ESCUELA POLITÉCNICA NACIONAL

ESCUELA DE INGENIERÍA

"IMPLEMENTACIÓN DE UN SISTEMA SIMULADOR DE ENCRIPTACIÓN DE AUDIO UTILIZANDO EL ENTORNO DE PROGRAMACIÓN MATLAB"

PROYECTO PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN ELECTRÓNICA Y TELECOMUNICACIONES

HUGO ALEX RODRÍGUEZ TOBAR

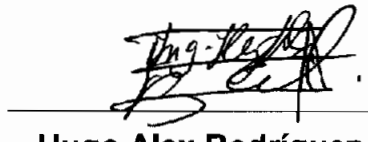
DIRECTOR: ING. PABLO HIDALGO L.

Quito, Julio 2001

DECLARACIÓN

Yo, Hugo Alex Rodríguez Tobar, declaro que el trabajo aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

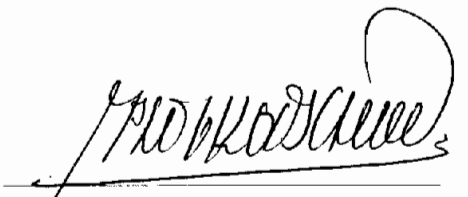
La Escuela Politécnica Nacional, puede hacer uso de los derechos correspondientes a este trabajo, según lo establecido por la Ley, Reglamento de Propiedad Intelectual y por la normatividad institucional vigente.



Hugo Alex Rodríguez Tobar

CERTIFICACIÓN

Certifico que el presente trabajo fue desarrollado por Hugo Alex Rodríguez Tobar, bajo mi supervisión.



Ing. Pablo Hidalgo L.
DIRECTOR DE PROYECTO

CONTENIDO

PRESENTACIÓN i

CAPÍTULO I: FUNDAMENTOS TEÓRICOS 1

1.1. FUNDAMENTOS DE AUDIO 1

1.1.1. CARACTERÍSTICAS DEL AUDIO ANALÓGICO 1

1.1.2. INTRODUCCIÓN AL AUDIO DIGITAL 2

1.2. CONVERSIÓN DE SEÑALES 3

1.2.1. CONCEPTOS BÁSICOS 3

1.2.2. CONVERSIÓN DIGITAL – ANALÓGICA 5

1.2.2.1. Salida analógica 6

1.2.2.2. Factores de ponderación de entrada 6

1.2.2.3. Resolución 6

1.2.2.4. Porcentaje de resolución 7

1.2.2.5. Especificaciones de los DAC 8

1.2.3. CONVERSIÓN ANALÓGICA – DIGITAL 10

1.2.3.1. Conversor A/D de rampa digital 11

1.2.4. ADQUISICIÓN DE DATOS 15

1.2.4.1. Reconstrucción de una señal digitalizada 16

1.3. CARACTERÍSTICAS DEL AUDIO DIGITAL 17

1.3.1. FRECUENCIA DE MUESTREO 17

1.3.1.1. Sobremuestreo 20

1.3.2. RECONSTRUCCIÓN DE LA SEÑAL 21

1.3.2.1. *Dither* 22

CAPÍTULO II: CRIPTOGRAFÍA 24

2.1. INTRODUCCIÓN 24

2.1.1. TRANSMISOR, RECEPTOR Y ENTIDAD 24

2.1.2. MENSAJES Y ENCRIPTACIÓN 24

2.1.3. OBJETIVOS DE LA CRIPTOGRAFÍA 26

2.1.4. ALGORITMOS Y CLAVES 27
2.1.5. ALGORITMOS SIMÉTRICOS 28
2.1.6. ALGORITMOS DE CLAVE PÚBLICA 29
2.1.7. CRIPTOANÁLISIS 30
2.1.8. SEGURIDAD DE LOS ALGORITMOS 33
2.2. PROTOCOLOS CRIPTOGRÁFICOS 35
2.2.1. INTRODUCCIÓN A LOS PROTOCOLOS 35
2.2.1.1. Propósito de los protocolos 36
2.2.1.2. Protocolos arbitrados 36
2.2.1.3. Protocolos adjudicados 37
2.2.1.4. Protocolos de auto detección 38
2.2.1.5. Ataques contra los protocolos 38
2.2.2. PROTOCOLOS DE CRIPTOGRAFÍA SIMÉTRICA 39
2.2.3. GENERACIÓN DE SECUENCIAS ALEATORIAS Y SEUDO - ALEATORIAS 41
2.2.3.1. Secuencias seudo-aleatorias 42
2.2.3.2. Secuencias seudo-aleatorias criptográficamente seguras 42
2.2.3.3. Secuencias aleatorias reales 43
2.2.4. PROTOCOLOS BÁSICOS 44
2.2.4.1. Intercambio de claves 44
2.2.4.1.1. Intercambio de claves en criptografía simétrica 44
2.2.4.1.2. Intercambio de claves con criptografía de clave pública 45
2.2.4.1.3. Ataque del hombre en la mitad 45
2.2.4.1.4. Protocolo entrelazado 46
2.2.4.1.5. Intercambio de claves con firmas digitales 48
2.2.4.1.6. Transmisión de claves y mensajes 49
2.2.4.1.7. Difusión de claves y mensajes 50
2.2.4.2. Autenticación 50
2.3. MANEJO DE CLAVES 51
2.3.1. GENERACIÓN DE CLAVES 51
2.3.1.1. Elección inadecuada de claves 52
2.3.1.2. Claves aleatorias 53
2.3.1.3. Frases de paso 54

2.3.2. TRANSFERENCIA DE CLAVES 54
2.3.2.1. Distribución de claves en redes grandes 55
2.3.2.2. Detección de errores durante la transmisión de claves 55
2.3.3. UTILIZACIÓN DE LAS CLAVES 55
2.3.4. TIEMPO DE VIDA DE LAS CLAVES 57
2.4. TIPOS Y MODOS DE LOS ALGORITMOS 58
2.4.1. MODO LIBRO DE CÓDIGO ELECTRÓNICO 58
2.4.1.1. Relleno 59
2.4.2. MODO DE ENCADENAMIENTO DE BLOQUES CIFRADOS 61
2.4.2.1. Vector de inicio 62
2.4.2.2. Relleno 63
2.4.2.3. Propagación de errores 63
2.4.2.4. Problemas de seguridad 64
2.4.3. CIFRADOS DE FLUJO 65
2.4.4. CIFRADOS DE FLUJO AUTO SINCRONIZADOS 67
2.4.4.1. Problemas de seguridad 68
2.4.5. MODO DE REALIMENTACIÓN DE CIFRADO 69
2.4.5.1. Vector de inicio 70
2.4.5.2. Propagación de errores 70
2.4.6. CIFRADOS DE FLUJO SINCRÓNICOS 71
2.4.6.1. Ataque de inserción 72
2.4.7. MODO DE REALIMENTACIÓN DE SALIDA 73
2.4.7.1. Propagación de errores 75
2.4.7.2. Problemas de seguridad 75
2.4.7.3. Cifrados de flujo en modo OFB 75
2.4.8. MODO CONTADOR 76
2.4.8.1. Cifrados de flujo en modo contador 76
2.4.9. ENTRELAZADO 76
2.4.10. CIFRADOS DE BLOQUE VERSUS CIFRADOS DE FLUJO 77
EPÍLOGO 77

CAPÍTULO III: PROGRAMACIÓN DEL SISTEMA SIMULADOR 78

3.1. INTRODUCCIÓN 78

3.2. DESCRIPCIÓN DE LOS ALGORITMOS 78

3.2.1. DES: *DATA ENCRYPTION STANDARD* 78

3.2.1.1. Descripción del DES 79

3.2.1.1.1. Perfil del algoritmo 79

3.2.1.1.2. Permutación inicial 82

3.2.1.1.3. Transformación de la clave 82

3.2.1.1.4. Permutación de expansión 83

3.2.1.1.5. Sustitución con cajas S 84

3.2.1.1.6. Permutación por caja P 87

3.2.1.1.7. Permutación final 87

3.2.1.1.8. Desencriptación del DES 88

3.2.2. IDEA: *INTERNATIONAL DATA ENCRYPTION ALGORITHM* 88

3.2.2.1. Generalidades de IDEA 88

3.2.2.2. Descripción de IDEA 89

3.2.3. A5/1 91

3.2.3.1. Registros de desplazamiento con realimentación lineal (LFSR) 92

3.2.3.2. Operación del algoritmo A5/1 94

3.2.4. RC4 94

3.3. PROGRAMACIÓN DEL SISTEMA SIMULADOR 96

3.3.1. DESCRIPCIÓN DE LA INTERFAZ GRÁFICA DE USUARIO 96

3.3.1.1. Pantalla de presentación 97

3.3.1.2. Pantalla de selección 98

3.3.1.3. Pantalla de control 99

3.3.1.3.1. Operación general de la pantalla de control 100

3.3.1.4. Operación general del programa de adquisición de audio 103

3.3.2. PROGRAMACIÓN DE LOS ALGORITMOS 104

3.3.2.1. Implementación del algoritmo DES 104

3.3.2.2. Implementación del algoritmo IDEA 108

3.3.2.3. Implementación del algoritmo A5/1 109

3.3.2.4. Implementación del algoritmo RC4 111

4.2.3.3.3. Pruebas realizadas al algoritmo A5/1 139
4.2.3.3.4. Pruebas realizadas al algoritmo RC4 141

CAPÍTULO V: CONCLUSIONES Y RECOMENDACIONES 143
5.1. CONCLUSIONES 143
5.2. RECOMENDACIONES 147

REFERENCIAS BIBLIOGRÁFICAS 149

ANEXO A: CÓDIGO FUENTE DE LOS ALGORITMOS IMPLEMENTADOS A-1
A.1. CÓDIGO FUENTE DEL ALGORITMO DES A-1
A.1.1. ALGORITMO DE ENCRIPTACIÓN DESCIP.M A-1
A.1.2. ALGORITMO DE DESENCRIPTACIÓN DESDEC.M A-11
A.1.3. SUSTITUCIÓN CON CAJAS S DESSBOX.M A-16
A.2. CÓDIGO FUENTE DEL ALGORITMO IDEA A-20
A.2.1. ALGORITMO DE ENCRIPTACIÓN IDEACIP.M A-20
A.2.2. ALGORITMO DE DESENCRIPTACIÓN IDEADEC.M A-26
A.3. CÓDIGO FUENTE DEL ALGORITMO A5/1 A-32
A.3.1. ALGORITMO DE ENCRIPTACIÓN A5CIP.M A-32
A.3.2. ALGORITMO DE DESENCRIPTACIÓN A5DEC.M A-32
A.4. CÓDIGO FUENTE DEL ALGORITMO RC4 A-33
A.4.1. ALGORITMO DE ENCRIPTACIÓN RC4CIP.M A-33
A.4.2. ALGORITMO DE DESENCRIPTACIÓN RC4DEC.M A-34

ANEXO B: "REAL TIME CRYPTANALYSIS OF A5/1 ON A PC" B-1
ABSTRACT B-2
1. INTRODUCTION B-2
2. DESCRIPTION OF THE A5/1 STREAM CIPHER B-3

3. PREVIOUS ATTACKS B-5
4. INFORMAL DESCRIPTION OF THE NEW ATTACKS B-6
5. DETAILED DESCRIPTION OF THE ATTACKS B-9
5.1. Efficient Sampling of Special States B-9
5.2. Efficient Disk Probing B-11
5.3. Efficient Disk Storage B-11
5.4. Efficient Tree Exploration B-12
5.5. The Biased Birthday Attack B-12
5.6. Efficient Determination of Initial States B-15
5.7. Reducing the Preprocessing Time of the Biased Birthday Attack B-15
5.8. Extracting the Key From a Single Red State B-16

ANEXO C: REPORTES DE PRUEBAS GENERADOS POR MATLAB C-1
C.1. INTRODUCCIÓN C-1
C.2. REPORTE DE PRUEBAS EN EL ALGORITMO DES C-1
C.3. REPORTE DE PRUEBAS EN EL ALGORITMO IDEA C-3
C.4. REPORTE DE PRUEBAS EN EL ALGORITMO A5/1 C-4
C.5. REPORTE DE PRUEBAS EN EL ALGORITMO RC4 C-6

ANEXO D: MANUAL DE USUARIO D-1
D.1. INTRODUCCIÓN D-1
D.2. REQUERIMIENTOS DEL SISTEMA D-1
D.3. INSTALACIÓN D-2
D.4. EJECUCIÓN D-3
D.5. RECOMENDACIONES DE USO D-7
D.6. EJEMPLO DE OPERACIÓN D-7
D.6.1. EJECUCIÓN DEL SISTEMA SIMULADOR D-7

PRESENTACIÓN

Un viejo proverbio chino dice: "Para que un hombre pueda decir que ha cumplido todas las metas de su vida, debe realizar tres cosas: plantar un árbol, tener un hijo y escribir un libro". El cumplimiento de la tercera tarea está representado en este trabajo. La culminación de un ciclo, la carrera de Ingeniería, se ve plasmada a través de las páginas de este proyecto, donde queda demostrado, una vez más, que el ingenio humano es inagotable, especialmente en lo que a carreras técnicas se refiere.

El mundo está convergiendo hacia los sistemas digitales en todas sus actividades diarias, lo cual conduce a plantear una pregunta: ¿son estos sistemas seguros?. La respuesta es incierta; hay varios sistemas que proclaman ser seguros, y cualquier persona puede penetrar en ellos (el caso Microsoft de noviembre del 2000 es el más reciente y preocupante). Existen otros sistemas que en apariencia son inseguros, pero es imposible violar su seguridad. La seguridad de un sistema siempre será relativa.

