656C7 E06D3215 78563412 0A3214E2
K02 : EE93467C 0907C326 60DFF660 525B4130
K03 : 5A0FD5C3 BA62E7D6 C234D3C4 C806C726
K04 : 197B29B8 107CEA9E 70A31CFE 22F85DCE
K05 : C94E9948 732C7E9E B118AD5A 791E6A7C
K06 : 09CD5BBE 19B1B120 6912ADDE 4BEAF010
K07 : 7AC91582 09E56B1C B8FDC646 C1E3AC3A
K08 : 89B54A27 9004FB07 F91656D9 B2FCA6C9
K09 : 4D79315F 449C5A43 FC619C05 3D82303F
K10 : FC925933 6C96A234 9580F4ED 277C5224
K11 : 81693169 C5F56B2A 3994F72F 0416C710
K12 : 36601ED5 5AF6BCE1 CF76480C E80A1A28
K13 : 1A0E935D DFFBF877 E66F0F58 E279C848
K14 : 64F8A87D 3E0E149C F1785C90 197246B8
```

Seguidamente representaremos los valores intermedios de cifrado y descifrado de un bloque de datos para estas tres claves. En cada línea se representa la matriz de estado (ver cuadro 10.4), de forma análoga a la que se ha empleado para representar la matriz de clave.

```
Clave  : 0123456789ABCDEF0123456789ABCDEF (128 bits)

CIFRADO:
Bloque   : 7563957A 7C6E9274 6E87F937 A2F4AB04  (128 bits)
Ronda 01 : 201D4467 70B06937 8FBFA93C 1D4757CF
Ronda 02 : 0486AEC2 951CEAA5 87BCD35D CE92939C
Ronda 03 : EDEF12D7 E6C5DB1E E2E45A51 8D1F89E9
Ronda 04 : C398674B C9822958 E84F1592 0C4556C0
Ronda 05 : C707CA8E A5C9F7EE C2BB119F D177A68A
Ronda 06 : D4D13E6C 46952EB2 F24BAAEC 6D5929FE
Ronda 07 : 508F2AEF 746D34C0 D13BF25D 288DCBBA
Ronda 08 : E500843A 4302ADE4 5E7E684E DE924E02
Ronda 09 : 5585CDD0 43ADC584 1B81F49C 1EBB3594
Ronda 10 : 74B460BC 4496A083 BDBF6D1A 5B297D80
Cifrado  : 74B460BC 4496A083 BDBF6D1A 5B297D80

DESCIFRADO:
```

```
Bloque    : 7563957A 7C6E9274 6E87F937 A2F4AB04  (128 bits)
Ronda 01  : B319F6D6 F00601B2 031D107C 1E876239
Ronda 02  : 3968DE25 C6266F04 A33BA0FF D7C06313
Ronda 01  : 9706478A 462565BA 164FF166 8FECC208
Ronda 02  : 87C9E8FB 25B34D03 D74DE19C 5FA360A5
Ronda 01  : C808ECD8 A3E29DAE 94293CCB 6304742C
Ronda 02  : 6528BC87 22719EE4 FD034F6F 2EF66891
Ronda 01  : 6FA21399 A1A4D30D 45B2E47D B5A718DF
Ronda 02  : 60C97EE2 7509D120 7C04EB6C 8DE033A3
Ronda 01  : 75C4C689 5B36142C A18AEADD 22F1EB70
Ronda 10  : 3E08FE25 DE23F126 F00782B7 1D64561D
Descifrado: 3E08FE25 DE23F126 F00782B7 1D64561D


Clave  : 8765F4765A8594E74635D869
         50B78432C756365A15326D0E (192 bits)

CIFRADO:
Bloque   : 7563957A 7C6E9274 6E87F937 A2F4AB04  (128 bits)
Ronda 01 : 160FB8C4 526A9EC9 D0AFCB25 70621BF8
Ronda 02 : 6FCAABF7 D15A8F7D 9A5EDF3E 37A5BC37
Ronda 03 : B1FE1D21 418746AA 9DCA21F6 FA2C13FA
Ronda 04 : C4A63E0D 9C5AAA4F B71F18E7 DCDA3D84
Ronda 05 : 3AD99ABB AD937C2E 81572FED D9E7C4E8
Ronda 06 : 726C6E54 FA30A491 CF114FD5 289E7E5A
Ronda 07 : E9DC1656 D1F328F5 5BEEFF85 55D84773
Ronda 08 : CCE9EE83 33D87F86 099585FE 6D8EC86F
Ronda 09 : 99765788 F3391287 2F36C0DD 7F13F5B7
Ronda 10 : D732AFDE BED82C86 D7A9B478 DDFE7792
Ronda 11 : 35EBB790 C52B1D57 C609E1EC 8927113C
Ronda 12 : 53C657C8 41EB61D4 1BC2421F 0CC6F928
Cifrado  : 53C657C8 41EB61D4 1BC2421F 0CC6F928

DESCIFRADO:

