

Práctica 3: OpenSSL y Criptografía Pública

Introducción	2
OpenSSL	2
Cifrados simétricos	2
Cifrados asimétricos	10
Generación de claves privadas y públicas	12
Diferencias entre velocidades de cifrados simétricos y asimétricos	13
Certificados X.509	16
RSA	18
Potenciación de grandes números	18
Generación de números primos: Miller-Rabin	18
Factorización del módulo del RSA mediante el conocimiento de d	20

Introducción

En esta práctica vamos a familiarizarnos con las herramientas de OpenSSL, relacionadas con las generación de claves y utilización de los diferentes cifrados simétricos y asimétricos disponibles. Posteriormente, trabajamos con la criptografía pública, más concretamente el algoritmo RSA. Relacionado con esto, también vamos a desarrollar una serie de funciones en GMP para la potenciación modular de números de gran tamaño, generación de números primos mediante Miller Rabin, y la factorización del módulo RSA conociendo el parámetro d .

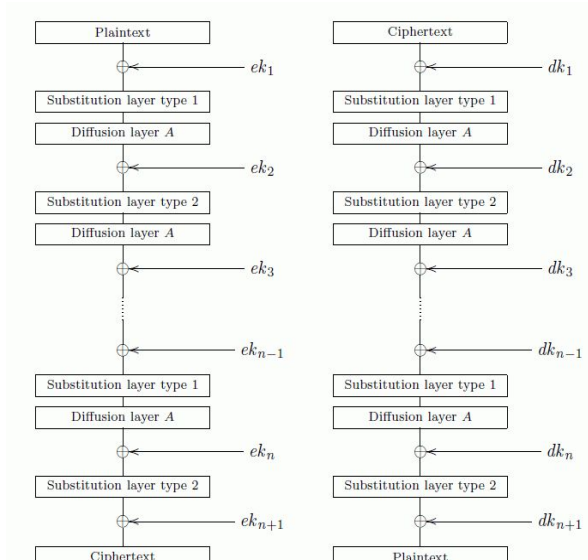
1. OpenSSL

a. Cifrados simétricos

Openssl soporta una gran variedad de cifrados simétricos, es decir, en los que los emisores tienen la misma clave secreta para compartir los mensajes. Cabe destacar que en la mayoría de estos hay variantes de cifrado, por ejemplo, en la cantidad de bits utilizados en la clave y bloque (128, 192, 256) o en los modos de operación (CBC, ECB, OFB...).

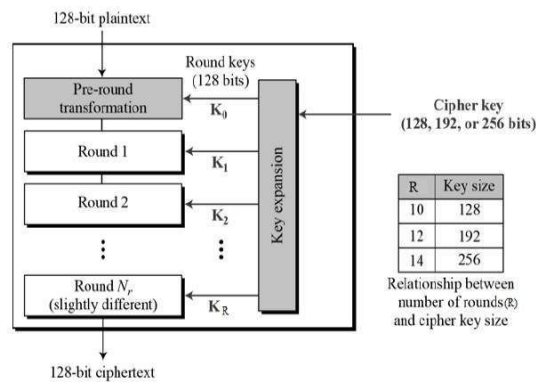
ARIA

ARIA es un algoritmo de cifrado de bloque desarrollado por criptógrafos coreanos en 2003. El algoritmo utiliza una estructura de red de permutación de sustitución basada en AES. La interfaz es la misma que AES: tamaño de bloque de 128 bits con tamaño de clave de 128, 192 o 256 bits. El número de rondas es 12, 14 o 16, dependiendo del tamaño de la clave. ARIA usa dos cajas S de 8×8 bits y sus inversos en rondas alternadas. Es seguro y adecuado para la mayoría de las implementaciones de software y hardware en procesadores de 32 y 8 bits.



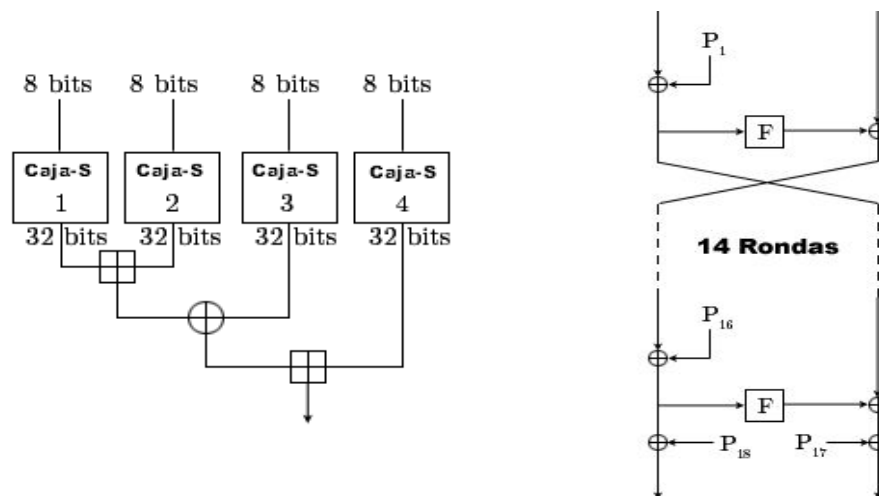
AES

Advanced Encryption Standard (AES), también conocido como Rijndael, es un esquema de cifrado simétrico por bloques adoptado como estándar de cifrado por el gobierno de Estados Unidos. Tiene un tamaño de bloque fijo de 128 bits y tamaños de clave de 128, 192 o 256 bits. La mayoría de los cálculos del algoritmo AES se hacen en un campo finito determinado. AES opera en una matriz de 4×4 bytes, llamada state.



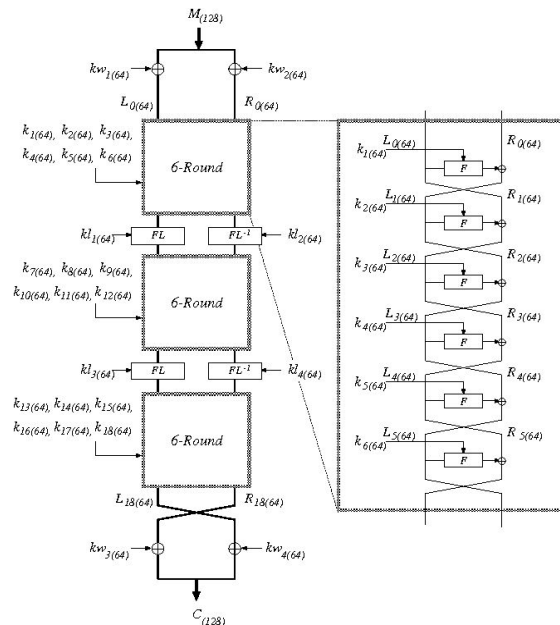
Blowfish

Es un algoritmo de cifrado por bloques simétrico creado por Bruce Schneier en 1993 como una alternativa para reemplazar a DES como estándar de cifrado. Usa bloques de 64 bits y claves que van desde los 32 bits hasta los 448 bits. Es un codificador de 16 rondas Feistel y usa claves que dependen de las Cajas-S. Tiene una estructura similar a CAST-128, el cual usa Cajas-S fijas. Es un proceso relativamente simple y altamente seguro ya que hasta la fecha no se conoce ningún tipo de criptoanálisis efectivo contra este algoritmo de cifrado.



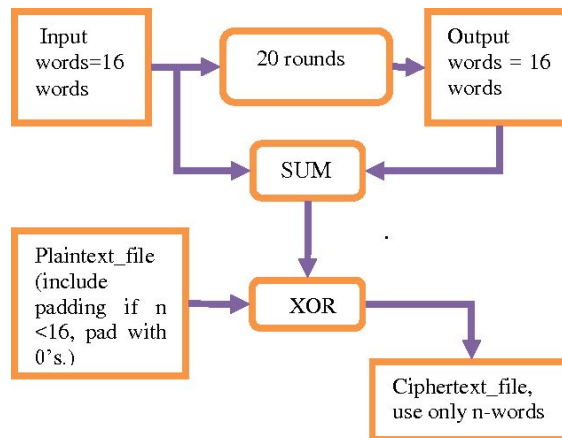
Camellia

Camellia es un cifrado simétrico con un tamaño de bloque de 128 bits y claves de 128, 192 y 256 bits desarrollado en Japón. Sigue la estructura de Feistel con 18 o 24 rondas y tiene 4 cajas de sustitución. Cada 6 rondas se realizan unas operaciones lógicas llamadas FL. Es un sistema idóneo para las implementaciones de software y hardware, así como por su alto nivel de seguridad. Además, su tiempo de configuración de clave es excelente y la rapidez de manejo de clave es superior a la del AES.