En años recientes, los sistemas de audio digital se han vuelto un tema de amplio estudio. El crecimiento acelerado de la compresión de audio, así como de Internet, han traído a la palestra nuevos tipos de servicios tales como telefonía IP, distribución de música a través de programas como Napster y Gnutella, emisiones radiales en vivo a través de Real Player, etc, lo que lleva a otras preguntas: ¿es adecuada la seguridad de estas emisiones?, y ¿qué tipo de seguridad necesitan?. De estas preguntas nace la idea para el presente proyecto de titulación.

El objetivo del presente proyecto de titulación es implementar un Sistema Simulador de Encriptación de Audio, a través del entorno de programación MATLAB. Este Sistema Simulador servirá para estudiar los efectos de la encriptación en sistemas de audio digital, lo que permitirá generar conclusiones tendientes a recomendar la adecuada aplicación de los algoritmos criptográficos a ciertas tecnologías de utilización de audio digital.

El presente proyecto de titulación se encuentra dividido en cinco capítulos: Fundamentos Teóricos, Criptografía, Programación del Sistema Simulador, Pruebas Realizadas y Conclusiones y Recomendaciones, cada uno de los cuales se explica a continuación.

El primer capítulo, Fundamentos Teóricos, describe las características del audio digital, sus frecuencias de muestreo estándar, la forma de reconstrucción de la señal, y sobre todo, la teoría básica de conversión de señales analógicas a digitales y viceversa.

El segundo capítulo, Criptografía, trata acerca de la teoría fundamental de criptografía, señalando sus objetivos, las clases de algoritmos y protocolos criptográficos, la forma de manejar las claves de los algoritmos, punto esencial de los mismos, y los modos en que éstos operan.

El tercer capítulo, Programación del Sistema Simulador, presenta los cuatro algoritmos implementados: DES, IDEA, A5/1 y RC4, para luego describir la forma en la cual cada uno de ellos fue implementado en MATLAB. Se describen además los archivos auxiliares que sirven para cumplir funciones de manejo de datos necesarias para complementar el funcionamiento del sistema en MATLAB.

El cuarto capítulo, Pruebas Realizadas, describe las pruebas a las que se sometió al Sistema Simulador en cuanto a tiempo de ejecución, efectividad de la operación de encriptación/desencriptación, ancho de banda utilizado por las señales adquirida, encriptada y desencriptada, así como pruebas de escritorio tendientes a demostrar la eficacia de los algoritmos implementados.

El quinto capítulo, Conclusiones y Recomendaciones, muestra las conclusiones a las cuales llegó el autor de este proyecto de titulación después de realizar todo el trabajo y realizar las pruebas más indicadas para el efecto. Después de generar las conclusiones, el autor realiza recomendaciones para la construcción de equipos, y la utilización de determinados algoritmos en situaciones específicas.

Con el permiso de los autores, Dr. Alexander Byriukov, Prof. Adi Shamir y Dr. David Wagner, se incluye como anexo un estudio realizado por ellos acerca de la forma de romper el algoritmo A5/1, utilizado en el enlace entre el teléfono y la radiobase en el esquema de comunicaciones GSM. También se incluye como referencia bibliográfica de consulta un artículo de IEEE *Spectrum*, el cual describe la nueva generación de computadoras cuánticas, las cuales representan un gran reto para la criptografía tal y como se la conoce ahora.

Para finalizar, deseo agradecer de manera especial a las personas que me apoyaron durante la realización de este proyecto de titulación. Empiezo por agradecer a mis dos mejores amigos, Damián y Juan Francisco, por apoyarme y soportarme durante la realización de este proyecto. Al Ing. Ramiro Campos y a Mauricio por su ayuda decidida en el aprendizaje y aplicación de MATLAB. Al Ing, Raúl Cubillo, por ayudarme en la edición del programa instalador del Sistema Simulador de Encriptación de Audio. Agradezco a todos mis profesores, los cuales me enseñaron todo lo que sé y todo lo que se puede lograr con una buena educación. Un agradecimiento especial a mi director de proyecto, Ing. Pablo Hidalgo, por su paciencia y acertada dirección, sin las cuales no hubiera podido concluir con éxito mi carrera.

A mi madre, que me supo enseñar el valor del tiempo y de los sueños, agradezco y dedico este proyecto de titulación.

ALEX RODRÍGUEZ T.

CAPITULO I

FUNDAMENTOS TEÓRICOS

1.1. FUNDAMENTOS DE AUDIO

1.1.1. CARACTERÍSTICAS DEL AUDIO ANALÓGICO[12]

El sonido, o audio, es un agente físico que consiste en vibraciones mecánicas de puntos materiales que, al propagarse en el medio circundante, llegan al oído y perturban su equilibrio provocando la sensación sonora. Se habla de sonido propiamente dicho cuando las oscilaciones son uniformes, y de ruido en caso contrario. Las frecuencias audibles están comprendidas entre 16 y 20000 Hz. Las frecuencias inferiores a los 16 Hz corresponden a los llamados subsonidos y las superiores a 20000 Hz a los ultrasonidos. Se llama sonido puro o tono a aquel cuya oscilación es sinusoidal. A él corresponde, por consiguiente, una única frecuencia, y puede representarse matemáticamente por una fluctuación de presión de la forma:

$$p = p_0 \cdot \sin\left(2 \cdot \pi \cdot \frac{t}{T} + \varphi\right) \qquad \text{(Ec. 1.1)}$$

donde p es el valor instantáneo de la presión, p_0 es la amplitud de la oscilación de presión, t es el tiempo, T es el período y φ es el ángulo de fase inicial. Un sonido cualquiera puede considerarse como la suma de sonidos puros de períodos submúltiplos del período T propio del sonido. El tono que oscila con período T se denomina primer armónico o armónico fundamental. Los demás términos se denominan armónicos superiores. Se denomina altura de un sonido a la frecuencia de su primer armónico. El timbre depende de la estructura del espectro y es tanto más agradable cuanto menor sea la intensidad de los armónicos de orden más elevado. La intensidad del sonido, o energía de oscilación, difiere de la sensación de intensidad, o sonoridad, experimentada por el oído.

El sentido del oído es el resultado de una serie de procesos acústicos, mecánicos, nerviosos y mentales dentro de la combinación oído/cerebro. La agudeza del oído humano es asombrosa, ya que puede detectar cantidades minúsculas de distorsión y aceptar un gran rango dinámico de amplitud. Normalmente, el grado de sensibilidad del oído humano es mayor entre los 2 kHz y los 5 kHz aproximadamente y, aunque algunas personas pueden detectar 20 kHz a nivel alto, existen muchas pruebas que sugieren que la mayoría de los oyentes no pueden distinguir si el límite superior de frecuencia del sonido se encuentra en los 20 kHz o los 16 kHz. El rango dinámico del oído tiene una respuesta logarítmica que sobrepasa claramente los 100 dB.

1.1.2. INTRODUCCIÓN AL AUDIO DIGITAL[9]

Cualquier fuente analógica de audio puede estar caracterizada por un determinado ancho de banda y una relación señal a ruido dada. Si se pone en serie con dicha fuente[1] un canal digital con un ancho de banda superior y una relación señal a ruido mayor, sólo es necesario ajustar los niveles correctamente y la señal analógica no estaría sujeta a ninguna pérdida de información en absoluto. El sonido transmitido por un sistema digital fluye en forma de una corriente de bits. Una transmisión perfecta es aquella en la cual se puede enviar una señal sin pérdida de información y sin captar señales no deseadas a lo largo de la trayectoria de transmisión; sin embargo, esto no es posible debido a la presencia inherente de ruido en todo canal de transmisión. El ruido en un canal se define como una señal aleatoria que no se correlaciona con la señal transmitida, y que se sobrepone a la misma.

La elección de la frecuencia de muestreo es importante para cualquier sistema; si es demasiado baja, la señal se verá degradada, y si es demasiado alta, aumentará innecesariamente el número de muestras requeridas, incrementando las necesidades de ancho de banda y otros elementos.

[1] Antes de conectar la fuente analógica de audio con el canal digital, debería ponerse un conversor A/D

A continuación, se introduce la teoría general de conversión de señales, para luego señalar ciertas particularidades del audio digital.

1.2. CONVERSIÓN DE SEÑALES[4]

Debido al uso cada más frecuente de dispositivos que procesan en forma digital todo tipo de señales, es necesaria una introducción acerca de la forma en la cual las señales analógicas (voz, variables físicas, etc). son convertidas a un formato digital, y como éstas pueden volver a su estado normal. Este proceso se denomina *conversión de señales*, y se explica a continuación.

1.2.1. CONCEPTOS BÁSICOS

Una cantidad digital es aquella cuyo valor se puede especificar mediante el uso de varios estados discretos, pero, en general, se usan solamente dos de ellos: verdadero y falso, lo cual genera señales digitales binarias. En las aplicaciones del mundo real, una cantidad digital se representa mediante voltajes, los cuales tienen un valor que está dentro de ciertos rangos específicos; los valores exactos de los voltajes no son significativos, ya que los circuitos digitales responden de la misma manera para todos los voltajes que se encuentren dentro de un rango especificado.

Por otro lado, una cantidad analógica puede tomar un valor cualquiera dentro de un rango continuo de valores; en este caso, el valor exacto de una cantidad analógica es totalmente significativo. Muchas variables físicas son de naturaleza analógica, por lo tanto, cualquier información que tenga que introducirse en un sistema digital, primero debe convertirse a una forma digital; de manera similar, las salidas de un sistema de esta naturaleza siempre son digitales, y deben volver a su estado analógico. Cuando un sistema digital se utiliza para controlar un proceso físico, el problema básico que hay que enfrentar para el diseño del mismo, es la diferencia entre la naturaleza digital del sistema y la analógica de las variables del proceso. Para comprender mejor el

funcionamiento de la conversión de señales, se expone un diagrama de bloques básico, mostrado en la figura 1.1, y que se explica a continuación:

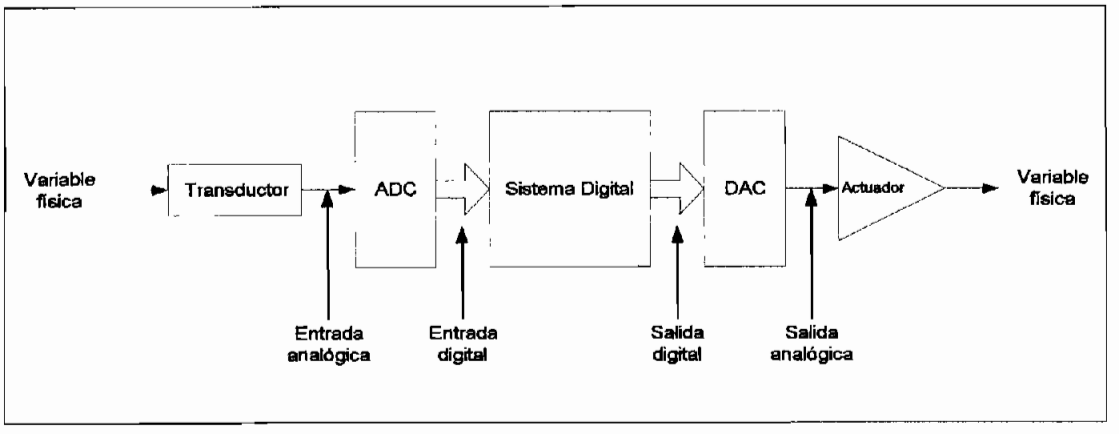


Figura 1. 1: Diagrama de bloques de un sistema de conversión de señales

- *Transductor.* Por lo general, las variables físicas no son cantidades eléctricas. Un transductor es un dispositivo que convierte una variable física en eléctrica u óptica, en este caso, sólo se hablará de variables eléctricas. La salida eléctrica de un transductor es una corriente o un voltaje analógico proporcional a la variable física que se está convirtiendo.
- *Conversor analógico-digital (ADC).* La salida del transductor se conecta a un ADC, que convierte esta señal en una cantidad digital, la cual está compuesta de varios bits que representan el valor de la señal analógica.
- *Sistema digital.* La representación digital de la variable analógica se transmite desde el ADC hacia un sistema de procesamiento digital, el cual almacena y procesa los datos digitales de acuerdo a las instrucciones del programa en ejecución. El programa efectuará cálculos u otras operaciones sobre la representación digital de la variable física, de acuerdo con la acción específica que se desee realizar con la misma.
- *Conversor digital-analógico (DAC).* La salida del sistema digital se conecta a un DAC, que la convierte de nuevo en una corriente o voltaje analógico proporcional a la cantidad digital de salida.
- *Actuador.* A menudo, la señal analógica que proviene del DAC está conectada a algún circuito o dispositivo que sirve como actuador para el control de variables físicas.

Del esquema general de un sistema conversor de señales se puede sacar una conclusión: los ADC y DAC funcionan como interfaces entre sistemas totalmente digitales y el mundo analógico. Esta función reviste cada vez mayor importancia a medida que los sistemas digitales siguen bajando su costo, e incorporando más funciones.

1.2.2. CONVERSIÓN DIGITAL-ANALÓGICA

Básicamente, la conversión D/A es el proceso de tomar un valor representado en código digital (como binario o BCD) y convertirlo en un voltaje o corriente que sea proporcional a este valor digital. La figura 1.2 muestra el esquema de un conversor D/A común de cuatro bits.

