Criptografía y seguridad informática

Práctica 1

Grupo 4

Mario Valdemaro García Roque
Alberto Cabello Álvarez

Consideraciones sobre la entrega y el código

La entrega se encuentra estructurada en subdirectorios con el contenido de la práctica. En **src** se encuentran los fuentes principales para la creación de cada programa y los ficheros para crear el filtro. En **lib** hay una librería sobre tablas hash utilizada en la parte de criptoanálisis. Esta librería está tomada de https://github.com/watmough/jwHash

Se puede bajar fácilmente usando el comando:

git clone https://github.com/watmough/jwHash.git

Comentar sobre este código que se han tenido que añadir unas cuantas funciones, ya que no estaban desarrolladas, por tanto ambos ficheros no son exactamente iguales, solo cambian en el detalle de que hay 2 o 3 funciones mas para insertar y buscar.

Este código se compilara en una librería que una vez ejecutado el **make all** guardara un archivo .a en el directorio **lib**.

Aparte de estos directorios se encuentran los directorios **includes** donde están todos los ficheros .h, que son los de la librería ya mencionada y los necesarios para crear el filtro, y los directorios **obj** donde se compilan todos los programas (una vez usados se borran automáticamente) y el directorio **doc** donde se encuentra este documento.

Para compilar, utilizar make sobre el directorio raíz. En ese directorio se generarán todos los ejecutables necesarios. Estos son los ejecutables propios de cada programa más un filtro programado en flex para transformar los ficheros de entrada al formato deseado (todo letras minúsculas sin caracteres extraños o espacios).

Cabe mencionar que este filtro no es necesario ejecutarlo a mano. Cada programa que recibe texto de fichero tiene una llamada al filtro con system para limpiar el texto de entrada. Esto creará un nuevo fichero del tipo “Fx” donde x es el nombre del fichero original. Por ejemplo, en nuestras pruebas, utilizando el man de gcc (man gcc > gcc.txt) se crea un “Fgcc.txt” al llamar a un programa con ese fichero como parámetro.

El método de ejecución de cada programa corresponde al método indicado en el guión de la práctica, utilizando getopt para el control de parámetros. Se puede consultar este método ejecutando sin parámetros o con parámetros incorrectos, en cuyo caso se detendrá y mostrará la ejecución correcta. Esto es cierto para todos salvo para el programa de criptoanálisis, que tiene un flag, -nl (no limit), que al emplearlo no parará la ejecución para tamaños de clave superiores a 32, que con el método estadístico usado re-

sulta lento para tamaños de clave superiores (Consultar apartado correspondiente para más información).

En el makefile se proporcionan targets como ejemplos de entrada para cada programa, para testeo rápido de los mismos, del tipo:

```
testafin:

	./afin -C -m 26 -a 17 -b 24 -i gcc.txt -o afinc.txt

	./afin -D -m 26 -a 17 -b 24 -i afinc.txt -o afind.txt
```

1) Sustitución Monoalfabeto.
    - Método afin.
    - Aplicación método afín, con flujo de claves basado en una semilla.
2) Sustitución Polialfabeto.
    - Método de Vigenere.
    - Criptoanálisis del Vigenere.
3) Método de transposición.
4) Producto de criptosistemas permutación.

Método afín:

En cuanto al método afín comentaremos que los principales problemas que nos hemos encontrado han sido al usar la librería gmp ya que tiene algún que otro fallo, como que mpz_inits y mpz_clears, no funcionan muy bien y por tanto muchas de las variables que las podriamos haber reservado o liberado de una linea las hemos tenido que reservar en varias. No entendemos muy bien este fallo quizá sea por fallo nuestro.

El código presenta una función llamada inversa que es la que calcula de el valor inverso de un número y una base dados. El método por el que se calculan estos valores es mediante Euclides extendido. Si se le pasase a esta función un número para el que no hay inverso entonces el programa terminara indicando este fallo.

Alguna consideración que hay que tener en cuenta es que es muy recomendable pasarle las pruebas con textos en ingles, primero porque si se le pasa un fichero el filtro que hemos desarrollado va a eliminar o convertir (mayúsculas) todos los caracteres que reconozca como extraños y operara sobre ese archivo, y segundo porque para hacer bien los módulos hemos tenido mover las letras del ASCII hasta la referencia de modulo adecuada, es decir mover los caracteres ASCII 97-122 a la zona 0-26.

