Criptografía y Seguridad en Computadores

Tercera Edición (Versión 1.00). Junio de 2001

Manuel José Lucena López

e-mail:
mlucena@ujaen.es
mlucena@kriptopolis.com

—¿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...

El Señor de Los Anillos

J.R.R. Tolkien

Copyright

Agradecimientos

A Loles, ella sabe por qué.

A los chicos de Kriptópolis, por darme esta oportunidad.

A mis alumnos, por aguantarme cada año.

A todos aquellos que, enviando sugerencias o correcciones, han ayudado a mejorar esta obra.

A todos los que alguna vez han compartido sus conocimientos, por enriquecernos a todos.

Prefacio

El presente documento ha sido elaborado originalmente como apoyo a la asignatura "Criptografía y Seguridad en Computadores", de 3^{er} Curso de *Ingeniería Técnica en Informática de Gestión*, de la Universidad de Jaén, empleando el procesador de textos LaTeX 2_ε, y los editores gráficos XFig y Gimp. Puede descargarse su última versión y eventuales correcciones en las siguientes direcciones *web*:

`http://www.kriptopolis.com`
`http://wwwdi.ujaen.es/~mlucena`

No se pretende que estos apuntes sustituyan a la bibliografía de la asignatura, ni a las clases teóricas, sino que sirvan más bien como complemento a las notas que el alumno debe tomar en clase. Asimismo, no debe considerarse un documento definitivo y exento de errores, si bien ha sido elaborado con detenimiento y revisado exhaustivamente. El autor pretende que sea mejorado y ampliado con cierta frecuencia, lo que probablemente desembocará en sucesivas versiones, y para ello nadie mejor que los propios lectores para plantear dudas, buscar errores, y sugerir mejoras.

Jaén, Junio de 2001.

Índice General

Parte I

Preliminares

Capítulo 1

Introducción

A lo largo de 1995 y principios de 1996, los profesores José Ignacio Peláez Sánchez, Antonio Sánchez Solana y Manuel José Lucena López elaboraron una Colección de Apuntes para la asignatura 'Criptografía y Seguridad en Computadores', impartida en tercer curso de Ingeniería Técnica en Informática de Gestión, en la Universidad de Jaén. Varios años han pasado desde entonces, y, como cabía esperar en una disciplina de tan rápida evolución, las cosas han cambiado. Algunos algoritmos han perdido parte de su interés —como es el caso de DES, que fue *vencido*[1] el verano de 1998—, nuevas técnicas han surgido o se han popularizado —PGP es un claro ejemplo de que para el usuario de a pie se puede conseguir auténtica privacidad—, temas que antes tenían un interés limitado se han convertido en fundamentales —la rápida expansión de Internet obliga no sólo al profesional, sino al usuario medio, a tener ciertos conocimientos básicos sobre seguridad—, etc.

La presente colección de apuntes nació con la vocación de intentar cubrir en la medida de lo posible ese vacío. Sin embargo, la escasez de documentación en Español sobre Criptografía, y las dificultades que encontraban muchos alumnos para complementar bibliográficamente la asignatura, unido todo ello a la sorprendente difusión del anterior texto hizo surgir en el autor la idea de distribuir esta obra en formato electrónico y de forma gratuita.

1.1 Cómo Leer esta Obra

Esta obra ha sido organizada en cinco partes:

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

[1] En realidad, el ataque que logró tener éxito se realizó por la fuerza bruta, por lo que no podemos hablar en un sentido estricto de *derrota*. De hecho, aún no se ha encontrado ninguna debilidad seria en su diseño; lo que sí quedó patente es que su longitud de clave es demasiado pequeña.

a lo largo de la obra.

2. *Fundamentos Teóricos de la Criptografía*. Se desarrollan brevemente los resultados teóricos sobre los que se va a apoyar el resto del libro. Si usted no domina las Matemáticas, o simplemente no tiene interés en estos fundamentos, puede pasarla por alto.

3. *Criptografía de Llave Privada*. Se introduce la Criptografía Clásica, así como los algoritmos simétricos de cifrado.

4. *Criptografía de Llave Pública*. En esta parte se estudian los algoritmos asimétricos y los métodos de autentificación. Además se incluye un capítulo sobre PGP.

5. *Seguridad en Redes de Computadores*. Esta es la parte menos teórica y quizá más práctica desde el punto de vista de la seguridad (no desde el punto de vista criptográfico). Se estudian en ella brevemente los problemas que se dan en redes de computadoras, para que el lector pueda comenzar a elaborar por sí mismo estrategias de protección de la información.

Este texto no tiene por qué 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 *creerse* 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 imprescincibles 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 documento, 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 el más sofisticado ingenio de codificación ideado hasta entonces: la máquina ENIGMA. Este grupo de científicos empleaba el que hoy se considera el primer computador —aunque esta información permaneció en secreto hasta mediados de los 70—. Su uso y la llegada del polaco Marian Rejewski tras la invasión de su país natal cambiarían para siempre el curso de la Historia.

Desde entonces hasta hoy ha habido un crecimiento espectacular de la tecnología criptográfica, si bien la mayor parte 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 nuevas plataformas de distribución de contenidos multimedia.

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. Ejemplos claros de oscurantismo y de sus nefastas consecuencias han sido la caída del algoritmo que emplean los teléfonos GSM en menos de cuarenta y ocho horas desde que su código fuera descubierto, el controvertido y manifiestamente inútil algoritmo de protección de los DVD, o las infructosas iniciativas de las casas discográficas encaminadas a impedir la *piratería*, por citar algunos. La seguridad no debe basarse en mantener los algoritmos ocultos —puesto que éstos, aparte de suscitar poca confianza en los usuarios, tarde o temprano acaban siendo analizados y descritos— sino en su resistencia demostrada tanto teórica como prácticamente, y la única manera de demostrar la fortaleza de un algoritmo es sometiéndolo a todo tipo de ataques. Uno de los capítulos más conocidos de esta historia ocurrió en el verano de 1999, cuando un programador denunció una supuesta *puerta trasera* en el código criptográfico de todas las versiones del sistema operativo Windows. Como este código permanece en secreto, y se considera delito su análisis —¿Qué pensaría usted si se compra un coche y se le prohíbe desarmarlo para ver cómo funciona?—, es desgraciadamente imposible que Microsoft pueda despejar cualquier sombra de duda sobre la seguridad de sus sistemas operativos. La pretensión de establecer sistemas de patentes sobre el *software* viene a agravar la situación, con un claro perjuicio tanto de los usuarios como de las pequeñas empresas frente al poder de las grandes corporaciones. Por desgracia, parece que a nuestros gobiernos les interesan más los beneficios de las multinacionales que los intereses de los ciudadanos.

Es imposible desligar la Criptografía moderna de todas las consideraciones políticas, filosóficas y morales que suscita. Recordemos, por ejemplo, que el software criptográfico está sujeto en EE.UU. a las mismas leyes que el armamento nuclear, y que en Europa se pretende elaborar legislaciones parecidas. Una consecuencia inmediata de esto es que las versiones que se exportan de los exploradores de Internet más extendidos (Netscape y Explorer), incorporaban hasta hace poco tiempo una seguridad *débil*, impidiendo, por ejemplo, establecer conexiones realmente seguras para conectarse a un banco —con el agravante de que no se informaba de ello adecuadamente al usuario—. Si bien esta situación se ha visto aliviada, debido a la relajación de estas restrictivas leyes, aún queda mucho camino por recorrer. Otra de las líneas de debate es 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 —con el pretexto, por supuesto, de luchar contra el terrorismo y el narcotráfico—.

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.

En el ojo del huracán se encuentra la red *Echelon*, que constituye una de las mayores amenazas contra la libertad de toda la Historia de la Humanidad. 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 políticas 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. a respecto pasan por el despliegue de su propia red de vigilancia electrónica, llamada *Enfopol*, en la que lleva trabajando unos diez años.

La red *Enfopol* tendría características tan aterradoras como la posesión de todas las claves privadas de los ciudadanos europeos, la monitorización de teléfonos, buzones de voz, faxes, chats y correo electrónico, además de la incorporación de puertas traseras a los proveedores de Internet. Existe un ducumento de la U.E. —ENFOPOL 112 10037/95—, firmado por todos y cada uno de los miembros de la Unión, en el que se aprueba la creación de esta red. Sin embargo, ninguno de los estados miembros ha confirmado ni desmentido nada al respecto. Este secretismo es más que preocupante, pues deja a la supuesta democracia en que vivimos maniatada frente a un asunto tan importante, impidiendo que se abra el imprescindible debate público que debería anteceder a tan grave asunto. ¿Qué nos queda de la Libertad y la Democracia si se nos ocultan asuntos tan importantes *por nuestro bien*?. Afortunadamente, el proyecto Enfopol se encuentra paralizado, pero, ¿por cuánto tiempo?

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 la tabla 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 aparecen términos como *encriptar* y *desencriptar*, adoptados con toda probabilidad del verbo anglosajón *encrypt*. El lector podrá comprobar que este tipo de expresiones ha sido evitado en el presente texto, debido a la existencia de palabras perfectamente válidas que pertenecen al idioma castellano, como son *cifrar-descifrar* y *codificar-decodificar* (o *descodificar*). La opinión del autor es que sólo deben emplearse términos foráneos cuando nuestro riquísimo idioma carezca de expresiones adecuadas para representar las ideas en cuestión. Esta última es la situación en la que se encuentra la palabra *esteganografía*, hispanización del término inglés *steganography* —que a su vez proviene del título del libro '*Steganographia*', escrito por Johannes Trithemius en 1518—.

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 empleo el primero de ellos simplemente porque me gusta más que el otro.

1.5 Notación Algorítmica

En este libro se describen varios 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

Valor	Número
Probabilidad de ser fulminado por un rayo (por día)	1 entre 9.000.000.000 (2^{33})
Probabilidad de ganar el primer premio de la Lotería Primitiva Española	1 entre 13.983.816 (2^{23})
Probabilidad de ganar el primer premio de 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
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	5.9×10^{24} (2^{82}) Kg.
Masa del Sol	2×10^{30} (2^{100}) 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

Tabla 1.1: Algunos números *grandes*

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 sobre Criptografía

2.1 Criptografía

Según el Diccionario de la Real Academia, la palabra Criptografía proviene del griego $\kappa\rho\upsilon\pi\tau\acute{o}\varsigma$, que significa oculto, y $\gamma\rho\acute{a}\varphi\epsilon\iota\nu$, que significa 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 de Llave Pública, abriendo enormemente el abanico 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 olivdemos 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 tanto la Criptografía como el 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 emisior 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, que sólo sirve para cifrar—, 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, burlando 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. Como es de suponer, existen tantos mecanismos para llevar a cabo este *camuflaje* como nuestra imaginación nos permita.

Mención especial merece el uso de la esteganografía para exportar información sin violar las leyes restrictivas que, con respecto a la Criptografía *fuerte*, existen en algunos países. El mensaje se envía como texto claro, pero entremezclado con cantidades ingentes de *basura*. El destinatario empleará técnicas esteganográficas para separar la información útil del resto. Esta técnica se conoce como *chaffing and winnowing*, que vendría a traducirse como *llenar de paja y separar el grano de la paja*. En consecuencia, tenemos un mecanismo para transmitir información no cifrada, pero que sólo puede ser reconstruida por el destinatario, con lo que en realidad

hemos logrado protegerla sin usar en ningún momento ningún algoritmo de codificación. Este sistema surgió en marzo de 1998, propuesto por Ronald L. Rivest —uno de los creadores de RSA—, como desafío a la política restrictiva del Gobierno de los EE.UU. con respecto a la Criptografía. No deja de ser en cierto modo una curiosidad, debido a lo enormemente grandes que son los mensajes en comparación con la cantidad de texto útil que se puede incluir en ellos.

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. Este método y todos los que buscan exhaustivamente por el espacio de claves K, se denominan *ataques por la fuerza bruta*, y en muchos casos no suelen considerarse como auténticas técnicas de criptoanálisis, reservándose este término para aquellos mecanismos que explotan posibles debilidades intrínsecas en el algoritmo de cifrado. Se da por supuesto que el espacio de claves para cualquier criptosistema digno de interés ha de ser suficientemente grande como para que un ataque por la fuerza bruta sea inviable. 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 frente a ataques por 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 un ataque de este tipo. Por ejemplo, si diseñáramos una máquina capaz de recorrer todas las combinaciones que pueden tomar 256 bits, cuyo consumo

fuera mínimo en cada cambio de estado[1], no habría energía suficiente en el Universo para que pudiera completar su trabajo.

Un par de métodos de criptoanálisis que han dado interesantes resultados son el *análisis diferencial* y el *análisis lineal* (ver sección 10.7, página 130). El primero de ellos, partiendo de pares de mensajes con diferencias 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. 44) 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

[1] Según las Leyes de la Termodinámica existe una cantidad mínima de energía necesaria para poder modificar el estado de un sistema.

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 seguiridad 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 supercomputadoras. 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 existen 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

El concepto de seguridad en la información es mucho más amplio que la simple protección de los datos a nivel *lógico*. Para proporcionar una seguridad real hemos de tener en cuenta múltiples factores, tanto internos como externos. En primer lugar habría que caracterizar el sistema que va a albergar la información para poder identificar las amenazas, y en este sentido podríamos hacer la siguiente subdivisión:

1. *Sistemas aislados.* Son los que no están conectados a ningún tipo de red. De unos años a esta parte se han convertido en minoría, debido al auge que ha experimentado Internet.

2. *Sistemas interconectados.* Hoy por hoy casi cualquier ordenador pertenece a alguna red, enviando y recogiendo información del exterior casi constantemente. Esto hace que las redes de ordenadores sean cada día más complejas y supongan un peligro potencial que no puede en ningún caso ser ignorado.

En cuanto a las cuestiones de seguridad que hemos de fijar podríamos clasificarlas de la siguiente forma:

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 *backup*, etc. También se suelen tener en cuenta dentro de este punto aspectos relacionados con la restricción de acceso físico a las computadoras únicamente a personas autorizadas.

2. *Seguridad de la información*. En este apartado prestaremos atención a la preservación de la información frente a observadores no autorizados. Para ello podemos emplear tanto criptografía simétrica como asimétrica, estando la primera únicamente indicada en sistemas aislados, ya que si la empleáramos en redes, al tener que transmitir la clave por el canal de comunicación, estaríamos asumiendo un riesgo excesivo.

3. *Seguridad del canal de comunicación*. Los canales de comunicación rara vez se consideran seguros. Debido a que en la mayoría de los casos escapan a nuestro control, ya que pertenecen a terceros, resulta imposible asegurarse totalmente de que no están siendo escuchados o intervenidos.

4. *Problemas de autentificación*. Debido a los problemas del canal de comunicación, es necesario asegurarse de que la información que recibimos en la computadora viene de quien realmente creemos que viene. Para esto se suele emplear criptografía asimétrica en conjunción con funciones *resumen* (ver sección 13.1, página 157).

5. *Problemas de suplantación*. En las redes tenemos el problema añadido de que cualquier usuario autorizado puede acceder al sistema desde fuera, por lo que hemos de confiar en sistemas fiables para garantizar que los usuarios no están siendo suplantados por intrusos. Normalmente se emplean mecanismos basados en *password* para conseguir esto.

6. *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. En este sentido es fundamental impedir que el emisor pueda *repudiar* un mensaje, es decir, negar su autoría sobre él.

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 nos permitirá efectuar una aproximación teórica al estudio de la seguridad de cualquier algoritmo criptográfico.

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 nos 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 no nos aporta 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, nos dan más información que si nos dicen que el número que ha salido es par.

Se puede decir que la cantidad de información que obtenemos al conocer un hecho es directamente proporcional al número posible de estados que éste tenía a priori. Si inicialmente teníamos diez posibilidades, conocer el hecho 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 ca-

so, puesto que el número de posibilidades a priori —y por tanto la incertidumbre, suponiendo sucesos equiprobables— es menor.

También la cantidad de información es proporcional a la probabilidad de un suceso. En el caso de las bolas tenemos dos sucesos: sacar bola negra, que es más probable, y sacar bola blanca, que es menos probable. Sacar una bola negra aumenta nuesto 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 nos 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 nos 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, obtener un x_j diferente nos aportará bastante información, como ya hemos visto. Este concepto de información es cuantificable y se puede definir de la siguiente forma:

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

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 nos 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:

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

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

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

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

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 nos 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), nos sale:

  $$H(M) = -(0.6 \log_2(0.6) + 0.4 \log_2(0.4)) = 0.970$$

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

$$H(M) = -(0.9\log_2(0.9) + 0.1\log_2(0.1)) = 0.468$$

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 $H(X/Y = y_j)$ obtemos 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:

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

Esto quiere decir que la cantidad de información que nos aporta el hecho de conocer X al medir la incertidumbre sobre Y es igual a la disminución de entropía que este conocimiento conlleva. Sus propiedades 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 nos aporta el hecho de conocer el mensaje cifrado c sobre la entropía del texto claro m vale cero. Es decir:

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

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 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 posibles mensajes de longitud k. Estamos midiendo el número de bits de información que nos aporta cada carácter en mensajes de una longitud determinada. Para idiomas como el Español, r_k suele valer alrededor de 1.4 $bits/letra$ 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 español, puesto que tenemos 27 símbolos, podríamos codificar 4.7 $bits/letra$ 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:

$$D = R - r$$

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

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

Desgraciadamente, 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:

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

Esta expresión nos permite saber la incertidumbre que nos queda sobre cuál ha sido mensaje enviado m si conocemos su criptograma asociado c. Si esa incertidumbre fuera la misma que desconociendo 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. 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 nos aporta 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 albsoluta certeza sobre el valor de m.

Esta magnitud se puede medir también en función del conjunto K de claves, y entonces nos dirá 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 Propuestos

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

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

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

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

4. Demuestre la Ley de Entropías Totales.

5. Suponga que un equipo de fútbol gana por regla general el 65% de sus partidos, pero que cuando llueve sólo gana el 35%. La probabilidad de que llueva en un partido es del 15%. ¿Cuál es la cantidad de información que nos aporta la variable aleatoria *lluvia* sobre la variable *ganar un partido*?

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.

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

Capítulo 4

Introducción a la 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, prácticamente todas 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, ¿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

[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.

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. Por lo tanto 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 factor de proporcionalidad —recuérdese que $\log_a(x) = \log_b(x) \cdot \log_a(b)$—. Dicho factor no afectará 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 $\mathbf{P} \subseteq \mathbf{NP}$ y que $\mathbf{P} \subseteq$ **co-NP**. Sin embargo, aún no se sabe si $\mathbf{P} = \mathbf{NP}$, si $\mathbf{NP} =$ **co–NP**, o si $\mathbf{P} = \mathbf{NP} \cap$ **co–NP**. Si bien muchos expertos consideran que ninguna de estas tres igualdades se cumple, este punto no ha podido ser demostrado matemáticamente.

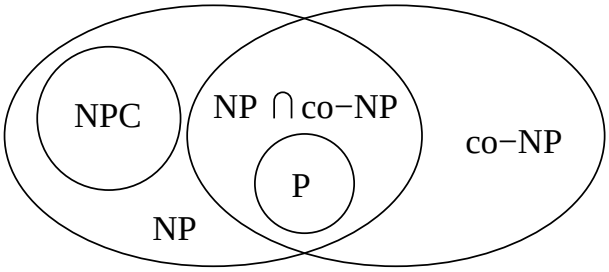

Figura 4.1: Relación entre las clases de complejidad **P, NP, co–NP y NPC.**

Dentro de la clase **NP**, existe un subconjuto 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.

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 los 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 asímismo 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

Fundamentos de Aritmética Modular

5.1 Aritmética Modular. Propiedades

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 tres números $a, b, n \in \mathbb{N}$, 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 en este conjunto cumple las propiedades asociativa y conmutativa y posee elementos neutro y simétrico, por lo que el conjunto tendrá 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 es uno de los más útiles. Permite obtener de forma eficiente el máximo común divisor de dos números.

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 mod b.

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

$$m|(b \bmod c)$$

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 artiméticas modulares es la misma que la de las no modulares:

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

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: Dos números enteros a y b se denominan *primos entre sí* (o *coprimos*), 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: Supongamos que $\text{mcd}(a,n) = 1$, y que existen $i \neq j$ tales que $ai \equiv aj (\bmod n)$. Se cumple, pues:

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

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

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

con lo que hemos alcanzado una contradicción.

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 nos 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:

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*.

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

Corolario: Si n es primo, el grupo finito que genera 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 *Campos de Galois*, y se notan $GF(n)$.

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:

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

Existe una expresión que nos permite calcular el número de elementos —el *cardinal*— del conjunto reducido de residuos módulo 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. Por ejemplo, si n fuera el producto de dos números primos p y q, $|\mathbb{Z}_n^*| = (p-1)(q-1)$.

Se define la *función de Euler sobre n*, y se escribe $\phi(n)$, como el cardinal de $\mathbb{Z}_n^*$, es decir:

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

Teorema: 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:

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

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

$$a^{p-1} \equiv 1 \pmod{p} \tag{5.4}$$

Como consecuencia de este último teorema podemos deducir que si $r \equiv s \pmod{p-1}$, 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 $\text{ord}(a)$, como el menor entero positivo t tal que $a^t \equiv 1 \pmod{n}$.

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

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:

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

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:

$$g_i = nu_i + av_i$$

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 ($v_i \bmod n$) 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 prctica 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)
  { int c,i;
    int g[],u[],v[];

    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

$$x \equiv x_i \pmod{p_i} \qquad i = 1, \ldots, r \tag{5.5}$$

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

$$x = \sum_{i=1}^{r} \frac{n}{p_i} \left[(n/p_i)^{-1} \pmod{p_i} \right] x_i \pmod{n} \tag{5.6}$$

Demostración: Para cada i, $\text{mcd}\left[p_i, \frac{n}{p_i}\right] = 1$. 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 (5.5), 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 (5.5), lo cual puede resultarnos de gran utilidad para ciertas aplicaciones, como por ejemplo, el algoritmo RSA (ver sección 12.2).

5.4 Exponenciación. Logaritmos Discretos

Muchos de los algoritmos de llave 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. También comentaremos brevemente el problema inverso, el cálculo de los logaritmos discretos, puesto que en su dificultad intrínseca se apoyan muchos algoritmos criptográficos.

