

UNIVERSIDAD AUTÓNOMA DE MADRID



**FUNDAMENTOS DE CRIPTOGRAFÍA Y
SEGURIDAD INFORMÁTICA
(2020-2021)**

PRÁCTICA 1

Alba Ramos Pedroviejo
Ana María Cidoncha Suárez

Grupo 1461

Madrid, 28 de octubre de 2020

ÍNDICE DE CONTENIDOS

Aclaraciones previas	3
Sustitución monoalfabeto	3
Método afín	3
Aplicación práctica	4
Sustitución polialfabeto	6
Método de Vigenere	6
Criptoanálisis	7
Cifrado de flujo	13
Implementación	13
Criptoanálisis	14
Producto de criptosistemas permutación	15

Aclaraciones previas

Todos los programas desarrollados permiten introducir el mensaje a cifrar/descifrar tanto por línea de comandos (que es lo que se muestra en la mayoría de figuras de la memoria por simplicidad) como por ficheros, así como mostrar el resultado en pantalla o en fichero también.

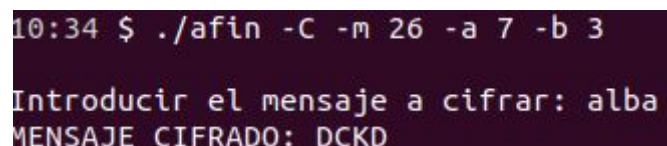
Sustitución monoalfabeto

Método afín

El programa *afin.c* cifra y descifra aplicando el cifrado afín sobre un texto que puede introducirse por pantalla o leerse de un fichero.

Antes de comenzar a cifrar o descifrar se debe comprobar que a y m sean coprimos, lo que determinará que la función de cifrado y descifrado sea inyectiva. Para ello se ha implementado la función *coprimos* en el fichero *euclides.c*, la cual calcula el máximo común divisor entre dos números siguiendo el algoritmo de euclides y devuelve un 1 si son coprimos (es decir, si su mcd es 1) o un 0 en caso contrario.

Para cifrar se ha creado la función *cifrar*, que transforma el texto original a un texto cifrado mediante el método afín: $ax_i + b \pmod m$. Para poder mostrar los resultados en formato ASCII printable, cada carácter se pasa a mayúsculas y se le resta 65, de forma que el ASCII de la A sea 0, el de la B 1, y así hasta $z = 25$: de esta forma tenemos los caracteres en Z_m . Sin embargo, es importante sumarle 65 a la hora de guardarlos e imprimirlos, para asegurar que estén mostrándose caracteres ASCII del alfabeto asociados a cada número resultante de las operaciones. En la figura 1 se puede observar un ejemplo de ejecución de esta función.



```
10:34 $ ./afin -C -m 26 -a 7 -b 3
Introducir el mensaje a cifrar: alba
MENSAJE CIFRADO: DCKD
```

Figura 1: cifrado con el programa *afin.c*.

Para descifrar se ha utilizado la función *descifrar* la cual transforma un texto cifrado en un texto plano. Para ello se ha despejado la x de la fórmula de cifrado obteniendo la fórmula: $x = (y - b) * a^{-1} \pmod m$, siendo a^{-1} el inverso multiplicativo de a . El inverso módulo m de $[a]$ es el número entero $[a^{-1}]$ de tal modo que $[a].[a^{-1}] \pmod m = 1$. Para calcular este inverso multiplicativo se utiliza la función *obtener_inverso* que utiliza el algoritmo de Euclides extendido. Nos aseguramos de que si $(y-b)$ es negativo le tenemos que sumar el módulo m para hacerlo positivo y poder calcular su módulo. En la figura 2 se puede observar un ejemplo de ejecución de esta función.

```
10:34 $ ./afin -D -m 26 -a 7 -b 3
Introducir el mensaje a descifrar: dckd
MENSAJE DESCIFRADO: ALBA
```

Figura 2: descifrado con el programa *afin.c*.

Aplicación práctica

El cifrado afín utiliza la función $y = ax + b$ para cifrar. Para desarrollar un método de cifrado basado en afín pero más robusto que éste, podemos utilizar el producto de criptosistemas. Sea $y_1 = \pi(x)$, donde $\pi(x)$ es una permutación de los m elementos del alfabeto, y $y_2 = ax + b \bmod m$, la composición $S_2(S_1)$ produce como resultado $y = a\pi(x) + b \bmod m$. En este nuevo criptosistema se aplica el cifrado afín sobre el mensaje al que previamente se le ha aplicado una permutación. De esta forma, el algoritmo es más robusto operacionalmente, ya que a la suma y el producto le añadimos la permutación. Asimismo, el número de claves posibles también aumenta: solo con el cifrado afín tendríamos $m \cdot \phi(m)$ claves debido a la combinación producto-desplazamiento, pero al añadirle también la permutación incluimos $m!$ claves posibles más.