Método afín mejorado:

Para llegar al cifrado escogido, primero ponderamos diferentes otras opciones. Una manera intuitiva y segura hubiese sido realizar producto de criptosistemas, pero tras discutirlo, acabamos decidiendo hacer un método similar a los de flujo de claves.

El parámetro para cifrar de este método o llave será una semilla que pasar a srand(). Cifraremos letra a letra escogiendo los parámetros a y b para el afín a través de rand(). De esta manera, se cifra cada carácter con parámetros diferentes, pero siguiendo siempre el mismo orden de cifrado y descifrado para la misma semilla.

Para cifrar se elige el parámetro 'a' según rand entre un array con todos los inversos posibles para zm (26). 'b', al igual se elige entre 0-25.

```
mpz_set_si(a, inversos[rand() % ninversos]);

mpz_set_si(b, rand() % 26);
```

El procedimiento es idéntico para el descifrado salvo que hay que operar con el inverso multiplicativo de a, que se obtiene de la siguiente manera.

```
mcd=inverso(a,zm);

mpz_set(inv, mcd[1]);

if(mpz_sgn(mcd[0]) == -1){

     mpz_sub(inv, inv, mcd[1]);

     mpz_sub(inv, inv, mcd[1]);

     mpz_add(inv, inv, zm);

}
```

El propósito de la condición del if es negar el inverso(mcd[1]) y sumarle 26. Por algún motivo, la función inverso, al aplicarla para gmp devuelve en ocasiones el mcd como negativo, cambiando al mismo tiempo el signo del inverso. Esto no ocurre así al aplicar la función inverso para enteros. Creemos que es por la manera de gmp de tomar módulos.

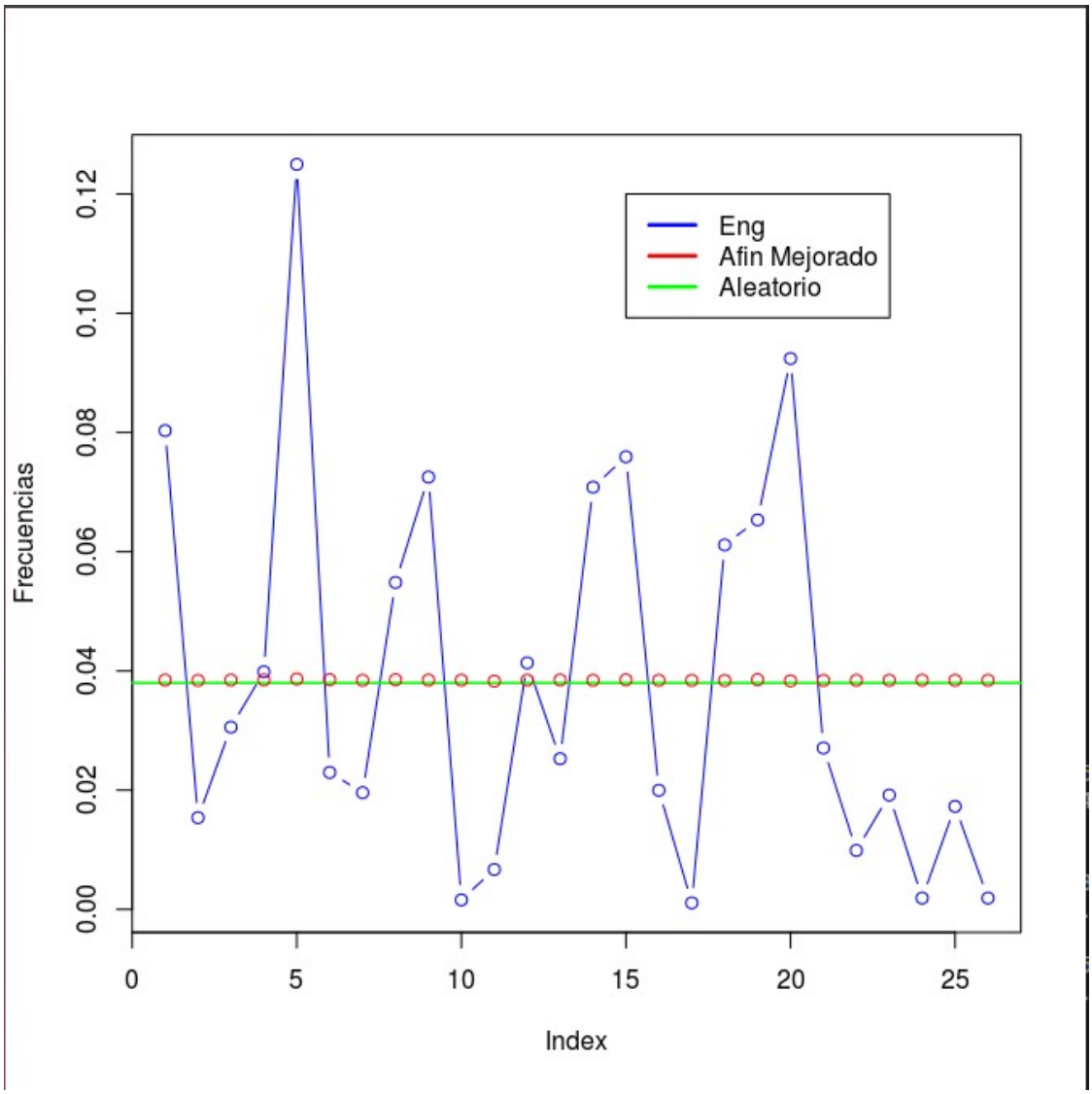
Este método soluciona varios problemas del afín, ya que en la versión simple una letra siempre se cifra de una misma manera. Al variar los parámetros se pierde la trazabilidad sencilla de cada carácter, rompiendo la estadística.