5.4.1 Algoritmo de Exponenciación Rápida

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 de Exponenciación Rápida queda como sigue:

```
int exp_rapida(int a, int b)
    { int z,x,resul;

      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.

5.4.2 El Problema de los Logaritmos Discretos

El problema inverso 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:

$$c = \log_b(a) \pmod{n} \Longleftrightarrow a \equiv b^c \pmod{n} \tag{5.7}$$

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 no se ha podido demostrar).

5.4.3 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 12.3.1) y el de ElGamal (apartado 12.3.2).

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

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

El enunciado del problema es el siguiente: dado un número primo p, un número α que sea un generador de $\mathbb{Z}_n^*$, 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 $(\alpha^b)^a = \alpha^{ab}$.

5.5 Importancia de los Números Primos

Para explotar la dificultad de cálculo de logaritmos discretos, muchos algoritmos criptográficos de llave 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, siempre y cuando los factores —como veremos más adelante— hayan sido escogidos correctamente. Esto nos permite confiar en que los factores de n serán imposibles de calcular aunque se conozca el propio n.

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 $n = p_1^{e_1} \cdot p_2^{e_2} \dots p_k^{e_k}$, siendo p_i números primos y e_i números enteros mayores que 1, 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 Método de Fermat

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

$$n = x^2 - y^2 \tag{5.8}$$

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—.

5.6.2 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. 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 $p-1$. 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 $x = (a^m \mod n)$.
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 este algoritmo tiene un 50% de probabilidades de encontrar un valor de a que permita obtener un factor de n. Ejecutándolo, pues, varias veces, es bastante probable que podamos hallar algún factor de n.

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 $x = (2^{10368} \mod 187) = 69$. 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.

5.6.3 Métodos Cuadráticos de Factorización

Los métodos cuadráticos de factorización se basan en la ecuación

$$x^2 \equiv y^2 \pmod{n} \tag{5.9}$$

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

$$n|(x-y)(x+y) \tag{5.10}$$

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

$$x_i^2 \equiv z_i \pmod{n} \tag{5.11}$$

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 (5.11).

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

$$q(x) \equiv 0 \pmod{p} \tag{5.12}$$

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 (5.12). 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.

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.

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 hemos definido para 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 Campo 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 campo 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

$$\begin{gathered} x^9 + x^8 + x^7 + x^4 + x^3 + x = x(x^8) + x^8 + x^7 + x^4 + x^3 + x = \\ = x(x^4 + x^3 + x + 1) + (x^4 + x^3 + x + 1) + x^7 + x^4 + x^3 + x = \\ = x^5 + x^4 + x^2 + x + x^4 + x^3 + x + 1 + x^7 + x^4 + x^3 + x = \\ = x^7 + x^5 + x^4 + x^4 + x^4 + x^3 + x^3 + x^2 + x + x + x + 1 = \\ = x^7 + x^5 + x^4 + x^2 + x + 1 \end{gathered}$$

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, por lo que se obtienen sistemas con mayores prestaciones, y a la vez más baratos.

5.9 Ejercicios Propuestos

1. Comprobar las propiedades de la suma en grupos finitos.
2. Comprobar las propiedades del producto en grupos finitos.
3. Calcular el valor de la función ϕ de Euler para los siguientes números: 64, 611, 2197, 5, 10000.
4. Resolver el siguiente sistema de ecuaciones en congruencias:

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

5. ¿Cómo calcularía el valor de $(2^{10368} \bmod 187)$, empleando únicamente lápiz, papel y calculadora?
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 $f(x) = x^6 + x + 1$.

Capítulo 6

Introducción a las Curvas Elípticas

La Criptografía de Curva Elíptica es una de las disciplinas más prometedoras 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 que presentaba la aritmética modular, lo cual las hace válidas para aplicar algunos de los algoritmos asimétricos más conocidos (ver capítulo 12). Si bien su estructura algebraica es algo compleja, su implementación suele resultar tanto o más eficiente que la aritmética modular, y además con claves mucho más cortas se puede alcanzar el mismo nivel de seguridad que con otras técnicas.

Las primeras propuestas de uso de las curvas elípticas en Criptografía fueron hechas por Neal Koblitz y Victor Miller en 1985. Precisamente el principal argumento que esgrimen los detractores de estas técnicas es que, si bien las curvas elípticas han sido objeto de estudio y análisis durante más de un siglo, las propiedades que pueden estar directamente relacionadas con su calidad como base para un sistema criptográfico, apenas llevan quince años siendo consideradas.

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 constituían un conjunto para el que podíamos definir una serie de operaciones, y éstas tenían unas propiedades concretas. Estos conjuntos y operaciones mostraban una estructura que hacía surgir problemas computacionalmente difíciles de tratar. Vamos a hacer exactamente lo mismo 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

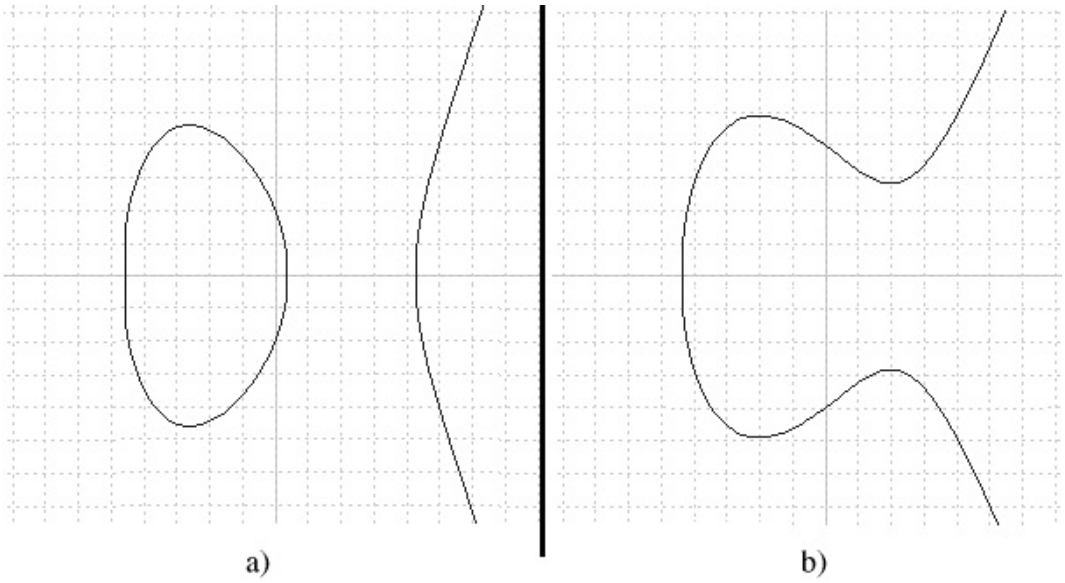

Figura 6.1: Gráficas de curvas elípticas. a) $y^2 = x^3 - 5x + 1$; b) $y^2 = x^3 - 3x + 4$.

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 abcisas 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

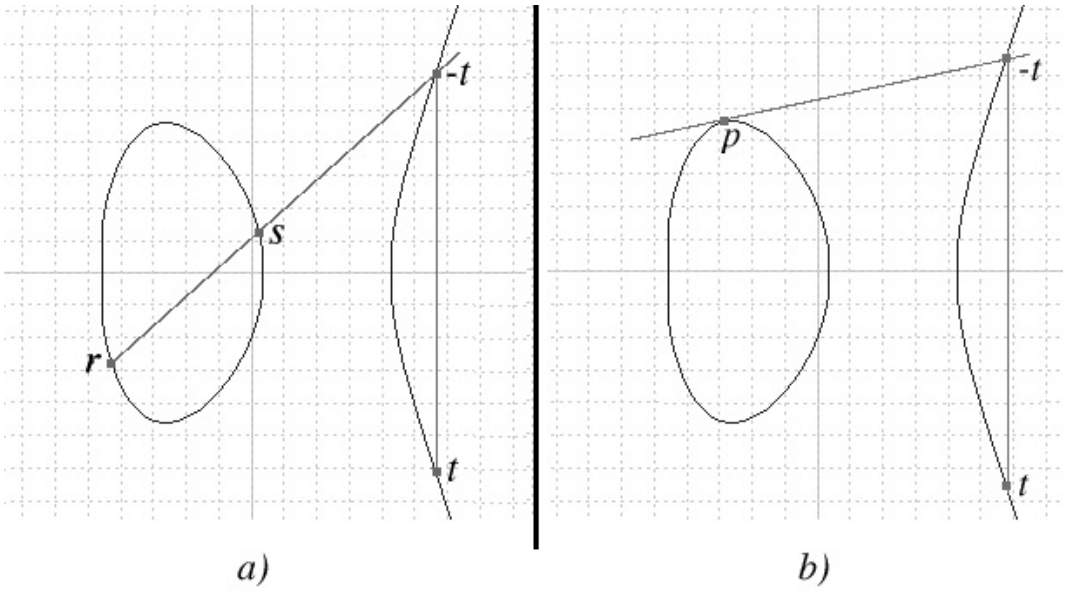

Figura 6.2: Interpretación gráfica de la suma de dos puntos en una curva elíptica.

$\mathbf{r} + \mathbf{s} = \mathbf{s} + \mathbf{r} = \mathcal{O}$ 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 abcisas, por lo que 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:

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

Algebraicamente, la suma de curvas elípticas se define 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 $\mathbf{r} + \mathbf{s} = \mathbf{t}$ donde

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

y cuando queremos sumar un punto consigo mismo, tenemos que $2\mathbf{p} = \mathbf{t}$ 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 $\mathbf{p}$.

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.

6.2 Curvas Elípticas en $GF(n)$

Recordemos que un Campo 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 $\mathbb{R}$, por lo que nada nos impide ver qué puntos cumplen la ecuación

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

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 $(x, -y) \pmod{n}$ 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):

$$d = (3-16)/(2-3) = 13 \pmod{17}$$
$$x = 13^2 - 2 - 3 = 11 \pmod{17}$$
$$y = -3 + 13 \cdot (2-3) = 1 \pmod{17}$$

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 vió 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. Nada nos impediría, pues, definir 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):

$$y^2 + xy = x^3 + ax^2 + b \tag{6.4}$$

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 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

Tomemos un punto $\mathbf{p}$ cualquiera de una curva elíptica. Denominaremos $\langle \mathbf{p} \rangle$ al conjunto $\{\mathcal{O}, \mathbf{p}, 2\mathbf{p}, 3\mathbf{p}, \dots\}$. 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 Ejercicios Propuestos

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.

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

3. 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 $b = 010$.

4. 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, el número en base 10 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:

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

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 $a > b$:

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

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 Multiplicación

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 despues 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) => Siendo x un digito, devuelve el numero
                               de bits de los digitos quitando los
                               ceros de la izquierda.
     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);
     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. $b = 100101$; $a = ----$; $d1 = 111000$; $b \not> d1 \longrightarrow a = 0---$
2. $b = 100101$; $a = 0---$; $d1 = 11100$; $b > d1 \longrightarrow a = 01--$
3. $b = 001001$; $a = 01--$; $d1 = 110$; $b \not> d1 \longrightarrow a = 010-$
4. $b = 001001$; $a = 010-$; $d1 = 111$; $b > d1 \longrightarrow a = 0101$
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) => Siendo x un digito, devuelve el numero
                             de bits de los digitos quitando los
                             ceros de la izquierda.
   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 sumar el módulo n al minuendo, restarle el sustraendo, y si el resultado es mayor que n, restar el módulo.
- 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 Propuestos

1. Efectúe el trazado del algoritmo de la división con $B = 8$ para calcular el siguiente cociente: $c = 35240$, $d = 234$.
2. 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 multplicación mediante esta técnica y calcule su umbral correspondiente.
3. Elabore la especificación de una Estructura de Datos que permita almacenar números enteros *largos* y defina sus primitivas básicas.
4. Proponga una especificación para la estructura del ejercicio anterior y discuta su eficiencia.

Capítulo 8

Criptografía y Números Aleatorios

Los algoritmos de llave pública, debido a su mayor orden de complejidad, suelen ser empleados en conjunción con algoritmos de llave privada de la siguiente forma (ver capítulo 12): el mensaje primero se codifica empleando un algoritmo simétrico y la llamada *clave de sesión*, que será diferente cada vez. Es la clave de sesión la que se codifica empleando criptografía asimétrica. La única manera de que estas claves sean seguras es 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 diferentes tests de aleatoriedad —como el denominado *test* ψ^2, que puede ser consultado en casi cualquier libro de Estadística—. Los generadores tradicionales no nos 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 el siguiente—. Si bien las series que producen estos generadores superan los test estadísticos de aleatoriedad, son totalmente previsibles, y esa condición es inadmisible para aplicaciones criptográficas. Un famoso ejemplo de este problema ocurrió en una de las primeras versiones de Netscape, que resultaba insegura debido al uso de un generador pseudoaleatorio demasiado *previsible*.

En este capítulo vamos a caracterizar diferentes tipos de secuencias aleatorias, 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*

8.1.1 Secuencias pseudoaleatorias

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. Todos los generadores pseudoaleatorios producen secuencias finitas y periódicas de números empleando operaciones aritméticas y/o lógicas. Lo único que podremos conseguir es que estas secuencias sean lo más largas posible antes de comenzar a repetirse y que superen los tests estadísticos de aleatoriedad. En este sentido podríamos hablar de:

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

Un generador *congruencial lineal*[1] cumple esta propiedad, pero en Criptografía será del todo inútil, 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:

- *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 números anteriores y del algoritmo de generación empleado.

Existen generadores pseudoaleatorios criptográficamente resistentes que cumplen esta propiedad. Sin embargo no son suficientes para nuestros propósitos, debido a que se necesita una semilla para inicializar el generador. Si un atacante lograra averiguar la semilla que estamos empleando en un momento dado, podría de nuevo comprometer nuestro sistema. Necesitamos para ella 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. Necesitamos, pues, valores auténticamente aleatorios.

8.1.3 Secuencias totalmente aleatorias

Como ya se ha dicho antes, no existe la aleatoriedad cuando se habla de computadoras. En realidad se puede decir que no existen en el Universo sucesos cien por cien aleatorios. En

[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.

cualquier caso, y a efectos prácticos, consideraremos un tercer tipo de secuencias pseudoaleatorias:

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

Llegados a este punto parece claro que nuestro objetivo no va a ser generar secuencias aleatorias puras, sino más bien secuencias impredecibles e irreproducibles. Será suficiente, pues, con emplear un generador criptográficamente aleatorio alimentado por una semilla totalmente aleatoria.

8.2 Generación de Secuencias Aleatorias Criptográficamente Válidas

Para poder obtener secuencias a la vez impredecibles e irreproducibles, haremos uso de generadores de secuencias criptográficamente aleatorias, en conjunción con algún mecanismo de recolección de bits aleatorios, que nos va a permitir inicializar la semilla del generador. Un esquema de este tipo será seguro siempre que se salvaguarde adecuadamente la semilla empleada. Comentaremos en primer lugar algunos mecanismos para obtener los bits de la semilla, y después nos centraremos en los generadores criptográficamente aleatorios propiamente dichos.

8.2.1 Obtención de Bits Aleatorios

Como hemos dicho antes, las operaciones aritméticas y lógicas que realiza una computadora son completamente deterministas. Sin embargo, los ordenadores, como veremos a continuación, poseen elementos menos *deterministas* que pueden ser útiles para nuestros propósitos.

Para obtener n bits aleatorios bastaría con que una persona lanzara una moneda al aire n veces y nos fuera diciendo el resultado. En la actualidad apenas hay computadores que incorporen hardware específico para esta tarea, aunque existe y sería bastante barato y sencillo incorporarlo a la arquitectura de cualquier ordenador.

Existen valores obtenidos del hardware de la 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 (*hardware*) 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 emplear algunos dispositivos hardware relativamente comunes en los ordenadores actuales:

- *Tarjetas digitalizadoras de sonido o vídeo.* Un dispositivo digitalizador de audio (o vídeo) sin ninguna entrada conectada, siempre que tenga ganancia suficiente, 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. Si se dispone de un método para medir el tiempo de acceso de la unidad con suficiente precisión, se pueden obtener bits aleatorios de la calidad necesaria.

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. Teniendo 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, hemos de buscar un mecanismo para llevar a cabo esa combinación. Emplearemos a tal efecto las denominadas *funciones de mezcla fuertes.*

Una función de mezcla 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 Simétricos* (ver capítulo 10). Un algoritmo simétrico de cifrado 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 codificar m bits usando como clave los n restantes, y así obtener como salida un bloque de m bits con mejor *aleatoriedad*. Así, por ejemplo, si usamos DES, podemos reducir a 64 bits un bloque de 120.

- *Funciones Resumen* (ver sección 13.1) . Una función resumen puede ser empleada para obtener un número fijo de bits a partir de una cantidad arbitraria de bits de entrada.

8.2.2 Eliminación del Sesgo

En la mayoría de los casos, los bits obtenidos de las fuentes aleatorias están sesgados, es decir, que hay más unos que ceros o viceversa. Esta situación no es deseable, puesto que necesitamos una fuente aleatoria no sesgada, que presente igual probabilidad tanto para el 0 como para el 1. Como veremos a continuación, esta circunstancia no constituye un problema serio, ya que existen diversas técnicas para solucionarla.

Bits de Paridad

Si tenemos una secuencia de valores cero y uno, con un sesgo arbitrario, 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.

Uso de Funciones Resumen

Si calculamos la entropía de una secuencia sesgada (ecuación 3.2, página 40), obtendremos el número n de bits *reales* de información que transporta. Entonces podremos aplicar una función resumen y quedarnos exactamente con los n bits menos significativos del resultado obtenido.

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 calcular la firma MD5 o SHA de dicha secuencia y considerar los 24 bits menos significativos del resultado como bits aleatorios válidos.

8.2.3 Generadores Aleatorios Criptográficamente Seguros

Suponiendo que ya tenemos una cantidad suficiente de bits auténticamente aleatorios (impredecibles e irreproducibles), vamos a ver un par de generadores pseudoaleatorios que permiten obtener secuencias lo suficientemente seguras como para ser empleadas en aplicaciones criptográficas.

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. El algoritmo para obtener cada uno de los valores g_n de la secuencia es el siguiente:

$$\begin{aligned} g_n &= DES(k, DES(k,t) \oplus s_n) \\ s_{n+1} &= DES(k, DES(k,t) \oplus g_n) \end{aligned}$$

donde k es una clave aleatoria 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—. $DES(K, M)$ representa la codificación de M mediante el algoritmo DES, 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.

8.2.4 Generador Blum Blum Shub

Es quizá el algoritmo 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*—. Consiste en escoger dos números primos *grandes*, p y q, que cumplan la siguiente propiedad:

$$p \equiv 3 (\text{mod } 4) \qquad q \equiv 3 (\text{mod } 4)$$

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:

$$\begin{aligned} s_0 &= (x^2)(\text{mod } n) \\ s_{i+1} &= (s_i^2)(\text{mod } n) \end{aligned}$$

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.

Parte III

Criptografía de Llave Privada

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 un sencillo algoritmo para evitar que sus comunicaciones militares fueran incerceptadas. 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 empleando 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 Inteligencia aliado *rompe* la máquina de cifrado del ejército alemán, llamada ENIGMA.

Todos los algoritmos criptográficos clásicos son simétricos, ya que hasta mediados de los años setenta no nació la Criptografía asimétrica, y por esa razón este capítulo se engloba dentro del bloque de la asignatura dedicado a los algoritmos de llave privada.

9.1 Algoritmos Clásicos de Cifrado

Estudiaremos en esta sección algunos criptosistemas que en la actualidad han perdido su eficacia, debido a que son fácilmente criptoanalizables empleando cualquier computadora doméstica, 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 mantienen

un interés teórico, ya que nos van a permitir explotar algunas de sus propiedades para entender mejor los algoritmos modernos.

9.1.1 Cifrados Monoalfabéticos

Se engloban dentro de este apartado todos los algoritmos criptográficos que, sin desordenar los símbolos dentro del mensaje, establecen una correspondencia única para todos ellos en todo el texto. Es decir, si al símbolo A le corresponde el símbolo D, esta correspondencia se mantiene a lo largo de 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:

$$C = (M + 3) \bmod 26$$

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:

$$E_{(a,b)}(M) = (aM + b) \bmod N$$

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).

9.1.2 Cifrados Polialfabéticos

En los cifrados polialfabéticos la sustitución aplicada a cada carácter varía en función de la posición que ocupe éste dentro del texto claro. En realidad corresponde a la apliación cíclica de n cifrados monoalfabéticos.

Cifrado de Vigènere

Es un ejemplo típico de cifrado polialfabético que debe su nombre a Blaise de Vigènere, 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 que 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. Para estimar d, buscaremos la periodicidad de los patrones comunes que puedan aparecer en el texto cifrado. Obviamente, para el criptoanálisis, necesitaremos al menos d veces más cantidad de texto que con los métodos monoalfabéticos.

9.1.3 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 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, lo que imposibilita un ataque basado en frecuencias. A diferencia de lo que se puede pensar en un principio, este método presenta demasiados inconvenientes para ser útil en la práctica: además del problema de necesitar un alfabeto de salida mayor que el de entrada, para aplicarlo hace falta conocer la distribución estadística *a priori* de los símbolos en el texto claro, información de la que, por desgracia, no siempre se dispone.

9.1.4 Cifrados de Transposición

Este tipo de mecanismos de cifrado no sustituye unos símbolos por otros, sino que cambia su orden dentro del texto. Quizás el más antiguo conocido sea el *escitalo*, 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, podría consistir en colocar el texto en una tabla de n columnas, y dar como texto cifrado los símbolos de una columna —ordenados de arriba a abajo— concatenados

con los de otra, etc. La clave k se compondría 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 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 trucos, 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 mensaje cifrado se irían iluminando en un panel. El destinatario copiaba dichas letras en su propia máquina y el mensaje original

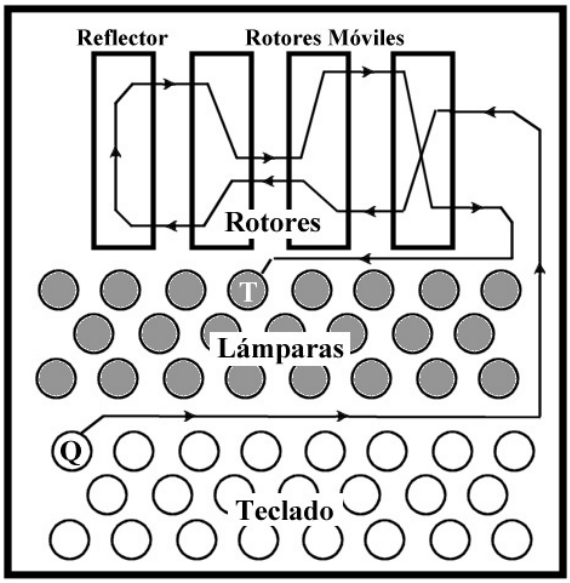

Figura 9.1: Esquema de la máquina Enigma.

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 posteriormente.

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 previo de permutación de letras, llamado *Stecker*, hacer que los rotores fueran intercambiables —se podían elegir y colocar en cualquier orden tres de entre cinco disponibles—, e incluso se amplió el número de rotores a cuatro.

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 1928 se recibió accidentalmente en ese país un ejemplar por correo ordinario[1]. Al ser reclamado urgentemente por las autoridades alemanas, despertó el interés del servicio secreto, que dispuso de un fin de semana entero para desmontar, analizar y volver a empaquetar cuidadosamente la máquina antes de devolverla a sus dueños el lunes.

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 tal en el mensaje que permitía obtener sin demasiados problemas 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 ordenaciones de los rotores que llevaran dos letras consecutivas iguales a la misma letra. 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 Africa, 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 Africa, 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. En su desarrollo participó también Max Von Neumann, y se llegaron

[1]La teoría del error postal, no obstante, es vista por muchos como una simple invención de aquellos que se empeñan en minusvalorar las habilidades criptoanalíticas de los matemáticos polacos, e intentan arrogarse todo el mérito en lo referente al descifrado de ENIGMA.

a construir varios cientos de ellas. 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.

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 radicaba en que tras codificar cada letra se giran los rotores, lo cual hace que la permutación que se aplica a cada letra sea diferente, y que esa permutación además no se repita hasta que los rotores recuperen su posición inicial. 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* también presenta un número bastante alto de combinaciones, existe una cantidad enorme de posibles disposiciones iniciales del la máquina (al menos para aquella época). La potencia del método de criptoanálisis empleado radica en que se podía identificar un emparejamiento válido entre el criptograma y el texto claro, 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. Estos dispositivos son mucho menos conocidos por diversas razones.

