2014

POLITÉCNICA

Algoritmo Criptográfico KASUMI

Yolanda De La Hoz Simón, t110359.

53826071E.

Ejercicio Práctico de Seguridad de las Tecnologías de la Información.

Marzo, 2014.

Facultad de Informática.

Universidad Politécnica de Madrid.

Índice

Introducción

KASUMI, es un cifrador por bloques, usado en los estándares UMTS, GSM y GPRS de comunicaciones móviles.

En UMTS se utiliza en algoritmos de confidencialidad, f8, e integridad, f9, con nombres UEA1 y UIA1, respectivamente. En GSM es usado como generador de claves en el algoritmo A5/3 y en GPRS se usa también como generador de claves para el GEA3 [1]. Fue diseñado por el grupo SAGE y está basado en MISTY1 con algunas modificaciones para su implementación hardware.

Funcionamiento del algoritmo

En la fig.7 he hecho una representación gráfica del algoritmo, para facilitar su posterior explicación. Para ello me he basado en algunas imágenes generadas, fórmulas y conceptos de Wikipedia que he comprobado mediante su implementación en lenguaje c.

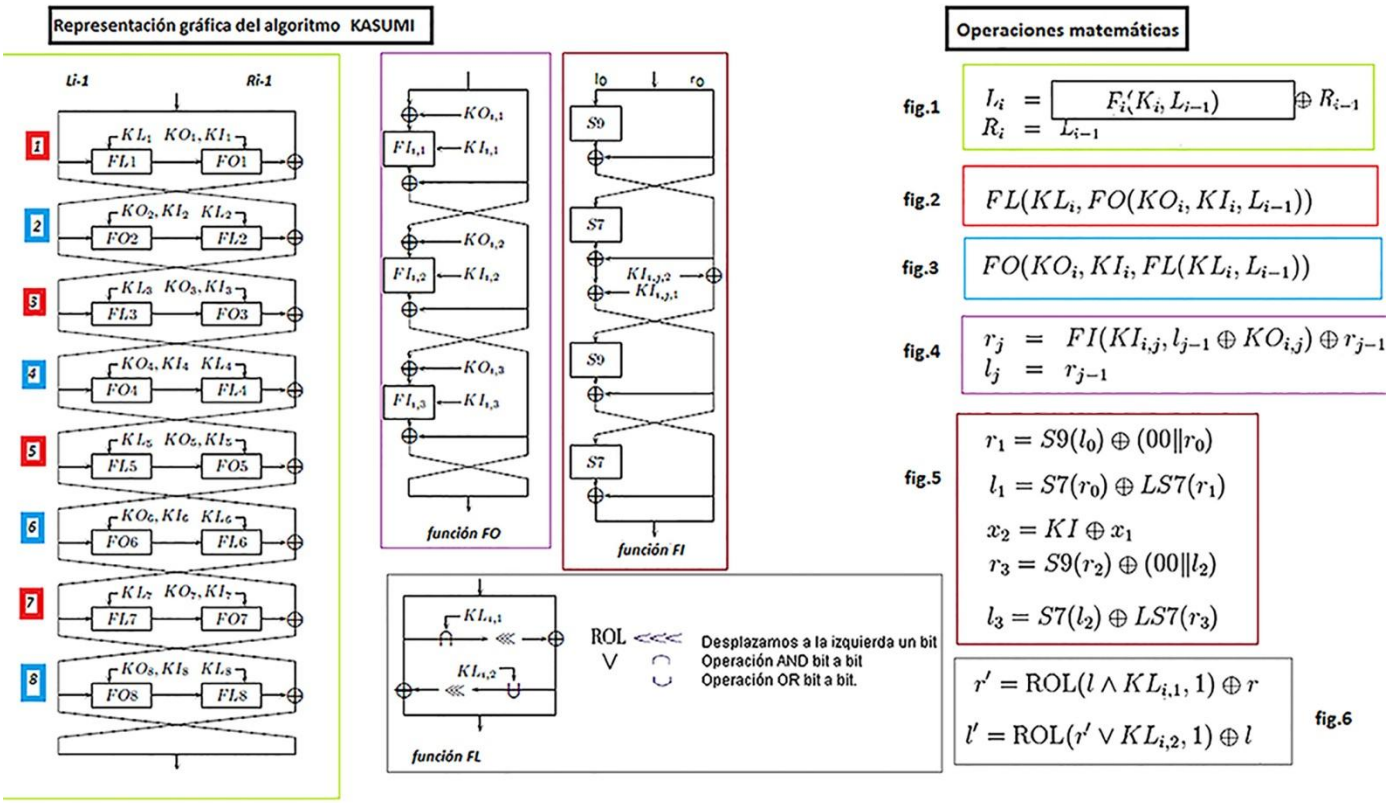

Fig.7 Representación del algoritmo Kasumi.