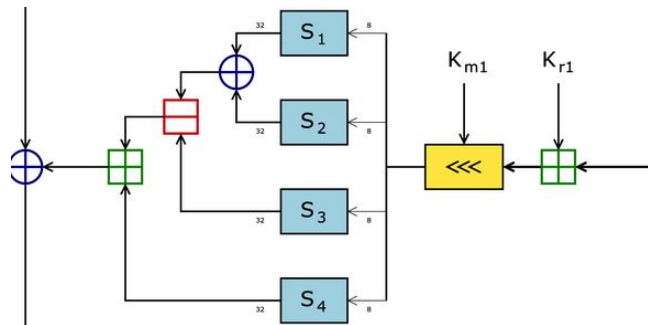
Chacha20

ChaCha es una modificación de Salsa20 publicada por Bernstein en 2008. Utiliza una nueva ronda que aumenta la difusión y el rendimiento en algunas arquitecturas, usando 20 rondas. Ambos sistemas de cifrado se basan en una función pseudoaleatoria basada en operaciones suma de 32 bits (add), adición bit a bit (xor) y operaciones de rotación (rotate). Utiliza claves de tamaño 128 o 256 y tamaños de bloque de 512 bits.

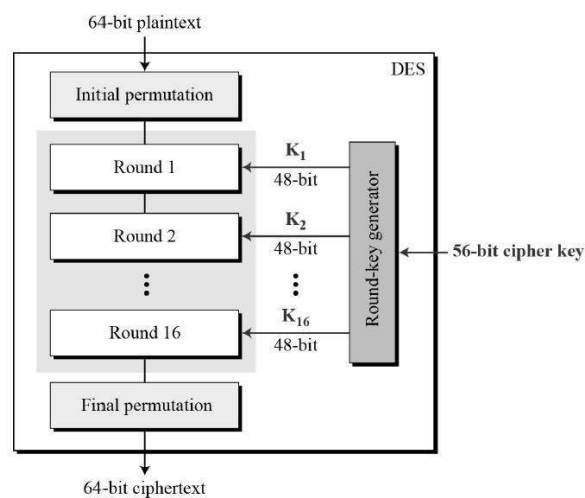


Cast

CAST-128 (o Cast5) es un cifrador de 12 o 16 rondas que se basa en Feistel con bloques de 64 bits y tamaños de clave entre 40 y 128 bits (pero con solo incrementos de 8 bits). Las 16 rondas completas se usan cuando la clave tiene un tamaño mayor de 80 bits. Incluye unas largas S-Boxes de 8x32 bits basadas en funciones booleanas, rotaciones de clave, adición y sustracción modular y operaciones xor.

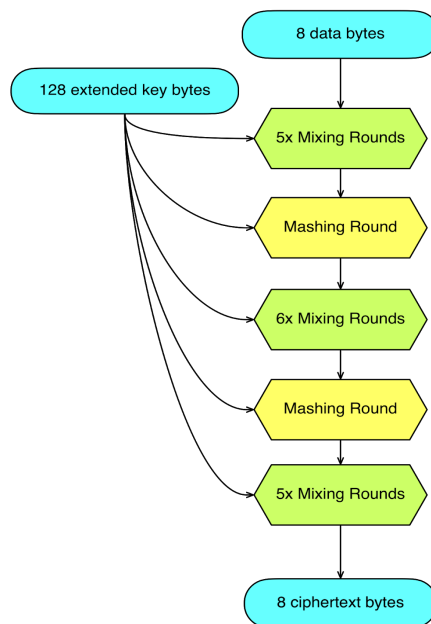
DES

DES es el algoritmo prototipo del cifrado por bloques, es decir, un algoritmo que toma un texto en claro de una longitud fija de bits y lo transforma, mediante una serie de operaciones básicas, en otro texto cifrado de la misma longitud. En DES, el tamaño de bloque y la clave son de 64 bits, aunque en realidad, sólo 56 bits de la clave son empleados por el algoritmo. Los ocho bits restantes se utilizan únicamente para comprobar la paridad, y después son descartados. Las rondas (16 en total) están basadas en Feistel.



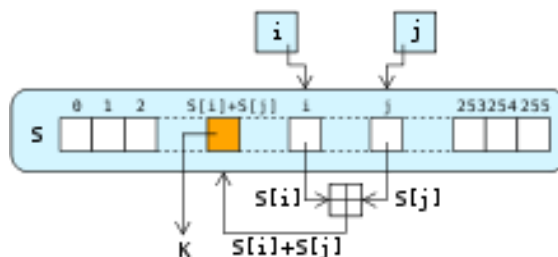
RC2

RC2 (también conocido como ARC2) es un cifrado de bloque de clave simétrica diseñado por Ron Rivest en 1987. Es un cifrado de bloque de 64 bits con una clave de tamaño variable. Sus 18 rondas están dispuestas como una red Feistel desequilibrada de gran densidad de fuente, con 16 rondas de un tipo (MIXING) marcadas por dos rondas de otro tipo (MASHING).



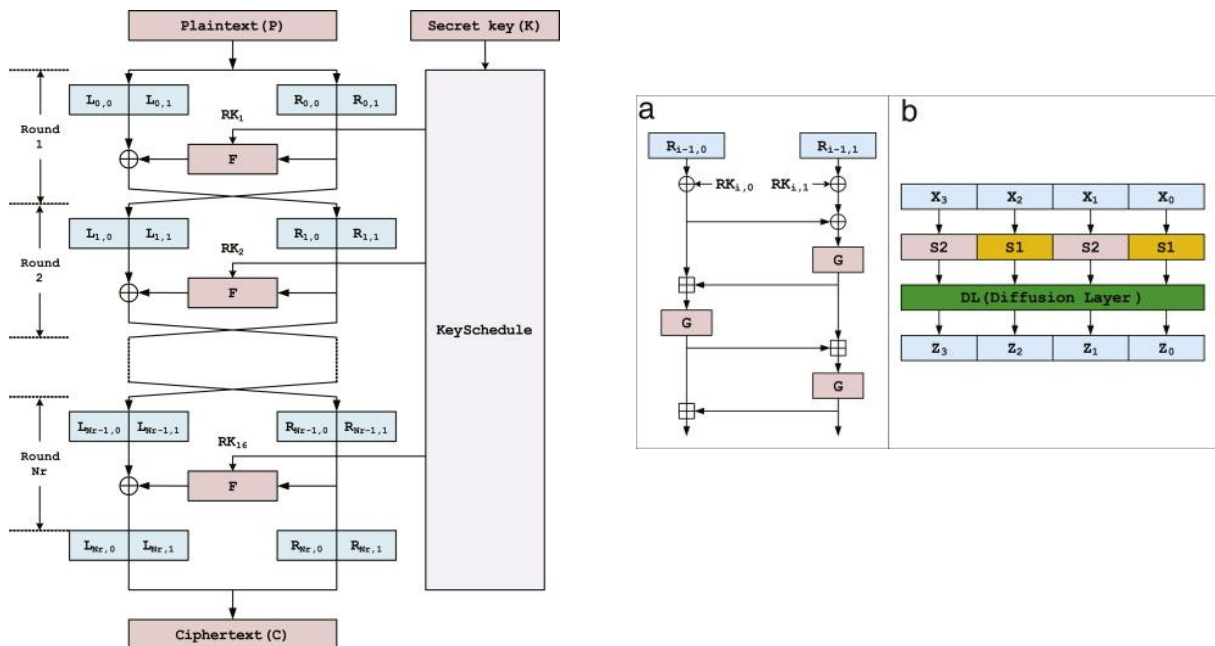
RC4

RC4 o ARC4 es el sistema de cifrado de flujo más utilizado y se usa en algunos de los protocolos más populares como Transport Layer Security (TLS/SSL) y Wired Equivalent Privacy (WEP). RC4 fue excluido de los estándares de alta seguridad por los criptógrafos y algunos modos de usar el algoritmo lo han llevado a ser un sistema de criptografía muy inseguro. Es un algoritmo muy simple. Consiste en 2 algoritmos: 1-Key Scheduling Algorithm (KSA) y 2- Pseudo-Random Generation Algorithm (PRGA). Cada uno de estos algoritmos usa 8-by-8 S-box. El KSA se encarga de realizar la primera mezcla en el S-Box, basado en el valor de la semilla dada.