Bloque    : 7563957A 7C6E9274 6E87F937 A2F4AB04  (128 bits)
```

```
Ronda 01  : 8A102DA6 32EE44E5 0F5EA9B9 85A8D1DB
Ronda 02  : EAC1F79A C3EE67FB F8AAA566 5C1EF22D
Ronda 01  : 109FC072 45BC7406 7AE5206B 0DBD735E
Ronda 02  : FD4EEFDE 3CC42E4F 50BB5BE9 673BA16D
Ronda 01  : 623F847F 2246E5C3 FDADA89E 5AA2D81C
Ronda 02  : 8ADB4E04 97319AB8 52A9E478 F16FEFB9
Ronda 01  : 48A546C4 56732D30 A735D297 8292A0A3
Ronda 02  : C253B1A9 D32607F4 E6D6C966 623A15C6
Ronda 01  : 6076F92C D62A52EA 10204094 B9CB8884
Ronda 02  : 88241F51 3CBD888F 6CBEEFBC F7BB9655
Ronda 01  : 7DA56D33 B33A0C47 7BAA5759 51C5B996
Ronda 12  : 94622E60 11AC4FF2 45976B5C 20D50554
Descifrado: 94622E60 11AC4FF2 45976B5C 20D50554


Clave  : 8765F4765A8594E74635D86950B78432
         C756365A15326DE012345678E214320A (256 bits)
CIFRADO:
Bloque   : 7563957A 7C6E9274 6E87F937 A2F4AB04  (128 bits)
Ronda 01 : 160FB8C4 526A9E27 67C2C272 6DAAE23A
Ronda 02 : 51EA071B CD262D8C 3E4861B7 99CCC7EB
Ronda 03 : 7E32CCE3 2800F0B7 C7C7F049 02E624F7
Ronda 04 : 04FB5028 8002D19E 02A99DAD F2D8E262
Ronda 05 : DACD92A2 DD89451C 4FE6B50C CF2A40F9
Ronda 06 : AEA43CAB 0356A2B2 2AB55277 535718FA
Ronda 07 : B4A5EBA6 412FAC38 A684D752 EF68376F
Ronda 08 : BF57D255 45579B83 B0DFB737 F7DD1C5F
Ronda 09 : C4A02922 46505017 D1CA1979 8C482CE5
Ronda 10 : F35D1EF6 FE10F4BA 326AB6DB 32AE9F4F
Ronda 11 : FCE5A501 D8E0274E D865B039 841FCCFB
Ronda 12 : 7E4AF5E5 C3E6C807 BC97AAF4 38B13938
Ronda 13 : 828F3938 6332099E F21541F6 70E4B9B0
Ronda 14 : 6E8B7B83 674D5839 19356AFA E935735B
Cifrado  : 6E8B7B83 674D5839 19356AFA E935735B

DESCIFRADO:
```

```
Bloque    : 7563957A 7C6E9274 6E87F937 A2F4AB04  (128 bits)
Ronda 01  : D90293DE EFCBC692 87620BEC A9E1A3D9
Ronda 02  : 6DF747AF 78006F1F 40DAFBE8 D333B4C3
Ronda 01  : 496618FA C59E36F5 3ABC05F3 7011CFA5
Ronda 02  : 13502465 4FB09CFA 6745440A BFC062A8
Ronda 01  : 639BEB46 25C9AD76 242C9AA9 39066FDC
Ronda 02  : F87CDE96 69CB5302 C8AE6B76 B2FEAF5B
Ronda 01  : FC6D0433 C8E51A5D DE349F93 2D113855
Ronda 02  : 8F872F53 D54D5DAA 1E8CB849 E8B2DC30
Ronda 01  : 33CB011A 1DE03C16 A468722A 2C2A38AA
Ronda 02  : 68CE0A4D 3FB38D7D FC8060FB BCCD1AB9
Ronda 01  : 14D2CABB 7D3AAFE8 48675BF3 B5133A20
Ronda 02  : BEF20489 FF3AD947 5B211677 4EB766DA
Ronda 01  : 28C8A02E B3526182 0C735A92 ACDA0765
Ronda 14  : 691FB267 1134AC93 C77D9FD5 FA385CF1
Descifrado: 691FB267 1134AC93 C77D9FD5 FA385CF1
```

B.4. MD5

En esta sección detallaremos todos los valores intermedios que se obtienen al aplicar el algoritmo MD5 a cuatro ejemplos diferentes. El primer campo es la cadena que se va a procesar, excluyendo las comillas. El segundo es el bloque de 512 bits de entrada —todos los ejemplos que se han incluido producen un único bloque— escrito en hexadecimal, que dicha cadena genera, en el que se puede apreciar cómo tras los códigos ASCII correspondientes aparece el valor 80, es decir, un uno seguido de ceros, y cómo los últimos 64 bits —correspondientes a los dieciséis últimos dígitos hexadecimales— representan la longitud total, en bits, de la cadena. Seguidamente, se especifican los valores de los registros a, b, c y d que se obtienen en cada paso, y para terminar se da el resultado final de 128 bits, en formato hexadecimal. Nótese que, en este caso, la representación como valores enteros de los registros de 32 bits es de tipo *little endian*, es decir, que el *byte* que primero aparece en el bloque es el menos significativo del valor entero

correspondiente.