Como se puede observar en la fig.7 se trata de un cifrado de Feistel de 8 rondas. Opera con una entrada de 64 bits, usa una clave de 128 bits y devuelve como resultado un criptograma de 64 bits.

La clave de 128 bits a su vez es subdivida en 8 claves Ki de 16 bits cada una. Haciendo un XOR con una constante aleatoria se generan las claves derivadas Ki' con las cuales se generan las claves de ronda mediante rotaciones hacia la izquierda bit a bit y las claves derivadas como se muestra en la fig.8

$$
\begin{aligned}
KL_{i,1} &= \mathrm{ROL}(K_i, 1) \\
KL_{i,2} &= K'_{i+2} \\
KO_{i,1} &= \mathrm{ROL}(K_{i+1}, 5) \\
KO_{i,2} &= \mathrm{ROL}(K_{i+5}, 8) \\
KO_{i,3} &= \mathrm{ROL}(K_{i+6}, 13) \\
KI_{i,1} &= K'_{i+4} \\
KI_{i,2} &= K'_{i+3} \\
KI_{i,3} &= K'_{i+7}
\end{aligned}
$$

fig.8 Claves de ronda del algoritmo Kasumi

Estas claves de rondas junto con los datos de entrada de la ronda se aplican a una función F. Está función F o función de etapa de nuestra cadena Feistel, consiste a su vez en dos subfunciones FLi y FOi, que son computadas de forma diferente dependiendo de si la ronda es par o impar, como se puede observar en la fig.7 en colores rojo y azul.

Mientras la función FLi es de complejidad relativamente sencilla, aplica operaciones lógicas y desplazamientos fácilemente computables fig.6. La función FOi presenta a su vez una estructura de Feistel de tres rondas, para la cual aplica una subfunción FI que consta a su vez de otras tres rondas, tratándose de una cadena de Feistel irregular, ya que la entrada se divide en **lo** de 9 bits y **ro** de 7 bits.

Estas entradas se barajan con las cajas de sustitución S9 y S7, aplicando en cada etapa operaciones lógicas XOR en las que se va realimentado alternativamente las salidas de la etapa anterior como se muestra en la fig.6

Por último el resultado es igual a la concatenación de $\mathbf{L}_8$ y $\mathbf{R}_8$, salidas de la ronda 8 de la estructura Feistel.

Implementación del algoritmo en C.

Para la implementación del algoritmo he elegido el lenguaje C, por ser más cercano al hardware de la máquina, aunque no proporciona tanta flexibilidad como ensamblador a la hora de optimizar el código, es más legible y fácil de depurar.

En un principio me he basado en el código del estándar especificado en el documento *ETSI TS 135 202 v7.0.0. (2007-06)* por su simplicidad, legibilidad, corrección y documentación sobre el mismo.

Posteriormente he optimizado el código mediante distintas técnicas que he investigado para la optimización de código en C [2][3] y otras técnicas usadas en otros algoritmos criptográficos.

Primero he hecho un análisis estático evaluando la estructura del código y diseño del mismo, atendiendo al uso de librerías, llamadas al preprocesador o criterios utilizados por el programador para la optimización del código y operaciones usadas por su adaptación al hardware.

En este análisis he declarado las cajas de sustitución como constantes y he sustituido algunas expresiones comunes u operaciones de conversión de datos por macros mediante la directiva #define.

Debido al pequeño tamaño y uso constante de la función FL he decidido declararla de tipo ‘inline’ para evitar pasar parámetros y ejecutar instrucciones de salto y retorno.

También he reducido el código de la función FO he guardado las variables globales de las claves en una matriz para reducir el espacio y poder simplificar los tres pasos en un bucle.

Optimización de la función FO.

```
static u32 FO( u32 in, int index )
{
        u16 left, right;

        left  = (u16)(in>>16);
        right = (u16) in;

        left ^= KOi1[index];
        left  = FI( left, KIi1[index] );
        left ^= right;

        right ^= KOi2[index];
        right  = FI( right, KIi2[index] );
        right ^= left;

        left ^= KOi3[index];
        left  = FI( left, KIi3[index] );
        left ^= right;

        in = (((u32)right)<<16)+left;

        return( in );
}
```

```
static u32 FO( u32 in, int index )
{
  u16 x, y, temp;
  int i;
  x = (u16)(in>>16);
  y = (u16) in;

  for(i=0; i<3; i++)
  {
   x ^= KOi[i][index];
   temp = FI( x, KIi[i][index] );
   x = y;
   y ^= temp;
  }

 return RESULT(x,y);
}
```

En la función KeySchedule he unificado la generación de las claves derivadas en un bucle, ya que el primero establece el formato en la variable key que posteriormente puede ser usada para hacer la operación XOR con la constante aleatoria en el mismo paso.