Apliquemos este nuevo sistema al siguiente texto: EL PERRO DE SAN ROQUE. Si elegimos $\pi(x) = (26, 25, 24, \dots, 1)$, es decir, darle la vuelta al alfabeto, el texto queda VO KVIIL WV HZM ILJFV. Esta es la permutación por defecto en el programa *afin_ext.c*, pero el usuario puede introducir la suya como argumento. Es importante, en este punto, remarcar que el usuario debe introducir la permutación completa: si el usuario introduce “4 3 2 1”, el programa es capaz de completar el resto de permutación, pero en el caso “4 3 2 1 7 5” no podrá, por lo que el funcionamiento correcto del programa está limitado a que el usuario introduzca la permutación completa.

Ahora a cada carácter hay que aplicarle la función $ax + b \bmod m$, con $a = 7$ y $b = 2$, por ejemplo, y tomando $m = 26$ para el alfabeto español sin ñ. El texto cifrado finalmente es TW UTGGB AT ZVI GBNLT. Podemos ver la ejecución del programa para este ejemplo en la figura 3.

```
10:35 $ ./afin_ext -C -m 26 -a 7 -b 2
Introducir el mensaje a cifrar: el perro de san roque
MENSAJE PERMUTADO: VO KVIIL WV HZM ILJFV
MENSAJE CIFRADO: TW UTGGB AT ZVI GBNLT
```

Figura 3: cifrado con el programa *afin_ext.c*, que realiza un producto de criptosistemas y combina permutación y cifrado afín.

Nótese que para el descifrado simplemente basta cambiar el orden de aplicación de operaciones, descifrando primero el mensaje recibido y permutándolo después, como podemos ver que resulta en la figura 4.

```
10:37 $ ./afin_ext -D -m 26 -a 7 -b 2  
Introducir el mensaje a descifrar: TW UTGGB AT ZVI GBNLT  
MENSAJE DESCIFRADO: EL PERRO DE SAN ROQUE
```

Figura 4: descifrado con el programa afin_ext.c.

Si ahora queremos criptoanalizar el anterior texto para romper el cifrado que hemos creado, primero habrá que romper el cifrado afín y para ello es necesario conocer dos parejas (x_1, y_1) y (x_2, y_2) : así podremos resolver el sistema de dos ecuaciones con dos incógnitas formado por $y_1 = ax_1 + b \bmod m$ y $y_2 = ax_2 + b \bmod m$. Con esto podremos obtener a y b , pero todavía faltaría hallar qué permutación se ha utilizado, y hay $26!$ posibilidades. No sólo podríamos permutar una vez, sino que el algoritmo podría realizar varias permutaciones y todavía sería más robusto, añadiendo $m!$ claves cada vez. Sin embargo, para esta práctica se ha realizado una única permutación, puesto que ya es bastante robusto y la idea queda clara.

Nótese que es bastante complicado realizar un análisis de frecuencias de los caracteres sobre el texto cifrado sin conocer nada más. Por ejemplo, podríamos pensar que la T del texto cifrado se ha correspondido con una E porque es la que más se repite... Sin embargo, el texto permutado no contiene ninguna E. Otro ejemplo: podríamos pensar que la A se ha correspondido con una G en el texto cifrado, pues es la segunda que más se repite... Y nos equivocaríamos, ya que hay solo una A realmente en el texto original, y ninguna en el texto permutado. Como vemos, es bastante complicado atacar desde esta situación: trabajamos a ciegas en lo que a las frecuencias se refiere.

En vista del análisis anterior, viendo que realizar un análisis de frecuencias es bastante complicado y que, incluso conociendo dos parejas también quedaría hallar la permutación utilizada (eso si hemos logrado descubrir dos parejas), podemos concluir que el algoritmo creado es más robusto y complejo que el afín simplemente, ya que añade mayor complejidad operacional y más claves.

También habría sido posible aprovechar el uso de GMP, que permite trabajar con números muy grandes, para hacer un cifrado afín que cifre grupos de 2 o más caracteres. GMP dispone de la función *gen_primey*, contenida en el fichero *cipher/primegen.c* que genera números primos muy grandes en dos fases, primero generando un número aleatorio candidato y luego comprobando la primalidad mediante test sucesivos. Teniendo dos números coprimos muy grandes se complica la factorización, es decir, la descomposición de un número compuesto en divisores no triviales que cuando se multiplican dan el número original. Los casos más difíciles de factorizar son aquellos en los que los factores son dos números primos, elegidos al azar y de aproximadamente el mismo tamaño que sería lo que se podría conseguir usando la librería de GNU.

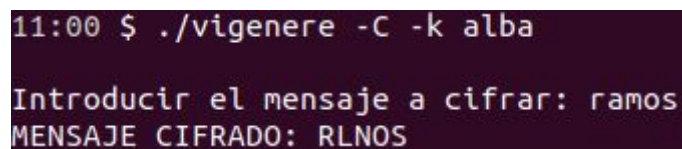
Sustitución polialfabeto

Método de Vigenere

El programa *vigenere.c* implementa el método de Vigenere. Para ello recibe una clave en formato cadena. El tamaño de dicha cadena, al que llamaremos n , constituirá el tamaño de bloques para cifrar y descifrar el texto.