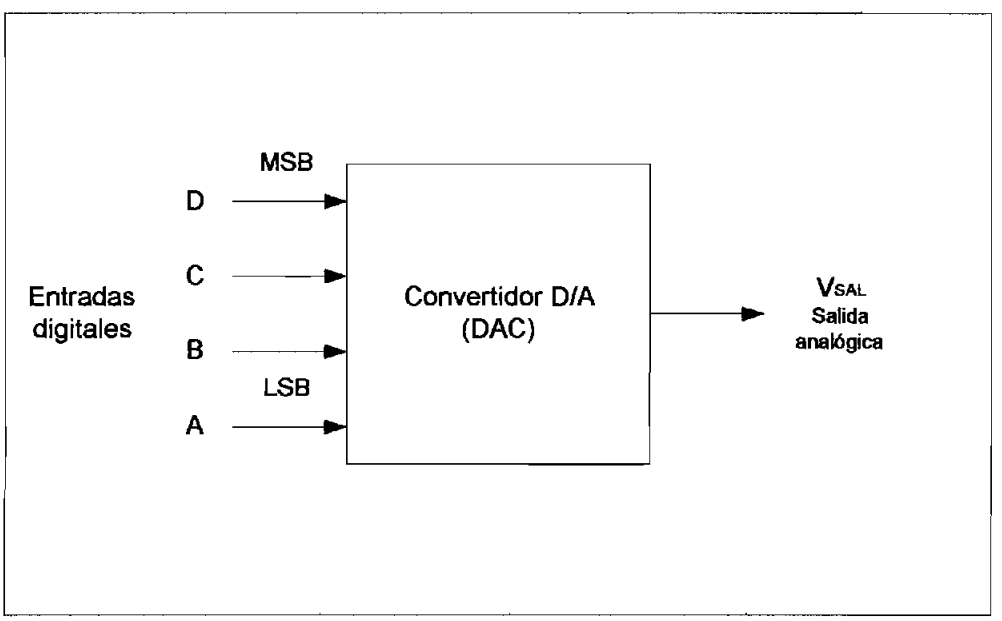


Figura 1. 2: Esquema de un conversor D/A

Las entradas digitales D, C, B y A generalmente se derivan del registro de salida de un sistema digital. Al tener 4 bits, se pueden representar 16 valores binarios diferentes; por cada valor de entrada, el voltaje de salida del conversor D/A es distinto. El voltaje de salida analógico V_{SAL}[2] será igual en voltios al valor binario, pudiendo estar multiplicado por algún factor de proporcionalidad.

[2] En lugar de un voltaje de salida, se puede utilizar una corriente de salida I_{SAL}

En general:

$$salida\ analógica = K \times entrada\ digital \qquad \text{(Ec. 1.2)}$$

donde K es el factor de proporcionalidad y tiene un valor constante para un DAC específico.

1.2.2.1. Salida analógica

Desde un punto de vista técnico, la salida de un DAC no es una cantidad analógica ya que sólo puede tomar valores específicos; de este modo, la salida es digital en realidad. Sin embargo, se puede reducir la diferencia entre dos valores consecutivos al aumentar el número de salidas diferentes, mediante el incremento del número de bits de entrada; esto permite producir una salida cada vez más similar a una cantidad analógica, la cual varía sobre un rango continuo de valores.

1.2.2.2. Factores de ponderación de entrada

A las contribuciones de cada entrada digital se les asigna factores de ponderación, según su posición en la cantidad binaria; así, A, que es el LSB tiene un factor de ponderación de 1, B de 2, C de 4 y D, el MSB, de 8. Los factores de ponderación se duplican sucesivamente por cada bit de entrada, comenzando con el LSB; por ende, se puede considerar a V_{SAL} como la suma de los factores de ponderación de las entradas digitales.

1.2.2.3. Resolución

La resolución de un conversor D/A se define como la mínima variación que puede ocurrir en la salida analógica como resultado de un cambio en la entrada digital. La resolución siempre es igual al factor de ponderación del LSB, la cual se conoce también como tamaño de paso ya que es la cantidad V_{SAL} que variará cuando el código de entrada pase de un valor al siguiente. Para ilustrar el método,

se propone un ejemplo, donde las salidas de un contador binario de cuatro bits se conectan a un DAC, y que se muestra en la figura 1.3.

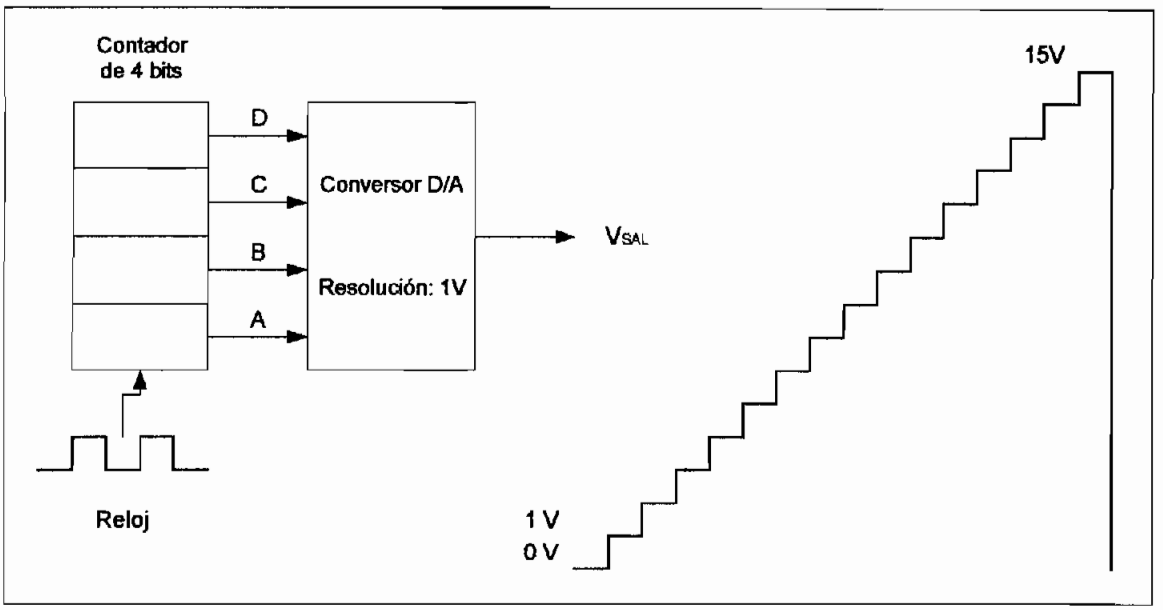


Figura 1. 3: Forma de onda de salida de un conversor D/A conectado a un contador

Conforme el contador pasa por sus 16 estados al aplicarle los pulsos de reloj, se observa que la salida del DAC es una forma de onda en escalera, que cambia un voltio por paso. Cuando el estado del contador es 1111, la salida del DAC alcanza su máximo valor, 15 V; este valor se denomina salida a escala completa. La resolución es el tamaño de los escalones en la forma de onda en escalera. Debe notarse que la escalera tiene 16 niveles, que corresponden a los 16 estados de entrada, pero existen 15 pasos entre el nivel 0 V y el de escala completa. En general, en un DAC de *N* bits, el número de estados diferentes es 2^N, mientras que el de pasos es $2^N - 1$.

1.2.2.4. Porcentaje de resolución

Aunque la resolución puede expresarse como la cantidad de voltaje o corriente por paso, resulta más útil expresarla como un porcentaje de la salida a escala completa, lo que resulta en la ecuación:

$$resolución\ porcentual = \frac{tamaño\ de\ paso}{escala\ completa} \times 100\% \qquad \text{(Ec. 1.3)}$$

La resolución porcentual se hace menor conforme se incrementa el número de bits de entrada; en realidad, la resolución porcentual puede calcularse a partir de:

$$resolución\ porcentual = \frac{1}{número\ total\ de\ pasos} \times 100\% \qquad \text{(Ec. 1.4)}$$

Esto significa que el número de bits es el que determina la resolución porcentual. Si se incrementa el número de bits, aumenta el número de pasos hasta llegar a la escala completa, de manera que cada paso sea una parte menor del voltaje de escala completa.

1.2.2.5. Especificaciones de los DAC

Al existir los DAC como circuitos integrados en el mercado, es necesario conocer las características básicas de los mismos:

a. Resolución

La resolución porcentual de un DAC depende exclusivamente del número de bits; por esta razón, los fabricantes especifican la resolución como el número de bits.

b. Precisión

Los fabricantes de DAC tienen varias formas de especificar la precisión. Las dos más comunes son: error de escala completa y error de linealidad, que se expresan como un porcentaje de la salida a escala completa del conversor (% F.S).

e. Monotonicidad

Un DAC es monótono si su salida aumenta a medida que la entrada binaria se incrementa de un valor a otro. Otra forma de describir esta característica es que la salida en escalera no tendrá pasos hacia abajo a medida que la entrada binaria aumenta desde cero hasta su valor de escala completa.

1.2.3. CONVERSIÓN ANALÓGICA-DIGITAL

Un conversor A/D toma un voltaje de entrada analógico y después de cierto tiempo produce un código de salida digital que representa la entrada analógica. El proceso de conversión A/D es generalmente mucho más complejo y largo que el proceso D/A.

La mayoría de los ADC utilizan un conversor D/A como parte de sus circuitos. La figura 1.4 ilustra un diagrama de bloques general de los ADC. La temporización para realizar esta operación la proporciona una señal de reloj de entrada. La unidad de control contiene los circuitos lógicos para generar la secuencia de operaciones adecuada en respuesta al comando Inicio, el cual hace que el proceso de conversión empiece. El comparador con amplificador operacional tiene dos entradas analógicas y una salida digital.

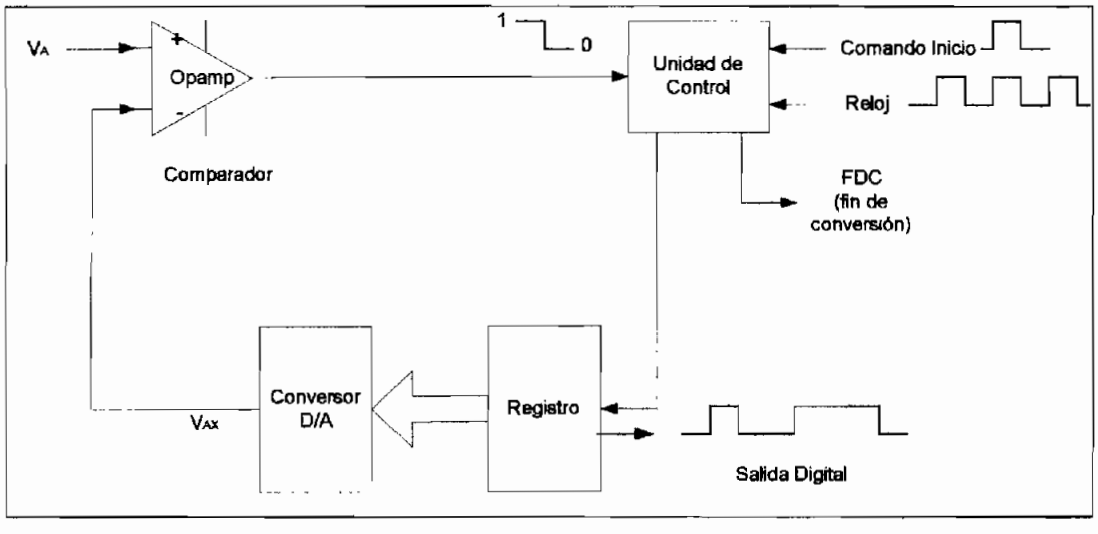


Figura 1. 4: Diagrama de bloques de un conversor A/D típico

La operación básica de los conversores A/D típicos se puede resumir en los siguientes pasos:

- El comando Inicio cambia de estado, dando comienzo a la operación.
- A una frecuencia determinada por el reloj, la unidad de control modifica continuamente el número binario que está almacenado en el registro.
- El número binario del registro es convertido en un voltaje analógico, V_{AX}, mediante el conversor D/A.
- El comparador compara V_{AX} con la entrada analógica V_A. Mientras $V_{AX} < V_A$, la salida del comparador permanece en su estado inicial. Cuando V_{AX} excede a V_A en por lo menos una cantidad V_T (voltaje de umbral), la salida del comparador cambia de estado y suspende el proceso de modificación del número del registro. En este punto, V_{AX} es un valor muy aproximado de V_A, y el número digital del registro, que es el equivalente digital de V_{AX}, es también el equivalente digital de V_A, en los límites de resolución y exactitud del sistema.
- La lógica de control activa la señal de fin de conversión, FDC, cuando se completa el proceso de conversión.

Todas las variantes de conversores A/D difieren básicamente en la forma como la sección de control modifica continuamente los números contenidos en el registro.

1.2.3.1. Conversor A/D de rampa digital

Una de las formas más fáciles de implementar un conversor A/D emplea un contador binario como registro y permite que el reloj incremente el estado del contador un paso a la vez hasta que $V_{AX} \geq V_A$. Este tipo de conversor recibe el nombre de ADC de rampa digital debido a que la forma de onda en V_{AX} es una rampa.