```
Cadena: "a" (8 bits)
Bloque: 61800000000000000000000000000000000000000000000000
        00000000000000000000000000000000000000000000000000
        00000000000000000800000000000000
Inicio  : a=67452301   b=EFCDAB89   c=98BADCFE   d=10325476
Ronda  4: a=A56017F4   b=607D9686   c=E65857A7   d=F2D58361
Ronda  8: a=3A9D5BCC   b=A8AF6DA5   c=D31DDC83   d=E0A07DB7
Ronda 12: a=BE580957   b=68493D6A   c=F5FDD933   d=F386BEA6
Ronda 16: a=44244CF8   b=F01E3CE2   c=6360A45F   d=D0FE9B27
Ronda 20: a=9C341767   b=8D25CC66   c=E39FFD23   d=970AB3A9
Ronda 24: a=8C444930   b=373BEAB0   c=2DACB8A3   d=7267097A
Ronda 28: a=F175E3AD   b=C8F891B4   c=87B7F475   d=9D5DF67E
Ronda 32: a=93842E98   b=3745961F   c=94A2EBEE   d=C7043B64
Ronda 36: a=BD607D1E   b=DAF7F308   c=BF8B4F98   d=A6F72085
Ronda 40: a=35A82A7A   b=CF7E60DB   c=5ABE099C   d=89E0EC97
Ronda 44: a=75C151E2   b=CC6F5E9E   c=0C0E6AC4   d=942E0C86
Ronda 48: a=0AC50E18   b=918F93BB   c=8A4A6356   d=79CA7845
Ronda 52: a=CAB8FE42   b=1EE405EB   c=36269C3F   d=6A4DAEEE
Ronda 56: a=982C7861   b=893501C0   c=71FC7709   d=6812A362
Ronda 60: a=FEBD62FD   b=AA4D8AE3   c=53E33526   d=28936A74
Ronda 64: a=52309E0B   b=B8E94637   c=49DEE633   d=50F422F3
Resultado: 0CC175B9C0F1B6A831C399E269772661


Cadena: "test" (32 bits)
Bloque: 74657374800000000000000000000000000000000000000000
        00000000000000000000000000000000000000000000000000
        00000000000000002000000000000000
Inicio  : a=67452301   b=EFCDAB89   c=98BADCFE   d=10325476
Ronda  4: a=DED2A12E   b=DAF27C2C   c=F1824515   d=0F74EDAC
Ronda  8: a=C5ADAD00   b=E95CAA49   c=480530DA   d=B7AC6179
Ronda 12: a=D2B0528F   b=39C7F222   c=E81C99B1   d=3A68633F
Ronda 16: a=70426956   b=02F9BE0B   c=1C3DC813   d=6C99C85B
Ronda 20: a=E6BCA679   b=DCE63C0F   c=A1551890   d=95200EE0
Ronda 24: a=090098DD   b=EB97FA59   c=04BA62B4   d=15C03EC7
```

```
Ronda 28: a=7F3420DE   b=E2416EB4   c=89F5CC1E   d=D933566E
Ronda 32: a=1DFC71B1   b=377D2CE4   c=8841B1FD   d=4CB69E35
Ronda 36: a=4F880ED5   b=796304FE   c=52B55B13   d=38CC24FB
Ronda 40: a=EF7A4FEE   b=42FB05F0   c=0F3B052F   d=A79F8A38
Ronda 44: a=6A509FA0   b=4995D409   c=190065DE   d=9009C912
Ronda 48: a=95B45DE9   b=A5B6C91B   c=412D4C7B   d=D02E07C9
Ronda 52: a=F09D8296   b=32C92920   c=10F833EA   d=FAA53851
Ronda 56: a=79F7507F   b=CA8F6F9D   c=19E4244E   d=3DE059DA
Ronda 60: a=1176200C   b=82BC77EB   c=997A7EAD   d=08C989F3
Ronda 64: a=66266C08   b=840575BD   c=EA9401CC   d=E681D2B0
Resultado: 098F6BCD4621D373CADE4E832627B4F6


