Criptografía y seguridad informática

Práctica 3

Grupo 4

Mario Valdemaro García Roque
Alberto Cabello Álvarez

1) OpenSSL
    - Cifrados simétricos.
    - Cifrados asimétricos.
    - Generación de claves privadas y públicas.
    - Diferencias entre velocidades de cifrados simétricos y asimétricos.
    - Certificados X.509.
2) RSA
    - Potenciación de grandes números.
    - Generación de números primos: Miller-Rabin.

Cifrados simétricos

Openssl es un paquete de herramientas que contiene la implementación de multitud de funciones relacionadas con la criptografía y utilidades varias.

En este primer apartado, comenzamos investigando sobre las funciones de cifrado si - métricas. Consultando los comandos de ayuda como referencia, probamos alguno de estos cifrados.

En primer lugar, cifremos con des en el modo de operación ecb para observar el mane - jo que se hace del padding.

```
openssl enc -des-ecb -K 0000000000000000 -in Downloads/me-
moria.odt -out Downloads/cifrado.cosa
```

```
openssl enc -des-ecb -K 0000000000000000 -in Downloads/ci-
frado.cosa -out Downloads/memoriadescifrada.odt
```

```
tail memoria.odt -n 1
```

```
[…] <META-INF/manifest.xmlPK####=
```

```
tail memoriadescifrada.odt -n 1
```

```
[…] <META-INF/manifest.xmlPK###  #  =~B("w6f
```

Se añaden varios caracteres al descifrar el fichero en comparación con el original. Además, con esta misma prueba, hemos comprobado que el cifrado con una clave dé - bil es invertible cifrando por segunda vez con la misma clave.

Para contrastar este comportamiento, repetimos esta misma prueba con una clave cual - quiera.

```
openssl enc -des-ecb -K 0123456789ABCDEF -in memoria.odt
-out clavenormal.cosa
```

```
openssl enc -d -des-ecb -K 0123456789ABCDEF -out memoria -
normal.odt  -in clavenormal.cosa
```

```
openssl enc -des-ecb -K 0123456789ABCDEF -out memorianor -
malcifrada2veces.odt  -in clavenormal.cosa
```

En efecto. La memoria cifrada dos veces con una clave cualquiera es ilegible, mientras que la descifrada utilizando **enc -d** sí que se lee correctamente.

Continuamos comprobando el funcionamiento de las claves semidébiles. Estas son claves para las que el cifrado con una clave es descifrable por el cifrado de la segunda clave. El DES tiene 6 pares de claves semidébiles.

First key in the pair	Second key in the pair
01FE 01FE 01FE 01FE	FE01 FE01 FE01 FE01
1FE0 1FE0 0EF1 0EF1	E01F E01F F10E F10E
01E0 01E0 01F1 01F1	E001 E001 F101 F101
1FFE 1FFE 0EFE 0EFE	FE1F FE1F FE0E FE0E
011F 011F 010E 010E	1F01 1F01 0E01 0E01
E0FE E0FE F1FE F1FE	FEE0 FEF0 FEF1 FEF1

```
openssl enc -des-ecb -K 01fe01fe01fe01fe -in memoria.odt
-out semidebiles.cosa
```

```
openssl enc -des-ecb -K fe01fe01fe01fe01 -out memoriasemi-
debil.odt -in semidebiles.cosa
```

Con estos ejemplos, hemos comprobado que el cifrado con una clave semidébil se puede deshacer con el cifrado con la clave semidébil correspondiente de la pareja.

Openssl soporta múltiples algoritmos de cifrado, cada uno con varios modos de operación y/o diferentes tamaños de clave aceptados.

Por mencionar algunos, los algoritmos de cifrado simétricos disponibles son AES, Blowfish, Camellia, CAST, DES, RC2, RC4 y seed.

Cifrados asimétricos

Los principales cifrados que encontramos de tipo asimétrico son:

- RSA: Es un algoritmo que sirve tanto para cifrar mensajes como para firmar.
- DSA: Es un algoritmo que firma claves en vez de mensajes, aunque no es un algoritmo de cifrado asimétrico como tal nos ha parecido importante destacarlo ya es una parte importante de este tipo de cifrados.
- Diffie-Hellman: Al igual que DSA es mas un algoritmo de generación de claves que de comunicación.