La función *cifrar* utiliza Vigenere para cifrar un texto. Esta recorre el texto carácter a carácter y realiza la operación $x_i + k_i \bmod m$, donde $m = 26$ porque usamos el alfabeto español sin ñ. Para poder mostrar los resultados en formato ASCII printable, cada carácter se pasa a mayúsculas y se le resta 65, de forma que el ASCII de la A sea 0, el de la B 1, y así hasta $z = 25$: de esta forma tenemos los caracteres en Z_m . Sin embargo, es importante sumarles 65 a la hora de guardarlos e imprimirlos, para asegurar que estén mostrándose caracteres ASCII del alfabeto asociados a cada número resultante de las operaciones. Podemos ver un ejemplo del cifrado en la figura 5.

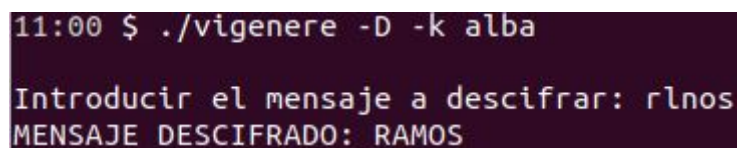
Hay que destacar que este algoritmo utiliza una clave que va repitiendo con periodo n , es decir, que ciframos el carácter x_i con la clave k_i . Sin embargo, cuando hayamos completado un bloque de n caracteres habrá que volver a comenzar a cifrar con k_1 : es por eso que, para cada carácter, comprobamos si hemos llegado al final del bloque y reiniciamos el contador que indexa la clave en caso positivo.



```
11:00 $ ./vigenere -C -k alba
Introducir el mensaje a cifrar: ramos
MENSAJE CIFRADO: RLNOS
```

Figura 5: cifrado con el programa *vigenere.c*.

Para descifrar seguimos el proceso inverso en la función *descifrar*: realizamos la operación $y_i - k_i \bmod m$ para cada carácter. Como en aritmética modular sólo se define la suma, la operación se transformará en $y_i + (m - k_i) \bmod m$. Todo este proceso se realiza siguiendo el formato de los ASCIIs explicado anteriormente, y podemos ver un ejemplo de ejecución en la figura 6.



```
11:00 $ ./vigenere -D -k alba
Introducir el mensaje a descifrar: rlnos
MENSAJE DESCIFRADO: RAMOS
```

Figura 6: descifrado con el programa *vigenere.c*.

Criptografía

El primer paso para criptoanalizar un cifrado de Vigenere es hallar la **longitud de la clave (n)** utilizada. Para ello existen dos alternativas:

- a) **Test de Kasiski:** el programa *kasiski.c* implementa este test. En primer lugar, se deben buscar subcadenas repetidas dentro del texto. Por defecto, una longitud de 3-4 caracteres es suficiente, pero se permite al usuario especificar la longitud deseada como argumento del programa. La función *buscarSubcadenas* es la encargada de realizar esta tarea: esta recorre carácter a carácter el texto (previamente pasado a un formato uniforme que únicamente contiene caracteres alfabéticos mayúsculos) y compara cada carácter con los posteriores. Cuando encuentra una coincidencia, compara lo siguiente de cada aparición para ver si la repetición tiene la longitud requerida. Es importante, en este punto, comprobar que no estamos ante un caso “aaaaaaaaa”, por ejemplo: se deben comprobar los índices para ver que se trate, efectivamente, de cadenas diferentes. Las cadenas encontradas se guardan en el array *cadena*s, y tienen asociado un contador de cuántas veces aparecen repetidas en el array *contadores*. Finalmente, se almacenan también las distancias entre cada aparición en el array *distancias*. Todos estos arrays se indexan por el índice que tenga cada cadena en el array *cadena*s.

Una vez tenemos este paso, tenemos que ver cuál es la cadena que más se repite de todas las encontradas y hallar el mcd de las distancias, ya que el test de Kasiski se basa en que $n \mid \text{mcd}(d_1, d_2, \dots, d_n)$. De hallar este mcd se encarga la función *obtenerTamanoClave*, que hace uso del algoritmo de euclides para hallar el mcd. El programa va mostrando por pantalla todo este proceso con el objetivo de ir viendo los pasos del algoritmo claramente. Finalmente, acaba mostrando el valor n encontrado.

En la figura 7 podemos ver un ejemplo de ejecución del programa para el texto CHR... que aparece en las diapositivas de Vigenere proporcionadas en el material docente. Podemos ver el proceso de hallar el mcd de todas las distancias: 5. Este es, muy probablemente, el tamaño de clave utilizado, y es correcto según las diapositivas.

```
09:56 $ ./kasiski -i kasiski.txt
---->Buscando cadenas de tamaño 3 en el texto...
+++++
La cadena que mas se repite es CHR (4 veces)
---->n|mcd(dist_1, dist_2, ... , dist_n)
mcd(165, 70) = 5
mcd(5, 40) = 5
mcd(5, 10) = 5
---->n = 5 segun Kasiski
```