También se amplía el espacio de claves. Se pasa de 26*Zm* al intervalo de todas las semillas posibles para srand(), MAX_INT, esto es, 2^{31}

En el siguiente gráfico se pueden ver las frecuencias de aparición de cada carácter para varios modelos, para el inglés, el modelo de lenguaje aleatorio teórico y las frecuencias obtenidas tras encriptar con este método un texto.

Se realizaron dos pruebas, encriptando una misma cadena con diferentes semillas (de 1 a 10^6) y otra encriptando un texto grande. En ambos casos las frecuencias son muy similares, así que se representa solo una vez.

Se puede ver que la estadística se pierde completamente.


Eng
Afin Mejorado
Aleatorio
Frecuencias
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0
5
10
15
20
25
Index

La debilidad de este sistema reside en el rand() de c. Desconocemos si se puede analizar de alguna manera o predecir su salida. Frente al ataque a fuerza bruta, dependerá de si se puede probar a tiempo razonable 2^{31} posibilidades de claves y analizar su salida. Aún así, la mejora frente al afín en este aspecto es sustancial.

Opciones para mejorarlo aún más sería utilizar un generador de números aleatorios más sofisticado, o tal vez combinarlo con otro criptosistema, como el de permutación.

Método de Vignere:

En cuanto a los errores que nos hemos podido encontrar a la hora de realizar este cifrado, comentar que no nos hemos encontrado casi ninguno ya que es un método muy sencillo, quizá lo mas complicado de manejar es comprobar que estas manejando bien las letras por las que sustituyes ya que al ser ASCII hay que hacer una conversión, cosa que ya hemos comentado en otros apartados así que no nos vamos a entretener en este aspecto.

Recordar que si se usa la entrada sin fichero usar letras de la a-z (ingles) y en minúscula. Para la entrada de fichero como siempre hay un filtro que limpiara de caracteres inválidos el texto.