Optimización de la función KeySchedule: para la generación de subclaves.

```
for( n=0; n<8; ++n )
        key[n] = (u16)((k16[n].b8[0]<<8) + (k16[n].b8[1]));
for( n=0; n<8; ++n )
        Kprime[n] = (u16)(key[n] ^ C[n]);
```

```
for( n=0; n<8; ++n )  {
  key[n] = (u16)((k16[n].b8[0]<<8) + (k16[n].b8[1]));
  Kprime[n] = (u16)(key[n] ^ C[n]);
}
```

Por último he realizado un programa de prueba para comprobar que funciona correctamente tras los cambios aplicados. Aprovechando la irreversibilidad del algoritmo he implementado la función “kasumi_descifrar” para descifrar el texto cifrado.

En el programa de prueba que he implementado he utilizado unos tests ya realizados [4] y he implementado una función que me comprobase que las salidas de mis funciones kasumi_cifrar y kasumi_descifrar (texto cifrado y texto en claro respectivamente) fueran las mismas que la de los test.

A continuación muestro los pantallazos del conjunto de pruebas realizado fig.9 y fig.10.

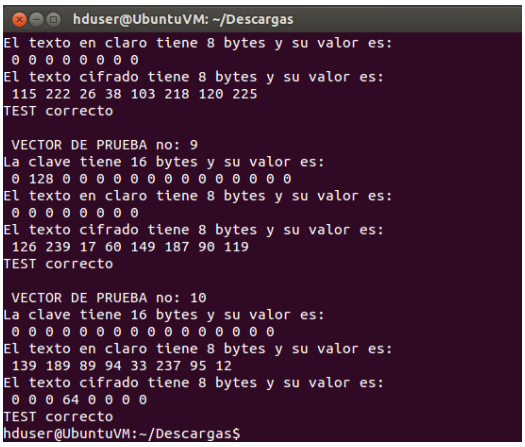

Fig.9

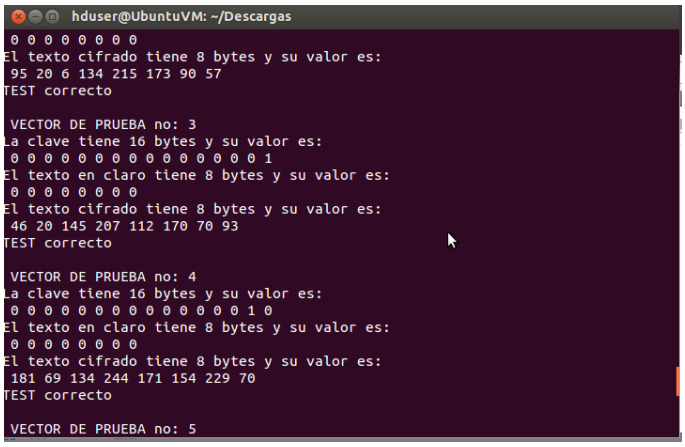

fig.10

Criptoanálisis y propiedades del algoritmo

La longitud de la clave aplicada sobre la estructura hace que en un principio podamos descartar ataques por fuerza bruta, siendo a su vez la longitud del bloque suficiente para descartar también ataques estadísticos básicos (análisis de frecuencias, palabras probables etc) sobre la estructura en cuestión.

En general en el algoritmo se realizan bastantes operaciones de desplazamiento en cada etapa y en las claves de ronda y se alternan las entradas en cada etapa de la estructura en las que se realimenta la salida de la anterior y que son a su vez subdivididas en cada función lo que aumenta considerablemente en la difusión o propagación del cambio de un bit. Además se realizan operaciones lógicas XOR, (también AND y OR en la función FLi) en cada etapa y en la generación de las subclaves, lo que aumenta la confusión en el sistema.