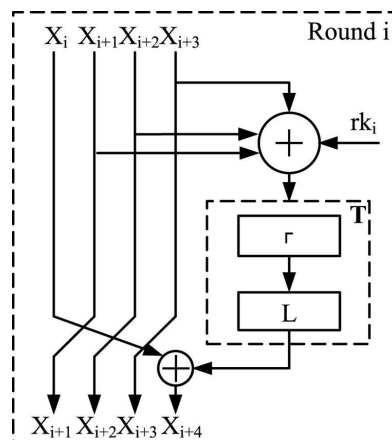
SEED

SEED es un cifrado de bloque desarrollado por la Agencia de Seguridad e Internet de Corea (KISA). Tiene la estructura Feistel de 16 rondas con bloques de 128 bits y una clave de 128 bits. Utiliza dos cajas S de 8×8 que se derivan de una exponenciación discreta. Tiene una función G en la cual cada palabra de 32 bits se considera como cuatro bytes de 8 bits, se pasan por las cajas S, y luego se combinan con un conjunto complejo de funciones booleanas, de manera que cada bit de salida depende de 3 de los 4 bytes de entrada. Tiene una programación de claves compleja, generando las 32 subclaves de 32 bits mediante la función G en una serie de rotaciones de la clave, combinadas con constantes.



SM4

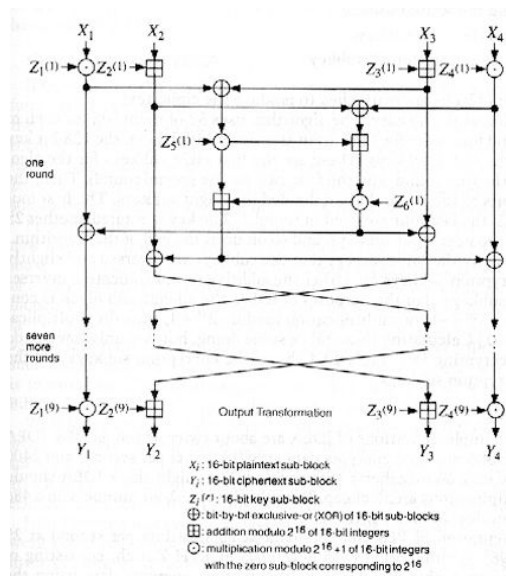
El algoritmo SMS4 fue inventado por el profesor LU Shu-wang y se desclasificó en enero de 2006. Tiene un tamaño de bloque y de clave de 128 bits. Utiliza una caja S de 8 bits y 32 rondas. Las únicas operaciones utilizadas son XOR, desplazamientos circulares de 32 bits y aplicaciones S-Box. Cada ronda actualiza un cuarto del estado interno (es decir, 32 bits). El descifrado usa las mismas claves pero en orden inverso.



IDEA

International Data Encryption Algorithm es un cifrador por bloques descrito por primera vez en 1991. Opera con bloques de 64 bits usando una clave de 128 bits y consiste en ocho transformaciones idénticas (cada una llamada ronda) y una transformación de salida (llamada media ronda). El proceso para cifrar y descifrar es similar. Gran parte de la seguridad de IDEA deriva del intercalado de operaciones de distintos grupos, es decir, adición y multiplicación modular y O-exclusivo (XOR) bit a bit. Este algoritmo presenta diferencias notables con el DES:

- El espacio de claves es mucho más grande: $2^{128} \approx 3.4 \times 10^{38}$
- Todas las operaciones son algebraicas
- No hay operaciones a nivel bit, facilitando su programación en alto nivel
- Es más eficiente que los algoritmos de tipo Feistel, porque a cada ronda se modifican todos los bits de bloque y no solamente la mitad.
- Se pueden utilizar todos los modos de operación definidos para el DES



Podemos probar el funcionamiento de alguno de estos cifrados simétricos con openssl y la terminal de comandos. Para ello, hemos creado el script llamado *cifrados_simetricos.sh*, el cual encripta un texto de entrada y desencripta para comprobar en cada caso que se obtiene el mismo resultado.

```
printf "Probando cifrados simetricos\n\n"

printf "AES\n"
openssl enc -e -aes-128-ecb -K FFFFFFFFFFFFFFFFFF0000000000000000 -in in.txt -out out_aes.txt
openssl enc -d -aes-128-ecb -K FFFFFFFFFFFFFFFFFF0000000000000000 -out in_aes.txt -in out_aes.txt

printf "Aria\n"
openssl enc -e -aria-128-ecb -K FFFFFFFFFFFFFFFFFF0000000000000000 -in in.txt -out out_aria.txt
openssl enc -d -aria-128-ecb -K FFFFFFFFFFFFFFFFFF0000000000000000 -out in_aria.txt -in out_aria.txt

printf "Blowfish\n"
openssl enc -e -bf-ecb -K FFFFFFFFFFFFFFFFFF0000000000000000 -in in.txt -out out_blowfish.txt
openssl enc -d -bf-ecb -K FFFFFFFFFFFFFFFFFF0000000000000000 -out in_blowfish.txt -in out_blowfish.txt
```


También podemos probar las claves débiles y semidébiles del DES, mediante el siguiente script, llamado *keys_des.sh*:

```
echo "Probando clave debiles DES"

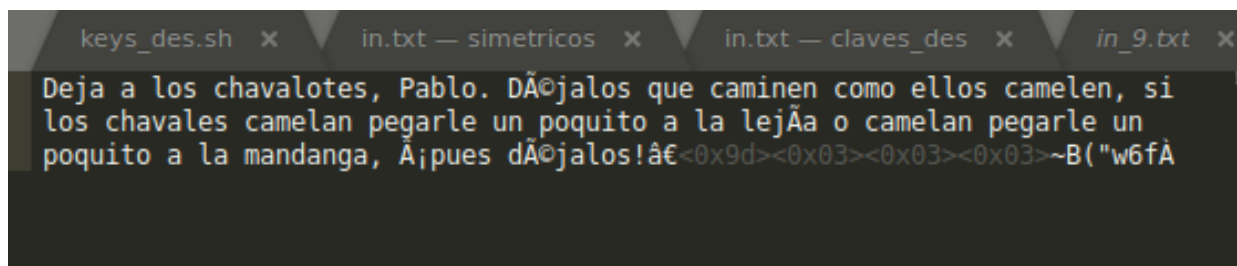
openssl enc -e -des-ecb -K 01010101010101 -in in.txt -out out_1.txt
openssl enc -e -des-ecb -K 01010101010101 -out in_1.txt -in out_1.txt

openssl enc -e -des-ecb -K FEF EFEFEFEFEFEFE -in in.txt -out out_2.txt
openssl enc -e -des-ecb -K FEF EFEFEFEFEFEFE -out in_2.txt -in out_2.txt

openssl enc -e -des-ecb -K E0E0E0E0F1F1F1F1 -in in.txt -out out_3.txt
openssl enc -e -des-ecb -K E0E0E0E0F1F1F1F1 -out in_3.txt -in out_3.txt

openssl enc -e -des-ecb -K 1F1F1F1F0E0E0E0E -in in.txt -out out_4.txt
openssl enc -e -des-ecb -K 1F1F1F1F0E0E0E0E -out in_4.txt -in out_4.txt
```

Observamos que al encriptar dos veces con la misma clave débil, obtenemos el mensaje original esencialmente, salvo por lo que suponemos, un padding añadido internamente por el algoritmo, como observamos en la siguiente imagen tras encriptar dos veces con una clave débil:



```
keys_des.sh x in.txt — simetricos x in.txt — claves_des x in_9.txt x
Deja a los chavalotes, Pablo. DÃojalos que caminen como ellos camelen, si
los chavales camelan pegarle un poquito a la lejÃa o camelan pegarle un
poquito a la mandanga, Ãípues dÃojalos!â€<0x9d><0x03><0x03><0x03>~B("w6fÀ
```