Criptoanálisis de Vignere:

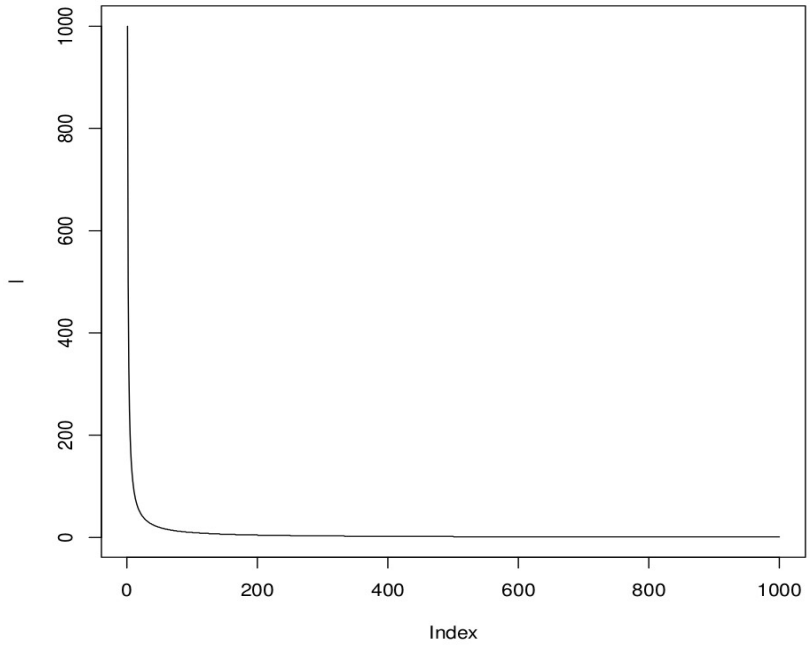
Para desarrollar este programa hemos usado 3 funciones principales que son los 3 apartados que se nos pide, **kasiski** (obtiene el tamaño de la clave), **IC** (obtiene el tamaño de la clave) e **ICclave** (obtiene la clave). Kasiski la discutiremos al final de este apartado ya que es la función es algo mas complicada que las otras 2 y además tiene mucha relación con las preguntas que se nos realizan sobre este apartado.

La función **IC** el tamaño de una clave con la que un texto **puede** estar cifrado, lo que realiza es que realiza ciertas hipótesis sobre el tamaño de la clave, con la flag desactivada comprueba tamaños de clave del 1 al 32, sino comprueba con tamaños de clave de hasta el tamaño del fichero, por eso le hemos puesto un limite. La forma en que realiza este método es ver todos los posibles tamaños de la clave y por cada clave calcular los índices de coincidencia que se han encontrado para esa clave y si la calve se aproxima mucho al índice del ingles guardarlo como máximo si el IC que hayamos calculado para un tamaño de clave menor se acerca menos al IC del ingles. Un detalle que hemos remarcado al inicio de este párrafo es el “puede” ya que este método devuelve un múltiplo del tamaño de clave original,es decir puede devolver el tamaño de la calve 2,3,.. veces. Esto se debe que tanto una clave como otra son validas para cifrar y descifrar, ya que es lo mismo cifrar con “a” que con “aa”.

La función **ICclave** obtiene una clave con la que se puede descifrar el texto dado el tamaño de esta, por defecto usa el tamaño obtenido en IC. El método por el que se obtiene esta clave es muy parecido al anterior solo que esta vez la palabra con la que se prueba ya la tenemos, así que ahora lo que mas nos interesa es ver para ese carácter en concreto cual es la letra que mas se ha repetido. Una vez obtenido esta letra si vemos la distancia que hay entre esta letra y la "e", sea esta distancia "n", el carácter para ese elemento de la clave sera la letra del abecedario que ocupe la posición "n".

La función **kasiski** calcula el tamaño de la clave usando este método, pero hay un problema, no con la función sino con este método y es que hay cierta probabilidad de que si vas probando y coges una cadena aleatoria (de tamaño n) y haces el mcd de la distancia de las 2 siguientes claves que haya en el texto, no te salga el tamaño de la clave sino un valor aleatorio. Esto se debe a que Kasiski mira si una palabra cifrada se encuentra en el texto en otra zona dando por supuesto que ambas palabras se descifraran igual, pero esto no es cierto ya que puede haber zonas del texto que coincidan pero no se descifren formando la misma palabra, por tanto hay una cierta probabilidad de que esto ocurra.