La máquina SIGABA, empleada por el ejército norteamericano, de la cual apenas se conoce nada por razones obvias.

También citaremos las máquinas japonesas empleadas en la II Guerra Mundial, que se denominaron PURPLE y RED, cuyas características son secretas o desconocidas. De hecho los norteamericanos declararon no haber tenido nunca ocasión de capturar nada más que un ejemplar, concretamente en la embajada japonesa tras la toma de Berlín.

Capítulo 10

Cifrados por Bloques

10.1 Cifrado de producto

La gran mayoría de los algoritmos de cifrado simétricos se apoyan en los conceptos de confusión y difusión inicialmente propuestos por Shannon (ver sección 3.8), que se combinan para dar lugar a los denominados *cifrados de producto*. Estas técnicas consisten básicamente en *trocear* el mensaje en bloques de tamaño fijo, y aplicar la función de cifrado a cada uno de ellos.

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.

Hemos de hacer notar que la confusión por sí sola sería 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 serían inviables. Por ejemplo, un algoritmo que codificara bloques de 128 bits empleando una clave de 80 bits necesitaría una tabla de aproximadamente 10^{63} entradas.

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.

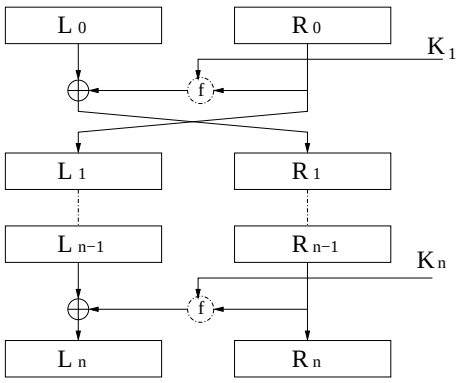

Figura 10.1: Estructura de una red de Feistel.

10.1.1 Redes de Feistel

Muchos de los cifrados de producto 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):

$$\begin{array}{rcl} L_i &=& R_{i-1} \\ R_i &=& L_{i-1} \oplus f(R_{i-1}, K_i) \end{array} \Bigg\} \quad \text{si } i < n. \qquad (10.1)$$

$$\begin{array}{rcl} L_n &=& L_{n-1} \oplus f(R_{n-1}, K_n) \\ R_n &=& R_{n-1} \end{array}$$

Este tipo de estructura se denomina *Red de Feistel*, y es empleada en multitud de algoritmos, como DES, Lucifer, FEAL, CAST, Blowfish, etcétera. 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 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 $D_n^L(E_n^L(L, R), E_n^R(L, R)) = L$ y $D_n^R(E_n^L(L, R), E_n^R(L, R)) = R$ para cualquier valor de n.

[1] Este principio nos garantiza que si demostramos el caso $n = 1$, y luego demostramos el caso $n+1$ suponiendo cierto el caso n, la propiedad en cuestión ha de cumplirse para cualquier valor de n igual o superior a 1

- Si $n = 1$ tenemos:

$$\begin{aligned} E_1^L(A,B) &= A \oplus f(B,K_1) \\ E_1^R(A,B) &= B \end{aligned}$$

y

$$\begin{aligned} D_1^L(A,B) &= A \oplus f(B,K_1) \\ D_1^R(A,B) &= B \end{aligned}$$

luego

$$D_1^L\Big(E_1^L(L,R), E_1^R(L,R)\Big) = E_1^L(L,R) \oplus f\Big(E_1^R(L,R), K_1\Big) =$$
$$= \Big(L \oplus f(R,K_1)\Big) \oplus f(R,K_1) = L$$

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:

$$\begin{aligned} E_{n+1}^L(L,R) &= E_n^R(L,R) \oplus f\Big(E_n^L(L,R), K_{n+1}\Big) \\ E_{n+1}^R(L,R) &= E_n^L(L,R) \end{aligned}$$

  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:

$$D_{n+1}\Big(E_{n+1}^L(L,R), E_{n+1}^R(L,R)\Big) = D_n\Big(E_{n+1}^R(L,R), E_{n+1}^L(L,R) \oplus f(E_{n+1}^R(L,R), K_{n+1})\Big)$$

  Sustituyendo $E_{n+1}^L(L,R)$ y $E_{n+1}^R(L,R)$ en la parte derecha de la anterior expresión nos queda:

$$D_{n+1}\Big(E_{n+1}^L(L,R), E_{n+1}^R(L,R)\Big) =$$
$$= D_n\Big(E_n^L(L,R), \Big(E_n^R(L,R) \oplus f(E_n^L(L,R), K_{n+1})\Big) \oplus f(E_n^L(L,R), K_{n+1})\Big)$$

  o sea,

$$\begin{aligned} D_{n+1}^L\Big(E_{n+1}^L(L,R), E_{n+1}^R(L,R)\Big) &= D_n^L\Big(E_n^L(L,R), E_n^R(L,R)\Big) = L \\ D_{n+1}^L\Big(E_{n+1}^R(L,R), E_{n+1}^R(L,R)\Big) &= D_n^R\Big(E_n^L(L,R), E_n^R(L,R)\Big) = R \end{aligned}$$

  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:

$$\forall\ k_1, k_2 \quad \exists\ k_3 \quad \text{tal que} \quad E_{k_2}(E_{k_1}(M)) = E_{k_3}(M) \tag{10.2}$$

esto es, si hacemos dos cifrados encadenados con k_1 y k_2, existe una clave k_3 que realiza la transformación equivalente.

Es interesante que un algoritmo criptográfico 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 algoritmos de producto, intercalaremos sustituciones sencillas (confusión), con tablas pequeñas, y permutaciones (difusión). Estas tablas pequeñas de sustitución se denominan de forma genérica S-Cajas.

Una S-Caja de m*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 elección de los valores de salida para que den lugar a un buen algoritmo 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—. Este algoritmo se ha demostrado resistente a las técnicas habituales de criptoanálisis, y sólo se conoce la fuerza bruta como mecanismo para atacarlo.

10.2 El Algoritmo DES

Es el algoritmo simétrico más extendido mundialmente. Se basa en el algoritmo LUCIFER, que había sido desarrollado por IBM a principios de los setenta, y fue adoptado como estándar

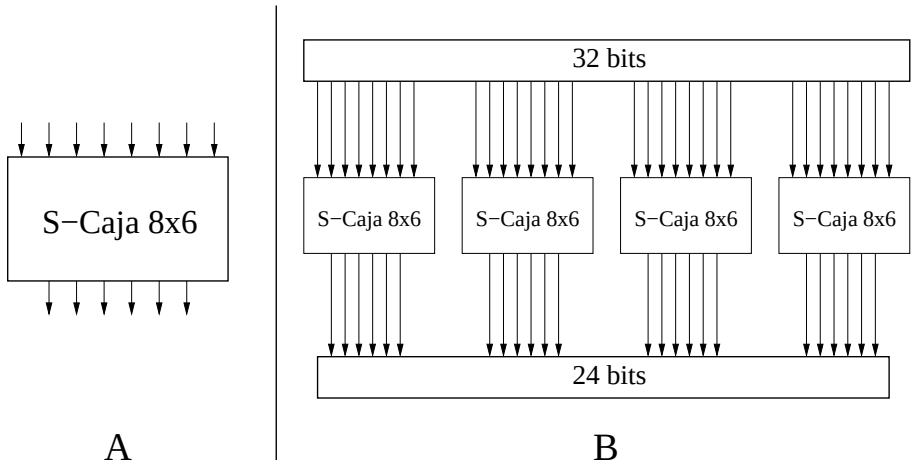

Figura 10.2: **A**: S-Caja individual. **B**: combinación de cuatro S-Cajas.

por el Gobierno de los EE.UU. para comunicaciones no clasificadas en 1976. En realidad 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.

El algoritmo 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 efectua 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

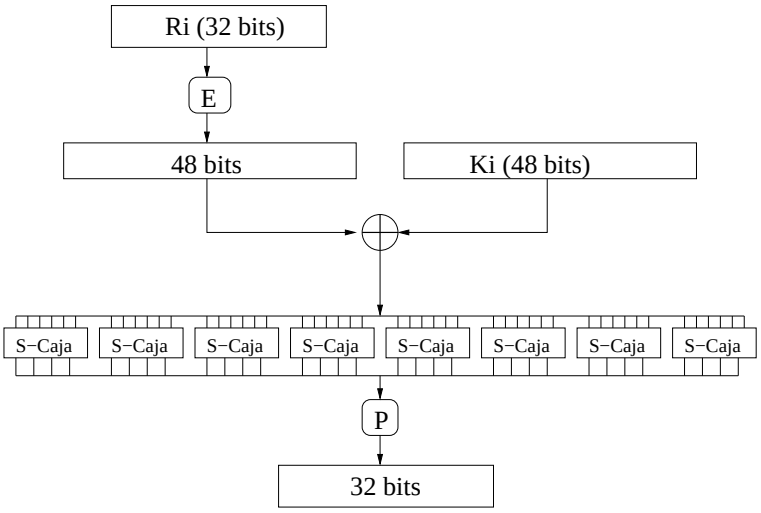

Figura 10.3: Esquema de la función f del algoritmo DES.

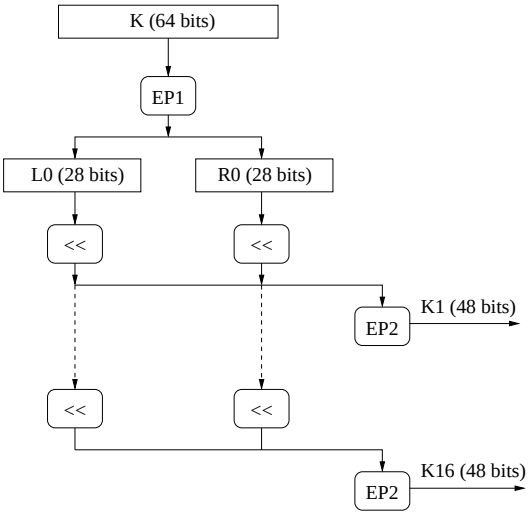

Figura 10.4: Cálculo de las K_i para el algoritmo DES. `EP1` representa la primera elección permutada, que sólo conserva 56 bits de los 64 de entrada, y `EP2` representa la segunda, que se queda con 48 bits. El signo "<<" representa un desplazamiento de bits a la izquierda (uno o dos dependiendo de la ronda).

Clave	Clave tras aplicar EP1
0101010101010101	0000000 0000000
1F1F1F1F0E0E0E0E	0000000 FFFFFFF
E0E0E0E0F1F1F1F1	FFFFFFF 0000000
FEFEFEFEFEFEFEFE	FFFFFFF FFFFFFF

Tabla 10.1: Claves débiles para el algoritmo DES (64 bits), expresadas en hexadecimal.

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

Tabla 10.2: Claves semi-débiles para el algoritmo DES (64 bits), expresadas en hexadecimal.

orden inverso.

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* (tabla 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* (tabla 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, una empresa sin ánimo de lucro, llamada EFF (Electronic Frontier Foundation), logró fabricar una máquina capaz de descifrar un mensaje DES en me-

nos de tres días. Curiosamente, pocas semanas antes, un alto cargo de la NSA 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 40 millones de pesetas.

A pesar de su *caída*, DES sigue siendo ampliamente utilizado en multitud de aplicaciones, 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 avance actual de las computadoras los ataques por la fuerza bruta comiencen a ser opciones realistas. Mucha gente se resiste a abandonar este algoritmo, precisamente porque ha sido capaz de *sobrevivir* durante veinte años sin mostrar ninguna debilidad en su diseño, y prefieren proponer variantes que, de un lado evitarían el riesgo de tener que confiar en algoritmos nuevos, y de otro permitirían 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. Se puede hacer ya que DES no presenta 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 propias 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 El algoritmo IDEA

El algoritmo IDEA (International Data Encryption Algorithm) es bastante más joven que DES, pues data de 1992. Para muchos constituye el mejor y más seguro algoritmo simétrico disponible en la actualidad. 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 algoritmo tanto para cifrar como para descifrar.

IDEA 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], y su longitud de clave hace imposible en la práctica un ataque por la fuerza bruta.

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. 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.

[2]En realidad, IDEA tiene un pequeñisimo 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.

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 la tabla 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.

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}		

Tabla 10.3: Subclaves empleadas en el algoritmo IDEA

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 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 ha efectuado de forma pública y abierta, por lo que, prácticamente por primera vez, toda la comunidad criptográfica mundial ha participado en su análisis, lo cual convierte a Rijndael en un algoritmo perfectamente digno de la confianza de todos.

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 campo 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 $GF(2^8)$.

10.5.1 Estructura de AES

AES, a diferencia de algoritmos como DES, *no* posee estructura de red de Feistel. En su lugar se ha definido cada ronda como una composición de cuatro funciones invertibles

[3]Gracias a Sven Magnus por la aclaración.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$

Tabla 10.4: Ejemplo de matriz de estado con N_b=5 (160 bits).

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Tabla 10.5: Ejemplo de clave con N_k=4 (128 bits).

diferentes, formando tres *capas*, diseñadas para proporcionar resistencia frente a criptoanálisis lineal y diferencial. 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 (tabla 10.4), N_b será igual a 5.

La llave 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 (tabla 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, $k_{0,0}, k_{1,0}, k_{2,0}, k_{3,0}, k_{0,1} \ldots$.

	$N_b = 4$ (128 bits)	$N_b = 6$ (192 bits)	$N_b = 8$ (256 bits)
$N_k = 4$ (128 bits)	10	12	14
$N_k = 6$ (192 bits)	12	12	14
$N_k = 8$ (256 bits)	14	14	14

Tabla 10.6: Número de rondas para AES en función de los tamaños de clave y bloque.

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
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 la tabla 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. $S \longleftarrow ByteSub(S)$
2. $S \longleftarrow DesplazarFila(S)$
3. $S \longleftarrow MezclarColumnas(S)$
4. $S \longleftarrow K_i \oplus S$

La última ronda es igual a las anteriores, pero eliminando el paso 3.

N_b	c_1	c_2	c_3
4	1	2	3
6	1	2	3
8	1	3	4

Tabla 10.7: Valores de c_i según el tamaño de bloque N_b

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. Después se aplica 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 consiste en desplazar a la derecha 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 la tabla 10.7.

La función inversa de *DesplazarFila* será, obviamente, un desplazamiento de las filas de la matriz de estado el mismo número de posiciones que en la tabla 10.7, pero a la izquierda.

Función *MezclarColumnas*

Para esta función, 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:

$$c(x) = \mathsf{03}x^4 + \mathsf{01}x^2 + \mathsf{01}x + \mathsf{02}$$

donde $\mathsf{03}$ 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:

$$d(x) = \mathsf{0B}x^4 + \mathsf{0D}x^2 + \mathsf{09}x + \mathsf{0E}$$

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. $\quad\quad tmp \leftarrow Sub(Rot(tmp)) \oplus R(i/N_k)$
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. $tmp \leftarrow W(i-1)$
5. Si $i \bmod N_k = 0$
6. $\quad tmp \leftarrow Sub(Rot(tmp)) \oplus Rc(i/N_k)$
7. Si $i \bmod N_k = 4$
8. $\quad tmp \leftarrow Sub(tmp)$
9. $W(i) \leftarrow W(i-N_k) \oplus tmp$

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 $x^{(i-1)}$.

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 a este algoritmo uno de los más seguros en la actualidad.

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. En primer lugar, independientemente del método empleado para codificar, hemos de tener en cuenta lo que ocurre cuando la longitud de la cadena que queremos cifrar no es un múltiplo exacto del tamaño de bloque. Entonces tenemos que añadir información al final para que sí lo sea. El mecanismo más sencillo consiste en rellenar con ceros (o algún otro patrón) el último bloque que se codifica. El problema ahora consiste en saber cuando se descifra por dónde hay que cortar. Lo que se suele hacer es añadir como último byte del último bloque el número de bytes que se han añadido (ver figura 10.5). 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, 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.

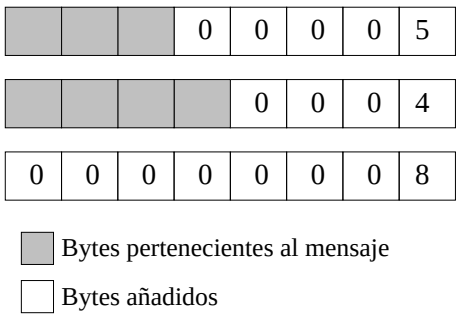

Figura 10.5: Relleno (*padding*) de los bytes del último bloque al emplear un algoritmo de cifrado por bloques.

10.6.1 Modo ECB

El modo ECB (*Electronic Codebook*) es el método más sencillo y obvio de aplicar un algoritmo de cifrado por bloques. Simplemente se subdivide la cadena que se quiere codificar 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 codificar 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 contrincante 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.

10.6.2 Modo CBC

El modo CBC (*Cipher Book Chaining Mode*) incorpora un mecanismo de retroalimentación en el cifrado por bloques. Esto significa que la codificación de bloques anteriores condiciona la

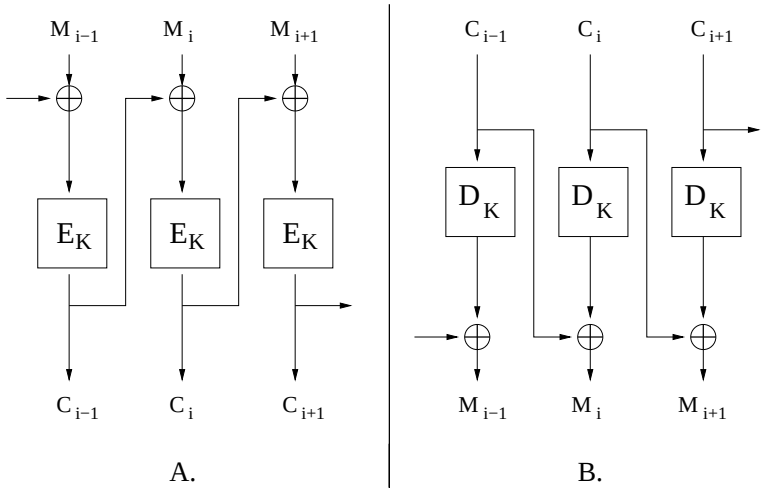

Figura 10.6: Modo de operación CBC. **A**: codificación, **B**: decodificación.

codificación del bloque actual, por lo que será imposible sustituir un bloque individual en el mensaje cifrado. Esto se consigue efectuando una operación XOR entre el bloque del mensaje que queremos codificar y el último criptograma obtenido (ver figura 10.6).

En cualquier caso, dos mensajes idénticos se codificarán de la misma forma usando el modo CBC. Más aún, dos mensajes que empiecen igual se codificarán igual hasta llegar a la primera diferencia entre ellos. Para evitar esto se emplea un *vector de inicialización*, que puede ser un bloque aleatorio, como bloque inicial de la transmisión. Este vector será descartado en destino, pero garantiza que siempre los mensajes se codifiquen de manera diferente, aunque tengan partes comunes.

10.6.3 Modo CFB

El modo CBC no empieza a codificar (o decodificar) hasta que no se tiene que transmitir (o se ha recibido) un bloque completo de información. Esta circunstancia puede convertirse en un serio inconveniente, por ejemplo en el caso de terminales, que deberían poder transmitir cada carácter que pulsa el usuario de manera individual. Una posible solución sería emplear un bloque completo para transmitir cada byte y rellenar el resto con ceros, pero esto hará que tengamos únicamente 256 mensajes diferentes en nuestra transmisión y que un atacante pueda efectuar un sencillo análisis estadístico para comprometerla. Otra opción sería rellenar el bloque con información aleatoria, aunque seguríamos desperdiciando gran parte del ancho de banda de la transmisión. El modo de operación CFB (*Cipher-Feedback Mode*) permitirá codificar la información en unidades inferiores al tamaño del bloque, lo cual permite aprovechar totalmente la capacidad de transmisión del canal de comunicaciones, manteniendo además un nivel de seguridad adecuado.

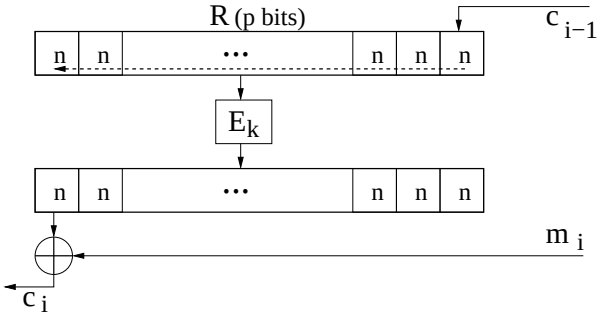

Figura 10.7: Esquema del modo de operación CFB.

En la figura 10.7 podemos ver el esquema de funcionamiento de este modo de operación. Sea p el tamaño de bloque del algoritmo simétrico, y sea n el tamaño de los bloques que queremos transmitir (n ha de divisor de p). Sea m_i el $i-$ésimo bloque del texto claro, de tamaño n. Empleamos entonces un registro de desplazamiento R de logitud p y lo cargamos con un vector de inicialización. Codificamos el registro R con el algoritmo simétrico y obtenemos en r sus n bits más a la izquierda. El bloque que deberemos enviar es $c_i = r \oplus m_i$. Desplazamos R n bits a la izquierda e introducimos c_i por la derecha.

Para descifrar basta con cargar el vector de inicialización en R y codificarlo, calculando r. Entonces $m_i = r \oplus c_i$. Desplazamos luego R e introducimos c_i por la derecha como hacíamos en el algoritmo de cifrado.

Nótese que si $n = p$, el modo CFB queda reducido al modo CBC.

10.6.4 Otros Modos

Existen protocolos criptográficos que no se basan en la transmisión de bloques, sino en un mecanismo secuencial de codificación de *streams* de tamaño variable. Estos algoritmos, que veremos con más detalle en el capítulo 11, permiten cifrar un mensaje bit a bit de forma continua y enviar cada bit antes que el siguiente sea codificado. Funcionan a partir de lo que se llama un *generador de secuencia de clave (keystream generator)*, un algoritmo que genera una clave continua de longitud infinita (o muy grande) bit a bit. Lo que se hace es aplicar una operación XOR entre cada bit del texto claro y cada bit de la clave. En el destino existe otro generador idéntico sincronizado para llevar a cabo el descifrado. El problema fundamental es mantener ambos generadores sincronizados, para evitar errores si se pierde algún bit de la transmisión.

Los algoritmos de codificación por bloques pueden ser empleados como generadores de secuencia de clave. Existen para ello otros modos de operación de estos algoritmos, como el OFB (*Output-Feedback*), que incorporan mecanismos para mantener la sincronía entre los generadores de secuencia origen y destino.

10.7 Criptoanálisis de Algoritmos Simétricos

Se podría decir que el criptoanálisis se comenzó a estudiar seriamente con la aparición de DES. Mucha gente desconfiaba (y aún desconfía) del algoritmo propuesto por la NSA. Se dice que existen estructuras *extrañas*, que muchos consideran sencillamente *puertas traseras* colocadas por la Agencia para facilitarles la decodificación de los mensajes. Nadie ha podido aún demostrar ni desmentir este punto. Lo único cierto es que el interés por buscar posibles debilidades en él ha llevado a desarrollar técnicas que posteriormente han tenido éxito con otros algoritmos.

Ni que decir tiene que estos métodos no han conseguido doblegar a DES, pero sí representan mecanismos significativamente más eficientes que la fuerza bruta para criptoanalizar un mensaje. Los dos métodos que vamos a comentar parten de que disponemos de grandes cantidades de pares texto claro-texto cifrado obtenidos con la clave que queremos descubrir.

10.7.1 Criptoanálisis Diferencial

Descubierto por Biham y Shamir en 1990, permite efectuar un ataque de texto claro escogido a DES que resulta más eficiente que la fuerza bruta. Se basa en el estudio de los pares de criptogramas que surgen cuando se codifican dos textos claros con diferencias particulares, analizando la evolución de dichas diferencias a lo largo de las rondas de DES.

Para llevar a cabo un criptoanálisis diferencial se toman dos mensajes cualesquiera (incluso aleatorios) idénticos salvo en un número concreto de bits. Usando las diferencias entre los textos cifrados, se asignan probabilidades a las diferentes claves de cifrado. Conforme tenemos más y más pares, una de las claves aparece como la más probable. Esa será la clave buscada.

10.7.2 Criptoanálisis Lineal

El criptoanálisis lineal, descubierto por Mitsuru Matsui, 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 podemos ver si se obtienen más ceros o más unos.

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.

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*—, combinándola mediante alguna función simple y reversible 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 de clave privada que explotan esta idea. Dichos algoritmos no son más que la especificación de un generador pseudoaleatorio, y permiten cifrar mensajes de longitud arbitraria, sin necesidad 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 era tan grande como el de posibles mensajes, cuando empleamos un generador tenemos como mucho tantas secuencias distintas como posibles valores iniciales de la semilla.

11.1 Secuencias Pseudoaleatorias

Como veíamos en el capítulo 8, los generadores *criptográficamente aleatorios* tenían la propiedad de que, a partir de una porción de la secuencia arbitrariamente grande, era computacionalmente intratable el problema de predecir el siguiente bit de la secuencia. Adicionalmente,

dijimos que no eran buenos como generadores aleatorios debido a que el conocimiento de la semilla nos permitiría 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 regenerarse 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 los parámetros que se empleen para calcular el valor de cada porción 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:

$$\begin{aligned} s_{i+1} &= g(s_i, k) \\ o_i &= h(s_i, k) \\ c_i &= w(m_i, o_i) \end{aligned} \tag{11.1}$$

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 sección 13.1).

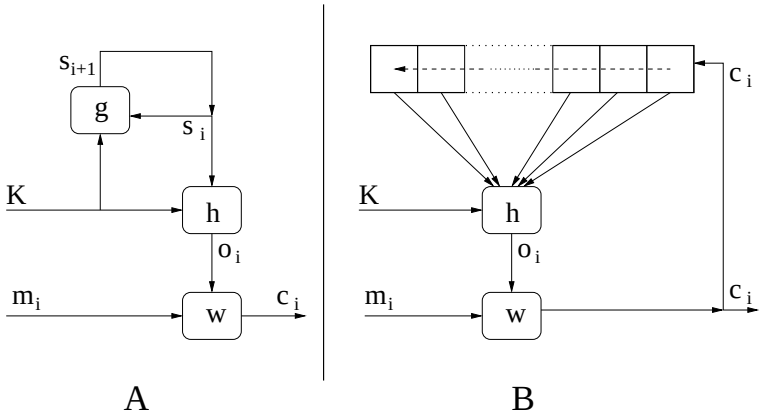

Figura 11.1: Esquema de generadores de secuencia: **A**: generador síncrono. **B**: generador asíncrono.

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 de la propia secuencia, como puede verse en la figura 11.1(b). Formalmente:

$$\begin{aligned} o_i &= h(k, c_{i-t}, c_{i-t+1}, \ldots, c_{i-1}) \\ c_i &= w(o_i, m_i) \end{aligned} \tag{11.2}$$

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 de forma automática, en cuanto llegan t bloques correctos de forma consecutiva. También será sensible a la alteración de un mensaje, ya que si se modifica la unidad de información c_i, el receptor tendrá valores erróneos de 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.

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 síncronos 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 para cifrados de flujo. 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 y su facilidad para ser implementados por *hardware*, 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 *estados*, $\{S_0, S_1, \ldots, S_{L-1}\}$, capaces de almacenar un bit cada uno (fig 11.2.a). Esta estructura viene controlada por un reloj que controla los flujos de información entre los estados. Durante cada unidad de tiempo se efectuan 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 1 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 *estados*, $\{S_0, S_1, \ldots, S_{L-1}\}$, 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 1 de un subconjunto de $\mathcal{L}$, obtenemos un registro de desplazamiento lineal.

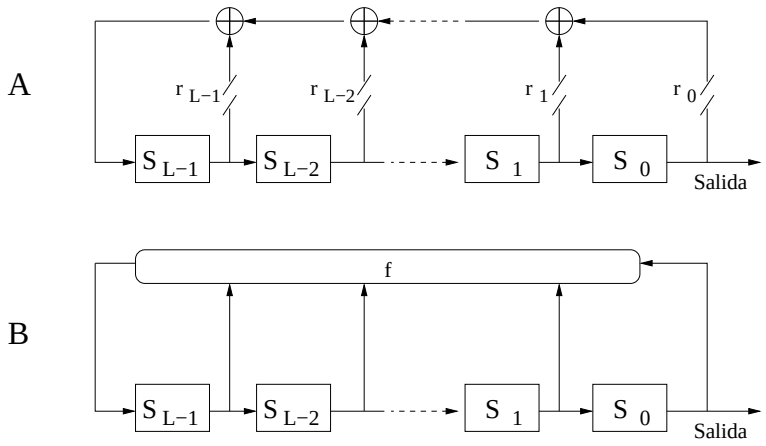

Figura 11.2: Registros de Desplazamiento Retroalimentados: **A**: Registro lineal, en el que cerrando el circuito en los puntos r_0 a r_{L-1} se puede seleccionar qué estados se emplearán para calcular el nuevo valor de S_{L-1}. **B**: Registro no lineal, donde se emplea una función f genérica.

11.3.3 Combinación de Registros de Desplazamiento

En general, los registros de desplazamiento retroalimentados no lineales presentan unas mejores condiciones como generadores de secuencia que los generadores de tipo lineal. Sin embargo, 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

$$f(R_1, R_2, \ldots, R_n) \tag{11.3}$$

siendo R_i la salida del i–ésimo registro de desplazamiento lineal.

11.4 Otros Generadores de Secuencia

Si bien los registros de desplazamiento son muy interesantes para generar secuencias mediante *hardware*, en realidad no son especialmente fáciles de implementar, ni eficientes, si se usan por *software*. Esto ha llevado a la comunidad a proponer algoritmos de generación de secuencia especialmente pensados para ser incorporados por *software*. Nosotros vamos a comentar dos de ellos: RC4 y SEAL.

11.4.1 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. Por desgracia es un algoritmo propietario, lo cual implica que no puede ser incluido en aplicaciones de tipo comercial sin pagar los *royalties* correspondientes.

El código del algoritmo no se ha publicado nunca oficialmente, pero en 1994 alguien difundió en los grupos de noticias de Internet una descripción que, como posteriormente se ha comprobado, genera las mismas secuencias. Dicha descripción consta de una S-Caja de 8*8 (ver capítulo 10), 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. $i = (i + 1) \bmod 256$