Destacamos que además de usar los comando de ayuda de OpenSSL hemos usado el libro “Network Security with OpenSSL”, que básicamente es una guía sobre como usar el programa. Contiene bastantes ejemplos sobre todas las opciones que tiene, ade - más de aportar ejemplos sobre como usar SSL junto con el lenguaje C para hacer una conexión segura a través de la red.

A continuación cifraremos y descifraremos un texto con el algoritmo RSA:

```
openssl rsautl -encrypt -pubin -inkey rsapublickey.pem -in
plaintext.txt -out cipher.txt
```

```
openssl rsautl -decrypt -inkey rsaprivatekey.pem -in ci  -
pher.txt -out dechiper.txt
```

```
cat dechiper.txt
```

```
> hola mundo
```

Generación de claves privadas y públicas

Básicamente es un esquema de cifrado en el que existe una clave con la que se cifra y otra con la que se descifra. La idea detrás de este esquema es que una persona "A" ge - nera una clave privada y una clave publica, la clave privada la sabe solo el y la clave pública la pude conocer cualquier persona. Lo que se hace habitualmente es que una persona "B" quiere mandarle un mensaje a "A" entonces lo que se hace es que B cifra un mensaje con la clave pública de A y se lo manda a A, una vez recibido este mensaje A lo descifra con su clave privada, asegurándose así que nadie mas que el va a leer el mensaje que le mando B.

Lo primero que hacemos es generar la calve privada y en función de la clave privada generar la pública:

```
openssl genrsa -out rsaprivatekey.pem
```

```
openssl rsa -in rsaprivatekey.pem  -pubout -out rsapubli -
ckey.pem
```

Y después para probar si lo hemos hecho correctamente ciframos un texto en el que hay escrito hola mundo.

```
openssl rsautl -encrypt -pubin -inkey rsapublickey.pem -in
plaintext.txt -out cipher.txt
```

```
openssl rsautl -decrypt -inkey rsaprivatekey.pem -in ci  -
pher.txt -out dechiper.txt
```

```
cat dechiper.txt
```

```
> hola mundo
```

Como se puede observar se descifra lo que habíamos escrito.

Diferencias entre velocidades de cifrados simétricos y asimétricos.

Para este apartado utilizaremos la herramienta speed de openssl que permite realizar multitud de cifrados y obtener sus tiempos para así poder comparalos.

Queremos comprobar que como norma general, los cifrados simétricos son más rápidos que los cifrados de criptografía pública. Esto es debido al tamaño de grande de claves que se utiliza en los cifrados asimétricos, así como el tipo de operaciones.

Realizamos una serie de pruebas para los algoritmos AES, DES, RC4 y RSA, para diferentes tamaños de clave. Obtenemos los siguientes resultados.

openssl speed rsa4096

```
Doing 4096 bit private rsa's for 10s: 1267 4096 bit private RSA's in 9.99s
Doing 4096 bit public rsa's for 10s: 79424 4096 bit public RSA's in 10.00s
```

openssl speed rsa1024

```
Doing 1024 bit private rsa's for 10s: 67883 1024 bit private RSA's in 9.99s
Doing 1024 bit public rsa's for 10s: 987365 1024 bit public RSA's in 10.00s
```

openssl speed des

```
Doing des cbc for 3s on 16 size blocks: 14178911 des cbc's in 3.00s
Doing des cbc for 3s on 64 size blocks: 3680380 des cbc's in 2.99s
Doing des cbc for 3s on 256 size blocks: 927963 des cbc's in 3.00s
Doing des cbc for 3s on 1024 size blocks: 232759 des cbc's in 3.00s
Doing des cbc for 3s on 8192 size blocks: 29102 des cbc's in 3.00s
Doing des ede3 for 3s on 16 size blocks: 5546053 des ede3's in 2.99s
Doing des ede3 for 3s on 64 size blocks: 1395009 des ede3's in 3.00s
Doing des ede3 for 3s on 256 size blocks: 348787 des ede3's in 3.00s
Doing des ede3 for 3s on 1024 size blocks: 87317 des ede3's in 2.99s
Doing des ede3 for 3s on 8192 size blocks: 10918 des ede3's in 3.00s
```