Cadena: "experimento" (88 bits)
Bloque: 6578706572696D656E746F8000000000000000000000000
        0000000000000000000000000000000000000000000000000
        0000000000000005800000000000000
Inicio  : a=67452301   b=EFCDAB89   c=98BADCFE   d=10325476
Ronda  4: a=5D5C19A7   b=46C38F9E   c=9D66E599   d=C18D8C6C
Ronda  8: a=5A1B370A   b=F00B0D14   c=C2337BF6   d=2BBE5411
Ronda 12: a=999F88DB   b=DD5473D0   c=4E1035F1   d=860ED16D
Ronda 16: a=827A70F9   b=5225BF6B   c=D7A665AF   d=FE0775F8
Ronda 20: a=878C66F1   b=40BEF53B   c=8A5ABAE0   d=23DC997C
Ronda 24: a=0C045617   b=468F7087   c=46F386B6   d=960E2AF4
Ronda 28: a=E34880F5   b=B0C9B131   c=58BCA90E   d=0790302C
Ronda 32: a=6BEE14CD   b=538F2F39   c=FD147E9E   d=249FB3A8
Ronda 36: a=B2623128   b=34B78DF5   c=D3D94D7C   d=0AB7F770
Ronda 40: a=DB335B6F   b=5A3DCDEA   c=A5C3B46A   d=7E4B5806
Ronda 44: a=B27D89A2   b=6841550D   c=257A8EB5   d=B9C1C281
Ronda 48: a=7D088655   b=789F1C2C   c=060B818B   d=02DB24DB
Ronda 52: a=0E3F05A0   b=545B70C4   c=7660DA54   d=A86030D5
Ronda 56: a=AABCB829   b=47D564F9   c=6CDBF4D6   d=F1E81106
Ronda 60: a=A4298EC8   b=B5246707   c=B26DA205   d=DBB56A4C
Ronda 64: a=553D262F   b=A4F73804   c=55138A82   d=83130C35
Resultado: 304982BC8DE3C4948067CEEDAB604593
```

```
Cadena: "kriptopolis" (88 bits)
Bloque: 6B726970746F706F6C6973800000000000000000000000000
        00000000000000000000000000000000000000000000000000
        00000000000000058000000000000000
Inicio  : a=67452301   b=EFCDAB89   c=98BADCFE   d=10325476
Ronda  4: a=D9D91CAC   b=24B06776   c=E5B4EE28   d=3E1ABFD1
Ronda  8: a=2CC5E57D   b=C013F682   c=937C5146   d=5C1CDC32
Ronda 12: a=F17F3A1B   b=3AF41990   c=84045E90   d=45F1B4B2
Ronda 16: a=45781161   b=499471DB   c=9521F17B   d=2DD56DA0
Ronda 20: a=E1077423   b=E2B7D25E   c=288A1472   d=382A7B92
Ronda 24: a=DDD330B8   b=0C9BCC03   c=060CC302   d=2A9772C2
Ronda 28: a=96B3C004   b=7E2F1EAC   c=2F65093C   d=5B41A3FC
Ronda 32: a=55C59563   b=E8FC5DA2   c=25D8CAE4   d=3D5795BE
Ronda 36: a=4C297776   b=6518EC96   c=0F032874   d=078CF1A4
Ronda 40: a=441953E2   b=73C80FB2   c=2B495D85   d=B6DBBFED
Ronda 44: a=24C417D7   b=8A6D297C   c=C3FD834A   d=81C5AC48
Ronda 48: a=F94C1268   b=18268270   c=39A9E934   d=E9406B33
Ronda 52: a=55703A51   b=F4D6A7B3   c=EF9EDE35   d=460E123F
Ronda 56: a=42081E66   b=50A3C1E5   c=F85BC50F   d=ABE4D855
Ronda 60: a=F4D3A73C   b=9E487E08   c=BEAE5BB7   d=8DC8E081
Ronda 64: a=DC5B1FD3   b=C48CAC59   c=72B24EFA   d=3AAD460B
Resultado: D442A043E2575AB4F82B6D0B819ADF4A
```

B.5. SHA-1

Se incluyen aquí los valores intermedios del algoritmo SHA-1 para las mismas cuatro cadenas de la sección anterior. Recordemos que en este caso el orden de los *bytes* a la hora de representar enteros es *big endian*, de forma que, por ejemplo, en la representación de la longitud b del mensaje total el último *byte* es el menos significativo.

```
Cadena: "a" (8 bits)
Bloque: 6180000000000000000000000000000000000000000000000
        00000000000000000000000000000000000000000000000000
        00000000000000000000000000000008
Ini: a=67452301   b=EFCDAB89   c=98BADCFE   d=10325476   e=C3D2E1F0
R02: a=8D43E36D   b=013498B3   c=59D148C0   d=7BF36AE2   e=98BADCFE
```