2. $j = (j + S_i) \bmod 256$
3. Intercambiar los valores de S_i y S_j
4. $t = (S_i + S_j) \bmod 256$
5. $O_r = S_t$

Para calcular los valores iniciales de la S-Caja, se hace lo siguiente:

1. $S_i = i \qquad \forall 0 \leq i \leq 255$
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:
   $j = (j + S_i + K_i) \bmod 256$
   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.

A pesar de las dudas que existen en la actualidad sobre su seguridad, es un algoritmo ampliamente utilizado en muchas aplicaciones de tipo comercial.

11.4.2 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 13.1.4) 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.

Parte IV

Criptografía de Llave Pública

Capítulo 12

Algoritmos Asimétricos de Cifrado

Los algoritmos de llave pública, o algoritmos asimétricos, 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 multitud de 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 ataques, si bien necesita una longitud de clave considerable. Otros algoritmos son los de ElGamal y Rabin.

Los algoritmos asimétricos emplean generalmente longitudes de clave mucho mayores que los simétricos. Por ejemplo, mientras que para algoritmos simétricos se considera segura una clave de 128 bits, para algoritmos asimétricos —si exceptuamos aquellos basados en curvas elípticas— se recomiendan claves de al menos 1024 bits. Además, la complejidad de cálculo que comportan estos últimos los hace considerablemente más lentos que los algoritmos de cifrado simétricos. En la práctica los métodos asimétricos se emplean únicamente para codificar la *clave de sesión* (simétrica) de cada mensaje o transacción particular.

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*. Una de ellas se emplea para codificar, mientras que la otra se usa para decodificar. 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.

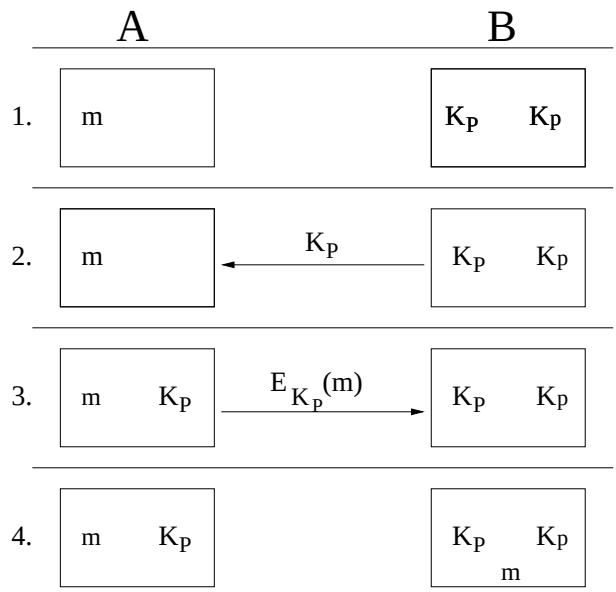

Figura 12.1: Transmisión de información empleando algoritmos asimétricos. 1. **A** tiene el mensaje m y quiere enviárselo a **B**; 2. **B** envía a **A** su clave pública, K_P; 3. **A** codifica el mensaje m y envía a **B** el criptograma $E_{K_P}(m)$; 4. **B** decodifica el criptograma empleando la clave privada K_p.

12.1.1 Protección de la Información

Una de las aplicaciones inmediatas de los algoritmos asimétricos es el cifrado de la información sin tener que transmitir la clave de decodificación, 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 $E_{K_P}(m)$. 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 llave que se hace pública es aquella que permite codificar los mensajes, mientras que la llave 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 *resumen* (ver sección 13.1), que nos permiten obtener una *firma digital* a partir de un mensaje. Dicha firma es mucho más pequeña que el mensaje original, y es muy difícil encontrar otro mensaje de lugar a la misma. 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.

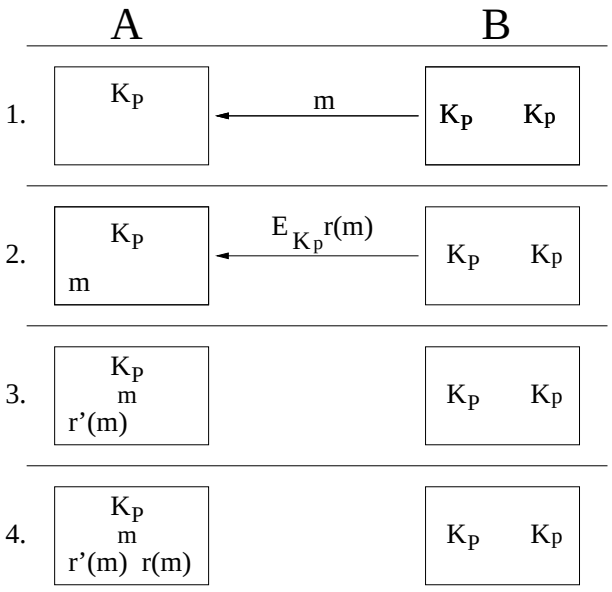

Figura 12.2: Autentificación de información empleando algoritmos asimétricos. 1. **A**, que posee la clave pública K_P de **B**, recibe el mensaje m y quiere autentificarlo; 2. **B** genera el resumen de m envía a **A** el criptograma asociado $E_{K_p}(r(m))$; 3. **A** genera por su cuenta $r'(m)$ y decodifica el criptograma recibido usando la clave K_P; 4. **A** compara $r(m)$ y $r'(m)$ para comprobar la autenticidad del mensaje m.

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.

En muchos de los 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 para codificar y autentificar.

12.2 El 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 codificar como para autentificar. Debe su nombre a sus tres inventores: Ronald Rivest, Adi Shamir y

Leonard Adleman, y estuvo bajo patente de los Laboratorios RSA 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 14) 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 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.

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 enfrenatará, si quiere recuperar un texto claro a partir del criptograma y la llave pública, a un problema de factorización (ver sección 5.6).

Para generar un par de llaves (K_P, K_p), en primer lugar se eligen aleatoriamente dos números primos grandes, p y q. Después se calcula el producto $n = pq$.

Escogeremos ahora un número e primo relativo con $(p-1)(q-1)$. (e, n) será la clave pública. Nótese que e debe tener inversa módulo $(p-1)(q-1)$, por lo que exisitrá un número d tal que

$$de \equiv 1 \pmod{(p-1)(q-1)}$$

es decir, que d es la inversa de e módulo $(p-1)(q-1)$. (d, n) será la clave privada. Esta inversa puede calcularse fácilmente empleando el Algoritmo Extendido de Euclides. Nótese que si desconocemos los factores de n, este cálculo resulta prácticamente imposible.

La *codificación* se lleva a cabo según la expresión:

$$c = m^e \pmod{n} \tag{12.1}$$

mientras que la *decodificación* se hará de la siguiente forma:

$$m = c^d \pmod{n} \tag{12.2}$$

ya que

$$c^d = (m^e)^d = m^{ed} = m^{k(p-1)(q-1)+1} = (m^k)^{(p-1)(q-1)}m$$

recordemos que $\phi(n) = (p-1)(q-1)$, por lo que, según la ecuación (5.3), $(m^k)^{(p-1)(q-1)} = 1$, lo cual nos lleva de nuevo a m, siempre y cuando m y n sean primos relativos.

Ya que en nuestro caso n es compuesto, puede ocurrir que no sea primo relativo con m. Para ver lo que ocurre, podemos llevar a cabo el siguiente razonamiento: buscamos un número a tal que

$$m^a \equiv 1 \pmod{n}$$

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 5.4), tenemos que a debe ser múltiplo de $(p-1)$ y de $(q-1)$, por lo que $a = \text{mcm}(p-1, q-1)$. Ya que el mínimo común múltiplo de $(p-1)$ y $(q-1)$ divide a $(p-1)(q-1)$, el razonamiento dado inicialmente para demostrar el buen funcionamiento del algoritmo sigue siendo válido. Por esta razón, en muchos lugares se propone obtener d de forma que:

$$de \equiv 1 \pmod{\text{mcm}(p-1, q-1)}$$

con lo que obtendremos valores más pequeños, y por lo tanto más manejables, para la clave de descifrado.

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 llave privada, se calcula $p_1 = p^{-1} \pmod{q}$, y cuando se desea descifrar un mensaje c, se plantea el siguiente sistema de congruencias:

$$\begin{aligned} [c \mod p]^{[d \mod (p-1)]} &\equiv m_1 \pmod{p} \\ [c \mod q]^{[d \mod (q-1)]} &\equiv m_2 \pmod{q} \end{aligned}$$

Evidentemente, 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—.

En la práctica, cogeremos p y q con un número grande de bits, por ejemplo 200, con lo que n tendrá 400 bits. Subdividiremos el mensaje que queramos enviar en bloques de 399 bits —de esta forma garantizamos que el valor de cada bloque sea menor que n— y efectuaremos la codificación de cada uno. Obtendremos un mensaje cifrado ligeramente más grande, puesto que estará compuesto por bloques de 400 bits. Para decodificar partiremos el mensaje cifrado en bloques de 400 bits —ya que en este caso sabemos que el valor de cada bloque ha de ser menor que n—, y obtendremos bloques de 399 bits.

El 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.

12.2.1 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 primitiva y que simultáneamente nos parta un rayo (tabla 1.1). Por otra parte, si p o q fueran compuestos, el algoritmo RSA simplemente no funcionaría.

12.2.2 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

Se puede demostrar matemáticamente que existen ciertos casos para los cuales el algoritmo RSA deja el mensaje original tal cual, es decir

$$m^e = m \pmod{n} \tag{12.3}$$

En realidad, siempre hay mensajes que quedan inalterados al ser codificados mediante RSA, sea cual sea el valor de n. Nuestro objetivo será reducir al mínimo el número de éstos. Se puede comprobar que, siendo $n = pq$ y e el exponente para codificar,

$$\sigma_n = [1 + \text{mcd}(e-1, p-1)] \cdot [1 + \text{mcd}(e-1, q-1)]$$

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 $\text{mcd}(e-1, p-1)$ 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 no deben preocuparnos. No obstante, como medida de precaución, se puede calcular σ_n a la hora de generar las llaves pública y privada.

Claves Demasiado Cortas

Actualmente se considera segura una clave RSA con una longitud de n de al menos 768 bits, si bien se recomienda el uso de claves no inferiores a 1024 bits. Hasta hace relativamente poco se recomendaban 512 bits, pero en mayo de 1999, Adi Shamir presentó el denominado dispositivo *Twinkle*, un ingenio capaz de factorizar números de manera muy rápida, aprovechando los últimos avances en la optimización de algoritmos específicos para esta tarea. Este dispositivo, aún no construido, podría ser incorporado en ordenadores de bajo coste y pondría en serio peligro los mensajes cifrados con claves de 512 bits o menos.

Teniendo en cuenta los avances de la tecnología, y suponiendo que el algoritmo RSA no sea roto analíticamente, deberemos escoger la longitud de la clave en función del tiempo que queramos que nuestra información permanezca en secreto. Efectivamente, una clave de 1024 bits parece a todas luces demasiado corta como para proteger información por más de unos pocos años.

Ataques de Intermediario

El *ataque de intermediario* (figura 12.3) puede darse con cualquier algoritmo asimétrico. 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. Cuando A codifique el mensaje, C lo interceptará de nuevo, decodificándolo con su clave propia y empleando K_B para recodificarlo y enviarlo a B. Ni A ni B son conscientes de que sus mensajes están siendo interceptados.

La única manera de evitar esto consiste en asegurar a A que la clave pública que tiene de B es auténtica. Para ello nada mejor que ésta esté *firmada* por un amigo común, que certifique la autenticidad de la clave. En la actualidad existen los llamados *anillos de confianza*, que permiten certificar la autenticidad de las claves sin necesidad de centralizar el proceso. Por eso se nos recomienda cuando instalamos paquetes como el PGP que firmemos todas las claves sobre las que tengamos certeza de su autenticidad, y sólamente esas.

Ataques de Texto en Claro Escogido

Existe una familia de ataques a RSA que explotan la posibilidad de que un usuario codifique y firme un único mensaje empleando el mismo par de llaves. Para que el ataque surta efecto,

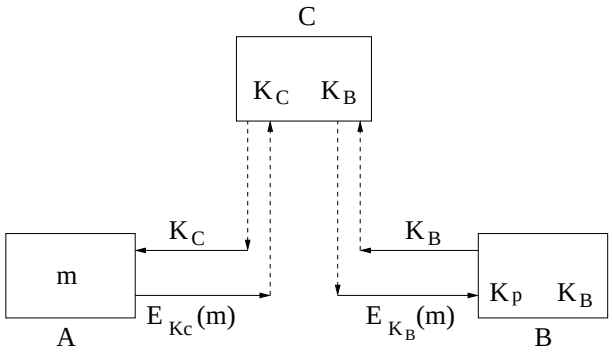

Figura 12.3: Ataque de intermediario para un algoritmo asimétrico.

la firma debe hacerse codificando el mensaje completo, no el resultado de una función *resumen* sobre él. Por ello se recomienda que las firmas digitales se lleven a cabo siempre sobre una función resumen del mensaje, nunca sobre el mensaje en sí.

Otro tipo de ataque con texto claro escogido podría ser el siguiente: para 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.

Ataques de Módulo Común

Podría pensarse que, una vez generados p y q, será más rápido generar tantos pares de llaves como queramos, en lugar de tener que emplear dos números primos diferentes en cada caso. Sin embargo, si lo hacemos así, un atacante podrá decodificar nuestros mensajes sin necesidad de la llave privada. Sea m el texto claro, que codificamos empleando dos claves de cifrado diferentes e_1 y e_2. Los criptogramas que obtenemos 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

$$c_1^r c_2^s = m^{e_1 r} m^{e_2 s} = m^{e_1 r + e_2 s} = m^1 \pmod{n}$$

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 mayoría 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 codificación e es demasiado bajo, existe la posibilidad de que un atacante pueda romper el sistema. Esto se soluciona *rellenando* los m que se codifican con bits aleatorios por la izquierda. Por ejemplo, si n es de 400 bits, una estrategia razonable sería coger bloques de 392 bits (que es un número exacto de bytes) e incluirles siete bits aleatorios por la izquierda. Cuando decodifiquemos simplemente ignoraremos esos siete bits.

Por otra parte, si d es demasiado bajo, también existen mecanismos para romper el sistema, por lo que se recomienda emplear valores altos para d.

Firmar y Codificar

Con el algoritmo RSA nunca se debe firmar un mensaje después de codificarlo, por el contrario, debe firmarse primero. Existen ataques que aprovechan mensajes primero codificados y luego firmados, aunque se empleen funciones *resumen*.

12.3 Otros Algoritmos Asimétricos

12.3.1 Algoritmo de Diffie-Hellman

Es un algoritmo asimétrico, basado en el problema de Diffie-Hellman (sección 5.4.3), 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 llaves públicas en el sentido estricto, sino una información compartida por los dos comunicantes.

Sean A y B los interlocutores en cuestión. En primer lugar, se calcula un número primo p y un generador α de $\mathbb{Z}_p^*$, 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.

12.3.2 Algoritmo de ElGamal

Fue diseñado en un principio para producir firmas digitales, pero posteriormente se extendió también para codificar mensajes. Se basa en el problema de los logaritmos discretos, que está íntimamente relacionado con el de la factorización, y en el de Diffie-Hellman.

Para generar un par de llaves, 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 llave pública es (p, y, n), mientras que la llave 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, tal que $\text{mcd}(k, n-1) = 1$, 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

$$y^a a^b = p^m \pmod{n} \tag{12.5}$$

Codificación de ElGamal

Para codificar 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

$$m = b \cdot a^{-x} \pmod n \tag{12.7}$$

12.3.3 Algoritmo de Rabin

El sistema de llave asimétrica de Rabin se basa en el problema de calcular raices 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 codificar un mensaje m, simplemente se calcula

$$c = m^2 \pmod n \tag{12.8}$$

La decodificación del mensaje se hace calculando lo siguiente:

$$\begin{array}{rcl} m_1 & = & c^{(p+1)/4} \pmod p \\ m_2 & = & (p - c^{(p+1)/4}) \pmod p \\ m_3 & = & c^{(q+1)/4} \pmod q \\ m_4 & = & (q - c^{(q+1)/4}) \pmod q \end{array}$$

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

$$\begin{array}{rcl} m_a & = & (am_1 + bm_3) \pmod n \\ m_b & = & (am_1 + bm_4) \pmod n \\ m_c & = & (am_2 + bm_3) \pmod n \\ m_d & = & (am_2 + bm_4) \pmod n \end{array} \tag{12.9}$$

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.

12.3.4 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 llave pública-llave privada

El algoritmo de generación de claves es el siguiente:

1. Seleccionar un número primo q tal que $2^{159} < q < 2^{160}$.
2. Escoger t tal que $0 \leq t \leq 8$, y seleccionar un número primo p tal que $2^{511+64t} < p < 2^{512+64t}$, y que además q sea divisor de $(p-1)$.
3. Seleccionar un elemento $g \in \mathbb{Z}_p^*$ y calcular $\alpha = g^{(p-1)/q} \bmod p$.
4. Si $\alpha = 1$ volver al paso 3.
5. Seleccionar un número entero aleatorio a, tal que $1 \leq a \leq q-1$
6. Calcular $y = \alpha^a \bmod p$.
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 resumen sobre el mensaje m (ver sección 13.1), la generación de una firma se hace mediante el siguiente algoritmo:

1. Seleccionar un número aleatorio k tal que $0 < k < q$.
2. Calcular $r = (\alpha^k \bmod p) \bmod q$.
3. Calcular $k^{-1} \bmod q$.
4. Calcular $s = k^{-1}(h + ar) \bmod q$.
5. La firma del mensaje m es el par (r, s).

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 el valor de h a partir de m.
3. Calcular $\omega = s^{-1} \bmod q$.
4. Calcular $u_1 = \omega \cdot h \bmod q$ y $u_2 = \omega \cdot r \bmod q$.
5. Calcular $v = (\alpha^{u_1} y^{u_2} \bmod p) \bmod q$.
6. Aceptar la firma si y sólo si $v = r$.

12.4 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 obtiene 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 muchos de los algoritmos asimétricos más populares. A modo de ejemplo, en esta sección veremos cómo se redefine el algoritmo de cifrado de ElGamal.

12.4.1 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

$$\begin{aligned} \mathbf{a} &= k\mathbf{p} \\ \mathbf{b} &= \mathbf{m} + k\mathbf{y} \end{aligned} \tag{12.11}$$

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

$$\mathbf{m} = -(x\mathbf{a}) + \mathbf{b} \tag{12.12}$$

12.5 Los Protocolos SSL y TLS

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. Su fundamento consiste en interponer una fase de codificación de los mensajes antes de enviarlos por 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.

TLS (descrito en el documento *RFC 2246*) es un nuevo protocolo muy similar a SSL, ya que de hecho se basa en la versión 3.0 de este último, mejorándolo en algunos aspectos. Si bien su nivel de implantación aún no es muy elevado, todo parece indicar que está llamado a ser su sustituto.

Una comunicación SSL o TLS consta fundamentalmente de dos fases:

1. Fase de saludo (*handshaking*). Consiste básicamente en una identificación mutua de los interlocutores, para la cual se emplean habitualmente los certificados X.509, que veremos en la sección 13.6. Tras el intercambio de claves públicas, los dos sistemas escogen una clave de sesión, de tipo simétrico.

2. Fase de comunicación. En esta fase se produce el auténtico intercambio de información, que se codifica mediante la clave de sesión acordada en la fase de saludo.

Cada sesión lleva asociado un identificador único que evita la posibilidad de que un atacante *escuche* la red y repita exactamente lo mismo que ha oído, aún sin saber lo que significa, para engañar a uno de los interlocutores.

Las ventajas de SSL —y en el futuro, de TLS— 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. Desde hace tiempo los principales navegadores de Internet incorporan un módulo SSL, que se activa de forma automática cuando es necesario. Hasta diciembre de 1999, debido a las restricciones de exportación de material criptográfico existentes en los EE.UU., la mayoría de los navegadores incorporaban un nivel de seguridad bastante pobre (claves simétricas de 40 bits), por lo que conviene comprobar qué nivel de seguridad soporta nuestro navegador, y actualizarlo si fuera necesario.

12.6 Ejercicios Propuestos

1. Suponga un sistema RSA con los siguientes parámetros:

   - $N = 44173$
   - $K_P = 25277$
   - $C = 8767, 18584, 7557, 4510, 40818, 39760, 4510, 39760, 6813, 7557, 14747$

   a) Factorizar el módulo N.

   b) Calcular la llave privada K_p.

   c) Descifrar el mensaje C.

Capítulo 13

Métodos de Autentificación

Por autentificación entenderemos cualquier método que nos permita comprobar de manera segura alguna característica sobre un objeto. Dicha característica puede ser su origen, su integridad, su identidad, etc. Consideraremos tres grandes tipos dentro de los métodos 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 mecanismo se conoce habitualmente como *firma digital*.
- Autentificación *de usuario mediante contraseña*. En este caso se trata de garantizar la presencia de un usuario legal en el sistema. El usuario deberá poseer una contraseña secreta que le permita identificarse.
- Autentificación *de dispositivo*. Se trata de garantizar la presencia de un dispositivo válido. Este dispositivo puede estar solo o tratarse de una *llave electrónica* que sustituye a la contraseña para identificar 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. De todas formas, en este capítulo únicamente trataremos métodos de autentificación basados en técnicas criptográficas.

13.1 Firmas Digitales. Funciones *Resumen*

En el capítulo 12 vimos que la criptografía asimétrica permitía autentificar información, es decir, poder asegurar que un mensaje m proviene de un emisor A y no de cualquier otro. Asímismo vimos que la autentificación debía hacerse empleando una *función resumen* y no

codificando el mensaje completo. En esta sección estudiaremos dichas funciones resumen, también conocidas como MDC (*modification detection codes*), que nos van a permitir crear *firmas digitales*.

Sabemos que un mensaje m puede ser autentificado codificando con la llave privada K_p el resultado de aplicarle una función resumen, $E_{K_p}(r(m))$. Esa información adicional (que denominaremos *firma* o *signatura* del mensaje m) sólo puede ser generada por el poseedor de la clave privada K_p. Cualquiera que tenga la llave pública correspondiente estará en condiciones de decodificar y verificar la firma. Para que sea segura, la *función resumen* $r(x)$ debe cumplir además ciertas características:

- $r(m)$ es de longitud fija, independientemente de la longitud de m.
- Dado m, es fácil calcular $r(m)$.
- Dado $r(m)$, es computacionalmente intratable recuperar m.
- Dado m, es computacionalmente intratable obtener un m' tal que $r(m) = r(m')$.

13.1.1 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 de personas que hay que poner en una habitación para que la probabilidad de que el cumpleaños de una de ellas sea el mismo día que el mío supere 50%? Debemos calcular n tal que

$$n\frac{1}{365} > 0.5$$

luego $n > 182$. Sin embargo, ¿cuál sería la cantidad de gente necesaria para la probabilidad de que dos personas cualesquiera tengan el mismo cumpleaños supere el 50%? Cada pareja tiene una probabilidad 1/365 de compartir el cumpleaños, y en un grupo de n personas hay $n(n-1)/2$ parejas diferentes de personas, luego

$$\frac{n(n-1)}{2} \cdot \frac{1}{365} > 0.5$$

Esto se cumple si $n > 19$, una cantidad *sorprendentemente* mucho menor que 182.

La consecuencia de esta *paradoja* es que aunque resulte muy difícil dado m calcular un m' tal que $r(m) = r(m')$, es mucho menos costoso buscar dos valores aleatorios m y m', tales que $r(m) = r(m')$.

En el caso de una firma de 64 bits, necesitaríamos 2^{64} mensajes dado un m para obtener el m', pero bastaría con generar aproximadamente 2^{32} mensajes aleatorios para que aparecieran

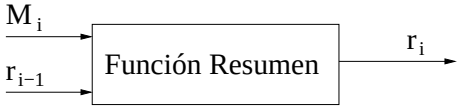

Figura 13.1: Estructura iterativa de una *función resumen*.

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 con una computadora que generara un millón de mensajes por segundo, mientras que el segundo necesitaría apenas una hora.

Hemos de añadir pues a nuestra lista de condiciones sobre las funciones resumen la siguiente:

- Debe ser difícil encontrar dos mensajes aleatorios, m y m', tales que $r(m) = r(m')$.

Hoy por hoy se recomienda emplear signaturas de al menos 128 bits, siendo 160 bits el valor más usado.

13.1.2 Estructura de una Función Resumen

En general, las funciones resumen se basan en la idea de *funciones de compresión*, que dan como resultado bloques de longitud n a partir de bloques de longitud m. Estas funciones se encadenan de forma iterativa, haciendo que la entrada en el paso i sea función del i-ésimo bloque del mensaje y de la salida del paso $i-1$ (ver figura 13.1). En general, se suele incluir en alguno de los bloques del mensaje m —al principio o al final—, información sobre la longitud total del mensaje. De esta forma se reducen las probabilidades de que dos mensajes con diferentes longitudes den el mismo valor en su resumen.

En esta sección veremos dos algoritmos de generación de firmas: MD5 y SHA-1.

13.1.3 Algoritmo MD5

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, diseñado por Ron Rivest, 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 se lleva a cabo añadiendo un 1 seguido de tantos ceros como sea necesario. En segundo lugar, se añaden 64 bits con el valor de b, empezando por el byte menos significativo. De esta forma tenemos el mensaje

como un número entero de bloques de 512 bits, y además le hemos añadido información sobre su longitud.

Seguidamente, se inicializan cuatro registros de 32 bits con los siguientes valores hexadecimales [1]:

$$\begin{aligned} A &= 67452301 \\ B &= EFCDAB89 \\ C &= 98BADCFE \\ D &= 10325476 \end{aligned}$$

Posteriormente comienza el *lazo 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:

$$\begin{aligned} FF(a,b,c,d,m_j,s,t_i) &\text{ representa } a = b + ((a + F(b,c,d) + m_j + t_i) \lhd s) \\ GG(a,b,c,d,m_j,s,t_i) &\text{ representa } a = b + ((a + G(b,c,d) + m_j + t_i) \lhd s) \\ HH(a,b,c,d,m_j,s,t_i) &\text{ representa } a = b + ((a + H(b,c,d) + m_j + t_i) \lhd s) \\ II(a,b,c,d,m_j,s,t_i) &\text{ representa } a = b + ((a + I(b,c,d) + m_j + t_i) \lhd s) \end{aligned}$$