Figura 7: ejecución sobre un texto donde mcd sale bien a la primera

Sin embargo, podría ocurrir que este mcd fuera 1 debido a que las distancias no son múltiplos entre sí. En este caso, esa subcadena no es correcta y probamos con otra, si hubiera, hasta conseguirlo. En la figura 8 podemos observar un ejemplo de un texto de El Quijote repetido varias veces, de forma que hay cadenas que no tienen distancias múltiplos y hay que ir probando. Finalmente, si en este proceso de repetición se acaban las subcadenas, el programa no habrá encontrado nada por este método y no podrá determinar el tamaño de clave.

```
10:09 $ ./kasiski -i quijote.txt
---->Buscando cadenas de tamaño 3 en el texto...
+++++
La cadena que mas se repite es AQU (6 veces)
---->n|mcd(dist_1, dist_2, ... , dist_n)
mcd(235, 202) = 1
Distancias no multiplos de n, volvemos a probar con otra cadena
+++++
La cadena que mas se repite es ALG (5 veces)
---->n|mcd(dist_1, dist_2, ... , dist_n)
mcd(154, 103) = 1
Distancias no multiplos de n, volvemos a probar con otra cadena
+++++
La cadena que mas se repite es QUE (5 veces)
---->n|mcd(dist_1, dist_2, ... , dist_n)
mcd(257, 30) = 1
Distancias no multiplos de n, volvemos a probar con otra cadena
+++++
---->n|mcd(dist_1, dist_2, ... , dist_n)
mcd(32, 91) = 1
Distancias no multiplos de n, volvemos a probar con otra cadena
+++++
La cadena que mas se repite es EST (5 veces)
---->n|mcd(dist_1, dist_2, ... , dist_n)
mcd(54, 229) = 1
Distancias no multiplos de n, volvemos a probar con otra cadena
+++++
La cadena que mas se repite es ABA (5 veces)
---->n|mcd(dist_1, dist_2, ... , dist_n)
mcd(48, 36) = 12
mcd(12, 44) = 4
mcd(4, 294) = 2
mcd(2, 277) = 1
Distancias no multiplos de n, volvemos a probar con otra cadena
+++++
La cadena que mas se repite es DEL (4 veces)
---->n|mcd(dist_1, dist_2, ... , dist_n)
mcd(70, 5) = 5
mcd(5, 70) = 5
mcd(5, 255) = 5
---->n = 5 segun Kasiski
```

Figura 8: ejecución sobre un texto donde hay que ir probando varias subcadenas porque las distancias no son múltiplos.

b) **Índices de Coincidencia:**

El Índice de coincidencia es la probabilidad de que dos letras tomadas al azar de un texto sean iguales. Si tomamos dos letras al azar de un texto en español, no seguidas si no de diferentes lugares del texto, es más probable que sean dos 'A' (probabilidad del 12%) a que sean dos 'N' (probabilidad del 7%). Esto significa que la probabilidad de obtener dos 'A' es $12 \cdot 12 / (100 \cdot 100) = 1,44\%$ y la de la 'N' sería $7 \cdot 7 / (100 \cdot 100) = 0,49\%$. La de obtener una 'A' y una 'N' es del $12 \cdot 7 / (100 \cdot 100) = 0,84\%$. Así pues tenemos que la probabilidad de que dos letras tomadas al azar de un texto con tamaño de N letras sean iguales, es decir, el Índice de Coincidencia es:

$$IC = \frac{n^{\circ} \text{ de pares de letras iguales}}{n^{\circ} \text{ de pares de letras posibles}} = \frac{\sum_{i=A}^Z \frac{f_i(f_i - 1)}{2}}{\frac{N(N - 1)}{2}} = \frac{\sum_{i=A}^Z f_i(f_i - 1)}{N(N - 1)}$$

En el lenguaje español, para un alfabeto de 26 letras, se obtiene un IC aproximado de 0,0775.

Cuando los textos son muy grandes el IC tiende a la sumatoria desde $i='A'$ hasta $i='Z'$ de p_i^2 siendo p la probabilidad de cada carácter.

Al ser el cifrado de Vigènere un cifrado de sustitución polialfabético con claves periódicas su vulnerabilidad reside en su naturaleza cíclica. Cuando se cifra en texto con sustitución monoalfabética el IC del texto en claro se mantiene en el texto cifrado mientras que con sustitución polialfabética, como es el de Vigènere, el IC en el texto cifrado será menor que en el texto en claro porque las frecuencias más altas de las letras del texto claro se reparten entre varias letras del criptograma. Por ejemplo, una 'A' en un texto cifrado podría corresponder a las letras 'G', 'E', 'A', 'Q', 'D', dependiendo del alfabeto que se use. Como el IC en el idioma español está próximo al 0,0775 únicamente obtendremos valores de IC próximos a 0,07 para la longitud de la clave realmente usada en el cifrado.

El programa *ic.c* implementado utiliza la fórmula anterior para calcular el IC de cada subcriptograma y compararlo con el IC español.

Se empieza calculando el IC de todo el texto y si el IC está próximo a 0,0775 la clave empleada podría tener longitud 1. Luego se forman dos subcriptogramas (C1 y C2) compuesto C1 por los caracteres 1º, 3º, 5º...del texto y el C2 por los caracteres 2º, 4º, 6º,...Igualmente si el IC fuese próximo a 0,0775 la clave empleada podría tener longitud 2 ya que los caracteres de C1 se habrían cifrado con el alfabeto correspondientes a la primera letra de la clave y los de C2 con el alfabeto correspondiente a la segunda.