```
R04: a=98ECF029   b=178D2F08   c=6350F8DB   d=C04D262C   e=59D148C0
R06: a=24FDDFA9   b=9531EFB8   c=663B3C0A   d=05E34BC2   e=6350F8DB
R08: a=1706BD51   b=628293E2   c=493F77EA   d=254C7BEE   e=663B3C0A
R10: a=B48452FD   b=E6E3DBB3   c=45C1AF54   d=98A0A4F8   e=493F77EA
R12: a=06A530AF   b=910E0091   c=6D2114BF   d=F9B8F6EC   e=45C1AF54
R14: a=94A32E14   b=DE9B35CA   c=C1A94C2B   d=64438024   e=6D2114BF
R16: a=2ECAE907   b=3CD2D518   c=2528CB85   d=B7A6CD72   e=C1A94C2B
R18: a=16FFEBFC   b=5FADB00B   c=CBB2BA41   d=0F34B546   e=2528CB85
R20: a=3AD1E596   b=AB5979F5   c=05BFFAFF   d=D7EB6C02   e=CBB2BA41
R22: a=57668EF8   b=0DD747B1   c=8EB47965   d=6AD65E7D   e=05BFFAFF
R24: a=587C3B69   b=57212656   c=15D9A3BE   d=4375D1EC   e=8EB47965
R26: a=37225DA7   b=94A32676   c=561F0EDA   d=95C84995   e=15D9A3BE
R28: a=D72C806E   b=C073A57E   c=CDC89769   d=A528C99D   e=561F0EDA
R30: a=229C3156   b=831D03EB   c=B5CB201B   d=B01CE95F   e=CDC89769
R32: a=94F65775   b=B4F37984   c=88A70C55   d=E0C740FA   e=B5CB201B
R34: a=C174A741   b=2A032F9B   c=653D95DD   d=2D3CDE61   e=88A70C55
R36: a=25E95D6B   b=481844C5   c=705D29D0   d=CA80CBE6   e=653D95DD
R38: a=42285BB1   b=FC08D9F3   c=C97A575A   d=52061131   e=705D29D0
R40: a=AC15A69D   b=A3B72B37   c=508A16EC   d=FF02367C   e=C97A575A
R42: a=504F64C1   b=E6CD1F2C   c=6B0569A7   d=E8EDCACD   e=508A16EC
R44: a=7D09F328   b=B45FC818   c=5413D930   d=39B347CB   e=6B0569A7
R46: a=B44DB2B6   b=7B7354D5   c=1F427CCA   d=2D17F206   e=5413D930
R48: a=F9ADD677   b=C43968F2   c=AD136CAD   d=5EDCD535   e=1F427CCA
R50: a=2D5EDB70   b=5432B748   c=FE6B759D   d=B10E5A3C   e=AD136CAD
R52: a=7C1B92B1   b=5C350FCB   c=0B57B6DC   d=150CADD2   e=FE6B759D
R54: a=C530EAD6   b=C60F44D9   c=5F06E4AC   d=D70D43F2   e=0B57B6DC
R56: a=AF4F737E   b=D7A11B36   c=B14C3AB5   d=7183D136   e=5F06E4AC
R58: a=68346C20   b=81923337   c=ABD3DCDF   d=B5E846CD   e=B14C3AB5
R60: a=C682F53F   b=80C84724   c=1A0D1B08   d=E0648CCD   e=ABD3DCDF
R62: a=CEEC2B04   b=193B35BD   c=F1A0BD4F   d=203211C9   e=1A0D1B08
R64: a=377A9914   b=829ED919   c=33BB0AC1   d=464ECD6F   e=F1A0BD4F
R66: a=CD6D6C13   b=DAC285B9   c=0DDEA645   d=60A7B646   e=33BB0AC1
R68: a=D3FCDF01   b=639762AA   c=F35B5B04   d=76B0A16E   e=0DDEA645
R70: a=C4B7981A   b=3E5A0CC1   c=74FF37C0   d=98E5D8AA   e=F35B5B04
R72: a=42650D26   b=26F94DF5   c=B12DE606   d=4F968330   e=74FF37C0
R74: a=65E8D295   b=9487B5C9   c=90994349   d=49BE537D   e=B12DE606
R76: a=CE8E1721   b=064CC166   c=597A34A5   d=6521ED72   e=90994349
R78: a=22890379   b=A6E259D2   c=73A385C8   d=81933059   e=597A34A5
R80: a=1FB2C136   b=0AD7FC73   c=48A240DE   d=A9B89674   e=73A385C8
Resultado: 86F7E437FAA5A7FCE15D1DDCB9EAEAEA377667B8


Cadena: "test" (32 bits)
Bloque: 7465737480000000000000000000000000000000000000000
        0000000000000000000000000000000000000000000000000
        0000000000000000000000000000020
Ini: a=67452301   b=EFCDAB89   c=98BADCFE   d=10325476   e=C3D2E1F0
R02: a=69F251EF   b=141A0C27   c=59D148C0   d=7BF36AE2   e=98BADCFE
R04: a=D698D619   b=AD78FF44   c=DA7C947B   d=C5068309   e=59D148C0
R06: a=27A84E9E   b=4FED19DC   c=75A63586   d=2B5E3FD1   e=DA7C947B
R08: a=D1725C31   b=8FBF195D   c=89EA13A7   d=13FB4677   e=75A63586
R10: a=C43997BA   b=985E8C80   c=745C970C   d=63EFC657   e=89EA13A7
R12: a=E671210F   b=DF9D4AEF   c=B10E65EE   d=2617A320   e=745C970C
```