Lo mismo ocurre con los pares de claves semidébiles.

```
echo "Probando claves semidebiles DES"

openssl enc -e -des-ecb -K 011F011F010E010E -in in.txt -out out_5.txt
openssl enc -e -des-ecb -K 1F011F010E010E01 -out in_5.txt -in out_5.txt

openssl enc -e -des-ecb -K 01E001E001F101F1 -in in.txt -out out_6.txt
openssl enc -e -des-ecb -K E001E001F101F101 -out in_6.txt -in out_6.txt

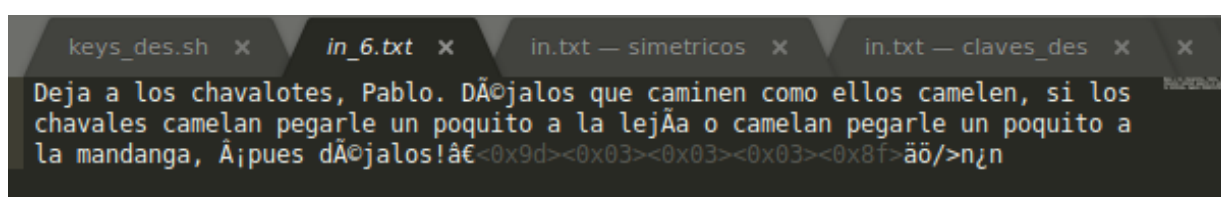
openssl enc -e -des-ecb -K 01FE01FE01FE01FE -in in.txt -out out_7.txt
openssl enc -e -des-ecb -K FE01FE01FE01FE01 -out in_7.txt -in out_7.txt

openssl enc -e -des-ecb -K 1FE01FE00EF10EF1 -in in.txt -out out_8.txt
openssl enc -e -des-ecb -K E01FE01FF10EF10E -out in_8.txt -in out_8.txt

openssl enc -e -des-ecb -K 1FFE1FFE0EFE0EFE -in in.txt -out out_9.txt
openssl enc -e -des-ecb -K FE1FFE1FFE0EFE0E -out in_9.txt -in out_9.txt

openssl enc -e -des-ecb -K E0FEE0FEF1FEF1FE -in in.txt -out out_10.txt
openssl enc -e -des-ecb -K FEE0FEE0FEF1FEF1 -out in_10.txt -in out_10.txt
```

Podemos observar el ruido al final del mensaje tras la doble encriptación con semidébiles:



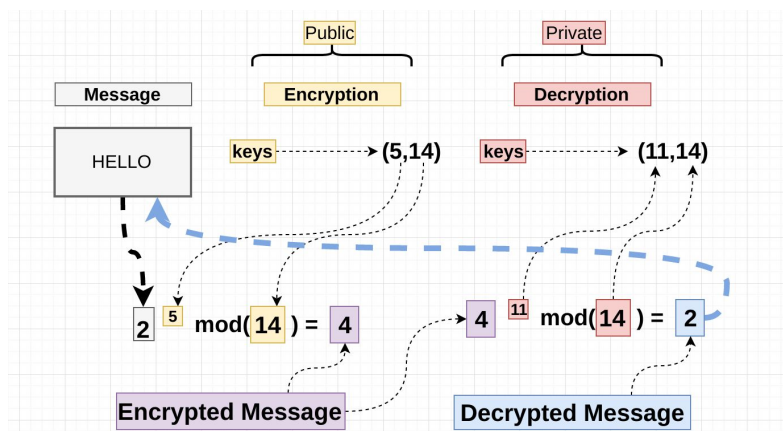
```
keys_des.sh x in_6.txt x in.txt — simetricos x in.txt — claves_des x x
Deja a los chavalotes, Pablo. DÃojalos que caminen como ellos camelen, si
los chavales camelan pegarle un poquito a la lejÃa o camelan pegarle un poquito a
la mandanga, Ãípues dÃojalos!â€<0x9d><0x03><0x03><0x03><0x8f>ãö/>n¿n
```

b. Cifrados asimétricos

RSA

RSA (Rivest, Shamir y Adleman) es un sistema criptográfico de clave pública desarrollado en 1979. Es el primero de este tipo y el más usado, siendo válido tanto para cifrar como para firmar digitalmente.

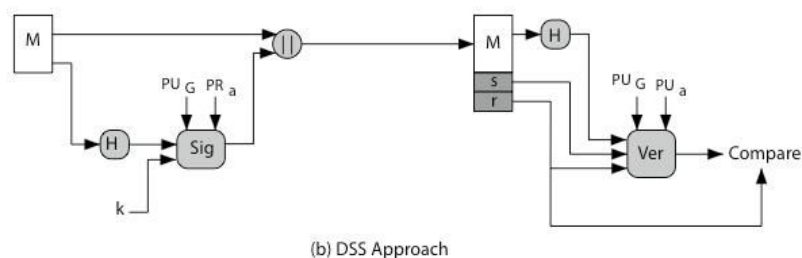
Su seguridad radica en la complejidad de factorizar números enteros. Los mensajes enviados se representan mediante números, y el funcionamiento se basa en el producto de dos números primos grandes al azar y mantenidos en secreto. Actualmente estos primos son del orden de 10^{200} , y se prevé que su tamaño crezca con el aumento de la capacidad de cómputo de los ordenadores.



Ejemplo RSA

DSA

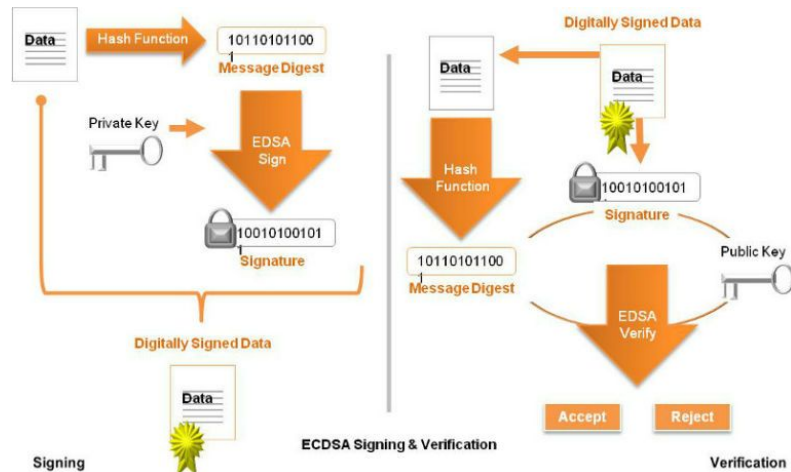
DSA (Digital Signature Algorithm) es un estándar del Gobierno Federal de los Estados Unidos de América o FIPS para firmas digitales. Fue un Algoritmo propuesto por el Instituto Nacional de Normas y Tecnología de los Estados Unidos para su uso en su Estándar de Firma Digital (DSS). DSA se hizo público el 30 de agosto de 1991. Este algoritmo como su nombre indica, sirve para firmar y no para cifrar información. Una desventaja de este algoritmo es que requiere mucho más tiempo de cómputo que RSA.



(b) DSS Approach

ECDSA

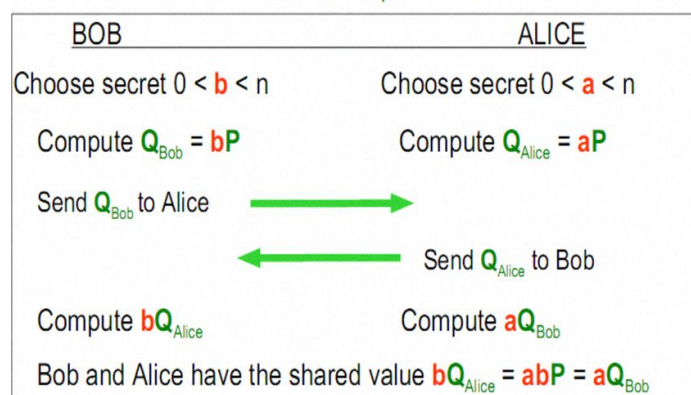
ECDSA. Elliptic Curve Digital Signature Algorithm es una modificación del algoritmo DSA que emplea operaciones sobre puntos de curvas elípticas en lugar de las exponenciaciones que usa DSA (problema del logaritmo discreto). La principal ventaja de este esquema es que requiere números de tamaños menores para brindar la misma seguridad que DSA o RSA. Existen dos tipos de curvas dependiendo del campo finito en el que se definan, que pueden ser $GF(P)$ o $GF(2^m)$.