donde la función $a \lhd s$ representa desplazar circularmente el valor a s bits a la izquieda.

Las 64 operaciones que se realizan en total quedan agrupadas en cuatro rondas.

- Primera Ronda:
  $FF(a,b,c,d,m_0,7,D76AA478)$
  $FF(d,a,b,c,m_1,12,E8C7B756)$
  $FF(c,d,a,b,m_2,17,242070DB)$
  $FF(b,c,d,a,m_3,22,C1BDCEEE)$
  $FF(a,b,c,d,m_4,7,F57C0FAF)$
  $FF(d,a,b,c,m_5,12,4787C62A)$
  $FF(c,d,a,b,m_6,17,A8304613)$

[1] Los números que aquí se indican son los valores enteros hexadecimales tal y como se introducirían en el código fuente de un programa, suponiendo que el byte menos significativo quede en la dirección de memoria más baja (*little endian*). Nótese que de esta forma, si escribimos los valores hexadecimales byte a byte de cada registro en el orden en que quedan en la memoria, tendríamos que en las posiciones correspondientes al registro A aparecerían los bytes 01-23-45-67, y así sucesivamente.

$FF(b, c, d, a, m_7, 22, FD469501)$
$FF(a, b, c, d, m_8, 7, 698098D8)$
$FF(d, a, b, c, m_9, 12, 8B44F7AF)$
$FF(c, d, a, b, m_{10}, 17, FFFF5BB1)$
$FF(b, c, d, a, m_{11}, 22, 895CD7BE)$
$FF(a, b, c, d, m_{12}, 7, 6B901122)$
$FF(d, a, b, c, m_{13}, 12, FD987193)$
$FF(c, d, a, b, m_{14}, 17, A679438E)$
$FF(b, c, d, a, m_{15}, 22, 49B40821)$

- 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, se procesa 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.

En los últimos tiempos el algoritmo MD5 ha mostrado ciertas *debilidades*, aunque sin implicaciones prácticas reales, por lo que se sigue considerando en la actualidad un algoritmo seguro, si bien su uso tiende a disminuir.

13.1.4 El 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 los algoritmos de cifrado propuestos por esta organización, SHA-1 se considera seguro y libre de *puertas traseras*, ya que favorece a los propios intereses de la NSA que el algoritmo sea totalmente seguro. Produce firmas de 160 bits, a partir de bloques de 512 bits del mensaje original.

El algoritmo es similar a MD5, y se inicializa igual que éste, 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 bytes del propio mensaje. A diferencia de MD5, SHA-1 emplea cinco registros de 32 bits en lugar de cuatro:

$$\begin{aligned} A &= 67452301 \\ B &= EFCDAB89 \\ C &= 98BADCFE \\ D &= 10325476 \\ E &= C3D2E1F0 \end{aligned}$$

Una vez que los cinco valores están inicializados (ver la nota al pie de la página 160), se copian en cinco variables, a, b, c, d y e. El lazo 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:

$$\begin{aligned} w_t &= m_t && \text{para } t = 0 \ldots 15 \\ w_t &= (w_{t-3} \oplus w_{t-8} \oplus w_{t-14} \oplus w_{t-16}) \lhd 1 && \text{para } t = 16 \ldots 79 \end{aligned}$$

Como curiosidad, diremos que la NSA introdujo el desplazamiento a la izquierda para corregir una posible debilidad del algoritmo, lo cual supuso modificar su nombre y cambiar el antiguo (SHA) por SHA-1.

El lazo principal del algoritmo es entonces el siguiente:

$$\begin{aligned} \text{FOR} \quad & t = 0 \text{ TO } 79 \\ & i = t \text{ div } 20 \\ & Tmp = (a \lhd 5) + A(b,c,d) + e + w_t + K_i \\ & e = d \\ & d = c \\ & c = b \lhd 30 \\ & b = a \\ & a = Tmp \end{aligned}$$

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.

13.1.5 Funciones de Autentificación de Mensaje

Frente a los MDC, vistos en la sección anterior, existe otra clase de funciones resumen, llamada genéricamente MAC (*message authentication codes*). Los MAC se caracterizan fundamentalmente 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 efecto conseguido es que el receptor puede, mediante el cálculo de dicha función, comprobar tanto la integridad como la procedencia del mensaje.

Existen multitud de MAC diferentes, pero lo más común es cifrar el mensaje mediante un algoritmo simétrico en modo CBC —ver sección 10.6—, y emplear la salida correspondiente al cifrado del último bloque.

13.2 Autentificación de Dispositivos

Los algoritmos simétricos pueden ser empleados para autentificar dispositivos, siempre que éstos permitan hacer operaciones de cifrado/descifrado a la vez que impidan acceder físicamente a la clave que llevan almacenada. Un ejemplo de este mecanismo de autentificación lo tenemos en las tarjetas que emplean los teléfonos GSM. Dichas tarjetas llevan implementado un algoritmo simétrico de cifrado, y usan una clave k almacenada en un lugar de la memoria que no se puede leer desde el *exterior*. En cada tarjeta se graba una única clave, de la que se guarda una copia en lugar seguro. Si la compañía quiere identificar una tarjeta simplemente genera un bloque de bits aleatorio X y calcula su criptograma $E_k(X)$ asociado. Posteriormente se envía X a la tarjeta para que lo codifique. Si ambos mensajes codificados coinciden, la tarjeta será auténtica. Esta técnica se conoce como *autentificación por desafío* (fig. 13.2). Nótese que la clave k en ningún momento queda comprometida.

13.3 Autentificación de Usuario Mediante Contraseña

El sistema de autentificación basa su funcionamiento en una información secreta conocida únicamente por el usuario, que le permite identificarse positivamente frente al sistema. Supondremos que el usuario se encuentra en un terminal *seguro*, es decir, libre de posibles ataques del exterior. Distinguiremos entonces dos casos claramente diferenciados:

a) El sistema se comunica con el usuario, pero éste no puede *entrar* en él. Piénsese en un cajero automático. El usuario carece de acceso a los archivos del sistema y no tiene posibilidad

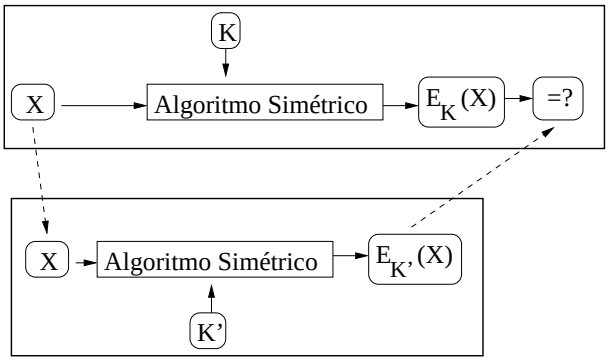

Figura 13.2: Esquema de autentificación por desafío.

ejecutar aplicaciones en él, únicamente puede llevar a cabo una serie de opearciones muy restringidas.

b) El sistema permite al usuario *entrar*. Este es el caso de los sistemas operativos como UNIX, que ofreen la posibilidad a los usuarios operar con el sistema desde terminales remotos. Normalmente el usuario tiene acceso más o menos restringido a los archivos del sistema y puede ejecutar programas.

El primer caso es el más simple y sencillo de resolver. Basta con que el sistema mantenga la lista de usuarios y sus contraseñas asociadas en un archivo. Como este archivo no puede ser consultado desde el exterior, es imposible averiguar la clave de un usuario. Para protegerse de los ataques por la fuerza bruta, será suficiente con limitar el número de intentos desde un terminal concreto e introducir retardos cuando la contraseña introducida sea errónea.

El caso b es considerablemente más complejo. Por un lado deberemos tomar las mismas medidas que en el caso anterior para protegernos de los ataques por la fuerza bruta, y por otro hemos de tener en cuenta que cualquier usuario puede acceder a algunos ficheros. En versiones *antiguas* de sistemas operativos UNIX el fichero con las contraseñas podía ser descargado por cualquier usuario anónimo —que tiene un nombre concreto y una contraseña genérica—, por lo que las palabras clave no pueden ser almacenadas como texto claro en dicho fichero. El mecanismo que surge entonces de manera inmediata consiste en almacenar en el fichero de claves la signatura de cada contraseña. De esta forma será difícil *adivinar* una contraseña que se ajuste a una signatura concreta. Los sistemas operativos modernos impiden además leer el fichero de claves de forma directa, pero eso no evita que en algunos casos éste pueda quedar comprometido.

13.3.1 Ataques Mediante Diccionario

El principal problema de las palabras clave son las elecciones *poco afortunadas* por parte de los usuarios. Desgraciadamente todavía hay personas que emplean su fecha de nacimiento,

el nombre de algún familiar o la matrícula del coche como contraseña. Un atacante avispado podría tratar de generar millones de claves y construir un *diccionario*. El siguiente paso sería precalcular las signaturas de todas las claves que hay en su diccionario. Si de alguna manera ha obtenido el fichero con las signaturas de las claves, bastaría con compararlas con las de su diccionario para obtener en pocos segundos una contraseña que le permita entrar en el sistema. Tengamos en cuenta que un diccionario con más de 150.000 claves de ocho caracteres cabe (sin comprimir) en un diskette de 3 pulgadas y media, y que ha habido casos en los que con diccionarios de este tamaño se ha conseguido averiguar un sorprendente número de claves.

Para protegerse frente a este tipo de ataques se introduce en el cálculo de la signatura de la contraseña la denominada *sal*, que no es ni más ni menos que un conjunto de bits aleatorios que se añaden a la palabra clave antes de calcular su firma. En el fichero de claves se almacenará, junto con la signatura, la sal necesaria para su obtención. Esto obligará al atacante a recalcular todas las signaturas de su diccionario antes de poder compararlas con cada una de las entradas del fichero de claves del sistema. De todas formas, si la contraseña buscada aparece en el diccionario, el único inconveniente para el atacante será que en lugar de obtenerla de forma casi instantánea, se tardará algún tiempo en conseguirla. Deberemos pues evitar a toda costa emplear contraseñas que puedan aparecer en un diccionario.

Además de estar bien salvaguardadas, las palabras clave han de cumplir una serie de condiciones para que puedan considerarse *seguras*:

1. Deben ser memorizadas. Una contraseña jamás debe ser escrita en un papel, por razones obvias.

2. Suficientemente complejas. Una buena contraseña debe constar de al menos ocho letras. Pensemos que si empleamos únicamente seis caracteres alfanuméricos (números y letras), tenemos *únicamente* 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.

3. 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.

4. Fáciles de recordar. Puesto que una palabra clave ha de ser memorizada, no tiene 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*.

5. Deben ser modificadas con frecuencia. Hemos de partir de la premisa de que toda palabra clave caerá 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 claves.

Para ilustrar este apartado usaremos como ejemplo la desafortunada revelación del archivo con las signaturas de todas las contraseñas de algunos proveedores de servicios de Internet. Muchos de los usuarios comprobaban horrorizados como sus claves eran reveladas en cuestión de pocos minutos por cualquier programa que realizara ataques por diccionario en un simple PC doméstico. Si bien el compromiso de este tipo de archivos es un grave inconveniente, si las claves que éste almacena hubieran tenido suficiente calidad, hubieran resistido sin problemas el ataque, permaneciendo a salvo.

13.4 Dinero Electrónico

Si hay un concepto contrario al de *autentificación*, éste es *falsificación*. Y cuando hablamos de falsificar casi inmediatamente nos viene a la cabeza el objeto más falsificado de la historia: el dinero. El objetivo de los países a la hora de fabricar dinero siempre ha sido evitar su falsificación —lo cual equivale a facilitar su autentificación—. Parece bastante razonable dedicar pues un breve comentario dentro de esta obra al dinero como objeto de autentificación. Obviamente, nos referimos al *dinero electrónico*.

El dinero *físico* es algo bastante engorroso. Es incómodo de transportar, se desgasta con facilidad y suele ser susceptible de falsificación. Además debe ser cambiado periódicamente debido a la renovación de las monedas. Para sustituirlo están las tarjetas de crédito y los cheques. El problema que éstos presentan es que rompen el anonimato de quien los emplea, por lo que la privacidad queda comprometida. Existen sin embargo protocolos que permiten el intercambio de capital de una forma segura y anónima. Es lo que denominaremos *dinero electrónico*.

Las ventajas que reportará su extensión en un futuro próximo son evidentes. Facilitará el comercio electrónico y las compras por Internet, y además garantizará el anonimato en las transacciones comerciales. Hoy por hoy no existe un único protocolo aceptado universalmente, aunque sí muchas propuestas. Por ello haremos únicamente una breve introducción acerca de cómo debería ser un protocolo a título meramente ilustrativo.

Supongamos que queremos enviar un cheque anónimo. Para ello creamos cien cheques por la misma cantidad, los metemos cada uno en un sobre y los enviamos al banco. El banco abre noventa y nueve al azar y se asegura de que todos llevan la misma cantidad. Al que queda le pone su sello sin abrirlo y nos lo devuelve, restando la cantidad de nuestra cuenta corriente. Tenemos ahora un cheque validado por el banco, pero del que el banco no sabe nada (la probabilidad de que tenga una cantidad diferente de la que el banco supone es únicamente del uno por ciento). Cuando entreguemos ese cheque y alguien quiera cobrarlo, bastará con que lo lleve al banco, que verificará su sello y abonará su importe, sin conocer su procedencia. Este protocolo se puede implementar mediante criptografía asimétrica de la siguiente forma: se construyen cien órdenes de pago anónimas y se envían al banco. Éste las comprueba y firma digitalmente una, restando además la cantidad correspondiente en nuestra cuenta. El destinatario podrá cobrar la orden de pago cuando quiera.

El problema que surge con el protocolo anterior es que una orden se puede cobrar varias

veces. Para evitar esto basta con incluir una cadena aleatoria en cada orden de pago, de forma que sea muy difícil tener dos órdenes con la misma cadena. Cada una de las cien órdenes tendrá pues una cadena de identificación diferente. El banco, cuando pague la cantidad, únicamente tendrá que comprobar la cadena de identificación de la orden para asegurarse de que no la ha pagado ya.

Ahora cada orden de pago es única, por lo que el banco puede detectar una orden duplicada, pero no sabe quién de los dos ha cometido fraude: el que paga o el que cobra. Existen mecanismos que permiten saber, cuando la orden de pago aparece duplicada, quién de los dos ha intentado engañar. Si lo ha hecho el cobrador, puede ser localizado sin problemas, pero ¿y si quien envía dos veces la misma orden es el pagador?. El protocolo completo de dinero electrónico hace que la identidad del pagador quede comprometida si envía dos veces la misma orden de pago, por lo que podrá ser capturado.

En cuanto a los diferentes protocolos propuestos hoy en día hay que decir que son muy dispares, y que todavía ninguno de ellos ha sido adoptado como estándar. Los clasificaremos en tres grupos, según el tipo de criptografía en que se basen:

- *Basados en Criptografía Simétrica*: NetBill y NetCheque.
- *Basados en Criptografía Asimétrica*: Proyecto CAFE, ECash, NetCash, CyberCash, iKP de IBM, y Anonymous Credit Cards (ACC) de los Laboratorios AT & Bell.
- *No basados en Criptografía*: ISN, Compuserve y FIRST VIRTUAL Holdings Incorporated.

Por desgracia, ninguno de estos protocolos acaba de imponerse, precisamente porque permiten recuperar el anonimato al comprador —y, por tanto, su privacidad—, y dificultan el rastreo de capital. Pero mucho nos tememos que hay una causa mucho más sutil y poderosa a la vez en esta sociedad mercantilizada. Supongamos por un momento que yo poseo una gran empresa que vende todo tipo de productos, y que mis clientes emplean sus tarjetas de crédito para comprar. Yo puedo almacenar un perfil de cada tarjeta de crédito con la que se pagan los productos, incluso desconociendo los datos personales del titular, en el que se reflejen detalladamente sus hábitos de consumo, lo cual me permite elaborar campañas de mercado mucho más eficientes, por ejemplo incluyendo publicidad de los productos que a cada cliente más le puedan interesar cada vez que se le sirva un pedido. Y todo ello sin violar la *intimidad* de los compradores, puesto que no se almacena ningún dato personal.

13.5 Esteganografía

La esteganografía consiste en almacenar información *camuflada* dentro de otra información. Supongamos que queremos enviar un mensaje secreto a un interlocutor que se encuentra en un lugar donde la Criptografía está prohibida. Podríamos, por ejemplo, enviarle una imagen de mapa de bits y utilizar el bit menos significativo del color de cada píxel para guardar cada

bit del mensaje secreto. La imagen será válida y un observador externo nunca sospechará que en realidad esconde un mensaje secreto.

Existen infinidad de métodos de esteganografía, sólo limitados por la imaginación, pero, ¿por qué incluir esta técnica dentro del capítulo dedicado a autentificación? La respuesta es sencilla: en general la esteganografía consiste en mezclar información *útil* con información de alguna otra naturaleza, que sólo sirve para despistar. Podríamos definirla entonces como el mecanismo que nos permite entresacar la información útil, reduciendo este problema a una simple autentificación.

Volvamos al ejemplo de la imagen de mapa de bits. Si el observador externo conociera el algoritmo que hemos empleado para camuflar nuestro mensaje dentro de la imagen, el sistema quedaría automáticamente comprometido. Alguien podría proponer entonces emplear la Criptografía y almacenar en los bits menos significativos de cada píxel la versión codificada del mensaje. Existe sin embargo una forma más elegante de proteger la información sin emplear ningún algoritmo criptográfico.

Podríamos generar una gran cantidad de información irrelevante y subdividirla junto con el mensaje original en pequeños *paquetes*, a los que añadiríamos un código de identificación (signatura), de forma que sólo los paquetes que corresponden al mensaje contengan una signatura correcta. Si enviamos una secuencia de paquetes en la que aparece el mensaje original entremezclado con la *basura*, sólo quien disponga del mecanismo de autentificación correcto —que podría depender de una *clave*— estará en condiciones de recuperar el mensaje original. Sin embargo, el mensaje ha sido enviado como texto claro, sin ser codificado en ningún momento. El ejemplo del mapa de bits no sería más que un caso particular de este esquema, en el que el algoritmo de autentificación simplemente considera válidos los bits menos significativos de cada píxel y descarta todos los demás.

13.6 Certificados X.509

Un certificado es esencialmente una clave pública y un identificador, firmados digitalmente por una *autoridad de certificación*, y su utilidad es demostrar que una clave pública pertenece a un usuario concreto. El formato de certificados X.509 (Recomendación X.509 de CCITT: *"The Directory - Autentication Framework"*. 1988) es el más común y extendido 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 de certificador.
- Identificador único de sujeto.
- Extensiones.
- Firma digital de todo lo anterior generada por el certificador.

Estos certificados se estructuran de forma jerárquica, de tal forma que nosotros podemos verificar la autenticidad de un certificado comprobando la firma de la autoridad que lo emitió, que a su vez tendrá otro certificado expedido por otra autoridad de rango superior. De esta forma vamos subiendo en la jerarquía hasta llegar al nivel más alto, que deberá estar ocupado por un certificador que goce de la confianza de toda la comunidad. Normalmente las claves públicas de los certificadores de mayor nivel se suelen publicar incluso en papel para que cualquiera pueda verificarlas.

El mecanismo que debe emplearse para conseguir un certificado X.509 es enviar nuestra clave pública —¡nunca la privada!— a la autoridad de certificación, después de habernos identificado positivamente frente a ella. Existen autoridades de certificación que, frente a una solicitud, generan un par llave pública-privada y lo envían al usuario. Hemos de hacer notar que en este caso, si bien tendremos un certificado válido, nuestro certificador podrá descifrar todos nuestros mensajes.

Capítulo 14

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 Phill Zimmerman. 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. 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 Zimmerman, en febrero de 2001, el cdigo fuente dejó de publicarse.

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 a codificar particiones enteras del disco duro (PGPDisk), pasando por la codificación automática y transparente de todo el tráfico TCP/IP (PGPnet).

14.1 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 fundmentales

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 Phill 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`).

14.2 Estructura de PGP

14.2.1 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 14.1) con una clave generada aleatoriamente (*clave de sesión*) y posteriormente codifica la clave haciendo uso de la llave pública del destinatario. Dicha clave es extraida 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 leido 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 mensaje. 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 criptográfi-

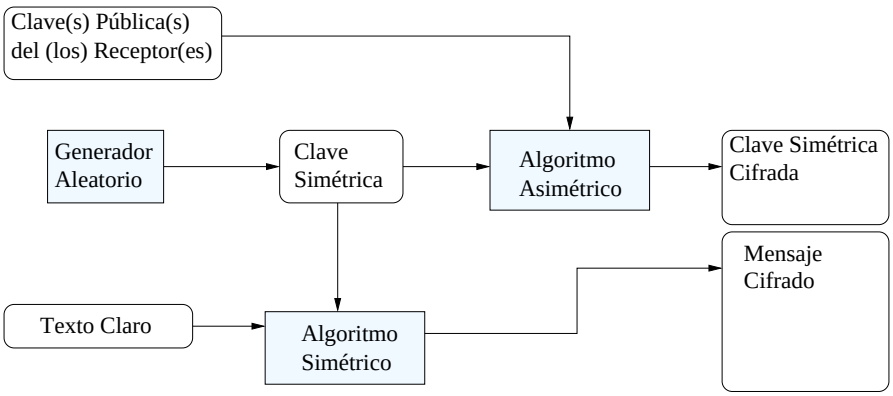

Figura 14.1: Codificación de un mensaje PGP

camente 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.

14.2.2 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 13.1.3), que posteriormente se codifica empleando la clave privada RSA correspondiente. Las versiones actuales implementan el algoritmo DSS, que emplea la función resumen SHA-1 y el algoritmo asimétrio DSA (secciones 12.3.4 y 13.1.4).

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.

14.2.3 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 extraidas 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.

[1] Algunas implementaciones de PGP emplean otras fuentes de aleatoriedad, como ocurre con GnuPG, por lo que no necesitan almacenar una semilla aleatoria.

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 incluyo mi clave pública PGP —firmada con la de Kriptópolis— en formato ASCII:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.0.4 (GNU/Linux)
Comment: For info see http://www.gnupg.org

mQGiBDRkk6kRBADKYHrNnFeXlggr14IVGy6FudLG2Cd1wb3yKOaNnodyjZa0a5oi
Ls9jDfDfEdq8K+W6QBLv06w7oVFPNMYsU+ufb0pa/bHWq6IrHxKkTVH4o4PUYTmH
W0jfGjoXEtAUZ0vp9wYR0Yqi7wXO3L/N5KuVNjLj7rXOT7rOmHsOjmY1cQCg//2w
OcyAnkaDCODFNif/VdowntcD/j5midszzU6M7BWmeDJoqEEGzSuxfmRSNyNZe6/6
5k8TFXIVpB0vnxwsZSh0POSlNgz1cmX6VbEmmUXoYsMRfq7iXHSAZ3DLB333yR2b
QUbkrH5WZF75G2vvTO7rKS5KtmROJ8E+vX/py6PGz1f3tBZJ94KwM787g6j43F4X
IYTAA/9L5GZzClHOGt01BtZkioH5YoHnDGHKC8mMXcykXA5KdJvl+9jGz3InUHiG
04StaMxMcDcWLzL5FVLz3LBzlOXGs7jikgH3BYBI3p7dIExfRADucDHyKL/CpIl5
zqHBI+5bxY3Tysu3UlA1UkQloJMsSInlkkjQhwihNYsj8Avr9LQmTWFudWVsIEx1
Y2VuYSBMb3BleiA8bWx1Y2VuYUB1amFlbi5lcz6IVgQTEQIAFgUCOHyzZAQLCgQD
AxUDAgMWAgECF4AACgkQSLJRYWmrV4TqngCgsDk/ysnBdpPwp/r2dL0LzcqO1J8A
nRxUUiS3SoVb3WfnaSQmdb6eaJ3qiEsEEBECAAsFAjTa4FoECwMBAgAKCRBIslFh
aatXhO9yAJ9vI1QWihIKMUa4g3S8t3EZZ9SXxgCaAjfnHx8Kayylm6XXjjsC6iJK
BmaIPwMFEDTa5h2buAet57tpPxEC8K4AoOTP5I1fJFN6KtZdmLtENKSRrKfxAJ4g
wl5R1MzpeTFiysWKab/PsU5GwohGBBARAgAGBQI3eQrfAAoJEPi4YmyN8qnzA1sA
niVQF6V/6gBVPq0Idt1Yrtuy4+aQAKDTuyVvfU1tRNy/U89FhzMmBVRL47QtTWFu
dWVsIEx1Y2VuYSBMb3BleiA8bWx1Y2VuYUBrcmlwdG9wb2xpcy5jb20+iFYEExEC
ABYFAjklG9wECwoEAwMVAwIDFgIBAheAAAoJEEiyUWFpq1eEMZEAnAx9TFz49xbM
dwTsGN0a+qvph5b3AJ4stMpniXd+hANnwygWwuOih3mIvLQsTWFudWVsIEx1Y2Vu
YSBMb3BleiA8ZXF1aXBvQGtyaXB0b3BvbGlzLmNvbT6IVwQTEQIAFwUCOhjzNQUL
BwoDBAMVAwIDFgIBAheAAAoJEEiyUWFpq1eEh9MAnAkTZVoNU1BhfjkIzXmJyWxq
+Ee/AKCw8dPJ7uxeoLe0pyy3ilXds7ocDrkCDQQ0ZJRfEAgAw/iGbTW9OaTyfV4R
NZdg1HRDGEyasZdEPCM9ihPkfvQyK44nH13OseaikIYoyoA/BFiWeTNcHvb/4KOu
CK2GnO/p/6ohFcAOK5anEygGrhUUttUw8kYZ0rUBFIJnurtDcxwawugbPFv3qA+s
n756q7XUxjnTtpou+lWyj6VkN/EvrZDf9E7ikPUqRuIsHzJ5PUwypWtXaKg2HfCl
```

```
KkZlYFqzdPDCssrXOfjZDx2q6GSek6Sgj5Ph3X4opoXIx6Cfmp4ELYmvdmnDu4oe
6A6l/XIQ8NNhj+GxdtOgTq8QKDRWl2f6M3pQgPnYzBHoDIqnr/ie8jK4seDezRPt
L1/TlQACAgf+JXw03Q1opLBAaO/WZlcs2SiEzqv+gCkFW9vk2bJbSY4PQHwiLc0H
wcPEDi7jIu9QxJfZcHkax8XgXkCvfFJFFmqgqarIOzXp/BgiYyma6GVAmXcI6lI9
ZSgzPvvaNFGe0/7R6Yroee7nJ/9RyxF89SI++5tZY+/bpLuKAbnX9SA3PEnUWiHD
2ah3cC3VXNrus3lsKA7MEh3q9xnoF/8Z7vwldrKUyLZdaDqSM7isyI5Fe0PWn/mt
W4+7/rjboaY7PGJCAqtn8cHDvByRYCZ8kLRlobQHzL8XN1fsdfBv6WDNeS9IqBCX
cPME7R21wytsi2WMDnYL7rQWU/CgLqFx2Ig/AwUYNGSUX0iyUWFpq1eEEQL3JACf
Tfvh6A70A9N2SbnRBmktuRBp9NsAn2ZQbpg0eaeVRuzejA2QM7ldrz53
=D9SU
-----END PGP PUBLIC KEY BLOCK-----
```