```
R14: a=3A4BF38D   b=4E12148F   c=F99C4843   d=F7E752BB   e=B10E65EE
R16: a=0690C1E0   b=4F049361   c=4E92FCE3   d=D3848523   e=F99C4843
R18: a=28D8607D   b=ED827927   c=01A43078   d=53C124D8   e=4E92FCE3
R20: a=A594E7C5   b=D7E2BB59   c=4A36181F   d=FB609E49   e=01A43078
R22: a=3E7F1747   b=89CF51DE   c=696539F1   d=75F8AED6   e=4A36181F
R24: a=08667DF4   b=C1714F43   c=CF9FC5D1   d=A273D477   e=696539F1
R26: a=F78CDC7E   b=63420FC9   c=02199F7D   d=F05C53D0   e=CF9FC5D1
R28: a=301C99FA   b=C11D04BC   c=BDE3371F   d=58D083F2   e=02199F7D
R30: a=31E08911   b=2563E943   c=8C07267E   d=3047412F   e=BDE3371F
R32: a=63D791B9   b=99B4D18E   c=4C782244   d=C958FA50   e=8C07267E
R34: a=66AA9C75   b=0525A937   c=58F5E46E   d=A66D3463   e=4C782244
R36: a=3E4E8518   b=BF1CD105   c=59AAA71D   d=C1496A4D   e=58F5E46E
R38: a=B32B1931   b=18AB89BD   c=0F93A146   d=6FC73441   e=59AAA71D
R40: a=80F549BF   b=ED3E0D75   c=6CCAC64C   d=462AE26F   e=0F93A146
R42: a=04BDFD86   b=B6353C2E   c=E03D526F   d=7B4F835D   e=6CCAC64C
R44: a=875C7539   b=02132FC0   c=812F7F61   d=AD8D4F0B   e=E03D526F
R46: a=567A94C8   b=64C62440   c=61D71D4E   d=0084CBF0   e=812F7F61
R48: a=CA454844   b=AE30686B   c=159EA532   d=19318910   e=61D71D4E
R50: a=D24F0B76   b=F039F33C   c=32915211   d=EB8C1A1A   e=159EA532
R52: a=E0D4BBAF   b=46A89C74   c=B493C2DD   d=3C0E7CCF   e=32915211
R54: a=3F042183   b=9E1A2F5D   c=F8352EEB   d=11AA271D   e=B493C2DD
R56: a=13496F6A   b=ACB50C2A   c=CFC10860   d=67868BD7   e=F8352EEB
R58: a=646566EF   b=B3E7F37D   c=84D25BDA   d=AB2D430A   e=CFC10860
R60: a=1387E9E4   b=2A106003   c=D91959BB   d=6CF9FCDF   e=84D25BDA
R62: a=FC461EB0   b=7A8ACB6F   c=04E1FA79   d=CA841800   e=D91959BB
R64: a=F00B178D   b=4E7E642E   c=3F1187AC   d=DEA2B2DB   e=04E1FA79
R66: a=3810D68A   b=39938A7E   c=7C02C5E3   d=939F990B   e=3F1187AC
R68: a=02152E1A   b=11876ADB   c=8E0435A2   d=8E64E29F   e=7C02C5E3
R70: a=DFEB6670   b=69F18CE7   c=80854B86   d=C461DAB6   e=8E0435A2
R72: a=25C49F67   b=4F6EC7D7   c=37FAD99C   d=DA7C6339   e=80854B86
R74: a=A3CEFF95   b=882DA0C9   c=C97127D9   d=D3DBB1F5   e=37FAD99C
R76: a=5173678F   b=24CEC91C   c=68F3BFE5   d=620B6832   e=C97127D9
R78: a=0E44ADD6   b=0D7E5447   c=D45CD9E3   d=0933B247   e=68F3BFE5
R80: a=42056CE4   b=DCE3F01D   c=83912B75   d=C35F9511   e=D45CD9E3
Resultado: A94A8FE5CCB19BA61C4C0873D391E987982FBBD3


Cadena: "experimento" (88 bits)
Bloque: 6578706572696D656E746F8000000000000000000000000
        0000000000000000000000000000000000000000000000000
        0000000000000000000000000058
Ini: a=67452301   b=EFCDAB89   c=98BADCFE   d=10325476   e=C3D2E1F0
R02: a=7EBB5D72   b=052D0918   c=59D148C0   d=7BF36AE2   e=98BADCFE
R04: a=75DD0E53   b=B4F0DF48   c=9FAED75C   d=014B4246   e=59D148C0
R06: a=BF3915DD   b=05A16415   c=DD774394   d=2D3C37D2   e=9FAED75C
R08: a=9B5914B1   b=0E916082   c=6FCE4577   d=41685905   e=DD774394
R10: a=E406B102   b=F304AC67   c=66D6452C   d=83A45820   e=6FCE4577
R12: a=7E5DA8FA   b=ADCB3390   c=B901AC40   d=FCC12B19   e=66D6452C
R14: a=C474F542   b=860F061D   c=9F976A3E   d=2B72CCE4   e=B901AC40
R16: a=14CC37BD   b=519A992D   c=B11D3D50   d=6183C187   e=9F976A3E
R18: a=B07CDCEE   b=DAD274C5   c=45330DEF   d=5466A64B   e=B11D3D50
R20: a=DBA8A460   b=3C5ABB3E   c=AC1F373B   d=76B49D31   e=45330DEF
R22: a=CF0B1462   b=D9B84C74   c=36EA2918   d=8F16AECF   e=AC1F373B
```