ECDH

El protocolo Elliptic-curve Diffie–Hellman (ECDH) es un protocolo de establecimiento de claves anónimo que permite a dos partes, cada una de las cuales tiene un par de claves pública-privada de curvas elípticas, establecer un secreto compartido en un canal inseguro. Este secreto compartido puede usarse directamente como clave o para derivar otra clave. La clave, o la clave derivada, pueden usarse entonces para encriptar sucesivas comunicaciones usando un cifrado de clave simétrica. Es una variante del protocolo Diffie-Hellman usando criptografía de curvas elípticas.

Public Knowledge: A group $E(F_p)$ and a point P of order n .



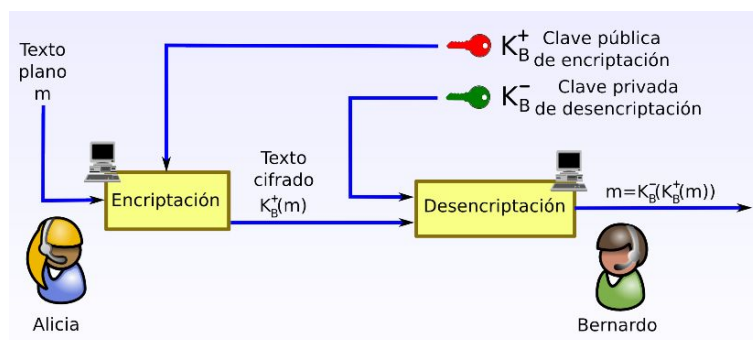
Podemos probar el funcionamiento de alguno de estos algoritmos mediante el script *cifrados_simetricos*. Comprobamos el encriptado y desencriptado de los ficheros y vemos que funcionan correctamente.

c. Generación de claves privadas y públicas

Definido en el apartado anterior qué es RSA, vayamos a explicar cómo funciona.

Cada usuario posee dos claves de cifrado: una pública y otra privada. Cuando se quiere enviar un mensaje, el emisor busca la clave pública del receptor, cifra su mensaje con esa clave, y una vez que el mensaje cifrado llega al receptor, este se ocupa de descifrarlo usando su clave privada.

Se cree que RSA será seguro mientras no se conozcan formas rápidas de descomponer un número grande en producto de primos. Se cree que con la computación cuántica se podría romper este algoritmo de cifrado.



Ahora vamos a crear un sencillo script para generar las claves pública y privada de RSA. En este script *RSA_keys*, irán los siguientes comandos:

Para crear el par de claves usamos *openssl genrsa* con el número de bits deseados.

```
openssl genrsa -out keypair.pem 2048
```

Luego, extraemos la parte de la clave pública y la guardamos en otro fichero

```
openssl rsa -in keypair.pem -pubout -out publickey.crt
```

Finalmente, podemos convertir el par original de claves en formato PKCS8:

```
openssl pkcs8 -topk8 -inform PEM -outform PEM -nocrypt -in keypair.pem -out pkcs8.key
```

Podemos ver que se han creado los ficheros correctamente y que en la cabecera podemos ver si se refiere a la clave pública o privada.

```
-----BEGIN RSA PRIVATE KEY-----
MIIEogIBAAKCAQEAub5bdKCtPhq2RWCvtP0chx0VlBvj37X+szwsfYtq4xdz9XJL
IUyRGNJHdYpDXLuXMnnCeg79HQgn3nYB6EPCYqRyWax7U1VI9j7iIrnS5/R7ANR7
GGv/w0uM8AnR41tHqfm10ZCyLLYXQ3M8d9vUJE5uto6gPeCJz97UviGndsoF89gA
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAub5bdKCtPhq2RWCvtP0c
hx0VlBvj37X+szwsfYtq4xdz9XJLIUyRGNJHdYpDXLuXMnnCeg79HQgn3nYB6EPC
YqRyWax7U1VI9j7iIrnS5/R7ANR7GGv/w0uM8AnR41tHqfm10ZCyLLYXQ3M8d9vU
```

d. Diferencias entre velocidades de cifrados simétricos y asimétricos

En este apartado vamos a comparar las velocidades de los cifrados simétricos y asimétricos. Intuitivamente, cabe pensar que los cifrados simétricos serán más rápidos que los asimétricos, ya que en los primeros se suelen utilizar operaciones básicas como xor, add, rotaciones de bit, que son bastante ligeras, mientras que en los cifrados asimétricos se suelen utilizar potenciación y módulos de números muy grandes (2048 bits), lo que supone que sean algoritmos mucho más pesados computacionalmente.

Para este proceso, utilizamos el comando *openssl speed* y volcamos la salida en un fichero llamado *speed.txt*, para su mejor manejo.

Al final de dicho fichero, tras la ejecución, nos aparece el siguiente resumen, en función de los tamaños de bloque utilizados:

The 'numbers' are in 1000s of bytes per second processed.

type	16 bytes	64 bytes	256 bytes	1024 bytes	8192 bytes	16384 bytes
md2	0.00	0.00	0.00	0.00	0.00	0.00
mdc2	0.00	0.00	0.00	0.00	0.00	0.00
md4	82316.30k	242827.71k	536240.64k	811809.79k	946094.08k	942228.20k
md5	101740.14k	249343.26k	438247.17k	545194.50k	584888.32k	586006.53k
hmac(md5)	46787.39k	145722.47k	318972.91k	490320.90k	571670.53k	577021.43k
sha1	107232.14k	253450.46k	527304.19k	639643.94k	789520.38k	756588.54k
rmd160	35505.55k	91774.36k	154302.71k	192166.47k	199069.78k	208224.26k
rc4	343312.81k	437881.41k	506360.32k	701254.66k	690704.38k	675495.94k
des cbc	63577.50k	62842.89k	63535.98k	65819.65k	64360.45k	66748.42k
des ede3	23386.92k	23473.72k	24302.72k	24576.00k	24694.78k	24109.06k
idea cbc	0.00	0.00	0.00	0.00	0.00	0.00
seed cbc	77710.06k	79333.06k	78900.22k	78170.62k	79572.99k	79585.28k
rc2 cbc	43925.14k	44283.07k	44920.06k	45171.26k	44921.50k	45293.57k
rc5-32/12 cbc	0.00	0.00	0.00	0.00	0.00	0.00
blowfish cbc	110653.97k	118177.22k	117688.58k	121049.09k	121114.62k	121643.01k
cast cbc	97943.50k	106331.52k	107841.02k	107336.19k	106938.37k	107003.90k
aes-128 cbc	110853.60k	119565.44k	110034.30k	106525.18k	125612.03k	127131.65k
aes-192 cbc	94514.30k	97297.06k	102746.88k	104645.12k	104620.03k	104947.71k
aes-256 cbc	70987.87k	82622.88k	89237.89k	88855.55k	89903.10k	86294.53k
camellia-128 cbc	87322.01k	140813.09k	163120.26k	168874.50k	171483.14k	170983.42k
camellia-192 cbc	77581.89k	110791.81k	123235.58k	126825.47k	128630.87k	128598.02k
camellia-256 cbc	77019.35k	110548.58k	123300.74k	127659.01k	128466.94k	128819.20k
sha256	64927.46k	145489.89k	276198.91k	346846.72k	369336.32k	371728.38k
sha512	44994.63k	180777.18k	314582.78k	469789.70k	539262.98k	548577.28k
whirlpool	29984.27k	64100.45k	108023.38k	129368.06k	134213.63k	130842.62k
aes-128 ige	112706.38k	118443.78k	119774.85k	121357.82k	121425.92k	120741.89k
aes-192 ige	95204.03k	100083.17k	100214.66k	101810.18k	100999.17k	101040.13k
aes-256 ige	82967.23k	86525.70k	86393.86k	86808.58k	86507.52k	87146.50k
ghash	1119786.48k	4321062.94k	8179307.01k	9156342.78k	9507094.53k	9904455.68k
rand	9110.19k	35456.58k	109041.72k	223644.67k	337481.73k	333971.46k

Observamos, en primer lugar, que en algunos cifrados, los miles de bytes procesados por segundo aumentan a medida que aumenta el tamaño de los bloques, hasta que se estabiliza a partir de los 1024 bytes por bloque, como por ejemplo en md4, md5, sha256, ...