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 un OCR para introducirla en su anillo de claves correspondiente, aunque es mejor descargarla a través de Internet.

14.2.4 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 la secuencia binaria correspondiente para el algoritmo concreto donde se emplee, posee una serie de datos, como son el identificador 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 una secuencia hexadecimal lo suficientemente larga como para que sea única, y lo suficientemente corta como para que pueda ser escrita en un papel, o leida de viva voz. La huella digital se emplea para asegurar la autenticidad de una clave. Por ejemplo, la huella digital de la clave pública anterior es:

```
9E2B 9D14 CBCE FE12 16A8  C103 48B2 5161 69AB 5784
```

Si alguien quisiera asegurarse de la autenticidad de dicha clave, bastaría con que llamara por teléfono al autor, y le pidiera que le leyera la su huella digital.

14.2.5 Distribución de Claves y Redes de Confianza

PGP, como cualquier sistema basado en clave pública, es susceptible a ataques de intermediario (sección 12.2.2). 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.

14.2.6 *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 Zimmerman. 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 estrtuctura 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.

14.3 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 13.3 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 permitiría 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, aunque suponemos que obligará a una revisión del protocolo.

Parte V

Seguridad en Redes de Computadores

Capítulo 15

Seguridad en Redes

La rápida expansión y popularización de Internet ha convertido a la seguridad en redes en uno de los tópicos más importantes dentro de la Informática moderna. Con tal nivel de interconexión, los virus y los *hackers* campan a sus anchas, aprovechando las deficientes medidas de seguridad tomadas por administradores y usuarios a los que esta nueva revolución ha cogido por sorpresa.

Las ventajas de las redes en Informática son evidentes, pero muchas veces se minusvaloran ciertos riesgos, circunstancia que a menudo pone en peligro la seguridad de los sistemas. En unos pocos años la inmensa mayoría de las empresas operarán a través de la Red, y esto sólo será posible si los profesionales de la Informática saben aportar soluciones que garanticen la seguridad de la información.

15.1 Importancia de las Redes

La Informática es la ciencia del tratamiento automático de la información, pero tanto o más importante que su procesamiento y almacenamiento es la posibilidad de poder transmitirla de forma eficiente. La información tiene un tiempo de vida cada vez menor y la rapidez con la que pueda viajar es algo crucial. Los últimos avances en compresión y transmisión de datos digitales permiten hoy por hoy transferir cantidades enormes de información a velocidades que hace tan solo unos años eran impensables. En este sentido las redes de computadoras desempeñan un papel fundamental en la Informática moderna.

Pero hemos de tener en cuenta que la complejidad de las grandes redes y su carácter público convierte la protección física de los canales de comunicación en algo tremendamente difícil. Hemos de depositar nuestra confianza en la Criptografía para garantizar la confidencialidad en las comunicaciones.

Uno de los mayores obstáculos que han tenido que ser superados para que las redes pudieran desarrollarse, ha sido encontrar *lenguajes* comunes para que computadoras de diferentes

tipos pudieran *entenderse*. En este sentido el protocolo TCP/IP se ha erigido como estándar *de facto* en la industria de la Informática. En general todas las redes de computadoras se construyen conceptualmente sobre diferentes capas de abstracción, que desarrollan tareas distintas y proporcionan un protocolo unificado a las capas superiores. La Criptografía podrá entonces ser empleada en diferentes niveles de abstracción. Por ejemplo, podemos cifrar un fichero antes de transmitirlo por la red, lo cual correspondería al nivel de abstracción mayor, o podemos enviarlo *en claro*, pero a través de un protocolo de bajo nivel que cifre cada uno de los *paquetes* de información en los que se va a subdividir el fichero en el momento de transmitirlo.

En función del tipo de red con el que trabajemos nos enfrentaremos a diferentes clases de riesgos, lo cual nos conducirá inevitablemente a medidas de diferente naturaleza para garantizar la seguridad en las comunicaciones. En este capítulo haremos una breve reflexión sobre algunos de los casos que pueden darse, sin tratar de ser exhaustivos —sería imposible, dada la inmensa cantidad de posibilidades—. Nuestro objetivo se centrará en aportar una serie de directrices que nos permitan analizar cada situación y establecer una correcta política de protección de la información.

Ya que no existe una solución universal para proteger una red, en la mayoría de los casos la mejor estrategia suele consistir en tratar de *colarnos* nosotros mismos para poner de manifiesto y corregir posteriormente los *agujeros de seguridad* que siempre encontraremos. Esta estrategia se emplea cada vez con mayor frecuencia, y en algunos casos hasta se contrata a *hackers* para que impartan cursillos de seguridad a los responsables de las redes de las empresas.

15.2 Redes Internas

El caso más sencillo de red que nos podemos encontrar corresponde al término LAN[1], de ámbito muy limitado geográficamente —usualmente un único edificio— con todos los computadores interconectados a través de unos cables de los que se es propietario. Esta última circunstancia nos va a permitir ejercer un control total sobre el canal de comunicaciones, pudiendo protegerlo físicamente, lo cual evita prácticamente cualquier peligro de falta de privacidad en la información.

Uno de los riesgos dignos de mención en estos casos son las posibles pérdidas de información debidas a fallos físicos, que pueden ser minimizados llevando a cabo una adecuada política de *copias de respaldo*, que deberán ser confeccionadas periódicamente, almacenadas en un lugar diferente de aquel donde se encuentra la red, y protegidas adecuadamente contra incendios y accesos no deseados.

Otro riesgo que se da en las redes locales, a menudo infravalorado, es el que viene del uso inadecuado del sistema por parte de los propios usuarios. Ya sea por mala fe o descuido, un usuario con demasiados privilegios puede destruir información, por lo que estos permisos deben ser asignados con mucho cuidado por parte de los administradores. Esta circunstancia es muy importante, ya que, sobre todo en pequeñas empresas, el dueño muchas veces cree que

[1] *Local Area Network*, red de área local.

debe conocer la clave del administrador, y luego es incapaz de resistir la tentación de *jugar con ella*, poniendo en serio peligro la integridad del sistema y entorpeciendo el trabajo del administrador o *superusuario*.

Existen redes internas en las que un control exhaustivo sobre el medio físico de transmisión de datos es en la práctica imposible. Piénsese en un edificio corporativo con un acceso no muy restringido, por ejemplo un aulario de una universidad, que posee conexiones *ethernet* en todas sus dependencias. En principio, nada impediría a una persona conectar un ordenador portátil a una de esas conexiones para llevar a cabo un análisis del tráfico de la red sin ser descubierta, o *suplantar* a cualquier otro computador. En estos casos será conveniente llevar a cabo algún tipo de control, como la deshabilitación dinámica de las conexiones de red no utilizadas en cada momento, la verificación del identificador único de la tarjeta de red concreta que debe estar conectada en cada punto, o la adopción de protocolos de autentificación de las computadoras dentro de la red, como por ejemplo *Kerberos*[2].

Uno de los últimos avances en redes locales son las denominadas WLAN[3], muy en boga en la actualidad, que basan su funcionamiento en el empleo de ondas de radio como canal de comunicaciones, proporcionando una flexibilidad sin precedentes a las redes. Sin embargo, los protocolos propuestos hasta la fecha (IEEE 802.11 y 802.11b) porporcionan un nivel de seguridad realmente deficiente, que podría permitir a un atacante analizar el tráfico con relativa facilidad, por lo que se recomienda emplear protocolos de cifrado en las capas de mayor nivel.