```
R24: a=00FA7488   b=B5017A00   c=B3C2C518   d=366E131D   e=36EA2918
R26: a=678F9C3D   b=55535315   c=003E9D22   d=2D405E80   e=B3C2C518
R28: a=36ED4BC6   b=2C053172   c=59E3E70F   d=5554D4C5   e=003E9D22
R30: a=D46C6C32   b=CEF800E2   c=8DBB52F1   d=8B014C5C   e=59E3E70F
R32: a=31B75696   b=7109F222   c=B51B1B0C   d=B3BE0038   e=8DBB52F1
R34: a=7CAE903B   b=5BC99A55   c=8C6DD5A5   d=9C427C88   e=B51B1B0C
R36: a=7B134BC7   b=522A73DB   c=DF2BA40E   d=56F26695   e=8C6DD5A5
R38: a=AF987330   b=A90C28E7   c=DEC4D2F1   d=D48A9CF6   e=DF2BA40E
R40: a=3A793DF9   b=8847DFE3   c=2BE61CCC   d=EA430A39   e=DEC4D2F1
R42: a=79238476   b=B7083A1D   c=4E9E4F7E   d=E211F7F8   e=2BE61CCC
R44: a=6381087C   b=860C3C86   c=9E48E11D   d=6DC20E87   e=4E9E4F7E
R46: a=4601F05C   b=2DCC49C0   c=18E0421F   d=A1830F21   e=9E48E11D
R48: a=8EB6C83B   b=1C149969   c=11807C17   d=0B731270   e=18E0421F
R50: a=D5D79282   b=5666238B   c=E3ADB20E   d=4705265A   e=11807C17
R52: a=0C947120   b=0823F989   c=B575E4A0   d=D59988E2   e=E3ADB20E
R54: a=C218BAA5   b=920A30EA   c=03251C48   d=4208FE62   e=B575E4A0
R56: a=B5022079   b=F30D765D   c=70862EA9   d=A4828C3A   e=03251C48
R58: a=E70F5DFB   b=766D14A3   c=6D40881E   d=7CC35D97   e=70862EA9
R60: a=110BCB99   b=46BA94E7   c=F9C3D77E   d=DD9B4528   e=6D40881E
R62: a=F200718D   b=63A74A72   c=4442F2E6   d=D1AEA539   e=F9C3D77E
R64: a=23F6D36A   b=21C432A4   c=7C801C63   d=98E9D29C   e=4442F2E6
R66: a=C210491B   b=7973CC2D   c=88FDB4DA   d=08710CA9   e=7C801C63
R68: a=34DC1778   b=383D5D93   c=F0841246   d=5E5CF30B   e=88FDB4DA
R70: a=674E55A5   b=463558B7   c=0D3705DE   d=CE0F5764   e=F0841246
R72: a=E452151C   b=DB6379D9   c=59D39569   d=D18D562D   e=0D3705DE
R74: a=CE40A960   b=95A1780C   c=39148547   d=76D8DE76   e=59D39569
R76: a=42B352EB   b=7994D2F8   c=33902A58   d=25685E03   e=39148547
R78: a=E3B07323   b=ADDFDC73   c=D0ACD4BA   d=1E6534BE   e=33902A58
R80: a=21D68E46   b=6F2351DF   c=F8EC1CC8   d=EB77F71C   e=D0ACD4BA
Resultado: 891BB1475EF0FD6891A6F9C6FBAA4B92947FB6AA


Cadena: "kriptopolis" (88 bits)
Bloque: 6B726970746F706F6C6973800000000000000000000000000
        00000000000000000000000000000000000000000000000000
        00000000000000000000000000000058
Ini: a=67452301   b=EFCDAB89   c=98BADCFE   d=10325476   e=C3D2E1F0
R02: a=400081DD   b=0B270223   c=59D148C0   d=7BF36AE2   e=98BADCFE
R04: a=61557CFE   b=D9886E7F   c=50002077   d=C2C9C088   e=59D148C0
R06: a=B62CB646   b=3145031C   c=98555F3F   d=F6621B9F   e=50002077
R08: a=2D45AF5F   b=46807E85   c=AD8B2D91   d=0C5140C7   e=98555F3F
R10: a=F43BF9F6   b=A85EF180   c=CB516BD7   d=51A01FA1   e=AD8B2D91
R12: a=072108C8   b=697D55A9   c=BD0EFE7D   d=2A17BC60   e=CB516BD7
R14: a=DF29879C   b=3503FAD9   c=01C84232   d=5A5F556A   e=BD0EFE7D
R16: a=0678F27A   b=481EB2E3   c=37CA61E7   d=4D40FEB6   e=01C84232
R18: a=A15AB731   b=3EE9ADE2   c=819E3C9E   d=D207ACB8   e=37CA61E7
R20: a=AB2F7D99   b=5704D5FE   c=6856ADCC   d=8FBA6B78   e=819E3C9E
R22: a=28A91B1F   b=D90DB07B   c=6ACBDF66   d=95C1357F   e=6856ADCC
R24: a=C2BE33DB   b=4B342F34   c=CA2A46C7   d=F6436C1E   e=6ACBDF66
R26: a=1DAB7331   b=27F1416F   c=F0AF8CF6   d=12CD0BCD   e=CA2A46C7
R28: a=D85BDF45   b=AB5927D5   c=476ADCCC   d=C9FC505B   e=F0AF8CF6
R30: a=43B8E8A4   b=74386A94   c=7616F7D1   d=6AD649F5   e=476ADCCC
R32: a=3DAAE44F   b=774E7172   c=10EE3A29   d=1D0E1AA5   e=7616F7D1
```