En segundo lugar, vemos que los cifrados simétricos, en bloques a partir de 1024 bytes, están en torno a 100.000K de bytes procesados por segundo, en algunos casos más (como Camellia) y en algunos casos menos (como des). De todas formas, el valor mínimo de eficiencia de estos cifrados es el correspondiente al triple des, que no baja de los 20.000K de bytes procesados.

			sign	verify	sign/s	verify/s
rsa	512 bits		0.000055s	0.000003s	18274.5	333771.4
rsa	1024 bits		0.000116s	0.000007s	8591.5	145243.0
rsa	2048 bits		0.000771s	0.000021s	1296.5	48327.6
rsa	3072 bits		0.002251s	0.000044s	444.3	22792.0
rsa	4096 bits		0.004938s	0.000073s	202.5	13790.0
rsa	7680 bits		0.046136s	0.000245s	21.7	4074.5
rsa	15360 bits		0.218000s	0.000958s	4.6	1044.0
			sign	verify	sign/s	verify/s
dsa	512 bits		0.000082s	0.000041s	12134.0	24234.0
dsa	1024 bits		0.000134s	0.000091s	7466.0	11012.5
dsa	2048 bits		0.000310s	0.000262s	3229.0	3811.5
			sign	verify	sign/s	verify/s
160 bits	ecdsa	(secp160r1)	0.0003s	0.0002s	3852.5	4113.0
192 bits	ecdsa	(nistp192)	0.0003s	0.0003s	2920.0	3085.5
224 bits	ecdsa	(nistp224)	0.0001s	0.0002s	13209.5	4321.0
256 bits	ecdsa	(nistp256)	0.0000s	0.0001s	37130.5	12308.5
384 bits	ecdsa	(nistp384)	0.0012s	0.0009s	847.5	1118.0
521 bits	ecdsa	(nistp521)	0.0004s	0.0008s	2505.5	1290.0
163 bits	ecdsa	(nistk163)	0.0003s	0.0006s	3444.5	1636.2
233 bits	ecdsa	(nistk233)	0.0004s	0.0008s	2346.7	1208.0
283 bits	ecdsa	(nistk283)	0.0008s	0.0014s	1306.5	729.0
409 bits	ecdsa	(nistk409)	0.0011s	0.0023s	908.0	432.5
571 bits	ecdsa	(nistk571)	0.0025s	0.0048s	404.5	209.0
163 bits	ecdsa	(nistb163)	0.0003s	0.0006s	3478.4	1749.5
233 bits	ecdsa	(nistb233)	0.0004s	0.0008s	2530.0	1274.5
283 bits	ecdsa	(nistb283)	0.0007s	0.0013s	1457.0	744.5
409 bits	ecdsa	(nistb409)	0.0012s	0.0023s	862.5	435.3
571 bits	ecdsa	(nistb571)	0.0026s	0.0052s	382.5	193.0
256 bits	ecdsa	(brainpoolP256r1)	0.0005s	0.0004s	1997.0	2272.5
256 bits	ecdsa	(brainpoolP256t1)	0.0005s	0.0004s	2008.5	2380.5
384 bits	ecdsa	(brainpoolP384r1)	0.0012s	0.0010s	835.5	1044.5
384 bits	ecdsa	(brainpoolP384t1)	0.0013s	0.0011s	790.0	950.0
512 bits	ecdsa	(brainpoolP512r1)	0.0020s	0.0015s	512.5	688.5
512 bits	ecdsa	(brainpoolP512t1)	0.0018s	0.0013s	558.0	752.0
			op	op/s		
160 bits	ecdh	(secp160r1)	0.0003s	3939.0		
192 bits	ecdh	(nistp192)	0.0003s	3081.5		
224 bits	ecdh	(nistp224)	0.0001s	8616.0		
256 bits	ecdh	(nistp256)	0.0001s	16038.2		
384 bits	ecdh	(nistp384)	0.0012s	8		

Respecto a los cifrados asimétricos, tomando como referencia los 1024 bits, nos encontramos con que están en torno 8000 firmas en RSA y DSA, o en 224 bits en ECDSA y ECDH en torno a 10000 firmas. En ningún caso se cifran más de 40000.

Por tanto, hemos comprobado que lo que intuíamos se ha cumplido, y que los cifrados asimétricos son más pesados de calcular, por los argumentos que hemos descrito anteriormente, la simplicidad de las operaciones bit a bit de los simétricos frente a la complejidad de la potenciación y módulos de números muy grandes de los asimétricos.

e. Certificados X.509

Los servicios de una autoridad de certificación, son principalmente utilizados para garantizar la seguridad de las comunicaciones digitales vía el protocolo TLS, utilizados en las comunicaciones web (HTTPS) o las comunicaciones de emails, así como para resguardar documentos digitales (por ejemplo, utilizando firmas electrónicas).

Los certificados de CA tienen gran utilidad en las comunicaciones web. Los navegadores web modernos integran de forma nativa (en forma estándar) una lista de certificados de diferentes autoridades de certificación, los que según los casos, son elegidos automáticamente de acuerdo a los criterios internos definidos por los desarrolladores del navegador.

Entonces, cuando una persona física o jurídica requiere configurar un servidor web mediante una comunicación segura HTTPS y securitizada por TLS, se genera una clave pública, y una clave privada es enviada a una de estas autoridades de certificación, con una solicitud de firma de certificado.

El proceso de crear un certificado autofirmado por una autoridad certificadora lo hemos realizado en un script llamado *certificados.sh*. Este fichero contiene los siguientes comandos:

Primero generamos la clave de la autoridad certificadora (CA).

```
openssl genrsa -out ca.key 4096
```

Ahora necesitamos crear el certificado de la CA (ponemos los campos ya rellenos).

```
openssl req -new -x509 -days 1826 -key ca.key -out ca.crt -subj "/C=US/ST=New  
York/L=Brooklyn/O=CA/CN=ca.com"
```

Lo siguiente es crear la clave para el servidor a certificar (en este caso nosotros mismos).

```
openssl genrsa -out server.key 4096
```

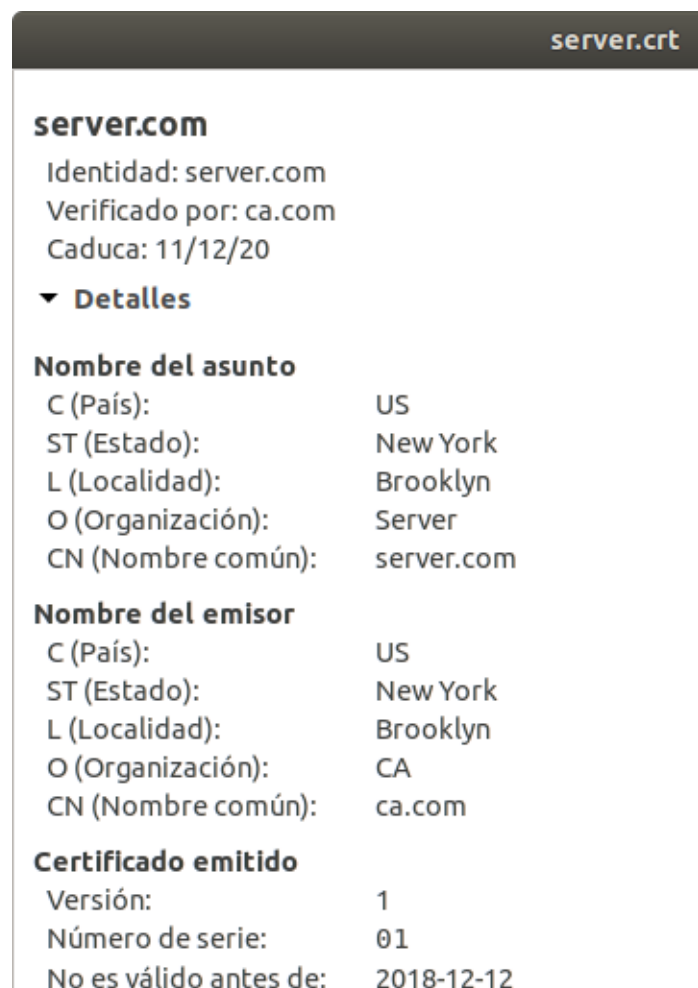
Ahora creamos el certificado del servidor correspondiente a esa clave.

```
openssl req -new -key server.key -out server.csr -subj "/C=US/ST=New  
York/L=Brooklyn/O=Server/CN=server.com"
```

Por último, firmamos el certificado del servidor utilizando el certificado y la clave de la CA.

```
openssl x509 -req -days 730 -in server.csr -CA ca.crt -CAkey ca.key -set_serial 01 -out  
server.crt
```


Ya tenemos nuestro certificado autofirmado, *server.crt*. Podemos ver la información sobre éste abriendo el archivo desde el explorador, observando que, efectivamente, ha sido verificado por nuestra CA y tiene la fecha de validez.