15.3 Redes Externas

Consideraremos red externa a aquella que, en todo o en parte, se apoye en un canal físico de comunicación ajeno. En la actualidad, la mayor parte de las redes externas e internas están interconectadas, formando lo que conocemos como Internet. Existirán redes externas de muy diferentes tipos, pero todas ellas tienen en común la característica de que que en algún momento la información viaja por canales sobre los que no se tiene ningún tipo de control. Todas las técnicas que nos van a permitir llevar a cabo protecciones efectivas de los datos deberán hacer uso necesariamente de la Criptografía.

Para identificar los posibles riesgos que presenta una red externa, hemos de fijarnos en cuestiones tan dispares como el sistema operativo que corre sobre los ordenadores o el tipo de acceso que los usuarios *legales* del sistema pueden llevar a cabo.

Una de las configuraciones más comunes consiste en el uso de una red local conectada al exterior mediante un *cortafuegos* —computadora que filtra el tráfico entre la red interna y el exterior—. Los cortafuegos son herramientas muy poderosas si se emplean adecuadamente, pero pueden entrañar ciertos riesgos si se usan mal. Por ejemplo, existen muchos lugares donde el cortafuegos está conectado a la red local y ésta a su vez a la red externa (ver figura 15.1, caso

[2] *Kerberos* es uno de los protocolos de autentificación que forma parte del proyecto *Athena*, en el MIT (Instituto Tecnológico de Massachusetts), y permite a un computador identificarse positivamente dentro de una red.

[3] *Wireless Local Area Network*, red de área local sin hilos.

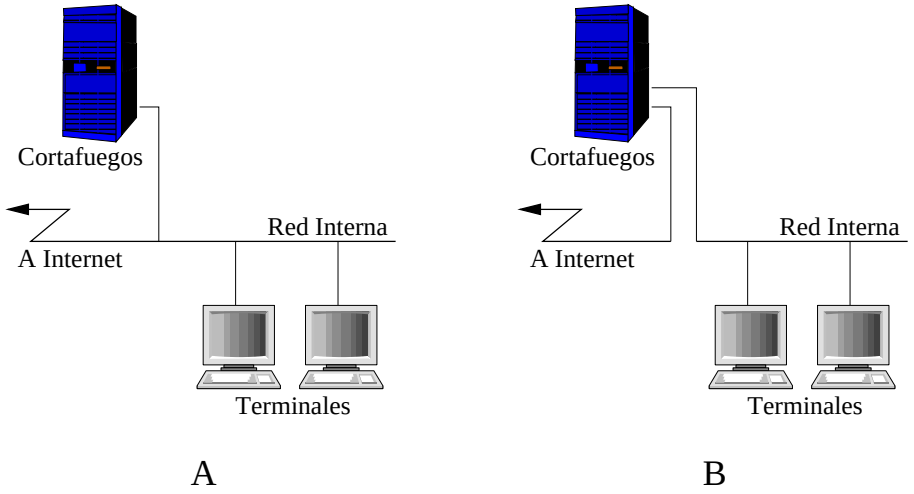

Figura 15.1: **A**: Configuración incorrecta, el cortafuegos tiene una única tarjeta de red, y los terminales están conectados físicamente a la red externa. **B**: Configuración correcta, el cortafuegos dispone de dos tarjetas de red y el resto de las computadoras está aislado físicamente de la red externa.

A). Esta configuración es la más sencilla y barata, puesto que sólo necesitamos una tarjeta de red en el cortafuegos, pero no impediría a un computador situado en el exterior acceder directamente a los de la red local. La configuración correcta se puede apreciar en el caso B de la figura 15.1, donde la red externa (y todos sus peligros) está separada físcamente de la red local.

Podemos distinguir dos grandes tipos de peligros potenciales que pueden comprometer nuestra información desde una red externa:

- Ataques indiscriminados. Suelen ser los más frecuentes, y también los menos dañinos. Dentro de esta categoría podemos incluir los troyanos y los virus, programas diseñados normalmente para *colarse* en cualquier sistema y producir efectos de lo más variopinto. Precisamente por su carácter general, existen programas específicos que nos protegen de ellos, como los antivirus. Conviene disponer de un buen antivirus y actualizarlo periódicamente.

- Ataques *a medida*. Mucho menos comunes que los anteriores, y también más peligrosos, son los ataques que generalmente llevan a cabo los *hackers*. En estos casos las víctimas son casi siempre grandes corporaciones, y muchas veces la información ni siquiera es destruida o comprometida, puesto que los *hackers* sólo persiguen enfrentarse al reto que supone para ellos entrar en un sistema grande. El problema es que para borrar sus huellas y dificultar el rastreo de sus acciones, suelen atacar en primer lugar sistemas pequeños para desde ellos cometer sus *travesuras*, lo cual convierte a cualquier sistema en potencial víctima de estos personajes. Lo que ocurre en la mayoría de los casos es que su necesidad

de emplear sistemas pequeños como plataforma les obliga a no dañarlos, para no dejar ningún tipo de rastro que permita localizarlos posteriormente.

En cuanto a la protección de las comunicaciones en sí, baste decir que existen protocolos de comunicación segura de *bajo nivel*, como SSL y TLS (ver sección 12.5), que permiten establecer comunicaciones seguras a través de Internet, haciendo uso de algoritmos simétricos y asimétricos simultáneamente. Estos protocolos son transparentes y pueden correr bajo otros ampliamente conocidos, como POP3, TELNET, FTP, HTTP, etc. De hecho, gran cantidad de aplicaciones los emplean en sus comunicaciones. Desgraciadamente, las restrictivas leyes norteamericanas en cuanto a la exportación de material criptográfico hacen que la gran mayoría de las aplicaciones *seguras* que se venden fuera de los EE.UU. y Canadá estén en realidad debilitadas, por lo que hemos de informarnos muy bien antes de depositar nuestra confianza en ellas.

15.3.1 Intranets

El término *intranet* se ha popularizado recientemente y hace alusión a redes externas que se comportan de cara a los usuarios como redes privadas internas. Obviamente, este tipo de redes ha de ser implementado haciendo uso de protocolos criptográficos de autentificación y codificación de las transmisiones, puesto que el tráfico que nosotros vemos como *interno* a nuestra red, en realidad viaja por Internet.

15.4 Conclusiones

Después de todo lo dicho parece una locura conectarse a una red externa, y ciertamente lo es si no se toman las precauciones adecuadas. La cantidad de posibles riesgos es enorme, y con toda seguridad en el futuro aparecerán nuevos peligros, pero no olvidemos que ante todo debemos ser racionales. Si bien puede ocurrir que un equipo de *hackers* trate de entrar en nuestro sistema, esta posibilidad suele ser remota en la mayoría de los casos, debido precisamente al escaso interés que va a despertar en ellos penetrar en una red pequeña. Por otro lado hay que tener en cuenta que cierto tipo de ataques requiere fuertes inversiones, por lo que si nuestra información no resulta realmente valiosa para el atacante, podemos considerarnos a salvo. No olvidemos que el coste de la protección en ningún caso puede superar el valor de la propia información que se desea proteger. Por lo demás, parece claro que las ventajas que nos proporcionará estar en la Red son claramente mayores que los inconvenientes, pero nunca se debe bajar la guardia.

En general, conviene estar preparado para el peor de los casos probables, que suele ser la pérdida de la información, casi siempre debida a fallos físicos o a la presencia de virus. En cuanto al resto de posibilidades, será suficiente con la adopción de protocolos seguros, además de llevar un registro de todas las operaciones que tienen lugar dentro del sistema, registro que deberá ser controlado periódicamente para detectar posibles anomalías. Otra práctica

bastante recomendable consiste en mantenerse al día sobre los fallos de seguridad detectados en los programas y sistemas operativos que empleemos, así como de los sucesivos *parches* que las empresas de software suelen distribuir periódicamente, con objeto de eliminar los agujeros de seguridad.

Capítulo 16

Hackers

Se consideran a sí mismos una casta, y su filosofía de la vida es casi una religión. Delincuentes para unos, héroes para otros, multitud de leyendas circulan sobre estos personajes. Al igual que en otras épocas había alquimistas, capaces de dominar los oscuros poderes de la materia, hoy los *hackers* están considerados por muchos como los nigromantes de la era tecnológica. Nos guste o no, los *hackers* son ya un mito en la cultura de finales del siglo XX.

No debemos confundir el término *hacker* con el de pirata informático, ya que este último es un concepto más amplio. Aquellos que conozcan la cultura *cyberpunk*, sabrán que además de *hackers* hay otros grupos, tales como los *crackers*, que se dedican a la copia ilegal de software, y los *phreakers*, que dirigen sus esfuerzos hacia las compañías telefónicas. Cada uno de ellos se especializa en algún tipo de actividad —curiosamente, las actuaciones de los *crackers* y *phreakers* suelen ser siempre delictivas, mientras que las de los *hackers* en algunos casos no lo son, mal que les pese a muchos—. Por supuesto, todos ellos justifican sus formas de pensar y actuar con argumentos de lo más variopinto, que tienen como punto común la lucha contra el *sistema establecido*.

Periódicamente los medios de comunicación nos sorprenden con alguna nueva *hazaña* de estos personajes, contribuyendo, junto con la industria cinematográfica, al crecimiento y propagación de su leyenda.

16.1 El Hielo y los Vaqueros

Un *hacker* es un individuo que se dedica a infiltrarse en sistemas informáticos. Su actividad, tan antigua como las redes de ordenadores, conoce diversas variantes. Desde aquellos que no tratan de hacer ningún daño, y que consideran estas actuaciones como un excitante reto a su inteligencia, hasta aquellos cuyo único objetivo es sabotear una red, llevándose toda la información que posea para luego venderla, podemos decir que hay *hackers* para todos los gustos.

En 1983, William Gibson escribió *"Neuromante"*, pieza clave de la literatura de Ciencia Ficción moderna y referencia obligada cuando se habla de cultura *cyberpunk*. En esta obra los vaqueros —término empleado por Gibson para referirse a los *hackers*— burlaban el *Hielo* de los sistemas informáticos —del inglés ICE, *Intrusion Countermeasures Electronics*— a través del ciberespacio, —también denominado matriz en el libro, o si se prefiere, *matrix*—.

Huelga decir que la magia y lo esotérico nada tienen que ver con estos sujetos. En general son individuos bastante ingeniosos y bien informados que se dedican a buscar y explotar fallos más o menos sutiles en los sistemas de seguridad. Puesto que cada sistema se puede decir que es único, los buenos vaqueros suelen elaborar ataques *a medida*, poniendo a prueba su profundo conocimiento sobre las redes de ordenadores. Es prácticamente imposible protegerse al cien por cien de un ataque de esta naturaleza, y debemos pensar que si, por alguna razón, nos convertimos en objetivo de un *hacker* lo suficientemente bueno, tarde o temprano acabaremos cayendo.

También hay que decir que aunque muchos actúan solos, los vaqueros suelen formar grupos, en los que cada uno tiene su alias, y que normalmente sólo establecen contacto a través de la Red, no conociéndose personalmente entre ellos. Suelen dominar bastante bien el uso de la Criptografía, y frecuentemente la emplean en sus comunicaciones. Alguno podría pensar que este es un buen argumento para imponer un control gubernamental sobre el uso privado de la Criptografía, pero en ese caso también deberíamos permitir que la policía entre en nuestras casas, lea nuestras cartas y escuche nuestras conversaciones telefónicas sin nuestro consentimiento, para evitar que cometamos delitos.

Pero no todo va a ser negativo. La gente que instala un dispositivo antirrobo en su casa sabe que puede ser burlado por un ladrón altamente especializado, pero aún así lo considera seguro, ya que un ladrón profesional asumirá el riesgo sólo si la casa despierta el suficiente interés. Así pues, basta con tomar unas medidas de seguridad proporcionales al valor que posea el sistema que queremos proteger. El problema es que en muchos casos, por simple desconocimiento, los sistemas están mal protegidos, hasta tal punto que es como si dejáramos abierta la puerta de nuestra casa. En esos casos, cualquiera puede entrar a fisgonear.

Por desgracia, las computadoras son tan heterogéneas que no existe un conjunto de medidas universal que nos permita protegernos de estos chicos traviesos. Intentaremos no obstante dar unas pautas sobre las técnicas que más emplean para que así cada cual pueda saber qué medidas debe tomar en su caso concreto.

16.2 Cómo actúa un *Hacker*

Comentaremos en esta sección muy brevemente algunas de las técnicas más comunes empleadas para infiltrarse en computadores ajenos a través de la Red. Esto no quiere decir que sean las únicas técnicas posibles, ni siquiera que sean las mejores, pero servirán para hacernos una idea del modo de actuar de estos individuos.

Puerto	Función
21	FTP
23	Telnet
25	SMTP (Mail)
37	Time
43	Whois
80	HTTP (Servidor Web)
110	POP3 (Mail)
117	UUCP
119	NNTP (News)
513	Login
514	Shell
515	Spooler

Tabla 16.1: Algunos puertos TCP/IP.

16.2.1 Protocolo TCP/IP. Demonios y Puertos

TCP/IP es el protocolo que se ha impuesto como norma universal *de facto* en las comunicaciones. Internet se basa en dicho protocolo, y aunque existan otros para redes locales, los sistemas operativos actuales permiten su coexistencia, por lo que podemos decir sin temor a equivocarnos que prácticamente cualquier máquina conectada a Internet *entiende* TCP/IP.

Un computador con TCP/IP puede establecer múltiples comunicaciones simultáneamente, a través de los denominados *puertos*. Un puerto se comporta como los canales de un televisor: a través de un único cable llegan muchas emisiones, de las cuales podemos escoger cuál ver con solo seleccionar el canal correspondiente.

Existen puertos dedicados a tareas concretas. Así por ejemplo el puerto 80 se emplea para las páginas *web*, y el 21 para la transferencia de ficheros. En la tabla 16.1 podemos ver algunos de los más usuales, aunque existen muchos más. Hay que decir que esta tabla es orientativa: nada nos impediría situar nuestro demonio de FTP en el puerto 300, por ejemplo, aunque eso obligaría a quienes quisieran establecer una comunicación FTP con nosotros a emplear dicho puerto. De hecho, ciertos servidores de acceso restringido emplean puertos no normalizados para evitar visitantes molestos.

Un *demonio* (*daemon*[1], en inglés) es un programa que escucha a través de un puerto a la espera de establecer comunicaciones. Así, por ejemplo, un servidor de páginas *web* tiene un demonio asociado al puerto 80, esperando solicitudes de conexión. Cuando nosotros cargamos una página en el navegador estamos enviando una solicitud al puerto 80 del servidor, que responde con la página correspondiente. Si el servidor *web* no estuviera ejecutando el demonio o éste estuviera escuchando en otro puerto, no podríamos consultar la página que buscamos.

[1]En realidad, el concepto de *daemon* es mucho más amplio, y se refiere a cualquier programa que, cuando se ejecuta, entra en un estado de espera hasta que algún suceso lo activa.

Una vez que se establece la comunicación en un puerto, los ordenadores *hablan* entre ellos, usando diferentes *idiomas*, como por ejemplo HTTP para las páginas *web*, FTP para las transferencias de ficheros, etc.

Ahora vamos a ver un ejemplo utilizando la orden `telnet` de *UNIX* y el protocolo SMTP de envío de correos electrónicos, que se ubica en el puerto 25:

```
usuario> telnet 1.2.3.4 25
Trying 1.2.3.4...
Connected to 1.2.3.4.
Escape character is '^]'.
220 host.dominio.pais ESMTP Sendmail 8.9.3; Fri, 10 Sep 1999 16:16:55
```

A partir de este momento el ordenador con IP[2] `1.2.3.4` (*host.dominio.pais*) está esperando nuestros mensajes a través de su puerto 25. Si escribimos

```
helo mlucena
```

el demonio responderá algo parecido a

```
250 host.dominio.pais Hello mlucena@host [3.2.5.6], pleased to meet you
```

Puesto que el demonio es un programa, puede que contenga errores, o que simplemente haya situaciones en las que no funcione adecuadamente. En los sistemas tipo *UNIX*, uno puede enviar mensajes extraños para los que el demonio no se encuentra preparado, y así lograr que aborte su ejecución, dejándonos una consola de texto con la que podremos tomar el control del sistema. En general, los fabricantes de software suelen actualizar periódicamente sus programas, subsanando paulatinamente los errores que éstos puedan contener, por lo que es de una importancia crucial que los demonios instalados en nuestro sistema sean fiables y se encuentren al día. Un fallo muy famoso y que dio mucho que hablar estaba en el demonio que escuchaba las comunicaciones SMB en las primeras versiones de Windows 95. Si uno enviaba un mensaje concreto a ese puerto podía bloquear de forma instantánea el ordenador de su víctima, y para ello sólo tenía que conocer su número de IP.

En general, el *hacker* se dedica a tratar de averiguar en qué puertos está escuchando el ordenador objetivo, y luego a localizar y explotar posibles fallos en los demonios correspondientes, para tomar el control del sistema. Muchas veces nuestro ordenador puede que esté escuchando algún puerto sin que nosotros lo sepamos. Existe un troyano (ver capítulo 17) que corre sobre los sistemas *Windows*, denominado *Back Orifice*, que escucha un puerto a la espera de que el ordenador atacante tome el control de nuestra máquina.

Por desgracia, existen programas cuya configuración por defecto no es lo suficientemente conservadora, y que habilitan ciertas características a no ser que se les diga lo contrario,

[2] El IP es un conjunto de cuatro números, separados por puntos, que identifica unívocamente a un ordenador conectado a Internet.

abriendo inevitablemente agujeros de seguridad. A modo de ejemplo, citaremos el problema que surgió en el verano de 1998, cuando se descubrió que era posible descargar, si el servidor corría bajo Windows NT, el código fuente de algunas páginas web de tipo ASP de una manera muy simple, a no ser que el administrador tomara ciertas precauciones. Afortunadamente, el problema se solucionó con rapidez, y gracias a estos pequeños sustos cada vez se pone más cuidado a la hora de elaborar software *sensible*.

16.2.2 Desbordamientos de *Buffer*

Uno de los mecanismos más empleados para hacer *saltar* a un demonio es el *desbordamiento de buffer*, que aprovecha una característica muy común en casi todos los compiladores de lenguajes de programación modernos. Cuando se ejecuta un procedimiento o subprograma, el ordenador reserva memoria para los datos de entrada al procedimiento, junto con información relativa a la dirección de memoria por la que debe continuar la ejecución una vez finalizado. En algunos casos, los datos de entrada pueden ser de mayor tamaño que el espacio que el programador estimó suficiente en su día para albergarlos, por lo que, al salirse de su lugar, pueden sobreescribir y modificar otros campos. Un atacante podría entonces pasar al demonio una cadena que incluyera código ejecutable, y que *machacara* el campo que indica dónde debe seguir la ejecución del programa, para que apuntara precisamente a su código malicioso. De esta forma se podría ejecutar virtualmente cualquier cosa en la máquina de la víctima. Evidentemente, la construcción de cadenas de este tipo no es una tarea sencilla, pero una vez elaboradas, podríamos emplearlas cuantas veces quisiéramos, y desde cualquier otro ordenador.

16.2.3 Suplantando Usuarios

Lo ideal para entrar en un sistema es hacerlo como administrador, lo cual proporciona suficientes privilegios como para alterar cualquier cosa sin ningún problema. A veces ocurre que el ordenador *víctima* no presenta vulnerabilidades en los puertos que escucha, por lo que debemos buscar otros medios para entrar en él.

La mayoría de los sistemas operativos permiten la existencia de usuarios genéricos, llamados *invitados*, que no necesitan contraseña para entrar en el sistema y que tienen unos privilegios de acceso bastante limitados. En muchos casos esos privilegios pueden llegar a ser suficientes como para perpetrar un ataque con garantías de éxito, debido a que un invitado puede acceder al fichero que almacena las contraseñas. Afortunadamente, esta circunstancia ha sido subsanada en casi todos los sistemas operativos, pero ha de ser tenida en cuenta, para evitar desagradables sorpresas.

Como ya indicamos en la sección 13.3.1, si un usuario posee el fichero de contraseñas, puede llevar a cabo un ataque con diccionario, y eventualmente llegar a averiguar las claves de cada usuario, lo cual le permitirá entrar en el sistema. Este tipo de ataque se volverá inútil si los usuarios escogen contraseñas adecuadas, del tipo que se propone en esta obra.

Una posibilidad bastante inquietante es la de, una vez que se han ganado suficientes privi-

legios, sustituir el fichero de contraseñas por otro elaborado por el *hacker*, lo cual dejaría sin acceso a todos los usuarios legítimos del sistema, ¡incluidos los administradores!. En tal caso habría que desconectar el sistema de la red y restaurarlo manualmente, con el consiguiente coste tanto de tiempo como de dinero.

16.2.4 Borrando las Huellas

Todos los sistemas operativos serios incorporan algún sistema de registro de los eventos del sistema que permite saber con detalle lo que en la computadora ha ido ocurriendo. Un vaquero que se precie debe eliminar todas las entradas de dicho registro relativas a su paso por el sistema, si no quiere que un policía llame a su puerta a los pocos días...

Además de tratar de borrar todas sus huellas, un *hacker* suele organizar sus ataques de forma que si queda algún rastro de su paso por el sistema elegido, éste sea realmente confuso. Para ello nada mejor que emplear otros ordenadores más modestos como plataforma para atacar al auténtico objetivo. Normalmente los vaqueros buscan ordenadores poco protegidos, entran en ellos, y controlándolos remotamente intentan encontrar las debilidades del objetivo real. Esta estrategia hará que en los registros del sistema atacado aparezcan datos sospechosos acerca del ordenador intermedio, pero pocas veces del auténtico enemigo. Las autoridades tendrán que ponerse en contacto con el ordenador empleado como plataforma para buscar en él indicios del verdadero atacante.

16.2.5 Ataques *Pasivos*

Recientemente se han detectado fallos de seguridad en los navegadores de Internet que permitirían a un hipotético atacante colocar en su página web código *malicioso*. De esta forma todos los que visitaran esa página y no hubieran tomado las adecuadas precauciones se verían afectados con problemas tales como la ejecución de algún programa —posiblemente un virus—, o el envío de algún fichero propio al atacante, todo ello de forma inadvertida, por supuesto. El atacante sólo tiene que esperar a que los incautos vayan cayendo...

Desgraciadamente, cada versión de los navegadores más populares presenta nuevos problemas de este tipo, si bien suelen ser identificados —que no siempre resueltos— con bastante celeridad, en gran parte gracias al excelente trabajo de Juan Carlos García Cuartango. Esperemos que poco a poco esta tendencia se vaya corrigiendo.

16.2.6 Ataques Coordinados

Uno de los tipos de ataque que más fama ha cobrado últimamente es el que se conoce como Denegación de Servicio (en inglés *Denial of Service*, o DoS). Éste se aprovecha de que los servidores, cuando reciben una solicitud de conexión, reservan memoria para atenderla. Basta, pues, con efectuar miles de solicitudes incompletas para bloquear la memoria del servidor, e impedir que acepte más conexiones. En la práctica, este ataque no permite robar información,

sino únicamente paralizar un servidor, lo cual puede hacer suficiente daño en *sitios web* que ofrecen servicios en línea, como los dedicados al comercio electrónico.