Así se continúa de manera que luego se formarían tres subcriptogramas (C1, C2 y C3) compuestos el primero por los caracteres 1º, 4º, 7º..., el segundo por los caracteres 2º, 5º, 8º,... y el tercero por el 3º, 6º, 9º,... y calculando su IC para ver si estuviesen próximos a 0,0775.

Se han dividido hasta $\text{tam_clave}=15$, es decir, 15 subcriptogramas para comparar cual de los tamaños estaría más próximo a ese valor de IC español y así poder conseguir la longitud de la clave.

En la figura 9 podemos ver algunos ejemplos de cifrado de texto con distintas longitudes de clave (clave 'quijote' con 7 y 'cripto' con 6) y el resultado del programa

```
ana@ana-UX305LA:~/Escritorio/P1_kasiski$ ./vigenere -c -k quijote -i quijote.txt
-o quijote_v_c.txt

ana@ana-UX305LA:~/Escritorio/P1_kasiski$ ./ic -i quijote_v_c.txt
+++++ Indice de Coincidencia +++++
Tamaño clave=7
ana@ana-UX305LA:~/Escritorio/P1_kasiski$ ./vigenere -c -k cripto -i quijote.txt
-o quijote_v_c.txt

ana@ana-UX305LA:~/Escritorio/P1_kasiski$ ./ic -i quijote_v_c.txt
+++++ Indice de Coincidencia +++++
Tamaño clave=6
ana@ana-UX305LA:~/Escritorio/P1_kasiski$
```

Figura 9: Ejemplos de obtención del tamaño de la clave

En cuanto se ha conseguido un tamaño cuya diferencia con el ic medio español ha sido menor de 0.01 nos hemos quedado con esa clave pues, según va aumentando la división del texto en subcriptogramas, es decir, más columnas con menos caracteres, llega un momento en que cada letra del alfabeto ocurre con la misma frecuencia (1/26) por lo tanto el IC se aproxima al 0,037 de un texto aleatorio, esto es, un texto que no sigue un patrón regular.

Como el cifrado de Vigènere utiliza una clave que va repitiendo con periodo n , es decir, que ciframos el carácter x_i con la clave k_i y los ataques se basan en la detección de esas repeticiones, si utilizamos una clave del tamaño del texto o incluso de una longitud mayor conseguimos que no se repita y el ataque de Kasiski no nos serviría. Podría atacarse por el método de análisis de las frecuencias o índice de coincidencia de los caracteres si el mensaje es lo suficientemente largo. Para evitarlo se podría usar una clave compuesta de caracteres aleatorios (cifrado de Vernam) y el mensaje se volvería completamente inatacable por cualquier método de criptoanálisis (seguridad incondicional).

Una vez tenemos el tamaño de la clave, procedemos a hallar cada componente k_i de la misma. En el fichero *obtenerClave.c* se implementa un programa basado en los índices de coincidencia visto en clase y que permite obtener la clave del cifrado Vigenere si el tamaño de clave asumido es correcto. Para ello se reorganiza el texto cifrado en n bloques, con n igual al tamaño de la clave, que conformarán los vectores y_1, y_2, \dots, y_n . Cada uno de estos vectores se asume que estará cifrado con la misma clave k_1, k_2, \dots, k_n , que son letras del alfabeto. Es decir, el texto se divide en bloques de tamaño n y el vector y_i estará formado por todos los caracteres $y_{i,j}$ siendo estos los caracteres i de cada bloque j en que dividimos el texto.

Cada uno de los vectores y_i está cifrado con la clave k_i y conserva la estadística del lenguaje español, que es el que se está utilizando y del que tenemos tanto las letras como las frecuencias de cada una (proporcionadas en la propia práctica). Al descifrar, tendremos un vector $x_i = y_i - k_i \pmod{m}$. Iremos guardando las frecuencias de cada carácter descifrado en un vector de frecuencias llamado *frecuenciasDescifrado*, y una vez

que tengamos recorrido todo el vector x_i transformaremos esas frecuencias al formato de los índices de coincidencias explicados en clase: $IC(x_j) = \sum [(P_i * F_{i+k_j}) / \text{lenTxt}/n]$.

Una vez tenemos el formato preparado, si la letra probada para k_i es la correcta, las frecuencias de los caracteres de x_i serán similares a las del alfabeto español. Pero... ¿Cómo cuantificar “similar”? Deberíamos quedarnos con aquella letra que proporcione las frecuencias más similares a las del alfabeto. Para ello podríamos tener un umbral de similitud, de forma que si este se supera se toma la letra que lo hace. Sin embargo, dos letras podrían superar el umbral y entonces no se sabría cual es más correcta. Para ser más precisos, por cada k_i probada calcularemos la distancia vectorial entre las frecuencias obtenidas por nosotros y las del alfabeto español proporcionadas en la práctica. Nos quedaremos con la k_i que más reduzca esta distancia entre ambos vectores, y repetiremos el proceso para el resto de componentes de la clave.