O bien mediante openssl, con `openssl x509 -noout -text -in server.crt`

```
Certificate:
  Data:
    Version: 1 (0x0)
    Serial Number: 1 (0x1)
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: C = US, ST = New York, L = Brooklyn, O = CA, CN = ca.com
    Validity
      Not Before: Dec 12 10:22:27 2018 GMT
      Not After : Dec 11 10:22:27 2020 GMT
    Subject: C = US, ST = New York, L = Brooklyn, O = Server, CN = server.com
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
      RSA Public-Key: (4096 bit)
      Modulus:
        00:c8:97:7c:44:be:e1:64:b0:3d:e5:0c:ab:95:03:
        e6:46:ed:c8:55:7a:c4:2f:83:3f:dd:5b:94:b3:83:
```

2. RSA

a. Potenciación de grandes números

Para realizar la potenciación, hemos usado el algoritmo visto en clase.

Este algoritmo se basa en la siguiente propiedad:

$$(a_1 a_2) \bmod n = [(a_1 \bmod n) * (a_2 \bmod n)] \bmod n$$

Es decir, es mucho menos costoso ir reduciendo modularmente cada vez que se multiplica que primero multiplicarlo y luego reducirlo al final.

Es más, nuestro algoritmo es eficiente bajo la idea de reutilizar potencias parciales:

$$\text{Sean la base } b \text{ y el exponente } e = \sum_{i=0}^{l-1} e_i 2^i \text{ con } e_i \in \{0, 1\} \text{ entonces } b^e \bmod n = \prod_{i=0, e_i \neq 0}^{l-1} b^{2^i} \bmod n$$

A continuación, veamos un ejemplo de ejecución:

```
carlos@carlos-SVF1521N6EW:~/Escritorio/practicas/Cripto/P3/g05$ ./potencia
Ejecucion: ./potencia -b base -e exponente -m modulo
carlos@carlos-SVF1521N6EW:~/Escritorio/practicas/Cripto/P3/g05$ ./potencia -b 123456789 -e 987654321 -m 456
Resultado: 189
El resultado coincide con el de la librería GMP
```

Como vemos, nuestro programa calcula el resultado de la potenciación, lo imprime por pantalla, y posteriormente lo compara con el de la librería GMP, indicando que con GMP se obtiene el mismo resultado.

b. Generación de números primos: Miller-Rabin

Para generar números aleatorios primos, antes de aplicarle el test de primalidad de Miller-Rabin, realizamos ciertas modificaciones y comprobaciones para que el número generado sea razonable que pueda ser un primo grande.

Para empezar, hacemos que sus bits menos significativo y más significativo estén a 1, así el número será impar y grande. Después, dividiremos el número por una tabla de primos menores que 2000 para descartar números compuestos.

En caso de que el número primo coincida con alguno de los primos de la tabla, lo usaremos como primo para el test.

A continuación procedemos a realizar el Test de Miller-Rabin tantas veces como hemos especificado en el comando.

Si el test detecta que el número es compuesto, se vuelve al primero paso y se genera otro candidato a número primo y se itera hasta que salga un número que pase todos los filtros incluido el Test de Miller-Rabin.

A continuación, veamos ejemplos de ejecución con 4096 bits y diferentes valores para el flag t. Observaremos que nuestro algoritmo y el de GMP llegan a las mismas conclusiones dados los mismos parámetros:

- **t = 1**

Si ejecutamos el test sólo una vez, nuestro algoritmo y el de GMP obviamente nos dan primos posibles con un error bastante elevado (0.999024), casi del 100%.

```
carlos@carlos-SVF1521N6EW:~/Escritorio/practicass/Cripto/P3/g05$ ./primo -b 4096 -t 1
Número: 650884473247516217351406824514515124297333517067591724032738324276734490
03118339194439486849827691695090011246381136595679288264927518607582239622948842
84744846753474243141533669606085000318993719565915133261808069110652106541690155
2395713719216094348028266302701299133311406779697128463776258893806333435379741
28900116767532006078338076229047801557241673997921963066038408765897986543566462
36685584504337350950619980796733959375361000978101915577100547314850934199406123
75812052543688054663898813186051299153978334207016613351262492680726587262645265
05469779791857648115090412492451166912215955464704085855718227228492487230794530
48403759317924963272551379502305964890721804655014036246996262999979638504652790
78396193025385248112211943093555978012540284677381111700405043206977707022403760
76999448199837682066311526003691071654607703649291462754968870063289029794341577
45505358916491635069575348056133185790141405682495826606683544872521655506793953
29375863032776383505276218267541660715678345906975460226834600662333417356539413
74898173468011100279183989138535361699880807413067177586139382807668339071077071
49493058767812247923171269419162794573053510104567499104069690181892257421760895
87869147669173593599550684912869643088939
Resultado de nuestro test: POSIBLE_PRIMO con error de: 0.999024
Resultado de GMP: POSIBLE_PRIMO
```

- **t = 10**

Al aumentar el número de repeticiones del test, el error baja muy rápidamente, siendo ya menor del 1%.

```
carlos@carlos-SVF1521N6EW:~/Escritorio/practicass/Cripto/P3/g05$ ./primo -b 4096 -t 10
Número: 894270563467966085721959726078276053110491925139144716827699653792510031
19470403147000276017504804637134624868996338622174365945209669344251464123563402
58783159378033918804583466459388498617927821942173668026258087894969285476827371
79571981638204247767750951690008589563621190388712571679308032155148694729747912
01029054575802419572042904714009339058982093074455501317011865664810578821011063
75952408924670742785194594789442611315251969565296265944575850748104442805147855
84795110161478080658075795261918072945919724708501292277845838425377744135079271
17491007541296217200208294787082659066718039859013664610560730309365441276822689
30331209175382870232837819808184725398215437203368492958352116288446862246644234
49423050400022167836442540382086085784077124299650115284621929494990169448582116
46721510941501075408328885552426081574335588178906059538201896208577848785634269
59831295071204570193673597597551845498517186418255050666532514981869260655739159
68772618575306052168959727610255064408024408059442618068914562734153800295320303
24621960417098684942470533536750190497942463537713552812198066529123370731506811
65235750865658011364198454611237746122141436200485404116495468335859518406663137
18877587962113108557898127839828754613281
Resultado de nuestro test: POSIBLE_PRIMO con error de: 0.003891
Resultado de GMP: POSIBLE_PRIMO
```

- **t = 100**

Si aumentamos el número de repeticiones a 100, ya tenemos un primo con una probabilidad de error prácticamente nula (no conseguimos representar más decimales de precisión).


```

carlos@carlos-SVF1521N6EW:~/Escritorio/practicass/Cripto/P3/g05$ ./primo -b 4096
-t 100
Número: 839889611945723624831503924466450820761253182752186017362612799195306203
24884523956192748424405311685885750378162072296430865260152338819957989385983018
72615122626822714755523794253816567887292445944423270755628719315529818457604100
80658822782414786253519181032607326469577556085423262645012099620213724466382720
29942077425483709603798899614508682496926034814728127181525109201729377326250342
11496571104009845382558351829857682515419315989947035381752286073551443669981952
4274985615141277734402400905187028311309735584469928009404704915346696998555221
74157812080835151699591389441702150446545077959320990587400769000003249753254533
42336120179803040479421241617863714604893460610422148319368786674694313577504404
31317237464689090507624735913353359205871647651808971687126107710970826490815150
36438552579040857226143532430174458517639100176610005016773069780592728176271511
93570820568338547678083521956169827722166185712403517451459692907816399294880695
07783110975721786784415251259513171774917320812785252716028069993370186202400466
66039564555545857707681341695771519839052288968305674991492288607519511885772705
84494111426870172071725674815171411090217891323125223608030104648401091784133736
39120019179556339389552741243501464931511
Resultado de nuestro test: POSIBLE_PRIMO con error de: 0.000000
Resultado de GMP: POSIBLE_PRIMO

```

c. Factorización del módulo del RSA mediante el conocimiento de d

Para realizar este apartado, hemos utilizado el algoritmo de las Vegas.