En realidad, la única modalidad efectiva de ataque DoS es aquella que se lleva a cabo, simultáneamente y de forma distribuida, por muchos computadores. El *quid* de la cuestión radica en que prácticamente cualquier tipo de ataque puede ser automatizado mediante un programa adecuado, por lo que un *hacker* podría construir un troyano que incorporara el código necesario para efectuar el ataque, y limitarse a coordinarlo cuando lo estime oportuno. Vulgarmente, se denomina *zombie* a aquella computadora que contiene un código capaz de contribuir en un ataque distribuido, de forma inadvertida para sus usuarios.

Obsérvese que el crecimiento casi explosivo de Internet, impulsado en muchas ocasiones por políticas poco realistas de los gobiernos, está conduciendo a la existencia de un número cada vez mayor de computadoras con una deficiente administración, conectadas a la Red —en bibliotecas, colegios, etc.—, perfectas candidatas a protagonizar el próximo ataque distribuido, que posiblemente provocará pérdidas millonarias y hará correr ríos de tinta.

16.3 Cómo Protegerse del Ataque de los *Hackers*

Después de haber leído las anteriores secciones, parece una auténtica locura tener ordenadores conectados a Internet. Nada más lejos de la realidad. Así como hay muchas formas de poder entrar fraudulentamente en un ordenador, también hay muchas formas de protegerse razonablemente contra estos ataques. Vamos a dar una serie de consejos prácticos que, si no nos protegen totalmente, ponen las cosas bastante difíciles a los *hackers*. No obstante, lo mejor es conocer bien nuestro propio sistema para poder adaptar estas medidas a nuestro caso concreto.

- *Sólo la Información Necesaria*. No almacene información *sensible* en su ordenador si ésta no necesita ser consultada desde el exterior. ¿Por qué colocar un premio extra para los *hackers*?

- *Instalación de Demonios*. Cuando instale cualquier software que incluya algún demonio, asegúrese de que se trata de la versión más reciente y actualizada, que debería ser la más segura. Desconfíe de las versiones *beta*, a no ser que sepa muy bien lo que hace.

  Configure sus servidores de la forma más conservadora posible. No habilite usuarios genéricos sin antes asegurarse de que no poseen excesivos privilegios. Si tiene alguna duda sobre alguna funcionalidad del servidor en cuestión, deshabilítela.

  Consulte periódicamente las páginas de los fabricantes de software y aquellas especializadas en alertar sobre fallos de seguridad, ellas le informarán de los agujeros más recientes y de cómo eliminarlos. Muchos *hackers* también las consultan, pero con otros propósitos.

  Ejecute periódicamente alguna utilidad que recorra los puertos de su sistema para saber en cuáles hay demonios. Esto le permitirá detectar programas del tipo *Back Orifice*.

- *Vigile su Software Criptográfico.* Emplee siempre que pueda SSL o TLS en sus comunicaciones, y asegúrese de que todos los programas de cifrado que usa funcionan con claves de al menos 128 bits.

- *Contra los Ataques por Diccionario.* Muchos sistemas operativos impiden que un administrador abra una consola remota, por lo que aunque alguien averigüe su contraseña, no podrá emplearla a no ser que disponga de acceso físico a la computadora. Si su ordenador está conectado a Internet, use esta característica, a no ser que necesite poder abrir consolas remotas como administrador.

  Asegúrese de que el fichero de contraseñas está protegido frente a accesos externos. Afortunadamente, casi todos los sistemas operativos modernos incorporan esta característica por defecto.

  Cambie periódicamente las contraseñas, y sobre todo, use *buenas* contraseñas. Existen utilidades para realizar ataques de diccionario (por ejemplo, la famosa *John The Ripper*) que nos permitirán saber si nuestras claves son satisfactorias.

- *Los Archivos de Registro.* Serán nuestra mejor defensa contra los *hackers*. Hay que consultarlos frecuentemente para detectar entradas *sospechosas*, y nunca bajar la guardia. Tampoco viene mal efectuar copias de seguridad en medios externos al ordenador, como pueden ser diskettes o cintas magnéticas. De esta forma el vaquero no podrá borrar totalmente sus huellas.

16.4 Conclusiones

Cuando hacemos un viaje solemos tomar ciertas precauciones, que reducen el riesgo de tener un accidente, o al menos el posible daño en caso de sufrirlo. Nadie que esté lo suficientemente equilibrado mentalmente y que conozca los medios de transporte emprende un viaje pensando que va a sufrir un accidente. Estos razonamientos son totalmente válidos cuando nos enfrentamos a los *hackers*. No hay que alarmarse, pero tampoco hemos de bajar la guardia.

Todos hemos oído historias acerca de personas que se infiltran en redes, que cambian la trayectoria de satélites de comunicaciones, o que venden secretos militares a países enemigos. Todo esto es muy espectacular, y en algunos casos puede que hasta cierto. Pero si razonamos un poco nos daremos cuenta de que los mejores *hackers* no han sido descubiertos, bien por razones de publicidad —pensemos en un banco al que le roban varios cientos de millones de pesetas—, bien porque ha resultado imposible localizarlos.

Capítulo 17

Virus

Quizá uno de los temas más famosos y sobre los que más mitos corren en el ámbito de la Informática sean los virus. Programas malignos que son capaces de parasitar un sistema, reproduciéndose y devorándolo por dentro, la imagen que la gente tiene acerca de los virus está deformada por el desconocimiento y las modernas leyendas urbanas.

Los virus estaban entrando ya en lo que parecía su definitivo declive cuando el auge de Internet provocó su relanzamiento, y la aparición de nuevas y más peligrosas formas de contagio. Hoy por hoy constituyen uno de los problemas de seguridad que más dinero e información hacen perder a los usuarios, por lo que dedicaremos un breve capítulo a estudiarlos.

17.1 Origen de los Virus

De origen incierto, los virus existen prácticamente desde los inicios de la Informática a gran escala. Desde aquellos programas *gusano*, capaces de copiarse a sí mismos hasta colapsar un sistema, hasta casos que han dado recientemente la vuelta al mundo, como *Melissa* y *I Love You*, que trajeron de cabeza a una gran cantidad de usuarios, el aumento de complejidad en los sistemas ha llevado aparejada nuevas e inquietantes formas de comprometer la seguridad de las computadoras.

Muchos dicen que los virus nacieron como una medida de las compañías de desarrollo de *software* para disuadir a la gente del uso de copias piratas de sus programas. Aunque este extremo no ha sido demostrado ni tampoco desmentido, muchos sugieren que algunos virus eran inoculados en las versiones legales de algunos programas, configurados para activarse cuando se intentara llevar a cabo una copia fraudulenta. El tiempo ha demostrado que los verdaderos perjudicados son las mismas compañías y los propios usuarios.

Lo cierto es que hoy por hoy existen miles de virus, y que cada día surgen más. Al igual que unos son creados con el único y dudoso ánimo de provocar una sonrisa en los afectados, no es menos cierto que otros se desarrollan con fines auténticamente destructivos, casi podría

decirse que terroristas.

17.2 Anatomía de un Virus

Pero, ¿qué es un virus? Denominaremos así a cualquier programa capaz de infiltrarse en un sistema y ejecutarse sin que el usuario tenga noticia de ello. Normalmente sacan copias de sí mismos de forma más o menos indiscriminada, con la intención de reproducirse, aunque esto no ocurre siempre. En realidad son dos las *habilidades* propias de un virus: la capacidad de propagarse, y sus efectos destructivos. Si únicamente presenta la primera de ellas, careciendo de la segunda, se suele denominar *gusano*. Aquellos programas que sólamente son destructivos, y que no se *contagian*, se denominan genéricamente *bombas lógicas*. El caso de los *troyanos*, como veremos más adelante, es ligeramente distinto, ya que si bien no pueden reproducirse por sus propios medios, sí que nacen con la intención de ser propagados.

El término *virus* se tomó prestado a los biólogos porque responde bastante bien a la filosofía de estos programas. Un virus biológico se infiltra en una célula y le inocula su código genético para, aprovechando el sistema de reproducción de su involuntaria anfitriona, duplicarse tantas veces como le sea posible. La célula muere en el proceso. Un virus informático se instala en una computadora y se ejecuta de manera inadvertida para el usuario. Cuando esa ejecución tiene lugar, primero se efectúan las copias y después el virus daña el sistema.

17.2.1 Métodos de Contagio

Al principio, cuando las redes informáticas eran pocas y estaban relativamente aisladas, los mecanismos de propagación de los virus se basaban en modificar los programas ejecutables, añadiéndoles el código del propio virus. El usuario apenas se daba cuenta de que sus ficheros ejecutables crecían ligeramente, y cuando los copiara en diskettes y los llevara a otros ordenadores, el contagio estaría asegurado. En otros casos se modificaban los denominados *sectores de arranque*, que son las zonas de los discos duros y diskettes que el ordenador carga en memoria y ejecuta en el momento de ser puesto en marcha. Este mecanismo, pensado originalmente para que una computadora pueda cargar el sistema operativo, resultaba más que atractivo como medio de transporte de los virus. Esto explica la insistencia de los expertos en que no se arranque el ordenador con diskettes dudosos en su interior, o en que se arranque con diskettes *limpios* ante cualquier sospecha sobre el estado de salud del disco duro.

Hoy las cosas han cambiado. Los sistemas operativos han crecido en complejidad, y programas tan inocentes como los paquetes de oficina —que agrupan procesador de textos, hoja de cálculo, gestor de correo electrónico, etc.— incorporan completísimos lenguajes de programación que nos permiten automatizar cualquier tarea. Esta característica, pese a ser extremadamente poderosa, puede ser terreno abonado para un virus. En marzo de 1999, el virus *Melissa*, que no es ni más ni menos que una *macro* de Microsoft Word —lo cual lo convierte, al menos en teoría, en un virus *multiplataforma*[1]—, colapsó las redes corporativas de varias empresas.

[1]Un programa se dice multiplataforma si se puede ejecutar en diferentes sistemas operativos. Así, por

Melissa venía camuflado en un inocente archivo de texto, que al ser abierto por el usuario, leía las primeras cincuenta entradas de la libreta de direcciones del gestor de correo electrónico, y se reenviaba por dicho medio a otras tantas nuevas víctimas. Como era de suponer, este *gusano* provocó la caída de varios servidores de correo electrónico en cuestión de horas. Lo más preocupante de todo esto es que la situación, lejos de ser corregida, volvió a repetirse apenas un año después con en archiconocido *I Love You*, de características similares.

Pero no todos los medios de contagio tienen por qué ser tan sofisticados. Algunos programas funcionan exactamente igual que el Caballo de Troya, y precisamente por eso se les denomina *troyanos*. Un *troyano* no tiene un mecanismo de contagio propiamente dicho, sino que dispone de un envoltorio más o menos atractivo —una felicitación navideña, una imagen simpática, un chiste, etc.— para que el usuario desprevenido lo ejecute. Un ejemplo bastante sorprendente de lo peligroso que puede ser un troyano se dio en 1999 cuando unos *hackers*, tras haberse introducido en un servidor bastante conocido que suministraba utilidades de monitorización de red, sustituyeron uno de los programas más usados por los admninistradores por otro que enviaba información confidencial acerca del sistema por correo electrónico.

17.2.2 La Fase Destructiva de un Virus

La mayoría de los virus posee una denominada *fase de letargo*, en la que permanece inactivo, o a lo sumo se dedica únicamente a reproducirse. Diferentes eventos pueden provocar que un virus despierte, como puede ser una fecha —los famosos virus *Viernes 13* y *Chernobyl* son un claro ejemplo—, un número determinado de ejecuciones, etc.

Cuando un virus entra en su fase *destructiva* puede ocurrir cualquier cosa. Desde el simple bloqueo del sistema, con algún mensaje en la pantalla, hasta la destrucción total y absoluta del contenido del disco duro del sistema, hay casi tantos posibles comportamientos como virus.

El tremendo auge que está teniendo Internet en los últimos años está propiciando la aparición de virus que permiten a un usuario remoto tomar literalmente el control del sistema, haciendo creer a la víctima que su ordenador está poco menos que poseído por algún fantasma. Este comportamiento lo podemos ver en el tristemente famoso *Back Orifice*. Otros comportemientos no menos desagradables pueden ser el envío de información privada a través de la red, y prácticamente cualquier cosa que una mente calenturienta pueda concebir.

17.3 Cuándo son Peligrosos los Virus

Hay personas que creen que con sólo acercar un diskette contaminado a un ordenador *sano*, éste último puede quedar contagiado, o que basta con leer un correo electrónico para que un virus se extienda inexorablemente por nuestro sistema aniquilándolo todo a su paso. Esto es rotundamente falso; un virus no es más que un programa, y como tal ha de ser ejecutado para

ejemplo, el lenguaje de programación Java está diseñado para que un mismo programa pueda correr en casi cualquier ordenador, independientemente del sistema operativo que posea.

que entre en acción.

En cuanto al mito de los correos electrónicos, diremos que leer un correo de texto es, como cabría esperar, del todo punto inofensivo. Ahora bien, si ese correo lleva incluido algún archivo ejecutable, o algún archivo capaz de llevar código ejecutable —un fichero de procesador de texto, o de hoja de cálculo, por ejemplo—, puede que el ordenador haga correr dicho código sin avisar al usuario. Este problema ha sido detectado recientemente en ciertos programas y constituye un agujero de seguridad ciertamente intolerable, para el que esperamos haya pronto una solución satisfactoria.

Pero el problema real de todo este asunto lo constituye el hecho de que los sistemas operativos que usa casi todo el mundo otorgan control total sobre la computadora a cualquier programa que lance el usuario, de forma consciente o no, y de esta forma permiten que un virus —o cualquier otro programa— pueda producir daños impunemente. Esto no ocurre en los sistemas operativos serios, donde cada usuario tiene diferentes privilegios, de forma que sólo los programas que ejecute el administrador de la computadora pueden llegar a ser peligrosos. Si un usuario normal ejecuta un virus, éste sólo podrá estropear, en el peor de los casos, los archivos de quien lo ejecutó, pero en ningún caso podrá llegar a afectar al sistema.

17.4 Protegerse frente a los Virus

Una vez que sabemos algo más sobre los virus, y que hemos visto por dónde se pueden infiltrar en un sistema y por dónde no, estamos en condiciones de elaborar un conjunto mínimo de medidas preventivas que nos va a permitir defendernos de este peligro.

- Trabaje habitualmente en su sistema como usuario, no como administrador. Si por error ejecuta un virus, éste no tendrá privilegios para dañar el sistema. Este consejo va dirigido a usuarios de sistemas operativos serios, como UNIX, Linux, FreeBSD, Windows NT y Windows 2000.
- No ejecute nunca programas de origen dudoso o desconocido.
- Utilice *software* original.
- Si emplea un paquete de oficina capaz de ejecutar macros, asegúrese de que tiene desactivada la ejecución automática de éstas. Si no puede desactivarla, emplee otro programa.
- Utilice frecuentemente un buen antivirus. Esto no le protegerá a usted, sino más bien a la comunidad. Suponga que todo el mundo emplea antivirus, entonces todos los virus conocidos se verán frenados en su contagio, quedando sólo los desconocidos, frente a los que un antivirus se vuelve inútil. Podría entonces pensarse en dejar de emplear antivirus, pero en ese caso los virus conocidos volverían a representar un peligro.
- Realice con frecuencia copias de seguridad de la información importante. De esta forma, si un virus destruye sus datos, siempre podrá echar mano de la copia para minimizar el daño.

Parte VI

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 características. 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.

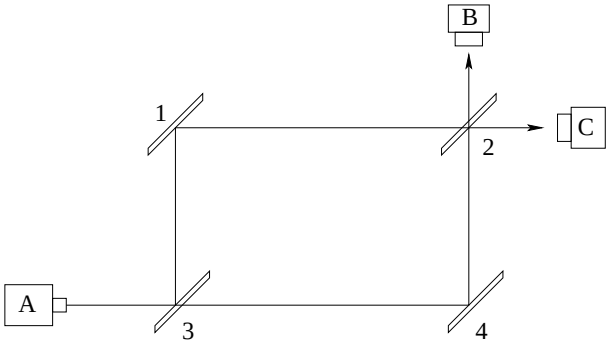

Figura A.1: Experimento con espejos para comprobar la superposición cuántica de estados en un fotón. A es una fuente emisora de fotones, B y C son receptores, 1 y 4 son espejos totalmente reflectantes, y 2 y 3 son espejos que reflejan exactamente la mitad de la luz y dejan pasar la otra mitad. Contrariamente a lo que diría la intuición, en B no se detecta nada.

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 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!.

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 $b_{58}b_{50}b_{42} \ldots b_7$.

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	
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8	S_1
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	
1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5	S_2
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	
1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1	S_3
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	
1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9	S_4
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	
1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6	S_5
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	
1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8	S_6
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	
1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6	S_7
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	
1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2	S_8
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	

Tabla 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

Tabla 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

Tabla B.3: Permutaciones E y P para la función f de DES.

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 64 bits, de los que conserva sólo 56, mientras que la segunda toma 56, y devuelve 48.

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

Tabla B.4: Permutaciones EP1 y EP2 para DES.

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, siendo el bit más a la izquierda el más significativo.

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, siendo el bit más a la izquierda 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 MD5

Incluiremos los resultados que se obtienen al aplicar el algoritmo MD5 a algunas cadenas de caracteres ASCII (excluyendo las comillas):

```
"a"             0cc175b9c0f1b6a831c399e269772661
"test"          098f6bcd4621d373cade4e832627b4f6
"experimento"   304982bc8de3c4948067ceedab604593
"kriptopolis"   d442a043e2575ab4f82b6d0b819adf4a
```

Recuerde que la implementación de este algoritmo es sensible al orden que emplee su microprocesador para representar números enteros. Dicho orden se denomina *little endian* si el byte de un entero que se almacena en la posición de memoria con el número de índice más bajo es el menos significativo, y *big endian* en el caso opuesto. Si sus resultados contienen los mismos valores, pero en diferente orden, probablemente haya considerado erróneamente esta característica.

Apéndice C

Ejercicios Resueltos

Capítulo 3

1. 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. 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. 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. Desarollemos 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:

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

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. Sea G la variable aleatoria que representa los partidos. Sea g_s el suceso correspondiente a que el equipo gana 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. 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. La entropía de los mensajes es igual a:

$$H(X) = -0.5 \cdot \log_2(0.5) - 10 \cdot 0.05 \cdot \log_2(0.05) = 2.660 \; bits$$

Capítulo 5

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

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

- *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. El producto en grupos finitos cumple las propiedades conmutativa y asociativa, además de la existencia de elementos neutro. Tendremos en cuenta, al igual que en el ejercicio anerior, que:

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

   - *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. Para calcular la función ϕ de Euler 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. Emplearemos la expresión (5.6) 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:

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

Sustityendo 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. Para calcular el valor de $(2^{10368} \bmod 187)$, se puede emplear el algoritmo de exponenciación rápida (apartado 5.4.1):

$r =$	1		$z =$	10368	$a =$	2
$r =$	1		$z =$	5184	$a =$	4
$r =$	1		$z =$	2592	$a =$	8
$r =$	1		$z =$	1296	$a =$	16
$r =$	1		$z =$	648	$a =$	32
$r =$	1		$z =$	324	$a =$	64
$r =$	1		$z =$	162	$a =$	128
$r =$	1		$z =$	81	$a =$	256 $\pmod{187} = 69$
$r =$	69		$z =$	40	$a =$	4761 $\pmod{187} = 86$
$r =$	69		$z =$	20	$a =$	7396 $\pmod{187} = 103$
$r =$	69		$z =$	10	$a =$	10609 $\pmod{187} = 137$
$r =$	69		$z =$	5	$a =$	18769 $\pmod{187} = 69$
$r =$	4761	$\pmod{187} = 86$	$z =$	2	$a =$	7396 $\pmod{187} = 86$
$r =$	86		$z =$	1	$a =$	10609 $\pmod{187} = 103$
$r =$	8858	$\pmod{187} = 69$				

6. 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:

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

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

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

Capítulo 6

1. Partiendo de la expresión

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

desarrollaremos el segundo término:

$$(x - q)^2(x - r) = (x^2 - 2qx + q^2)(x - r) = x^3 - 2qx^2 - rx^2 + q^2x + 2qrx - q^2r$$

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

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

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

   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

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

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

Capítulo 7

1. 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:

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

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

Capítulo 12

1. a) Para factorizar N, basta con emplear el método de tanteo (*prueba y error*) a partir de la raíz cuadrada de 44173, obteniendose 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 de Exponenciación Rápida:

$$
\begin{array}{rclrcl}
c_0 &=& 8767 & m_0 &=& 8767^{K_p} \pmod{44173} = 75 \\
c_1 &=& 18584 & m_1 &=& 18584^{K_p} \pmod{44173} = 114 \\
c_2 &=& 7557 & m_2 &=& 7557^{K_p} \pmod{44173} = 105 \\
c_3 &=& 4510 & m_3 &=& 4510^{K_p} \pmod{44173} = 112 \\
c_4 &=& 40818 & m_4 &=& 40818^{K_p} \pmod{44173} = 116 \\
c_5 &=& 39760 & m_5 &=& 39760^{K_p} \pmod{44173} = 111 \\
c_6 &=& 4510 & m_6 &=& 4510^{K_p} \pmod{44173} = 112 \\
c_7 &=& 39760 & m_7 &=& 39760^{K_p} \pmod{44173} = 111 \\
c_8 &=& 6813 & m_8 &=& 6813^{K_p} \pmod{44173} = 108 \\
c_9 &=& 7557 & m_9 &=& 7557^{K_p} \pmod{44173} = 105 \\
c_{10} &=& 14747 & m_{10} &=& 14747^{K_p} \pmod{44173} = 115
\end{array}
$$

Bibliografía

[1] Bruce Schneier. *Applied Cryptography. Second Edition*. John Wiley & sons, 1996.

[2] Alfred J. Menezes, Paul C. van Oorschot y Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[3] Seberry, J., Pieprzyk, J. *Cryptography. An Introduction to Computer Security*. Prentice Hall. Australia, 1989.

[4] Juan Manuel Velázquez y Arturo Quirantes. *Manual de PGP 5.53i*. 1998.

[5] John D. Lipson. *Elements of Algebra and Algebraic Computing*. Addison-Wesley, 1981.

[6] *RFC 2440: Open PGP Message Format*.
http://www.ietf.org/rfc/rfc2440.txt

[7] *RFC 1750: Randomness Recommendations for Security*.
http://www.it.kth.se/docs/rfc/rfcs/rfc1750.txt

[8] *Página Web de Kriptópolis*.
http://www.kriptopolis.com

[9] *Página Web de PGP International*.
http://www.pgpi.org

[10] *Página Web de Zedz Consultants (antes Replay)*.
http://www.zedz.net