La figura 1.5 muestra un diagrama de bloques de un ADC de rampa digital, el cual contiene un contador, un DAC, un comparador analógico y una compuerta

AND de control. La salida del comparador también proporciona la señal de fin de conversión, activa en bajo, $\overline{FDC}$. Si se supone que V_A es positivo, la operación del conversor es la siguiente:

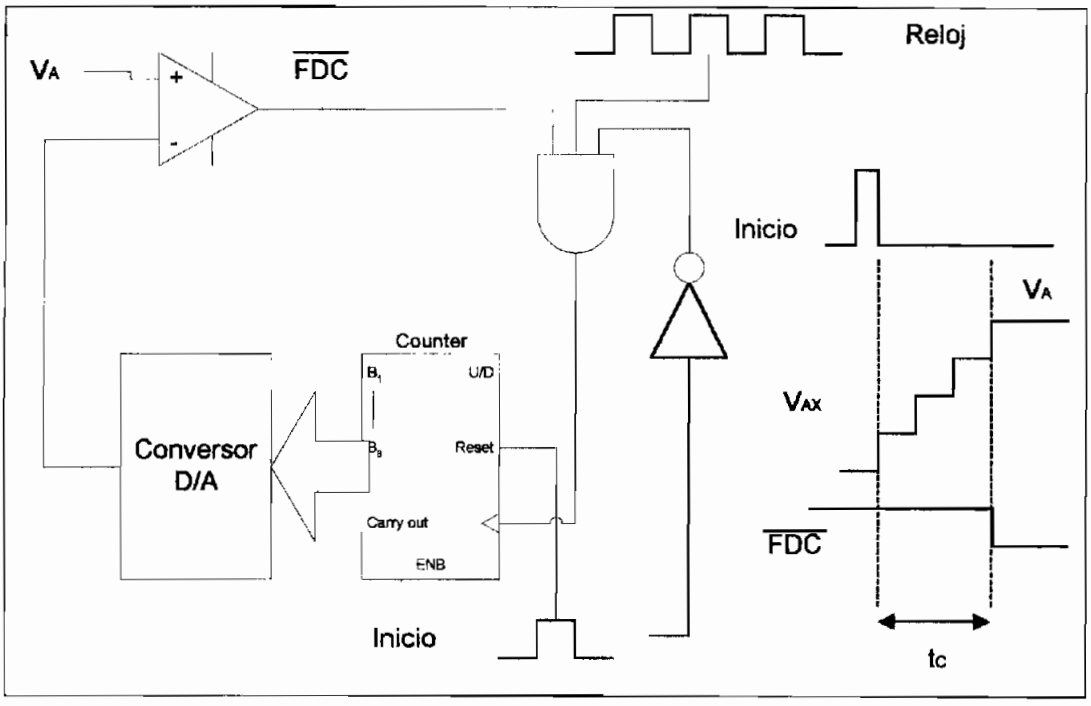


Figura 1. 5: Diagrama de bloques de un ADC de rampa digital

- Se aplica el pulso Inicio para poner el contador a cero. El estado alto de Inicio también inhibe el paso de los pulsos de reloj por la compuerta AND, y de ahí, hacia el contador.
- Cuando las entradas del DAC son cero, la salida de éste es $V_{AX} = 0$ V.
- Dado que $V_A > V_{AX}$, la salida del comparador, $\overline{FDC}$, está en alto.
- Cuando Inicio regresa el estado bajo, se habilita la compuerta AND y los pulsos de reloj entonces pasan hacia el contador.
- A medida que cambia el estado del contador, la salida del DAC, V_{AX}, aumenta un paso a la vez.
- Este proceso continúa hasta que V_{AX} alcanza un paso que excede a V_A por una cantidad igual o mayor que V_T. En ese momento $\overline{FDC}$ cambia hacia el estado bajo e inhibe el flujo de pulsos hacia el contador, motivo por el cual, éste deja de contar.

- Al terminar el proceso de conversión, lo cual es señalado por la transición de la señal $\overline{FDC}$, el contenido del contador es la representación digital de V_A.
- El contador retiene el valor digital hasta que el siguiente pulso Inicio da comienzo otra vez al proceso de conversión.

a. Resolución y exactitud de los ADC

La resolución de los ADC depende de la resolución del DAC que contienen. El voltaje de salida V_{AX} es una forma de onda en escalera que aumenta en pasos discretos hasta que alcanza un valor mayor a V_A; de este modo, V_{AX} es una aproximación al valor de V_A y lo mejor que se puede esperar es que V_{AX} se encuentre a no más de *n* mV de V_A, si la resolución es *n*. La resolución también puede definirse como un error inherente al conversor, al que a menudo se hace referencia como error de cuantización. Este error, que se puede reducir mediante el incremento del número de bits tanto en el contador como en el DAC, se especifica algunas veces como un error de +1 LSB; esto indica que el resultado puede tener un error igual al factor de ponderación asociado con el LSB. Desde otro punto de vista, la entrada V_A puede tomar un infinito número de valores, desde 0 hasta escala completa; sin embargo, la aproximación V_{AX} sólo puede tomar un número finito de valores discretos; por consiguiente, un pequeño rango de valores de V_A tendrá la misma representación digital.

Al igual que con los conversores D/A, la exactitud no se relaciona con la resolución, sino que depende de la precisión de los componentes del circuito. Una especificación de error de 0.01% F.S. indica que el resultado del conversor puede alejarse por 0.01% de la escala completa, debido a componentes no ideales. Este error existe junto con el error de cuantización, y estas dos fuentes de error son generalmente del mismo orden de magnitud para un ADC dado.

b. Tiempo de conversión

El tiempo de conversión es el intervalo entre el final del pulso Inicio y la activación de la salida $\overline{FDC}$. El contador empieza su cuenta desde cero hasta que V_{AX} sea mayor que V_A; en ese instante, $\overline{FDC}$ tiene una transición hacia abajo y termina el proceso de conversión. Debe estar sobreentendido que el valor del tiempo de conversión, t_C, depende de V_A. Un valor grande de V_A requerirá un número más grande de pasos para que el voltaje de la rampa digital sea mayor que V_A.

El tiempo máximo de conversión se presenta cuando V_A tiene un valor un poco menor que el de escala completa; en este caso V_{AX} tiene que esperar hasta el último paso para activar la señal $\overline{FDC}$. Para un conversor de *N* bits el tiempo de conversión está dado por:

$$t_C(\max) = 2^N - 1 \text{ ciclos de reloj} \qquad \text{(Ec. 1.5)}$$

Algunas veces se especifica el tiempo promedio de conversión, el cual es la mitad del tiempo máximo de conversión; para el conversor de rampa digital, la expresión sería:

$$t_C(\text{avg}) = \frac{t_C(\max)}{2} \approx 2^{N-1} \text{ ciclos de reloj} \qquad \text{(Ec. 1.6)}$$

La principal desventaja del método de rampa digital es que el tiempo de conversión se duplica esencialmente por cada bit que se agrega al contador, de manera que la resolución puede ser mejorada solamente con un tiempo de conversión más largo. Esta situación hace que este tipo de ADC sea inadecuado para aplicaciones donde tienen que efectuarse conversiones repetitivas de una señal analógica que cambia rápidamente; sin embargo, para aplicaciones de baja velocidad, la simplicidad relativa de este conversor es una ventaja sobre los ADC de alta velocidad.

1.2.4. ADQUISICIÓN DE DATOS

Existen varias aplicaciones en las que los datos analógicos tienen que ser digitalizados y transferidos a la memoria de un computador. El proceso por el cual un computador adquiere estos datos analógicos digitalizados se conoce como adquisición de datos. El computador puede ejecutar varias operaciones con los datos, dependiendo de la aplicación; por ejemplo, en una aplicación de almacenamiento, como una grabación de audio digital, el computador almacenará los datos y un tiempo después los transferirá a un DAC para reproducir los datos analógicos (en este caso, la señal de audio almacenada).

La figura 1.6 muestra la forma en que un microprocesador se conecta a un ADC de rampa digital con el fin de adquirir datos. El computador genera los pulsos Inicio que dan comienzo a cada proceso de conversión A/D. La señal $\overline{FDC}$ del ADC se alimenta al computador, el cual la utiliza para transferir los datos digitales de la salida del ADC a su memoria.

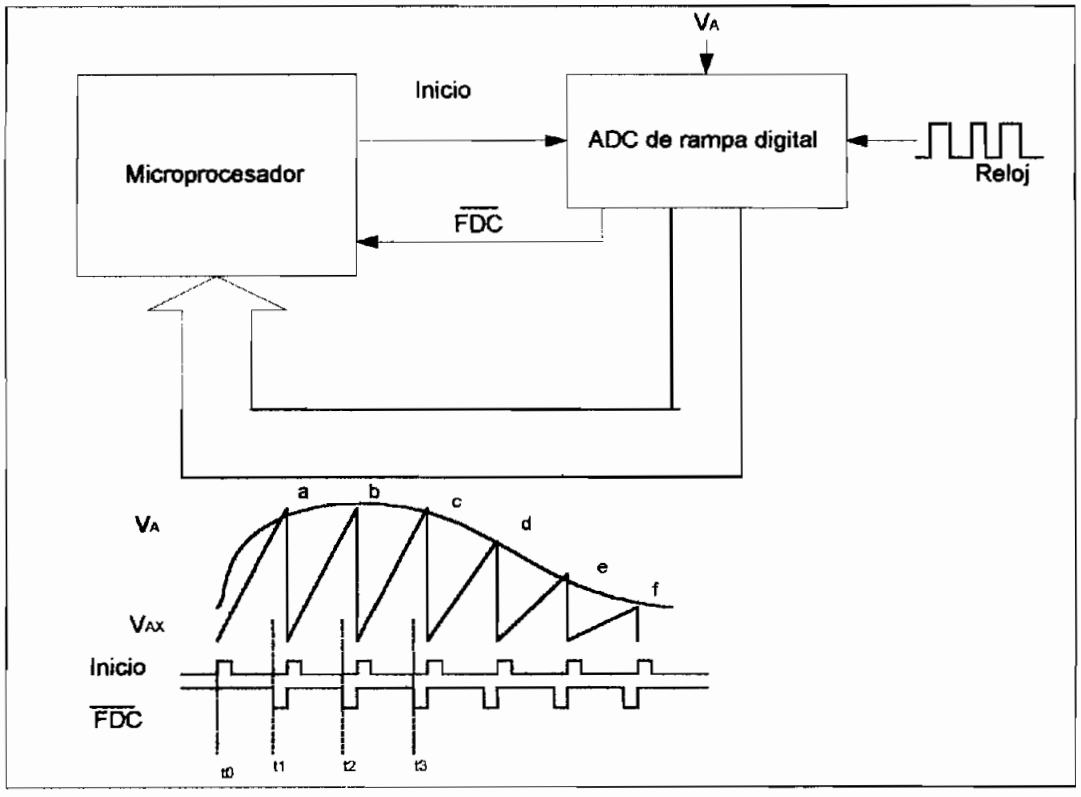


Figura 1. 6: Diagrama de bloques de un sistema de adquisición de datos

Las formas de onda de la figura 1.6 muestran la manera en la que un computador adquiere una versión digital de la señal analógica, V_A. La forma de onda cuantizada V_{AX} que se genera internamente en el ADC se muestra superpuesta en la forma de onda V_A. El proceso comienza en t_0, cuando la computadora genera un pulso Inicio para comenzar un ciclo de conversión A/D. La conversión se completa al tiempo t_1, cuando la rampa escalonada es mayor que V_A, y $\overline{FDC}$ pasa a bajo. Esta transición hacia abajo en $\overline{FDC}$ señala al computador que el ADC tiene una salida digital que ahora representa el valor de V_A en el punto a, y el computador cargará estos datos en su memoria.

El computador genera un nuevo pulso Inicio después de t_1, para comenzar un segundo proceso de conversión. Debe notarse que esto vuelve a poner en cero la rampa y a $\overline{FDC}$ en alto, ya que el pulso Inicio reinicia el contador del ADC. La segunda conversión termina al tiempo t_2 cuando la rampa vuelve a ser mayor que V_A. El computador carga después los datos digitales correspondientes en el punto b de su memoria. Estos pasos se repiten a t_3, t_4, etc.

El proceso que genera el pulso Inicio, examina $\overline{FDC}$ y carga datos del ADC en la memoria, se halla bajo el control del programa que el computador está ejecutando; este programa determinará cuántos puntos de datos de la señal analógica se almacenarán en la memoria del computador.

1.2.4.1. Reconstrucción de una señal digitalizada

Para el proceso de adquisición de datos, el ADC funciona a su máxima velocidad, dado que un nuevo pulso Inicio se genera inmediatamente después de que el computador adquiere el dato que produce el ADC como salida durante la conversión previa. Nótese que los tiempos de conversión no son constantes debido a los cambios en el valor de la entrada analógica. El computador almacena los datos digitales obtenidos en las conversiones previas; de esta manera, en su memoria se encuentra una versión digitalizada de la señal analógica; con frecuencia estos datos se emplean después para reconstruir una aproximación de la señal analógica original. Normalmente, la señal reconstruida es una buena

aproximación a la señal analógica original, y esto se debe a que no se presentan variaciones rápidas entre los puntos digitalizados. Si la señal analógica tuviese variaciones de alta frecuencia, el ADC no sería capaz de seguirlas, y la versión reconstruida sería mucho menos exacta; por esta razón, es importante mantener el tiempo de conversión lo suficientemente pequeño para que la señal analógica no cambie de manera significativa entre dos conversiones sucesivas.