Como observación, hemos hecho que si el algoritmo no responde, que ponga los valores p y q calculados a 0. En el programa principal, si detecta que están a 0 repetirá el algoritmo hasta que nos dé los 2 primos correspondientes a la clave.

Para poder realizar las pruebas con 4096b hemos creado un programa adicional llamado key_gen, donde dados p y q, nos calcula los valores n,d y e.

Su comando de ejecución tiene el siguiente formato: ./key_gen {-p primo} {-q primo}

Para probar este apartado con números de 4096b hemos hecho los siguientes pasos:

1. Generamos 2 primos de 4096b usando el programa de generación de primos del apartado anterior:

```
carlos@carlos-SVF1521N6EW:~/Escritorio/practicas/Cripto/P3/g05$ ./primo -b 4096 -t 100
Número: 60727008079212448534191653456567610207186570017110148968324863015138990016162542317425847893046247452911501948846582140383059607154625
1490622147168685575499329729643324095109662291533601090667241807242510181756806950907042556784286363713147974470150851291428894997008055548386
5584359083844562299982325509931471257225097146633087324862603790394905215039977944723439885180302166192148864549184591908861003172926182530042
1734909382319615654890404157715228626888159769892187407277867356416290769004577100656167460072218706749747140634331089267829577598120157084969
790881564126500216645756664246163042550460428778736514915971577108174460050308809598975440427650095398899666757354974445355501161637768219924
0743936652192015624927625156802136856232316135741496072798434844687652127424799238548160095291287994691544319927238323808584327052899544919708
1304259192215068878165013006753798249773172689152694067854268924149546437144296871928271114358868772880717686301172761843457057640796141643575
7208715883694536536618107963531220771193706218941109754269749688470242065125997905355221571807485403639866712303190833660817424899358442592942
24348856375055627285593134449424066676007275907571364302854569782919512157273392238695259317840091685043
Resultado de nuestro test: POSIBLE_PRIMO con error de: 0.000000
Resultado de GMP: POSIBLE_PRIMO
carlos@carlos-SVF1521N6EW:~/Escritorio/practicas/Cripto/P3/g05$ ./primo -b 4096 -t 100
Número: 10332263794597675049456096028048848184954940447236776702991850048498933048969610215923005480977531287236792367307396859150097897036503
20252024233262821443977136122303979088552613878149972978858085025415266983107654251596651648511931758938077550108490517461550892228096265995
2078699403806815786339577870558839729133942178869041975499726559324740155239638813314028556998243580879450338716440396520946774319150868355361
207431179974716318374283859039907076142911579322861065621188390969457619712197086162249574457616507250943331798192083055069196970419730100034
8983076974365882197190119953927503023837099718664435935045424833448594212102126105866000343986794224467385396564071236320055704520225712455715
759840407846252878982198358294494527478011959531991163109697489321362822466732465733663298054640963151389150703282742561557150949653739024286
12494078089257185409812756489213577176065457807372164194133524473067835358584167259276307522116237239140267517622739978062825580550626595182
564609844176558235536913693126105416291280331005688193364383182370140780605614147398157978253318829056876178923501542489503024496607573484418
7207351556189978285529669211873686228954033431841063645613381589094284255049042223771453950437173657482323
Resultado de nuestro test: POSIBLE_PRIMO con error de: 0.000000
Resultado de GMP: POSIBLE_PRIMO
```

2. Generamos los valores n, e y d con los primos anteriores con el programa key_gen, mediante el siguiente comando:

```
carlos@carlos-SVF1521N6EW:~/Escritorio/practicas/Cripto/P3/g05$ ./key_gen -p 60727008079212448534191653456567610207186570017110148968324863015
1389900161625423174258478930462474529115019488465821403830596071546251490622147168685575499329729643324095109662291533601090667241807242510181
7568069509970425567842863637131479744701508512914288949970080555483865584359083844562299982325509931471257225097146633087324862603790394905215
0399779447234398851803021661921488645491845919088610031729261825300421734999382319615654890404157715228626888159769892187407277867356416290769
004571700656167460072218706749747140634331089267829577598120157084969790881564265002166457566664246163042550460428778736514915971577108174460
503880895989754404276500953988996667573549744453555011616377682199240743566129602136856232316135741496072798438444687652127
4247992385481600952912879946915443199272383238085843270528995449107081304259192215068878165013006753798249773172689152694067854268924149546437
14429687192827111435886877288071768630117276184345705764079614164357572087518836945365366181079635312207711937062189410975469749688470242605
125997053552215718074854036398667123031908336608174248993584425929424344885637505562728559313444942406667600727590757136430285454697829195121
57273392238695259317840091685043 -q 1033226379459767504945609602804884818495494044723677670299185004849893304896961021592300548097753128723679
2367307396859150097897036503202520244232628214439771361223039790885526138781499729788580850254152669831076542515966516485119317589380775501084905
8490517461550892228096265995207869940380681578633957787055883972913394217886904197549972655932474015523963881331402855699824358087945033871644
039652094677431915806835536120743117997471631837428385903990707614291157932286106562118839096945761971219708616224957445761650725094333179819
2083055069196970419730100034898307697436588219719011995392750302383709971866443593504542483344859421210212610586600034398679422446738539656407
12363200557045202257124557157598404047846252878982198358294494527478011959531991163109697489321362822466732465733663298054640963151389150703282
274256155715094965373902428641249407808925718540981275648921357717606545780737216419413352447306783535858416725927630752211623723914026751762
739978062825580550626595182664609844176558235536913693126105416291280331005688193364383182370140780605614147398157978253318829056876178923501
54248950302444966075734844187207351556189978285529669211873686228954033431841063645613381589094284255049042223771453950437173657482323
```

La n generada es siempre la misma, pero los números e y d no son siempre los mismos, pues la e se genera aleatoriamente, y la d se calcula como la inversa de dicha e (y no mostramos la salida generada, pues es demasiado grande).

3. Ejecutamos el programa de las Vegas con los valores n, e y d generados anteriormente, y obtenemos esto como salida:

```
Factores primos son:
1033226379459767504945609602804884818495494044723677670299185004849893304896961021592300548097753128723679236730739685915009789703650320252024
423326282144397713612230397908855261387814997297885808502541526698310765425159665164851193175893807755010849051746155089222809626599520786994
0380681578633957787055883972913394217886904197549972655932474015523963881331402855699824358087945033871644039652094677431915086835536120743117
997471631837428385903990707614291157932286106562118839096945761971219708616224957445761650725094333179819208305506919697041973010003489830769
7436588219719011995392750302383709971866443593504542483344859421210212610586600034398679422446738539656407123632005570452022571245571575984044
0784625287898219835829449452747801195953199116310969748932136282246673246573366329805464096315138915070328274256155715094965373902428641249407
808925718540981275648921357717606545780737216419413352447306783535858416725927630752211623723914026751762273997806282558055062659518266460984
4176558235536913693126105416291280331005688193364383182370140780605614147398157978253318829056876178923501542489503024449660757348441872073515
56189978285529669211873686228954033431841063645613381589094284255049042223771453950437173657482323

6072700807921244853419165345656761020718657001711014896832486301513899001616254231742584789304624745291150194884658214038305960715462514906221
47168685575499329729643324095109662291533601090667241807242510181756806950907042556784286363713147974470150851291428894997008055548386558435908384456229998232550993147125722509714663308732486260379039490521503997794472343988518030216619214886454918459190886100317292618253004217349093
8231961565489040415771522862688815976989218740727786735641629076900457710065616746007221870674974714063433108926782957759812015708496979088156
412650021664575666424616304255046042877873651491597157710817446005030880959897544042765009539889966675735497444535550116163776821992407439366
5219201562492762515680213685623231613574149607279843484468765212742479923854816009529128799469154431992723832380858432705289954491970813042591
9221506887816501300675379824977317268915269406785426892414954643714429687192827111435886877288071768630117276184345705764079614164357572087518
8369453653661810796353122077119370621894110975426974968847024206512599790535522157180748540363986671230319083366081742489935844259294224348856
375055627285593134449424066676007275907571364302854569782919512157273392238695259317840091685043
```

Como vemos, los primos coinciden con los generados en el paso 1, luego el algoritmo de las Vegas funciona.