Como muestra hemos calculado la siguiente gráfica:



Para generar esta función hemos calculado mcd(1a100,1a100) donde el eje “x” son los posibles valores y el eje “y” las veces que se repiten estos valores.

Esta gráfica nos viene a decir por cada x que valor de error esperamos que tenga.

Como se puede ver es muy probable que si usamos una palabra corta obtengamos mas fallos. Así que nos ha parecido un factor importante a tener en cuenta a la hora de elegir el tamaño de la clave que devolverá nuestra función ya que como se observa si se analizamos el texto con una palabra corta es probables que nos encontremos con muchos errores que nos pueden romper la predicción. La solución que hemos elegido para este problema es generarnos todas las posibles palabras que se encuentran en el texto y vamos calculando todos los posibles resultados que nos devuelve Kasiski (esto hace que la función sea algo lenta), de todas esas posibles ocurrencia cogemos el dato que tiene la mayor ocurrencia menos la estimación que hacemos de errores que puede haber en el texto, esta estimación es la que obtenemos en la gráfica que es el tamaño del fichero entre el tamaño de la calve. Y gracias a este método podemos decir que estadísticamente vamos a devolver el tamaño de clave mas probable.

A continuación hacemos la siguiente prueba, con un fichero de unos 500KB ciframos el texto con una clave de tamaño 4 (abcd) y otra vez con una clave de tamaño 10 (lali-

lulelo) y obtenemos que los tamaños de clave que mas encontramos en el texto realizando el método de kasiski son los siguientes:

Clave de tamaño 10		Clave de tamaño 4	
Tamaño de clave	Repeticiones	Tamaño de clave	Repeticiones
10	1265156	4	7975363
20	317553	8	2010845
30	140714	12	885928
40	79154	16	492177
50	53794	20	317553
60	34647	24	220529
70	25631	28	166753
80	19088	32	126338
90	16147	36	100154
100	13316	30	79154
110	10340	44	69511

Como podemos observar las calves mas probables de ambos textos son las del tamaño real, pero nos parecía interesante poner las otras también, ya que en verdad cualquiera de estos tamaños nos dará una clave valida.

En cuanto al tamaño de la clave se puede observar como si la clave es mas pequeña es mas posible que encontremos mas fallos ya que si nos vamos a la gráfica que hemos puesto, se ve que los mcds mas pequeños son mas probables que ocurran aleatoriamente.

Método de transposición.

En primer lugar, para los métodos de transposición y permutación, tomamos unas funciones de manejo de permutaciones que teníamos de otra asignatura. Este método es relativamente sencillo de programar, el detalle a tener en cuenta, y el principal problema de este sistema por sí solo es qué hacer cuando el final del texto a permutar es menor que la permutación. La solución que hemos tomado es añadir caracteres aleatorios

al final, de tal manera que se mezclan con el final, añadiendo algo de ruido. Esto es ligeramente mejor que sustituirlos por espacios, que resultan más evidentes.

```
if(i+clave[j]>=size){
     texto[i+clave[j]]=(char)aleat_num(97,122);
}
```

La vulnerabilidad de este arreglo es que el tamaño del texto es divisible por el tamaño de la permutación. Parece que no hay una solución óptima, cada una tiene sus ventajas e inconvenientes. Por eso este método es mejor combinado con otros, donde se suplen sus inconvenientes.

Método de permutación.

Este método es similar al anterior, se realiza una doble permutación, siguiendo el esquema de una matriz en este caso. De hecho, sabiendo que cada elemento de la matriz permutará a otra posición única, podría realizarse una función que transformase de permutación por matriz a permutación lineal de tamaño k1*k2.

En cualuquier caso, tras ponderar varias opciones, seguimos un enfoque similar a la manera de indexado en ensamblador para 'transformar' de una cadena lineal a una matriz, sin coste adicional, solo en la manera de programación.

```
[...]
for(j=0;j<sj;j++){
if(s +i+j*k>=st){
     out[s + kj[j]*k+ki[i]]=(char)aleat_num(97, 122);


     }
else{
     out[s + kj[j]*k+ki[i]]=text[s +i+j*k];
     }
```

Este método presenta problemas similares al anterior en cuanto al final del texto, tomamos el mismo 'arreglo' que en el apartado anterior.