1.3. CARACTERÍSTICAS DEL AUDIO DIGITAL[9]

1.3.1. FRECUENCIA DE MUESTREO

Según lo expuesto en la sección precedente, se nota que la frecuencia de muestreo para audio debe estar en el rango de los[3] 40 kHz. En los inicios del audio digital, los equipos de grabación de video fueron adaptados para almacenar muestras de audio, creando formas de onda seudo-video que podían llevar información binaria mediante niveles de blanco y negro. La frecuencia de muestreo de tales sistemas se veía obligada a estar relacionada sencillamente con la frecuencia y estructura de campo del estándar televisivo empleado, de forma que un número entero de muestras pudiera almacenarse en cada línea de televisión utilizable en el campo. El almacenamiento puede realizarse en un grabador monocromático, siguiendo dos estándares: 525 líneas a 60 Hz y 625 líneas a 50 Hz; por tanto, es posible encontrar una frecuencia que sea un múltiplo común de las dos y que también sea adecuada como frecuencia de muestreo.

Es posible deducir las frecuencias de muestreo permisibles en un sistema seudo-video multiplicando la frecuencia de campo por el número de líneas activas de un campo, y después, por el número de muestras de una línea. Si se eligen con cuidado los parámetros, es posible utilizar los sistemas de video tanto de 525/60 como 625/50 con una frecuencia de muestreo de 44.1 kHz.

---

[3] El ancho de banda del audio analógico se considera entre 20 Hz y 20 kHz. Por el teorema de Nyquist, la frecuencia de muestreo debe ser por lo menos 2 veces la frecuencia de la señal analógica, por tanto $f_S = 2 \cdot 20$ kHz = 40 kHz

Para demostrar la validez del cálculo, se propone el siguiente ejemplo. En el sistema de video de 60 Hz, existen 35 líneas de supresión, quedando 490 líneas por cuadro ó 245 líneas por campo para las muestras. Si se almacenan tres muestras por línea, la frecuencia de muestreo será la siguiente:

$$60 \times 245 \times 3 = 44.1 \text{ kHz}$$

Para el sistema de video de 50 Hz, hay 37 líneas de supresión, quedando 588 líneas activas por cuadro o 294 por campo, así que la frecuencia de muestreo viene expresada por:

$$50 \times 294 \times 3 = 44.1 \text{ kHz}$$

El audio digital, en realidad, tiene varias frecuencias de muestreo, estándares *de facto* en la mayoría de los casos, y que se enumeran a continuación:

recuencia	Descripción
5500 Hz	Un cuarto de la frecuencia de muestreo para computadores Macintosh
7333 Hz	Un tercio de la frecuencia de muestreo para computadores Macintosh
8000 Hz	Estándar telefónico, utilizado por las leyes A y μ
11025 Hz	Un cuarto de la frecuencia de muestreo para CD
16000 Hz	Utilizado por el estándar de compresión G.722
18900 Hz	Estándar CD-ROM/XA
22050 HZ	Mitad de la frecuencia de muestreo para CD
32000 Hz	Radio digital, sistema NICAM
37800 Hz	Estándar de alta calidad para CD-ROM/XA
44100 Hz	Frecuencia de muestreo para CD. Calidad máxima
48000 Hz	Frecuencia de muestreo de DAT (Digital Audio Tape) para uso doméstico

Tabla 1. 1: Frecuencias de muestreo más comunes para audio digital[10]

El proceso de muestreo se origina con un tren de impulsos de amplitud y período constantes. La amplitud de la forma de onda de audio modula el tren de impulsos, tal como sucede en la modulación AM. Debe tenerse cuidado en no

sobremodular el tren de impulsos, para lo cual se aplica un desplazamiento de corriente continua a la forma de onda analógica, de manera que el silencio se relacione con un nivel que alcance la mitad de los impulsos; esto hace que los recortes producidos por niveles de entrada excesivos sean simétricos.

El muestreo produce bandas laterales en el dominio de la frecuencia, aunque la portadora es un tren de impulsos y genera una serie infinita de armónicos. Las bandas laterales se repiten por encima y por debajo de cada armónico de la frecuencia de muestreo. La señal muestreada puede reconstruirse mediante la aplicación de un filtro pasabajos. Este filtro tiene una respuesta en frecuencia que impide el paso de bandas laterales y sólo deja pasar la señal en banda base.

Según el teorema de Nyquist, la frecuencia de muestreo debe ser por lo menos el doble de la frecuencia más alta de la señal de entrada. Si es que la señal de entrada no es adecuada para las frecuencias de muestreo estándar indicadas anteriormente, se produce el efecto de *aliasing*, donde ciertas frecuencias de salida no corresponden a sus similares de entrada, sino que se convierten en frecuencias diferenciales. Este efecto hace que no se pueda reconstruir fielmente la señal de entrada, lo que provoca molestias en el oyente. La calidad de un sistema de audio se mide de forma subjetiva, es decir, que tan bien se oiga un sonido.

Si se presenta *jitter* en la señal de reloj de muestreo, los efectos son nefastos, ya que las muestras son tomadas en tiempos incorrectos. La magnitud de la señal no deseada es proporcional a la pendiente de la forma de onda de audio, así la cantidad de *jitter* que puede tolerarse cae a 6 dB por octava[9]. A medida que aumenta la resolución del sistema, se reduce el grado de tolerancia al *jitter*. La naturaleza de la señal no deseada depende del espectro del *jitter*; si éste es aleatorio, el efecto se asemeja al ruido y es relativamente benigno a menos que la amplitud sea excesiva. Con estos antecedentes, aunque se intente mejorar la calidad aumentando la resolución, esto no servirá si es que se tiene inestabilidad en la señal de reloj de muestreo.

1.3.1.1. Sobremuestreo

El sobremuestreo consiste en utilizar una frecuencia de muestreo mucho mayor que la mínima frecuencia enunciada por el teorema de Nyquist. No se requiere la utilización del sobremuestreo para obtener una determinada calidad de señal, pero la conversión de la frecuencia de muestreo exige una gran eficiencia por parte de los componentes cuando se implementa un conversor. El proceso de sobremuestreo permite alcanzar una determinada calidad de señal sin necesidad de una tolerancia muy precisa de los componentes.

La utilización de una frecuencia de muestreo mucho mayor que la frecuencia mínima permite la realización de filtros *anti-aliasing* y de reconstrucción con pendientes de corte mucho más suaves. Por tanto, hay menos probabilidad de que se produzcan problemas de linealidad de fase y de rizado en la banda de paso de audio.

La información contenida en una señal analógica es bidimensional y puede representarse como un área, resultado del producto entre el ancho de banda y la relación señal a ruido linealmente expresada. Una misma cantidad de información puede transmitirse a través de un canal con una SNR cuyo valor sea la mitad del original si el ancho de banda utilizado es el doble, siempre y cuando el sistema de modulación sea perfecto.

La información de una señal analógica puede transmitirse mediante algún sistema de modulación analógico con cualquier combinación entre el ancho de banda y la SNR que proporcione la capacidad de canal apropiada. Si se sustituye el ancho de banda por la frecuencia de muestreo y la SNR por una función del número de bits por muestra, también debe ser válido para una señal digital, puesto que no se trata más que de un valor numérico análogo. Por tanto, aumentar la frecuencia de muestreo permite potencialmente reducir el número de bits por muestra sin que se produzca pérdida de información[9].

1.3.2. RECONSTRUCCIÓN DE LA SEÑAL

La respuesta impulsiva de un filtro pasabajos ideal de fase lineal es una forma de onda del tipo sen(x)/x, en el dominio del tiempo. Esta forma de onda pasa por cero periódicamente. Si la frecuencia de corte del filtro es la mitad de la frecuencia de muestreo, la señal discreta pasa por cero en los puntos de todas las demás muestras. A la salida del filtro, la amplitud en el centro de la muestra se debe sólo a esa muestra, puesto que el valor de todas las demás muestras es cero en ese instante; en otras palabras, la forma de onda de salida de tiempo constante debe unir los picos de las muestras de entrada.

La salida del filtro es la suma de muchos impulsos, y la forma de onda une de manera uniforme los picos de las muestras. Como consecuencia de la limitación de banda del filtro original *anti-aliasing*, la forma de onda analógica filtrada sólo puede estar entre los puntos de la muestra de una forma única.

Los filtros de salida que se implementan deben ser de pendiente finita por cuanto se pueden implementar de forma práctica. La pendiente de corte comienza en el borde de la banda requerida y, en consecuencia, la frecuencia de muestreo se incrementa un poco para llevar los productos de *aliasing* a un nivel aceptablemente bajo. No hay un factor absoluto por el que deba elevarse la frecuencia de muestreo; depende de los filtros disponibles y del nivel de los productos de *aliasing* que son aceptables.

Aunque pareciera que el filtro de reconstrucción es innecesario, ya que todas las frecuencias imagen se encuentran fuera del rango de audición humano; sin embargo, cualquier falta de linealidad, por leve que sea, en etapas posteriores provocará una enorme distorsión de intermodulación. La mayoría de los amplificadores de potencia de audio pierden mucha linealidad cuando se introducen señales que están más allá de la banda de audio.

El proceso de reconstrucción funciona exactamente como se indica sólo si los impulsos tienen una duración despreciable; sin embargo, la mayoría de los

DAC no cumple con este principio, ya que mantienen constante la salida analógica durante una parte sustancial del período de la muestra o incluso hasta que otro valor de muestra sea introducido; esto genera una forma de onda que se asemeja más a una escalera que a un tren de impulsos. Al fenómeno que se produce cuando los impulsos han sido extendidos en amplitud para igualarse al período de la muestra se le conoce como sistema de retención de orden 0.

Para el sistema descrito, se presenta un espectro de la forma x/sen(x). La respuesta en frecuencia cae a cero en la frecuencia de muestreo, y se sitúa a aproximadamente 4 dB menos en el límite de la banda de audio. Si el ancho de los impulsos es estable, la reducción de las frecuencias altas es constante y previsible, y un circuito ecualizador adecuado puede hacer que la respuesta global sea plana.

1.3.2.1. *Dither*

A niveles altos de señal, el error de cuantización se convierte en ruido. A medida que disminuye el nivel de audio, el error de cuantización se correlaciona con la señal y el resultado es la distorsión. Si el error de cuantización se puede separar de la señal de entrada de alguna manera, el sistema podrá mantenerse lineal con un poco de ruido. El *dither* realiza la función de separación, haciendo que la acción del cuantificador sea imprevisible y le da al sistema una base de ruido similar a la de un sistema analógico.

Todos los sistemas de audio digital utilizan un *dither* no sustractivo, en el que la señal de *dither* es introducida antes del proceso de cuantización y no se intenta eliminarla en el DAC. La aplicación de *dither* con anterioridad a un cuantificador convencional provoca inevitablemente una ligera reducción en la relación señal a ruido, pero es un pequeño precio a pagar por evitar las no linealidades. La aplicación del *dither* hace que muestras sucesivas encuentren los intervalos de cuantización en diferentes sitios de la escala de amplitud. El error de cuantización se vuelve una función del *dither*, en lugar de una función previsible de la señal de entrada. El error de cuantización no se elimina, pero la distorsión

CAPÍTULO II

CRIPTOGRAFÍA

2.1. INTRODUCCIÓN

La proliferación de los medios electrónicos en la vida diaria ha traído consigo una multitud de beneficios y problemas a la vez. Uno de los problemas más grandes es la seguridad de la información. Al aumentar la transferencia electrónica de información, también aumentan los peligros: un ladrón que pueda interceptar una transacción electrónica de dinero y usarlo en su propio beneficio es un ejemplo. Para resolver este tipo de problemas se creó la criptografía, la cual ha tenido un desarrollo exponencial en los últimos años, a pesar de que los métodos para proveer seguridad a la información existen desde el tiempo de los antiguos egipcios, es decir, 4000 años atrás. A continuación se presentan ciertos conceptos que son necesarios para comprender de mejor manera a esta ciencia.

2.1.1. TRANSMISOR, RECEPTOR Y ENTIDAD[2]

Estos conceptos son diferentes de los utilizados tradicionalmente en telecomunicaciones: un *transmisor* es aquella entidad en una comunicación entre dos partes, la cual transmite información de forma legítima; y un *receptor* es una entidad en una comunicación entre dos partes, la cual recibe la información de forma intencionada. Los dos conceptos anteriores llevan al de entidad. Una *entidad* es algo o alguien que envía, recibe o manipula información.

2.1.2. MENSAJES Y ENCRIPTACIÓN[1]

Existe un grupo de ciencias que estudian la seguridad de los mensajes, y sus definiciones son las siguientes:

- *Criptografía:* Es el arte y la ciencia de mantener los mensajes seguros.

- *Criptoanálisis:* Es el arte y la ciencia de romper la seguridad aplicada a los mensajes.
- *Criptología:* Es la rama de las matemáticas que engloba tanto a la criptografía como al criptoanálisis.

El proceso de ocultar mensajes y esconder su contenido, se conoce como *encriptación*; el proceso inverso, es decir, recuperar el mensaje oculto, se conoce como *desencriptación*.