openssl speed aes

```
Doing aes-128 cbc for 3s on 16 size blocks: 25761337 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 64 size blocks: 7038974 aes-128 cbc's in 3.00s
```

```
Doing aes-128 cbc for 3s on 256 size blocks: 1795064 aes-128 cbc's in 2.99s
Doing aes-128 cbc for 3s on 1024 size blocks: 452510 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 8192 size blocks: 56665 aes-128 cbc's in 3.00s
Doing aes-192 cbc for 3s on 16 size blocks: 21942755 aes-192 cbc's in 3.00s
Doing aes-192 cbc for 3s on 64 size blocks: 5888249 aes-192 cbc's in 3.00s
Doing aes-192 cbc for 3s on 256 size blocks: 1495151 aes-192 cbc's in 3.00s
Doing aes-192 cbc for 3s on 1024 size blocks: 375278 aes-192 cbc's in 2.99s
Doing aes-192 cbc for 3s on 8192 size blocks: 47065 aes-192 cbc's in 3.00s
Doing aes-256 cbc for 3s on 16 size blocks: 19002071 aes-256 cbc's in 3.00s
Doing aes-256 cbc for 3s on 64 size blocks: 5054790 aes-256 cbc's in 3.00s
Doing aes-256 cbc for 3s on 256 size blocks: 1282161 aes-256 cbc's in 3.00s
Doing aes-256 cbc for 3s on 1024 size blocks: 322363 aes-256 cbc's in 2.99s
Doing aes-256 cbc for 3s on 8192 size blocks: 40367 aes-256 cbc's in 3.00s
```

`openssl speed rc4`

```
Doing rc4 for 3s on 16 size blocks: 88903692 rc4's in 3.00s
Doing rc4 for 3s on 64 size blocks: 34738502 rc4's in 2.99s
Doing rc4 for 3s on 256 size blocks: 10057672 rc4's in 3.00s
Doing rc4 for 3s on 1024 size blocks: 2614912 rc4's in 3.00s
Doing rc4 for 3s on 8192 size blocks: 330278 rc4's in 3.00s
```

Aunque para estos tests las medidas no son exactamente iguales, así que una comparación directa no es posible, sí que se puede observar que los algoritmos simétricos son significativamente más rápidos que los asimétricos. En estos tests, se ve que se puede ejecutar muchas más veces un algoritmo simétrico que uno asimétrico.

Entre los tamaños de clave para el RSA, se puede ver como disminuye mucho el rendimiento al aumentar el tamaño de clave. Para este ejemplo, un aumento de la clave de 1024 a 4096, 4x, supone una ralentización de aproximadamente 50x.

Por otro lado, debido a las optimizaciones y al diseño del AES, se puede ver en estas pruebas que además de ser un algoritmo más resistente que el DES, también es más rápido, incluyendo para tamaños de clave grandes.

Certificados X.509.

Se supone que la idea detrás de este sistema es que una entidad certificadora firma las calves de un “servidor” con el que vamos a entablar una comunicación. Este tipo de certificados son ampliamente utilizados en las paginas con las que entablamos una co - municacional segura, en general las paginas que usan https. El motivo por el que se hace esto es para asegurarnos que el servidor al que nos vamos a conectar es quien dice ser, es decir si no supiésemos que la clave con la que estamos cifrando el texto esta certificada no podríamos saber si realmente nos estamos comunicando con el ser - vidor que queremos o con un tercer agente que no es el servidor con el que nos quería - mos comunicar.

La mayoría de estas cosas las sabíamos ya por el hecho de que en la asignatura RE - DES II tuvimos que hacer un servidor y un cliente IRC y uno de los requisitos que te - nían era que se pudiese cifrar la comunicación mediante RSA, para ello tuvimos que generar nuestras propias claves autofirmadas mediante X509.

Hemos incluido por tanto las instrucciones para generar estas claves y certificados en nuestro makefile. Pero en resumidas cuentas para generar un certificado hay que usar los siguientes comandos.

```
certificado:

openssl genrsa -out rootkey.pem 2048

openssl req -new -x509 -key rootkey.pem -out rootcert.pem
-subj "/C=ES/ST=Madrid/L=Madrid/O=UAM/CN=CERTIFICATE"

cat rootkey.pem rootcert.pem > root.pem
```

Comentar también que toda es información o la mayoría de ella esta en el libro que ya hemos comentado “Network Security with OpenSSL”.

Potenciación de grandes números.

En este apartado programaremos una función que realice potenciación modular de grandes números. Hemos empleado el algoritmo de descomposición binaria del expo - nente, el aprendido en clase. Como curiosidad/investigación, buscamos además otros métodos, para comparar su eficacia.

Uno de los algoritmos simplemente multiplica sucesivamente por la base, y toma el módulo al final de cada paso. Este método es bueno con respecto al método básico de tomar la potencia completa y luego el módulo, ya que evita calcular la potencia entera, que ocuparía demasiado en memoria para números grandes. Este algoritmo tiene O(e) operaciones, donde e es el exponente.