En la figura 10 podemos observar un ejemplo de ejecución del cifrado de Vigenere con clave “cripto” sobre un texto introducido por el usuario. Si aplicamos el criptoanálisis sobre este texto cifrado indicando que la longitud de la clave es 6, la figura 11 muestra el proceso de criptoanálisis explicado anteriormente, y se aprecia que el programa devuelve la clave.

```
11:10 $ ./vigenere -C -k cripto
Introducir el mensaje a cifrar: pues nada aqui estoy probando esta ultima version de la funcion a ver si consigo que todo vaya sobre ruedas y bueno es
ta asignatura no me la esperaba yo tan teorica ni tan matematica de verdad es que es como si se me fuera a quedar frito el cerebro yo no se si me van
a quedar neuronas vivas al acabar la carrera
MENSAJE CIFRADO: RLMH GOFR IFNW GJBDR DTFJGRQ VAIT INKQBT JGIAHXB FV TP YIPTQDG O XVZ HB QQEAXZC SLM IHRQ MINT GQSZT KIGUIH R PWVVD XGVR IHBUPRBJKO P
F UT EO GJXTKODR GD MOP KMDKWER VX MOP DIIXACKQRT RG MMGWOF VA FNS GJ KDFC UZ AT FS HLMGT O SLMSTF HIQIH SN TMGXPTF GD GC UV AX FS XRV P JIGUIG GSWIWC
TG XZDPL ON RKPQUT CI RTFTVZP
```

Figura 10: cifrado Vigenere con clave “cripto”


```
11:17 $ ./obtenerClave -n 6
```

```
Introducir el mensaje sobre el que obtener la clave: RLMH GOFR IFNW GJBDR DTFJPGRQ VAIT INKQBT JGIAHXB FV TP YIPTQDG O XVZ HB QQEAXZC SLM IHRQ MINT GQ  
SZT KIGUIH R PWVVD XGVR IHBUPRBJKO PF UT EO GJXTKODR GD MOP KMDKWER VX MOP DIIXACKQRT RG MMGNOF VA FNS GJ KDFC UZ AT FS HLMGT O SLMSTF HIQIH SN TMGXPT  
F GD GC UV AX FS XRV P JIGUIG GSWIWCTG XZDPL ON RKPUOT CI RTFTVZP
```

```
Buscando k_0
```

```
Buscando k_0...
```

```
Si k_0 = A, distancia obtenida = 28.750381 y menor = -1.000000  
---> Tomamos k_0 = A  
Si k_0 = B, distancia obtenida = 28.697701 y menor = 28.750381  
---> Tomamos k_0 = B  
Si k_0 = C, distancia obtenida = 28.592897 y menor = 28.697701  
---> Tomamos k_0 = C  
Si k_0 = D, distancia obtenida = 28.681515 y menor = 28.592897  
Si k_0 = E, distancia obtenida = 28.707211 y menor = 28.592897  
Si k_0 = F, distancia obtenida = 28.693316 y menor = 28.592897  
Si k_0 = G, distancia obtenida = 28.707653 y menor = 28.592897  
Si k_0 = H, distancia obtenida = 28.757076 y menor = 28.592897  
Si k_0 = I, distancia obtenida = 28.759163 y menor = 28.592897  
Si k_0 = J, distancia obtenida = 28.739250 y menor = 28.592897  
Si k_0 = K, distancia obtenida = 28.778105 y menor = 28.592897  
Si k_0 = L, distancia obtenida = 28.707354 y menor = 28.592897  
Si k_0 = M, distancia obtenida = 28.680723 y menor = 28.592897  
Si k_0 = N, distancia obtenida = 28.707294 y menor = 28.592897  
Si k_0 = O, distancia obtenida = 28.716213 y menor = 28.592897  
Si k_0 = P, distancia obtenida = 28.647421 y menor = 28.592897  
Si k_0 = Q, distancia obtenida = 28.709793 y menor = 28.592897  
Si k_0 = R, distancia obtenida = 28.738997 y menor = 28.592897  
Si k_0 = S, distancia obtenida = 28.698614 y menor = 28.592897  
Si k_0 = T, distancia obtenida = 28.670092 y menor = 28.592897  
Si k_0 = U, distancia obtenida = 28.745970 y menor = 28.592897  
Si k_0 = V, distancia obtenida = 28.727995 y menor = 28.592897  
Si k_0 = W, distancia obtenida = 28.763620 y menor = 28.592897  
Si k_0 = X, distancia obtenida = 28.752750 y menor = 28.592897  
Si k_0 = Y, distancia obtenida = 28.739511 y menor = 28.592897  
Si k_0 = Z, distancia obtenida = 28.738962 y menor = 28.592897
```

```
Buscando k_5...
```

```
Si k_5 = A, distancia obtenida = 28.674318 y menor = -1.000000  
---> Tomamos k_5 = A  
Si k_5 = B, distancia obtenida = 28.690525 y menor = 28.674318  
Si k_5 = C, distancia obtenida = 28.695124 y menor = 28.674318  
Si k_5 = D, distancia obtenida = 28.684736 y menor = 28.674318  
Si k_5 = E, distancia obtenida = 28.696953 y menor = 28.674318  
Si k_5 = F, distancia obtenida = 28.735106 y menor = 28.674318  
Si k_5 = G, distancia obtenida = 28.722446 y menor = 28.674318  
Si k_5 = H, distancia obtenida = 28.774616 y menor = 28.674318  
Si k_5 = I, distancia obtenida = 28.756266 y menor = 28.674318  
Si k_5 = J, distancia obtenida = 28.780970 y menor = 28.674318  
Si k_5 = K, distancia obtenida = 28.636696 y menor = 28.674318  
---> Tomamos k_5 = K  
Si k_5 = L, distancia obtenida = 28.706146 y menor = 28.636696  
Si k_5 = M, distancia obtenida = 28.756069 y menor = 28.636696  
Si k_5 = N, distancia obtenida = 28.742512 y menor = 28.636696  
Si k_5 = O, distancia obtenida = 28.562996 y menor = 28.636696  
---> Tomamos k_5 = O  
Si k_5 = P, distancia obtenida = 28.727253 y menor = 28.562996  
Si k_5 = Q, distancia obtenida = 28.741283 y menor = 28.562996  
Si k_5 = R, distancia obtenida = 28.739891 y menor = 28.562996  
Si k_5 = S, distancia obtenida = 28.715082 y menor = 28.562996  
Si k_5 = T, distancia obtenida = 28.794952 y menor = 28.562996  
Si k_5 = U, distancia obtenida = 28.721964 y menor = 28.562996  
Si k_5 = V, distancia obtenida = 28.752228 y menor = 28.562996  
Si k_5 = W, distancia obtenida = 28.716070 y menor = 28.562996  
Si k_5 = X, distancia obtenida = 28.716356 y menor = 28.562996  
Si k_5 = Y, distancia obtenida = 28.721462 y menor = 28.562996  
Si k_5 = Z, distancia obtenida = 28.746176 y menor = 28.562996
```

```
La clave es: CRIPTO
```