En criptografía, a los mensajes se les denomina *texto llano*, el mismo que se denota por M o por P. El texto llano puede ser cualquier cosa: una corriente de bits, un archivo de texto, una imagen de mapa de bits, una corriente de voz digitalizada, una imagen de video digital... cualquier tipo de datos binarios, tanto para transmisión como para almacenamiento. En contraposición al texto llano, se encuentra el *texto cifrado*, resultado de un proceso de encriptación. El texto cifrado se denota por C, y se compone también de datos binarios; algunas veces puede ser del mismo tamaño que el texto llano, o puede ser más grande (al combinar la encriptación con la compresión, C puede ser menor que M; de todas maneras, no se debería usar compresión en mensajes encriptados; esto se debe a posibles errores en la corriente de bits, lo cual se trata más adelante, en la sección 2.4). La función de encriptación E, opera en M para producir C, o en notación matemática:

$$E(M) = C \qquad \text{(Ec. 2.1)}$$

En el proceso inverso, la función de desencriptación D opera en C para producir M:

$$D(C) = M \qquad \text{(Ec. 2.2)}$$

Debido a que el punto principal de encriptar y desencriptar un mensaje es recuperar el texto llano original, la siguiente identidad debe cumplirse:

$$D(E(M)) = M \qquad \text{(Ec. 2.3)}$$

2.1.3. OBJETIVOS DE LA CRIPTOGRAFÍA[2]

La criptografía tiene que cumplir cuatro objetivos básicos, para garantizar que la información en un canal sea enviada de forma segura, y por un transmisor autorizado. Los cuatro objetivos son los siguientes:

- *Confidencialidad.* La confidencialidad es un servicio utilizado para mantener el contenido de la información accesible para todos aquellos que estén autorizados a utilizarla. El secreto es un término sinónimo al de confidencialidad y privacidad. Existen varias aproximaciones a la provisión de confidencialidad, las cuales van desde la protección física hasta algoritmos matemáticos que hacen que los datos sean ininteligibles.
- *Integridad de datos.* Es un servicio que tiene que ver con la alteración no autorizada de datos. Para asegurar la integridad de los datos, se debe tener la habilidad de detectar la manipulación de los mismos por parte de entidades no autorizadas. Esta manipulación incluye cosas tales como la inserción, borrado y sustitución de datos.
- *Autenticación.* Es un servicio relacionado con la identificación; esta función se aplica tanto a las entidades como a la información. Dos entidades que se comunican entre sí, deberían ser capaces de identificarse una a la otra. La información que se encuentra en un canal de comunicaciones debería ser autenticada mediante su origen, fecha de envío, contenido, etc. Por estas razones este aspecto de la criptografía se subdivide en dos clases: autenticación de entidades y autenticación del origen de los datos, la cual provee integridad de datos de forma implícita.
- *No-repudio.* Es un servicio que evita que una entidad niegue sus actos o compromisos. Cuando se levanta una disputa debido a que una entidad niega haber tomado ciertas medidas, deben existir medios para resolver esta situación. Un procedimiento que envuelva una tercera entidad confiable es necesario.

Un objetivo fundamental de la criptografía es implementar adecuadamente estas cuatro áreas, tanto en la teoría como en la práctica.

Las cuatro condiciones anteriores son requerimientos vitales para la interacción "social" entre computadoras, y son análogas a las interacciones entre personas.

2.1.4. ALGORITMOS Y CLAVES

Un algoritmo criptográfico, denominado también cifrado, es la función matemática usada para la encriptación y la desencriptación; generalmente consta de dos funciones relacionadas: una para encriptar y la otra para desencriptar.

Si la seguridad de un algoritmo se basa en mantener secreta la forma en que el mismo trabaja, se trata de un algoritmo restringido. Este tipo de algoritmos tiene interés histórico, pero en la actualidad son utilizados sólo en aplicaciones de baja seguridad.[1]

La criptografía moderna combina los algoritmos con el uso de claves, cuya representación matemática es la letra K. Una clave puede tomar cualquier valor; su rango de posibles valores se denomina longitud de clave. La suscripción a la clave implica que las operaciones de encriptación y desencriptación dependen de la misma.[1] Las funciones de encriptación y desencriptación se convierten en:

$$\begin{aligned} E_K(M) &= C \\ D_K(C) &= M \\ D_K(E_K(M)) &= M \end{aligned} \qquad \text{(Ec. 2.4)}$$

Algunos algoritmos usan claves diferentes para las operaciones de encriptación y desencriptación[1], por consiguiente, la representación matemática de estas funciones se convierte en:

$$E_{K_1}(M) = C$$
$$D_{K_2}(C) = M \qquad \text{(Ec. 2.5)}$$
$$D_{K_2}(E_{K_1}(M)) = M$$

Toda la seguridad de los algoritmos modernos depende de la clave (o claves), no de los detalles de los mismos; así pues, es preferible que se publiquen y analicen, para poder descubrir su eficiencia real, mediante un severo criptoanálisis.

Un concepto que engloba todo lo expresado anteriormente es el de un criptosistema, el cual se compone de algoritmo, texto llano, texto cifrado, y claves. La figura 2.1. describe de forma gráfica un criptosistema:

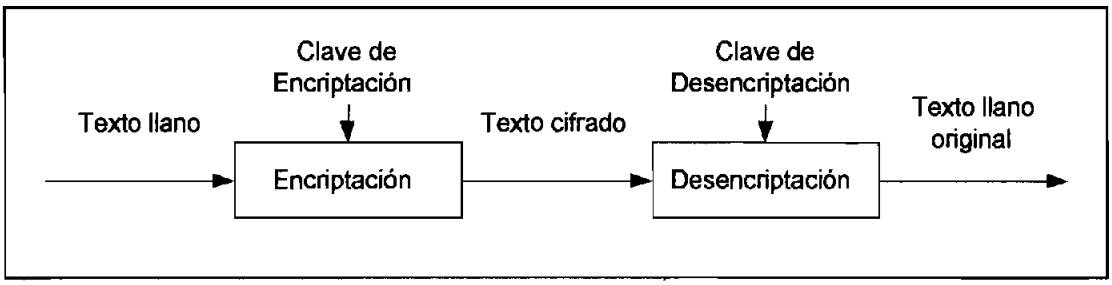


Figura 2. 1: Diagrama de bloques de un criptosistema[1]

2.1.5. ALGORITMOS SIMÉTRICOS

Existen dos tipos básicos de algoritmos basados en claves: simétricos y de clave pública. Los algoritmos simétricos, llamados también convencionales, son algoritmos en los cuales la clave de encriptación se puede calcular a partir de la clave de desencriptación, y viceversa, siendo una práctica común el uso de la misma clave para las dos operaciones; en este caso, los algoritmos se denominan de clave secreta o única, y requieren que el transmisor y el receptor acuerden el uso de la misma antes de poder comunicarse en forma segura. Debido a que toda la seguridad de este tipo de algoritmos descansa en la clave, ésta nunca debe ser divulgada.

La encriptación y desencriptación con algoritmos simétricos se notan mediante:

$$E_K(M) = C$$
$$D_K(C) = M \qquad \text{(Ec. 2.6)}$$

Los algoritmos simétricos se pueden dividir en dos categorías: los que operan sobre texto llano que se transmite un solo bit (o algunas veces un byte) a la vez; éstos son denominados algoritmos de flujo o cifrados de flujo. Otros operan en el texto llano sobre grupos de bits, los cuales se denominan bloques; estos algoritmos reciben el nombre de algoritmos de bloque. En los algoritmos de computador modernos, el tamaño típico de bloque es de 64 bits.

2.1.6. ALGORITMOS DE CLAVE PÚBLICA[1]

Los algoritmos de clave pública (también llamados algoritmos asimétricos) se diseñan de tal manera que la clave usada para la encriptación es diferente de aquella utilizada para desencriptación; además, ésta no puede ser calculada a partir de la clave de encriptación –al menos en un monto razonable de tiempo. Estos algoritmos toman su nombre debido a que la clave de encriptación se puede hacer pública: cualquier usuario puede usarla para encriptar un mensaje, pero sólo una persona con su respectiva clave de desencriptación puede leer este mensaje. En estos sistemas, la clave de encriptación a menudo se denomina clave pública, y la de desencriptación, clave privada.

La encriptación que usa clave pública se denota por:

$$E_K(M) = C \qquad \text{(Ec. 2.7)}$$

Aún cuando la clave pública y privada son diferentes, la desencriptación con su correspondiente clave privada se denota por:

$$D_K(C) = M \qquad \text{(Ec. 2.8)}$$

A veces, los mensajes serán encriptados con la clave privada y desencriptados con la clave pública, lo cual es común en las firmas digitales. A pesar de la posible confusión, estas operaciones se denotan respectivamente por:

$$E_K(M) = C$$
$$D_K(C) = M \qquad \text{(Ec. 2.9)}$$

2.1.7. CRIPTOANÁLISIS[1]

El objetivo principal de la criptografía es mantener el texto llano (o la clave, o ambos) oculto a los fisgones (también denominados adversarios, atacantes, interceptores, entrometidos, intrusos, oponentes, o simplemente el enemigo); se asume que los fisgones tienen acceso completo al canal de comunicaciones entre transmisor y receptor.

El criptoanálisis es el arte y la ciencia que permite recuperar el texto llano de un mensaje, sin necesidad de acceder a la clave. Los criptoanálisis exitosos pueden recuperar el texto llano o la clave, así como también pueden encontrar debilidades en un criptosistema que eventualmente conduzca a los resultados anteriores. (La pérdida de claves por métodos no criptoanalíticos se denomina compromiso). Un intento de criptoanálisis se denomina ataque.

Una suposición fundamental para el criptoanálisis, enunciada por el holandés A. Kerckhoffs en el siglo XIX[1],[2], es aquella que dice que la seguridad de un sistema debe residir enteramente en la clave; Kerckhoffs asume que el criptoanalista tiene los detalles completos del algoritmo criptográfico y su implementación; aún cuando los criptoanalistas en la realidad no siempre tienen información tan detallada, la primera suposición es enteramente válida.

Existen cuatro tipos fundamentales de ataques criptoanalíticos, pero cada uno de ellos supone que el criptoanalista posee un conocimiento completo del algoritmo de encriptación utilizado:

1. *Ataque sólo al texto cifrado.* El criptoanalista posee el texto cifrado de muchos mensajes, todos los cuales han sido encriptados usando el mismo algoritmo. El trabajo del criptoanalista es recuperar el texto llano de tantos mensajes como sea posible, o mejor aún, deducir la clave (o claves) utilizada para encriptar los mismos, y así poder desencriptar los siguientes mensajes que se logren interceptar.

   Dados: $C_1 = E_K(P_1),\ C_2 = E_K(P_2),\ \ldots C_i = E_K(P_i)$[1]

   Deducir: Cualquier $P_1, P_2, \ldots P_i; k$;o un algoritmo que permita inferir P_{i+1} de $C_{i+1} = E_K(P_{i+1})$

2. *Ataque a texto llano conocido.* En este ataque, el criptoanalista tiene acceso no solo al texto cifrado de muchos mensajes, sino también al texto llano de estos mensajes. Su trabajo es deducir la clave (o claves) utilizada para encriptar estos mensajes, o un algoritmo para desencriptar cualquier mensaje nuevo encriptado con la misma clave (o claves).

   Dados: $P_1, C_1 = E_K(P_1), P_2, C_2 = E_K(P_2), \ldots P_i, C_i = E_K(P_i)$

   Deducir: Cualquier *k*, o un algoritmo que permita inferir P_{i+1} de $C_{i+1} = E_K(P_{i+1})$

3. *Ataque a texto llano seleccionado.* En este tipo de ataque, el criptoanalista no solo tiene acceso al texto cifrado y al correspondiente texto llano de muchos mensajes, sino que puede escoger el texto llano adecuado. Este procedimiento es más poderoso que un ataque de texto llano conocido, ya que el criptoanalista puede escoger bloques de texto llano que pueden proporcionar más información acerca de la clave. Su trabajo es deducir la clave (o claves) utilizadas para encriptar los mensajes o un algoritmo para desencriptar cualquier mensaje nuevo encriptado con la misma clave (o claves).

---

[1] Como se explicó anteriormente, se puede utilizar indistintamente M o P para representar al texto llano

Dados: $P_1, C_1 = E_K(P_1), P_2, C_2 = E_K(P_2), \ldots P_i, C_i = E_K(P_i)$, de los cuales el criptoanalista puede escoger $P_1, P_2, \ldots P_i$

Deducir: Cualquier *k*, o un algoritmo que permita inferir P_{i+1} de $C_{i+1} = E_K(P_{i+1})$

4. *Ataque adaptable a texto llano seleccionado.* Este es un caso especial del ataque a texto llano seleccionado. El criptoanalista no solo puede escoger el texto llano, sino que puede modificar su elección basándose en los resultados de encriptaciones previas.

Existen tres tipos adicionales de ataques criptoanalíticos, los cuales son:

5. *Ataque a texto cifrado seleccionado.* El criptoanalista puede escoger entre varios textos cifrados diferentes, y tiene acceso al texto llano. Por ejemplo, el criptoanalista tiene acceso a una caja negra que realiza desencriptación automática; su trabajo es deducir la clave.

Dados: $C_1, P_1 = D_K(C_1), C_2, P_2 = D_K(C_2), \ldots C_i, P_i = D_K(C_i)$

Deducir: *K*

Este ataque se aplica principalmente a algoritmos de clave pública, aunque también puede resultar efectivo contra algoritmos simétricos.

6. *Ataque a clave seleccionada.* A pesar de su nombre, este ataque no implica que el criptoanalista puede escoger la clave, sino que posee conocimiento acerca de la relación entre claves diferentes. Es extraño, oscuro y no muy práctico.

7. *Criptoanálisis de arbitrariedad.* El criptoanalista amenaza, chantajea, o tortura a alguien hasta que le proporcione la clave. El soborno se denomina comúnmente ataque de compra de clave. Estas son formas muy poderosas de ataque y a menudo la mejor forma de romper un algoritmo.