Existen además otros métodos con matrices y otras posibilidades, pero nos decantamos finalmente por el método de descomposición binaria. Este método realiza O(log e) operaciones, lo que proporciona rapidez y ahorro en memoria.

Para comparar los resultados y la eficiencia, tomaremos como referencia la función de gmp:

```
mpz_powm(result, base, exponente, modulo);
```

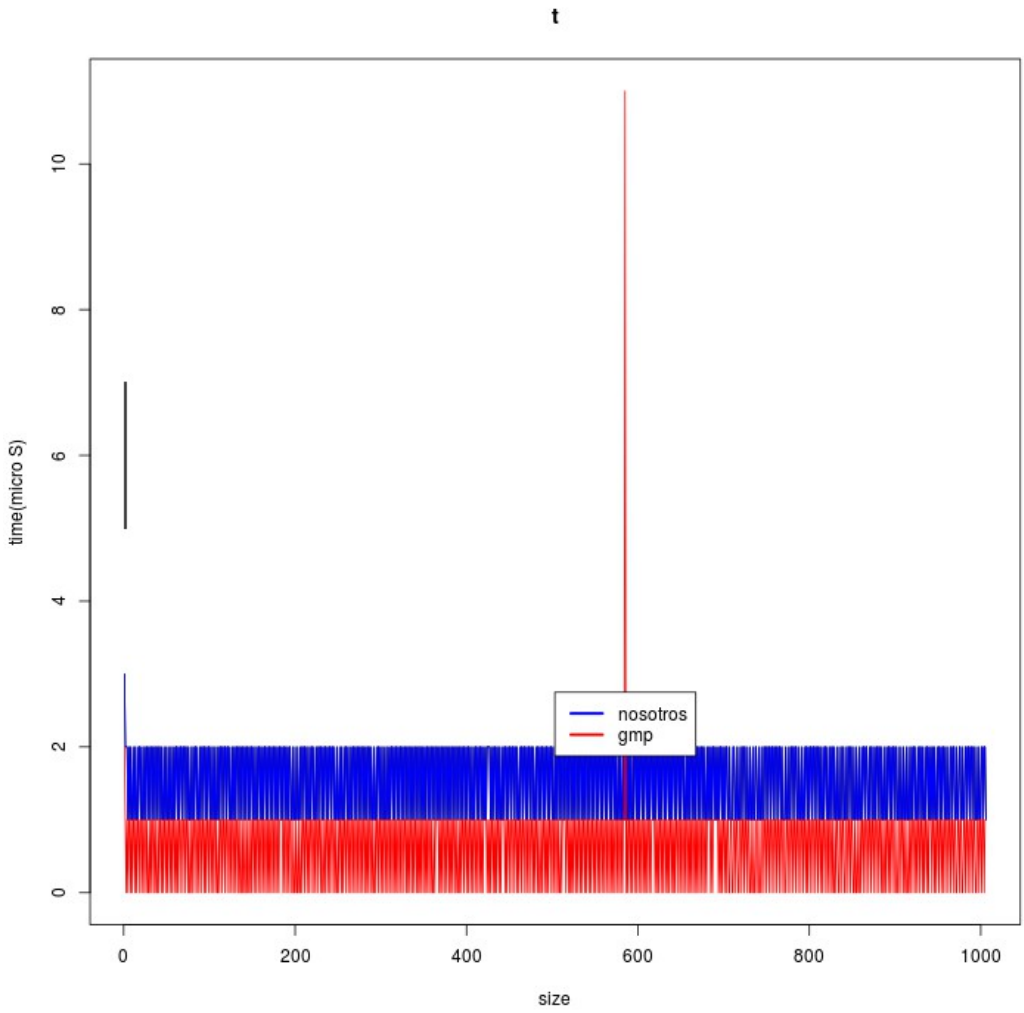
Implementamos un pequeño programa con el algoritmo de potenciación modular. Para las operaciones utilizamos las herramientas de gmp de multiplicación, módulo, and, etc. Un ejemplo de la salida de este programa es:

```
make testpotencia
potencia generado ejecutable
ejecutando potencia 23 11 243
./potencia 23 11 243
Con nuesto algoritmo:
23 elevado a 11 mod 243  == 92
Con GMP:
23 elevado a 11 mod 243  == 92
```

En cuanto al rendimiento de nuestro algoritmo comparado con el rendi - miento de la función powm de gmp:

Hay muy poca diferencia de rendimiento entre usar uno y otro algoritmo, siendo mejor el algoritmo de gmp, porque seguramente este mejor optimizado y nosotros al depen - der de la misma librería gmp podemos perder ciclos innecesarios en cosas que haga in - ternamente gmp.