Figura 11: proceso de criptoanálisis de cifrado de Vigenere.

Cifrado de flujo

Implementación

En esta ocasión hemos decidido implementar el algoritmo RC4, ya que lo hemos visto como un ejemplo en clase y nos ha parecido interesante porque, si bien está atacado, hay protocolos que lo siguen utilizando (como WEP, WPA o BitTorrent) y nos parecía adecuado conocerlo por eso mismo.

El programa flujo.c implementa el algoritmo RC4. En el caso de que queramos cifrar, el programa espera recibir una cadena de texto. En el caso de que queramos descifrar, espera recibir una cadena hexadecimal, la cual será procesada para convertir cada par de caracteres ASCII al número que realmente representan (porque "1A" no representa lo mismo que $1A_{16}$: uno es texto y el otro es el número en base 16 como tal, y nosotros queremos pasar del texto al número). El cifrado y descifrado se han probado utilizando los ejemplos de referencia de Wikipedia: https://en.wikipedia.org/wiki/RC4#Test_vectors.

El algoritmo RC4 consta de dos fases:

- **Key Scheduling Algorithm (KSA):** en esta fase se inicializa un vector de estado de 256 bytes denominado S. Se inicializa como permutación identidad y después se permuta 256 veces mezclando bytes de la clave. Este vector será el que se utilice para generar la secuencia de claves cifrantes.
- **Pseudo-Random Generation Algorithm (PRGA):** este algoritmo se realiza cada vez que necesitemos cifrar o descifrar, es decir, para cada carácter del texto recibido. Es la parte en la que se genera el flujo de claves cifrantes a partir del vector de estado S y de dos índices i y j. El vector de estado es modificado también mediante una permutación similar a la realizada en la parte de KSA, solo que esta vez no se introducen bytes de la clave: solo permuta el propio vector de estado y de él extrae cada clave de la secuencia de forma pseudo-aleatoria. Una vez tenemos la clave que usaremos, se realiza la XOR bit a bit de ese byte de clave y el byte a cifrar/descifrar correspondiente. La operación es la misma tanto para cifrar como para descifrar ya que son operaciones a nivel de bit.

En nuestra implementación hemos decidido ir mostrando por pantalla la secuencia de claves cifrantes, aunque esto en una implementación de producción no se realizaría: simplemente ha sido para facilitar la depuración y la comprensión del proceso que realiza el cifrado de flujo. Podemos ver un ejemplo en la figura 12, que muestra uno de los ejemplos de la Wikipedia de los mencionados anteriormente.

```
11:13 $ ./flujo -C -k Wiki
Introducir el mensaje a cifrar: pedia
Flujo de claves: 60 44 db 6d 41
Resultado: 10 21 bf 04 20
```

Figura 12: cifrado RC4 con el programa flujo.c.

Podemos comprobar que el programa descifra correctamente comparando nuestra salida con la de algún cifrador/descifrador online. En la figura 13 podemos ver nuestra salida y compararla con la salida generada por el recurso de la figura 14, observando que coinciden y que, por tanto, el programa es correcto.

```
11:13 $ ./flujo -D -k Wiki
Introducir el mensaje a descifrar en formato hexadecimal y con ceros rellenando
(ej. en vez de 1A B 3D poner 1A 0B 3D): 10 21 bf 04 20
Flujo de claves: 60 44 db 6d 41
Resultado: 70 65 64 69 61
```

Figura 13: descifrado utilizando el programa flujo.c

The screenshot shows an online encryption tool with three main sections: 'VIEW', 'ENCODE/DECODE', and another 'VIEW'. The first 'VIEW' section is set to 'Bytes' and shows the input '10 21 bf 04 20'. The 'ENCODE/DECODE' section is set to 'RC4' and shows a key of '57696b69' (hex for 'Wiki'). The 'DROP BYTES' is set to 0. The output shows 'Encoded 5 bytes'. The second 'VIEW' section is set to 'Ciphertext' and shows the output '70 65 64 69 61'.