Existen ciertas reglas que deben seguirse para asegurar que un criptosistema sea razonablemente invulnerable a los ataques criptoanalíticos, y son las siguientes:

- No suponer que el desconocimiento de los detalles de un algoritmo hará que el sistema sea invulnerable.
- No guardar en secreto un algoritmo, sino entregarlo a la comunidad científica para que lo analice.
- Si es que un algoritmo se está utilizando con software comercial, no debe creerse que alguien no podrá desensamblar el código del algoritmo y revertir su operación.

Los enunciados anteriores llevan a una conclusión muy importante: los mejores algoritmos son aquellos que se han publicado, han sido atacados por los mejores criptoanalistas del mundo por años, y aún no se pueden romper.

2.1.8. SEGURIDAD DE LOS ALGORITMOS[1]

Cada tipo de algoritmo posee diferentes niveles de seguridad, los cuales dependen de la dificultad para romperlo. La probabilidad de mantener datos seguros es más alta si se cumplen las siguientes condiciones: que el costo de romper un algoritmo sea más alto que el valor de los datos encriptados; que el tiempo requerido para realizar un ataque sea mayor que el período en el cual los datos encriptados deben permanecer secretos; y, que el total de los datos encriptados con una clave única sea mayor que la cantidad de datos empleados en ejecutar el ataque.

Lars Knudsen clasificó las formas de romper un algoritmo en varias categorías, las que se presentan aquí según su fortaleza:

1. *Ruptura total.* Un criptoanalista encuentra la clave, K, de tal forma que $D_K(C) = P$.

2. *Deducción global.* Un criptoanalista encuentra un algoritmo alternativo, A, equivalente a $D_K(C)$, sin conocer K.
3. *Deducción local.* Un analista encuentra el texto llano de un texto cifrado interceptado.
4. *Deducción de la información.* El criptoanalista logra obtener un poco de información acerca de la clave o del texto llano.

Un algoritmo es incondicionalmente seguro si no existe suficiente información para recuperar el texto llano, sin importar cuanto texto cifrado posea el criptoanalista; de hecho, solo los bloques de un sentido son invulnerables, aún teniendo recursos infinitos. Todos los criptosistemas son vulnerables a un ataque sólo al texto cifrado, simplemente intentando cada clave posible, una por una, y observando si el texto llano resultante es significativo. A este ataque se le denomina de fuerza bruta.

La criptografía está más interesada en los criptosistemas que son imposibles de romper por medio de computadores; así, un algoritmo se considera computablemente seguro (denominado en algunas ocasiones fuerte), si no puede ser vulnerado con los recursos disponibles, tanto presentes como futuros.

Se puede medir la complejidad de un ataque de diversas formas:

1. *Complejidad de los datos.* Cantidad de datos necesaria como entrada para el ataque.
2. *Complejidad de procesamiento.* Tiempo necesario para ejecutar el ataque. Se denomina a menudo *factor de trabajo*.
3. *Requerimientos de almacenamiento.* Cantidad de memoria necesaria para realizar el ataque.

Como regla general, para medir la complejidad de un ataque se toma el menor de estos tres factores; sin embargo, algunos ataques envuelven a los tres factores: un ataque más rápido sólo podría realizarse mediante recursos de almacenamiento grandes.

La complejidad se expresa en órdenes de magnitud, por ejemplo, si un algoritmo tiene una complejidad de procesamiento de 2^{128}, entonces se requieren 2^{128} operaciones para romper el algoritmo. Aún si se tuviera a disposición un millón de procesadores paralelos, operando a una velocidad de 100 MIPS (millones de instrucciones por segundo), tomaría 10^{17} años recuperar la clave. Este período de tiempo representa un billón de veces la edad del Universo.

2.2. PROTOCOLOS CRIPTOGRAFICOS[1]

2.2.1. INTRODUCCIÓN A LOS PROTOCOLOS

Un protocolo es una serie de pasos, que envuelven a dos o más entidades, diseñados para cumplir una tarea. Esta definición podría parecer muy vaga (se puede confundir un protocolo con un mecanismo), si no fuera acompañada por las siguientes características:

- Cada entidad envuelta en un protocolo debe conocerlo, así como todos sus pasos, para poder avanzar en el mismo.
- Cada entidad envuelta en un protocolo debe estar de acuerdo en seguirlo.
- El protocolo no debe ser ambiguo; cada paso debe estar bien definido, y no debe darse oportunidad a malas interpretaciones.
- El protocolo debe estar completo; debe existir una acción específica para cada situación posible.

Como una consecuencia de lo expresado, se puede definir que un protocolo criptográfico es aquel que usa criptografía, es decir, se ocupa de resolver problemas que envuelven los objetivos criptográficos planteados anteriormente. El punto principal de los protocolos criptográficos es prevenir o detectar intrusiones y fraudes, y por lo tanto, se debe añadir una característica adicional:

- No debe ser posible hacer o aprender más que aquello que se especifica en el protocolo.

2.2.1.1. Propósito de los protocolos

En la vida diaria, existen protocolos informales para casi todo; se han vuelto tan comunes, que nadie piensa en ellos. Estos protocolos han evolucionado con el tiempo, todo el mundo los conoce, y trabajan razonablemente bien.

Debido a que en la actualidad los computadores intervienen cada vez más en las relaciones humanas, los encuentros interpersonales son cada vez menores. Los computadores necesitan protocolos formales para hacer las mismas cosas que la gente hace por instinto. La mayoría de los protocolos interpersonales cuentan con la presencia de individuos para garantizar honradez y seguridad.

En la interacción remota se desconfía de todos: desde el usuario con quien se interactúa, hasta el administrador de la red. Por la vía de los protocolos formales, se pueden examinar las formas en las cuales la gente deshonesta los puede alterar, y, por consiguiente, hacer inmunes a los protocolos contra estas alteraciones.

2.2.1.2. Protocolos arbitrados

En este tipo de protocolos se tiene tres entidades: transmisor, receptor y una tercera entidad, interpuesta entre los dos, denominada árbitro, en la cual se confía para completar el protocolo. El árbitro no debe tener intereses creados en el protocolo, y ninguna alianza con cualquiera de las otras entidades envueltas en el mismo. La confianza depositada en el árbitro descansa en varios factores: todas las entidades envueltas en el protocolo deben aceptar lo que dice como verdadero, lo que hace como correcto, y que desempeñará a satisfacción su parte en el protocolo. Los árbitros pueden ayudar a completar protocolos entre dos entidades mutuamente desconfiadas.

Existen varios problemas con los árbitros, por ejemplo:

- Es fácil encontrar y confiar en una tercera parte neutral, si se conoce quien es y se puede ver su rostro. Dos partes sospechosas una de la otra, también sospecharán de un árbitro desconocido que se localice en cualquier punto de una red.
- Una red debe cubrir el costo para mantener un árbitro.
- Existe un retardo inherente en cualquier protocolo arbitrado.
- El árbitro debe negociar con cada transacción; es un "cuello de botella" en implementaciones a larga escala de cualquier protocolo. Incrementar el número de árbitros en la red puede mitigar el problema, pero eso aumentaría el costo.
- Debido a que cada parte de la red debe confiar en el árbitro, éste representa un punto vulnerable para cualquiera que intente trastornar la red.

2.2.1.3. Protocolos adjudicados

A causa del alto costo de contratar árbitros, los protocolos arbitrados se pueden subdividir en dos subprotocolos de bajo nivel. El primero es un subprotocolo no arbitrado, ejecutado cada vez que las entidades quieren completar el protocolo. El segundo es un subprotocolo arbitrado, ejecutado sólo en circunstancias excepcionales (cuando existe una disputa). Este tipo especial de árbitro se denomina adjudicador.

Un adjudicador es una tercera entidad confiable y desinteresada; de forma distinta a un árbitro, no está envuelto en todos los protocolos. El adjudicador es llamado sólo para determinar si un protocolo fue ejecutado de forma correcta.

Existen protocolos adjudicados para redes. Estos protocolos cuentan con que las entidades sean honestas, pero, si alguien sospecha que existen trampas, una comunicación puede existir sólo si es que una tercera entidad confiable puede determinar si alguien hizo trampa. En un buen protocolo adjudicado, el adjudicador puede determinar también la identidad del tramposo. En vez de prevenir las trampas, los protocolos adjudicados las detectan.

2.2.1.4. Protocolos de auto detección

Este tipo de protocolos es el mejor. El protocolo en sí mismo garantiza imparcialidad. No se requiere un árbitro para completarlo, ni un adjudicador para resolver las disputas. El protocolo se construye de tal forma que no pueden existir disputas. Si una de las entidades intenta hacer trampa, la otra inmediatamente lo detecta y el protocolo se detiene. Lo que sea que la entidad tramposa esperaba que pase, no sucede.

2.2.1.5. Ataques contra los protocolos

Se pueden dirigir ataques criptográficos contra los algoritmos utilizados en los protocolos, contra las técnicas criptográficas utilizadas para implementar los algoritmos y protocolos, o contra los protocolos mismos.

Las entidades envueltas en un protocolo pueden intentar diferentes maneras de atacarlo. Alguna entidad que no está envuelta en un protocolo puede fisgar en porciones del mismo, o en todo. A esto se denomina un ataque pasivo, ya que el atacante no daña el protocolo; todo lo que puede hacer es observarlo e intentar obtener información del mismo. Este tipo de ataque corresponde a uno de sólo texto cifrado. Debido a que los ataques pasivos son difíciles de detectar, los protocolos tratan de prevenir este tipo de ataques en vez de detectarlos.

De forma alternativa, un atacante podría intentar alterar un protocolo para su propio beneficio: podría fingir ser una entidad autorizada, introducir nuevos mensajes en el protocolo, borrar mensajes existentes, sustituir un mensaje por otro, reproducir mensajes antiguos, interrumpir un canal de comunicaciones, o alterar información almacenada en una red. Este tipo de actividades recibe el nombre de ataques activos, ya que requieren intervención activa. La forma de estos ataques depende de la red.

Los atacantes pasivos tratan de obtener información acerca de las entidades envueltas en un protocolo. Recogen los mensajes que están pasando

entre varias entidades e intentan criptoanalizarlos. Los atacantes activos, por otra parte, pueden tener objetivos diferentes. El atacante podría estar interesado en obtener información, degradar el desempeño del sistema, corromper la información existente, u obtener acceso no autorizado a los recursos de la red.

Los ataques activos son mucho más serios, especialmente en protocolos en los cuales las diversas entidades involucradas no confían una en la otra. El atacante no tiene que ser un completo extraño; puede ser un usuario legítimo del sistema, o el administrador del mismo. Podrían existir muchos atacantes activos trabajando juntos.

Es posible también que el atacante sea una de las entidades envueltas en un protocolo; puede mentir durante el mismo, o no seguirlo de ninguna manera. Este tipo de atacante se conoce como estafador. Los estafadores pasivos siguen un protocolo, pero tratan de obtener más información que aquella que el protocolo les permite. Por otra parte, los estafadores activos desbaratan un protocolo en progreso en su intento por hacer trampa.

Es muy difícil mantener la seguridad de los protocolos si la mayoría de las entidades envueltas son estafadores activos, pero algunas veces es posible legitimar a las entidades para detectar si una estafa activa se está desarrollando. Ciertamente, los protocolos deberían ser seguros contra la estafa pasiva.

2.2.2. PROTOCOLOS DE CRIPTOGRAFÍA SIMÉTRICA

Para ilustrar este tipo de protocolos, se ejecuta un ejemplo[1] entre dos entidades, Alice y Bob, con los siguientes pasos:

- Alice y Bob utilizan un criptosistema determinado.
- Alice y Bob acuerdan una clave.
- Alice toma su mensaje de texto llano y lo encripta utilizando el algoritmo y la clave acordados. Esto crea un mensaje de texto cifrado.
- Alice envía su mensaje de texto cifrado a Bob.

- Bob desencripta el mensaje de texto cifrado con el mismo algoritmo y clave y lo lee.

Si es que una tercera entidad estuviera interfiriendo la comunicación, intentaría obtener el mensaje para criptoanalizarlo. Esto es insuficiente ya que los algoritmos actuales son fuertes; por consiguiente, intentaría obtener la clave y el algoritmo, para así hacerse pasar por la entidad Bob en el protocolo, y así, enterarse de cualquier mensaje, sin tener que emprender ataques contra el texto cifrado.

Un buen criptosistema es aquel en el cual toda la seguridad es inherente al conocimiento de la clave; es por eso que el manejo de claves es tan importante en la criptografía. En un protocolo simétrico, Alice y Bob pueden acordar el algoritmo en público, pero la clave debe acordarse totalmente en secreto, y debe permanecer así antes, durante y después de ejecutar el protocolo (tanto tiempo como el mensaje deba permanecer secreto), de otra manera, el mensaje no estará seguro.

Un atacante activo, quien tomará el nombre de Mallory para los siguientes ejemplos, podría desarrollar otro tipo de actividades, como por ejemplo, romper el canal de comunicaciones, asegurándose que Alice no se pueda comunicar con Bob de forma alguna, y así, interceptar los mensajes de Alice y sustituirlos con los suyos. Si conoce la clave (interceptándola o criptoanalizándola), podría encriptar su propio mensaje y enviarlo a Bob en lugar del mensaje original. Bob no tendría forma de saber que el mensaje no viene de Alice. Si Mallory no conoce la clave, podría crear un mensaje de reemplazo que al desencriptarse produciría basura. Bob, pensando que el mensaje provino de Alice, podría concluir que la red, o Alice tienen serios problemas.