Estas propiedades se ven reforzadas con el uso de las claves de ronda para cada etapa y las cajas de substitución con las que se baraja las entradas de la función FIi

En las pruebas realizadas se ha intentado buscar un efecto Avalancha, es decir, si cambiando un bit del algoritmo Kasumi de los 64 bits de entrada o de los 128 bits de clave el efecto deseado sería a un cambio al menos en unos 32 bits de la salida en media. Posteriormente he ido comparando los criptogramas resultantes en los que se podía observar que efectivamente diferían al menos en el 80% de los bits en la mayoría de ellos y que no había ninguno de ellos que se parecieran más de un 50%.

Esta observación ha sido verificada con el programa Kasiski.c [6], guardando todos los criptogramas generados en un fichero y observando la salida que imprimía los bits repetidos y las distancias entre ellos.

Criptoanálisis realizado hasta el momento

En el estándar *ETSI TS 135 202 v7.0.0. (2007-06)* en el criptoanálisis inicial que se realizó se comprobó su firmeza ante el criptoanálisis diferencial y en un principio se consiguió optimizar el algoritmo para su implementación hardware sin afectar a la seguridad que ya tenía MISTY1. Por tanto, se proponen ataques derivados del criptoanálisis diferencial, el primero que tuvo éxito en 2001 por Kühn,Ulrich (Eurocrypt) fue un ataque diferencial imposible en la etapa 6 del algoritmo.

Mientras el ataque diferencial se basa en el análisis de la evolución de las diferencias de dos textos en claro relacionados cuando son encriptados con la misma clave, asignando probabilidades a cada una de las claves posibles para identificar la clave correcta o más probable. El ataque diferencial imposible sigue el mismo concepto pero asigna probabilidades con valor 0, es decir, determina el conjunto de claves que no pueden darse para el caso a analizar.

Finalmente se logró romper el algoritmo en 2010 con el denominado “sandwich attack” (Dunkelman, Keller, and Shamir) [5] para el cual se necesitaban las claves relativas pero que logra recuperar la clave completamente. Aunque en el estándar no contempla este tipo de ataques como amenaza ya que es difícil que se den las condiciones necesarias para recuperar las claves relativas necesarias, este tipo de ataque no tiene el mismo éxito en MISTY demostrando así que Kasumi es criptográficamente más débil al contrario de lo que se creía en un principio.

Conclusión

En general me ha parecido un trabajo muy interesante y ameno. Aproximadamente le he dedicado 80 horas de trabajo, aunque es cierto que me he entretenido mucho con artículos que no eran tan relevantes para este trabajo pero que aún así no he podido evitar leer e informarme dada la importancia del algoritmo en la actualidad.

En este trabajo me he dado cuenta de la importancia de buscar información contrastada por una organización especializada, ya que en la seguridad informática a mi parecer cuenta con muchas “aficionados” que malinterpretan la información o que simplemente se limitan a copiar la información de otros blogs omitiendo detalles para generar mayor expectación. De esto modo consiguen que un algoritmo que está bien construido pueda ser roto por cualquiera con el ordenador de su casa en menos de dos días lo que no es del todo cierto.

Además he aprendido mucho sobre técnicas de optimización en c. Aunque es cierto que me hubiese gustado usar la librería OpenMP no disponía de los conocimientos suficientes sobre paralelización en c, sólo los de la asignatura de Arquitectura de Computadores en una práctica optativa que realicé en dos clases.

Gracias a esta práctica me he dado cuenta de la importancia de saber manejarla por lo que probablemente curse al año que viene la optativa de Arquitectura de Computadores en la que si se ve este tipo de temas.

Finalmente he añadido el apartado “Criptoanálisis realizado hasta el momento” a pesar de que no se especificaba en la práctica. Por tanto entiendo que no sea evaluado ya que además supera la longitud permitida en la misma.

Referencias

1. http://en.wikipedia.org/wiki/KASUMI
2. http://www.ual.es/~jjfdez/IC/Practicas/optim.html
3. http://es.wikipedia.org/wiki/Funci%C3%B3n_inline

4. https://www.cosic.esat.kuleuven.be/nessie/testvectors/bc/kasumi/Kasumi-128-64.verified.test-vectors
5. http://www.math.huji.ac.il/~nkeller/Crypt-jour-Kasumi.pdf
6. http://pages.cs.wisc.edu/~yetkin/code/crypto/kasiski.c

Otras referencias y bibliografía consultada

Transparencias de la asignatura seguridad de las tecnologías de la información de la Facultad de informática (UPM). Temas de algoritmos actuales y criptoanálisis.

Criptografía y Seguridad, Jorge Dávila Muro.

http://www.tierradelazaro.com/cripto/3g.htm

http://www.tierradelazaro.com/cripto/KASUMI.pdf