Figura 14: comprobación del descifrado utilizando un recurso online y habiendo transformado la clave a formato hexadecimal.

Criptoanálisis

RC4 es el algoritmo utilizado por WEP, WPA o BitTorrent con el objetivo de proporcionar seguridad a un mensaje. De hecho, durante un tiempo, RC4 fue el único algoritmo resistente frente al ataque **Beast Attack**, descubierto en 2011 y que aprovechaba una vulnerabilidad presente en el cifrado de bloques en modo CBC utilizado en los protocolos SSL y TLS. Sin embargo, a partir del 2013 se descubrieron otras vulnerabilidades que hicieron que, actualmente, el uso de RC4 en TLS esté prohibido por el **RFC 7465** y los principales navegadores lo hayan eliminado. A continuación revisaremos la literatura y expondremos los principales ataques realizados al algoritmo, analizando en qué se basan para conocer sus puntos débiles.

Fluhrer y McGrew publicaron en 2001 un paper en el que analizaron el flujo de claves que crea RC4 y vieron que los primeros bytes están sesgados. Este descubrimiento dio pie a que se publicara el ataque de **Fluhrer, Mantin y Shamir**: a partir de las vulnerabilidades descubiertas previamente descubrieron que es posible determinar bytes del vector de estado S y que era posible recuperar la clave al analizar grandes cantidades de mensajes encriptados con la misma. Así lograron romper el protocolo WEP, pero no SSL, el cual genera claves distintas según la sesión. Estos sesgos son la base de los numerosos ataques sufridos por el algoritmo, de los que analizaremos algunos de los más conocidos.

El ataque **NOMORE**, descubierto en 2015 y demostrado de forma práctica en <https://www.rc4nomore.com/>, determina que es posible descryptar contenido enviado repetidamente vía RC4, por ejemplo cookies HTTP: el atacante inyecta código malicioso al navegador de la víctima de forma que este envíe repetidamente su cookie. De esta forma, el atacante puede hacer un análisis basado en los ataques de Fluhrer y Mantin y descryptar el contenido. Si este contenido es una cookie, el atacante ha logrado hacerse con la identidad de la víctima en alrededor de 75 horas. Este ataque también afecta a WPA-TKIP basándose en cómo combina el algoritmo la clave secreta con el vector de inicialización público.

Finalmente, destacaremos que en 2013 **Bernstein, Paterson, Poettering y Schuld** utilizaron los sesgos explicados anteriormente para romper RC4 cuando se utiliza en SSL/TLS. Esto fue registrado como **CVE-2013-2566**.

Producto de criptosistemas permutación

En el programa *permutacion.c* se implementa un producto de criptosistemas permutación que recibe dos claves, k_1 y k_2 , donde k_1 permutará filas y k_2 permutará columnas del texto colocado en forma de matriz. La dimensión de la matriz será $m \times n$, donde m es el tamaño del vector k_1 y n el tamaño del vector k_2 .

El texto recibido se colocará en forma de matrices de $m \times n$ hasta su final, y si sobran celdas se rellenarán con el carácter '.' (existen protocolos de *padding*, pero en esta ocasión simplemente lo añadiremos al final). Dichas celdas deberán permutarse según el formato explicado en la práctica. Es importante notar que la explicación es de cara al cifrado, y que para descifrar es necesario invertir la permutación de k_1 y k_2 . Por ejemplo, si ciframos con $k_1 = 231$ y $k_2 = 21$, como se muestra en la figura 15, debemos descifrar con la inversa, es decir, con $k_1 = 312$ y $k_2 = 21$, como se muestra en la figura 16. En ambos resultados vemos el padding explicado anteriormente.


```

11:44 $ ./permutacion -C -k1 231 -k2 21
Matriz de 3x2
231
21
Introducir el mensaje a cifrar: albaeshermosaole

Construida matriz
a  a
l  e
b  s

Permutada matriz:
e  l
s  b
a  a

Construida matriz
h  m
e  o
r  s

Permutada matriz:
o  e
s  r
m  h

Construida matriz
a  e
o  .
l  .

Permutada matriz:
.  o
.  l
e  a
MENSAJE CIFRADO: esalbaosmerh..eola

```

Figura 15: cifrado mediante doble permutación

```

11:51 $ ./permutacion -D -k1 231 -k2 21
Matriz de 3x2
231
21
Introducir el mensaje a descifrar: esalbaosmerh..eola

Modificado k1: 312
Modificado k2: 21
Construida matriz
e   l
s   b
a   a

Permutada matriz:
a   a
l   e
b   s

Construida matriz
o   e
s   r
m   h

Permutada matriz:
h   m
e   o
r   s

Construida matriz
.   o
.   l
e   a

Permutada matriz:
a   e
o   .
l   .
MENSAJE DESCIFRADO: albaeshermosaole..

```

Figura 16: descifrado mediante doble permutación