```
R34: a=4FDC1B88   b=5D5610F1   c=CF6AB913   d=9DD39C5C   e=10EE3A29
R36: a=7E58EA7B   b=EA9DF552   c=13F706E2   d=5755843C   e=CF6AB913
R38: a=B94B8784   b=7CDB656F   c=DF963A9E   d=BAA77D54   e=13F706E2
R40: a=19B56EA2   b=88361E28   c=2E52E1E1   d=DF36D95B   e=DF963A9E
R42: a=C5C402CE   b=B4AA1C2E   c=866D5BA8   d=220D878A   e=2E52E1E1
R44: a=087893C1   b=E5C8DD17   c=B17100B3   d=AD2A870B   e=866D5BA8
R46: a=10C39FA5   b=8781E3DC   c=421E24F0   d=F9723745   e=B17100B3
R48: a=FCA2E4E9   b=00DB97DB   c=4430E7E9   d=21E078F7   e=421E24F0
R50: a=29675B14   b=E07CFB83   c=7F28B93A   d=C036E5F6   e=4430E7E9
R52: a=30C83C92   b=E8C85C50   c=0A59D6C5   d=F81F3EE0   e=7F28B93A
R54: a=873B832C   b=4D5329BE   c=8C320F24   d=3A321714   e=0A59D6C5
R56: a=EA05D012   b=5EEE15EE   c=21CEE0CB   d=9354CA6F   e=8C320F24
R58: a=2E5309FD   b=C3E8ACC8   c=BA817404   d=97BB857B   e=21CEE0CB
R60: a=63129A6C   b=9D77469B   c=4B94C27F   d=30FA2B32   e=BA817404
R62: a=10600A4F   b=EB9C641B   c=18C4A69B   d=E75DD1A6   e=4B94C27F
R64: a=B67A32B2   b=7326BFF5   c=C4180293   d=FAE71906   e=18C4A69B
R66: a=844649E5   b=E8A707A8   c=AD9E8CAC   d=5CC9AFFD   e=C4180293
R68: a=F701EFA0   b=5EF992A0   c=61119279   d=3A29C1EA   e=AD9E8CAC
R70: a=57E47E05   b=70BF4F9C   c=3DC07BE8   d=17BE64A8   e=61119279
R72: a=A84D0A88   b=4B833CB9   c=55F91F81   d=1C2FD3E7   e=3DC07BE8
R74: a=0B3CD293   b=2B01A32B   c=2A1342A2   d=52E0CF2E   e=55F91F81
R76: a=C79E506D   b=A1584161   c=C2CF34A4   d=CAC068CA   e=2A1342A2
R78: a=761449FA   b=DB91E0C7   c=71E7941B   d=68561058   e=C2CF34A4
R80: a=8AB36355   b=2F2C07DB   c=9D85127E   d=F6E47831   e=71E7941B
Resultado: F1F886561EF9B364363FEF7C0716CCA735BA760B
```

Bibliografía

[1] Daniel J. Bernstein, *Salsa20 specification*. 2005.

[2] Daniel J. Bernstein, *Curve25519: new Diffie-Hellman speed records*. 2006.

[3] Daniel J. Bernstein, *The Poly1305-AES message-authentication code*. 2005.

[4] Certicom Research, *SEC 2: Recommended Elliptic Curve Domain Parameters*. 2010.

[5] Niels Ferguson, Bruce Schneier, Tadayoshi Kohno, *Cryptography Engineering*. Wiley, 2010.

[6] John D. Lipson, *Elements of Algebra and Algebraic Computing*. Addison-Wesley, 1981.

[7] Alfred J. Menezes, Paul C. van Oorschot y Scott A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.

[8] Bruce Schneier, *Applied Cryptography. Second Edition*. John Wiley & sons, 1996.

[9] Jennifer Seberry, Josef Pieprzyk, *Cryptography. An Introduction to Computer Security*. Prentice Hall. Australia, 1989.

[10] Phillip Rogaway, *Evaluation of Some Blockcipher Modes of Operation*. 2011.

[11] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2008

[12] William Stallings. *Cryptography and Network Security, Principles and Practice. Fifth Edition*. Prentice Hall, 2011.

[13] Juan Manuel Velázquez y Arturo Quirantes. *Manual de PGP 5.53i*. 1998.

[14] *Página web de Zedz Consultants (antes Replay)*. http://www.zedz.net

[15] *Página web de los Laboratorios RSA*. http://rsasecurity.com/rsalabs/

[16] *RFC 4880: Open PGP Message Format*. http://www.ietf.org/rfc/rfc4880.txt

[17] *RFC 1750: Randomness Recommendations for Security*. http://www.it.kth.se/docs/rfc/rfcs/rfc1750.txt

[18] *Wikipedia*. http://www.wikipedia.org