Ponemos las siguientes gráficas expresadas en micro-segundos para reflejar esto:



Esta prueba se realizo manteniendo la base (base=400) y el modulo (m= 100000) fijos y variando el exponente, (maxExp=201005 minExp=200000).

Como se observa no hay apenas diferencia entre uno y otro algoritmo, debemos estar haciendo alguna operación que se repite mas veces pero el coste de O() viene a ser el mismo.

Generación de números primos: Miller-Rabin.

Finalmente, desarrollaremos un programa de generación de números primos mediante el algoritmo de Miller Rabin.

Para ello, previamente realizaremos unos pasos para generar un número del tamaño deseado y pasaremos una criba inicial para descartar algunos números. El algoritmo es el siguiente.

1: Generar un N aleatorio entre 0 y 2^n -1

2: Poner a 1 los bits mas y menos significativos para asegurar que sea impar y que el tamaño sea de n bits.

3: Dividir por la tabla de los 2000 primeros números primos precalculada en el archivo primeros2kprimos.c

4: Realizar propiamente el test Miller-Rabin. La versión de este algoritmo que hemos implementado es la que aparece en el Stallings. El pseudocódigo de este test es el siguiente.:

```
1. Find integers k, q, with k > 0, q odd, so that n-1 = 2^kq

2. Select a random integer a, 1 < a < n - 1

3. if a^q mod n = 1 then return inconclusive

4. for j = 0 to k - 1 do

5. if a^((2^j) * q) mod n = n-q or =n-1 then return inconclusive

6. if a^((2^j) * q) mod n = 1 then return composite

7. return composite
```

Estos pasos resultarían en la evaluación de si el número generado en los pasos 1 y 2 es compuesto o primo, con una posibilidad de error en este juicio dada la estimación del error de Miller Rabin, calculada en el programa para los parámetros de entrada, el tamaño del número en bits y el número de repeticiones del test MR.

Comentar que hemos encontrado diferencias entre el pseudocódigo que muestra el libro y el que tenemos en los apuntes. Mientras que en los apuntes solo comparamos si

a^((2^j)*q)mod n =1 o si es =n-1 en el libro solo se compara si la operación anterior es igual a n-q.

Como el objetivo es utilizar estas herramientas para elaborar un generador de números primos, volveremos al paso 1 del algoritmo en el caso de que Miller Rabin devuelva que es un número compuesto.

Ejemplo de ejecución.

```
make testprimo
primo generado ejecutable
ejecutando primo
./primo -b 1024 -t 8
Buscando numero primos de 1024 bits
es posible primo:
131325435008081370226227698732603062406380170526371643821204782
845395601498572265218631878074241205124171823541027502741439784
765937988451415944576957294835950188390539814861091009418470279
382102946963482547573386051902958248324516732198917368914727240
144693321043307731587476841502305500736081578196893702669

GMP:
131325435008081370226227698732603062406380170526371643821204782
845395601498572265218631878074241205124171823541027502741439784
765937988451415944576957294835950188390539814861091009418470279
382102946963482547573386051902958248324516732198917368914727240
144693321043307731587476841502305500736081578196893702669 es
probablemente primo

Se probaron 16 numeros antes de que se pasase el test de miller
rabin para 1024 bits y 8 reps

Probabilidad de que el test MR haya fallado: 0.0107143833739508
```

Finalmente, repetimos la ejecución con variando los parámetros b y reps para mostrar la probabilidad de fallo en la siguiente tabla:

B	Reps	Prob fallo
1024	8	0.0107143833739508
1024	16	0.0000001652591386
1024	30	0.0000000000000006
2048	16	0.0000003305182226
2048	30	0.0000000000000012
4096	16	0.0000006610362266

El parámetro t influye en que afina la precisión del test. Esto es, hace más difícil que un número pase la serie de pruebas sobre la primalidad que pase. Resulta en una dis - minución de la probabilidad de fallo.

Por otro lado, es más difícil encontrar primos más grandes, así que la probabilidad de fallo es superior, así como el número de veces que se tiene que repetir todo el algoritmo hasta encontrar uno.