En suma, los criptosistemas simétricos tienen los siguientes problemas:

- Las claves deben ser distribuidas en secreto; son tan valiosas como los mensajes que encriptan, debido a que el conocimiento de las claves implica conocimiento de todos los mensajes.
- Si se compromete una clave (por robo, descifrado, extorsión, soborno, etc)., una tercera entidad puede desencriptar todo el tráfico de mensajes encriptados con esa clave, y además puede fingir ser una de las entidades legítimas y producir mensajes falsos para engañar a la otra entidad.
- Asumiendo que una clave individual se usa para cada par de usuarios en una red, el número total de claves se incrementa rápidamente con el número de usuarios. Una red de n usuarios requiere $n \cdot (n - 1)/2$ claves. Este problema se puede minimizar manteniendo el número de usuarios pequeño, pero esto no es siempre posible.

2.2.3. GENERACIÓN DE SECUENCIAS ALEATORIAS Y SEUDO-ALEATORIAS

Ciertos algoritmos y protocolos criptográficos utilizan números aleatorios para su operación; sin embargo, los generadores de números aleatorios no lo son debido a que no tienen que serlo; la mayoría de las aplicaciones simples, necesitan tan pocos números aleatorios para su operación, que raramente se nota que los que utilizan no lo son; sin embargo, la criptografía es extremadamente sensible a las propiedades de los generadores de números aleatorios. El uso de generadores de números aleatorios deficientes conduce a obtener resultados deficientes, y, si el criptosistema depende de números aleatorios, lo anterior es lo que menos se puede desear.

El problema es que un generador de números aleatorios no produce una secuencia aleatoria, ya que es imposible producir algo realmente aleatorio en un computador. Los computadores son máquinas determinísticas, y por tanto, sus operaciones son totalmente predecibles. Un computador sólo puede estar en un número finito de estados (un número finito grande), y por tanto, los datos de salida serán siempre una función determinística de los datos de entrada. Esto significa que cualquier generador de números aleatorios que sea implementado en un

3.3. PROGRAMACIÓN DEL SISTEMA SIMULADOR

3.3.1. DESCRIPCIÓN DE LA INTERFAZ GRÁFICA DE USUARIO

El sistema simulador se implementa mediante el paquete de software MATLAB® versión 5.3. Las razones para esta elección son las siguientes:

- Facilidad de implementación debido a su especialización en el manejo de matrices.
- Desarrollo de interfaces gráficas de usuario, lo cual permite manejar al sistema de una manera sencilla y organizada.
- Manejo directo de adquisición de voz, a través de *Data Acquisition Toolbox*, incluida en el paquete.

El sistema simulador puede visualizarse de mejor manera mediante el diagrama de bloques de la figura 3.9:

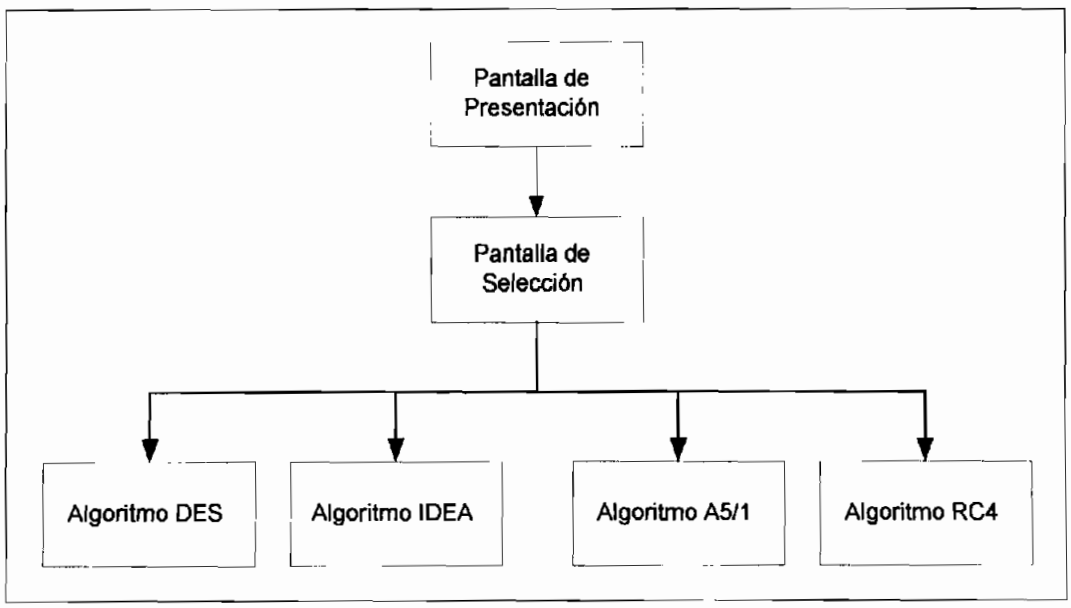


Figura 3. 9: Diagrama de bloques del sistema simulador

La descripción de cada uno de los bloques se ve en los siguientes puntos. Cuando se necesite describir un comando específico de MATLAB, se lo hará en la sintaxis de este entorno de programación.

3.3.1.1. Pantalla de presentación

La pantalla de presentación constituye la puerta de entrada hacia el sistema simulador. Consiste de una carátula que muestra el nombre del proyecto, el nombre del autor y la fecha de finalización. El botón "Entrar" permite al usuario acceder hacia la pantalla de selección, cerrando la pantalla de presentación. La forma de lograr este efecto es asignar a cada botón una llamada específica, así, el botón "Entrar" llama al programa Fase2.m y cierra la pantalla de presentación mediante el comando `close('Sistema Simulador de Encriptación de Audio')`. El botón "Salir" ejecuta exclusivamente el comando anterior.

La pantalla de presentación tiene la forma que se indica en la figura 3.10:

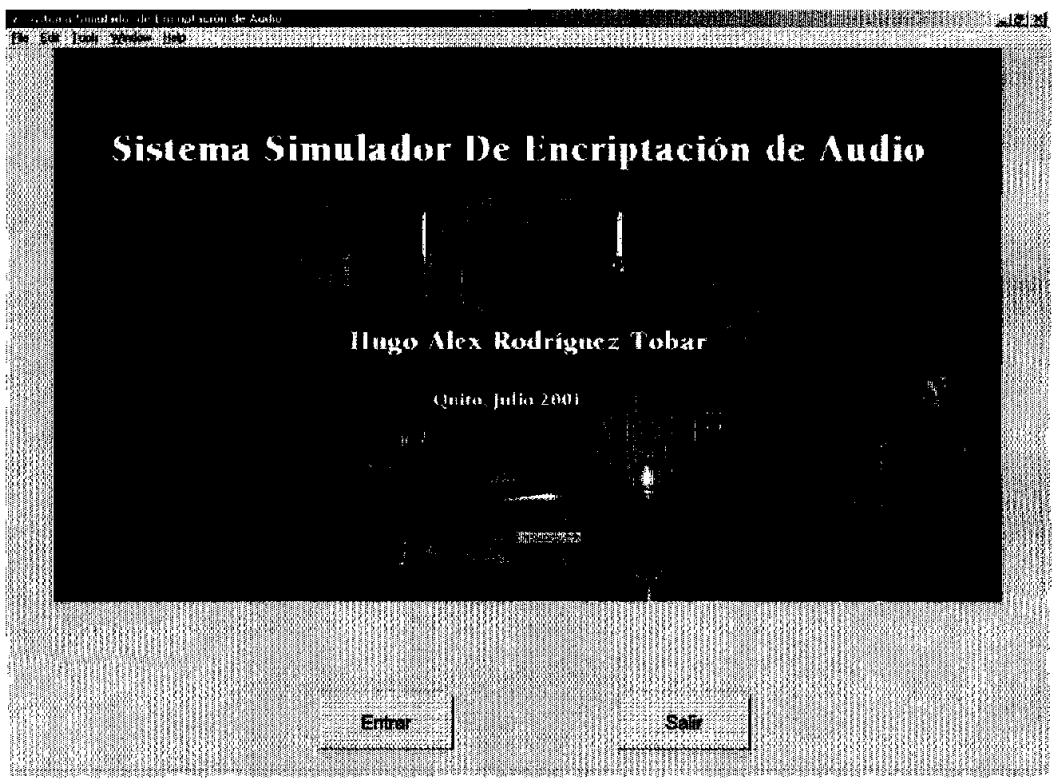


Figura 3. 10: Pantalla de presentación del Sistema Simulador

3.3.1.3. Pantalla de control

La pantalla de control tiene un formato único, que es el que se muestra en la figura 3.12

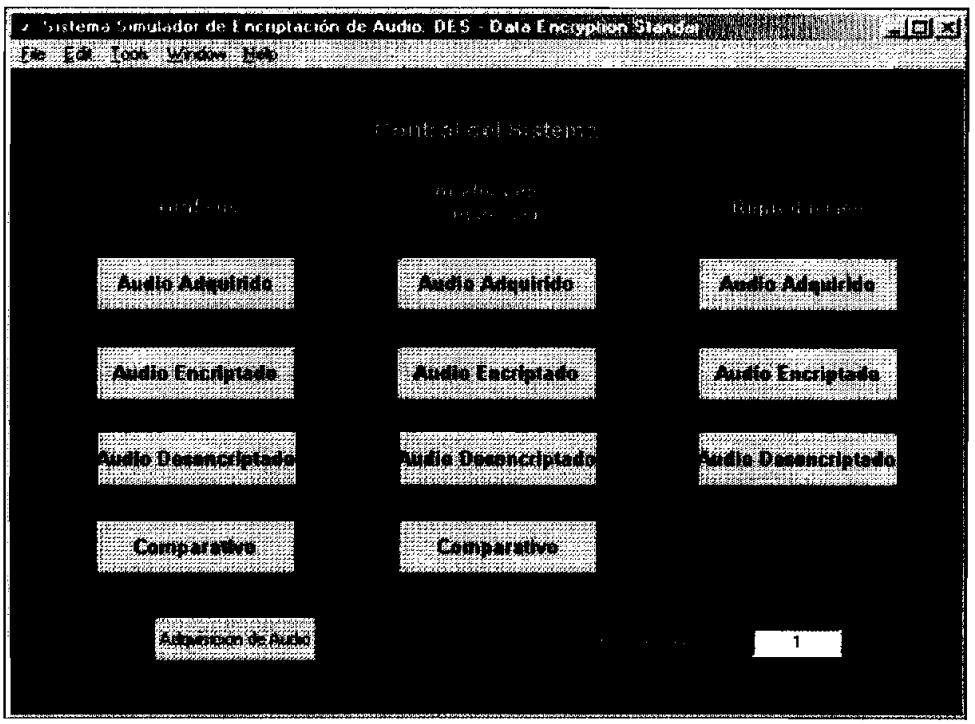


Figura 3. 12: Pantalla de control del Sistema Simulador

Aunque la pantalla de control tiene un formato único, las llamadas que realiza son específicas para cada algoritmo. Las opciones que posee esta pantalla son: Gráficos, Análisis en Frecuencia, Reproducción, Adquisición de Audio y Tiempo de Adquisición. La opción Gráficos permite visualizar las señales de audio adquirido, encriptado y desencriptado en el dominio del tiempo. La opción Análisis en Frecuencia permite realizar lo mismo que la opción anterior, pero en el dominio de la frecuencia. La opción Reproducción permite escuchar cada una de las señales, a través de los parlantes conectados a la tarjeta de sonido de un computador. La opción Adquisición de Audio llama a los respectivos archivos adqu*.m, donde * significa des, idea, a5 y rc4, cuyo argumento de entrada lo constituye el valor que se obtiene de la opción Tiempo de Adquisición, el cual tiene un valor predeterminado de 1 [s], pero puede ser alterado por el usuario a su conveniencia.

El sistema se provee con muestras adquiridas previamente, para poder presentarlas en las opciones Gráficos, Análisis en Frecuencia y Reproducción. En la primera ejecución aparecerán las muestras que se describen en el siguiente capítulo, sin embargo, para realizar otra encriptación es necesario que el usuario ingrese primero el tiempo de ejecución (si es diferente de 1 [s]) y luego presione el botón Adquisición de Audio.

En las secciones siguientes se explica la forma de operación de cada pantalla de control en forma general, citando las diferencias entre cada una de ellas.

3.3.1.3.1. Operación general de la pantalla de control

Aunque cada algoritmo posee su propia pantalla de control, la programación de las mismas es casi exactamente igual, excepto aquellas partes que llaman a operaciones específicas para cada algoritmo. La programación general de la pantalla de control se puede reducir al diagrama de flujo de la figura 3.13

Como se explica en el diagrama, cada botón tiene asignada una llamada para el archivo *op.m, donde * significa des, idea, a5 y rc4. Los archivos *op.m permiten ejecutar todas las funciones expuestas en la pantalla de control. Cada archivo *op.m es exclusivo para cada algoritmo de encriptación, aunque la programación de los mismos es prácticamente idéntica.

El diagrama de flujo de la figura 3.14 refleja la forma de operación general de los archivos *op.m. Aunque mirando el diagrama se puede pensar que se trata de una estructura de decisión múltiple común tal como if, en realidad se trata de una estructura switch ... case ... end, común en el lenguaje C. La ventaja de este tipo de estructura radica en el hecho de la ejecución de un solo parámetro a la vez, es decir, por ejemplo, si se aplasta el botón "Audio Adquirido" de la opción Gráficos en la pantalla de control, ejecutará solo la opción 'plotadqu' y se detendrá

la ejecución del programa, haciendo que la "corrida" de este programa sea más rápida y eficiente.

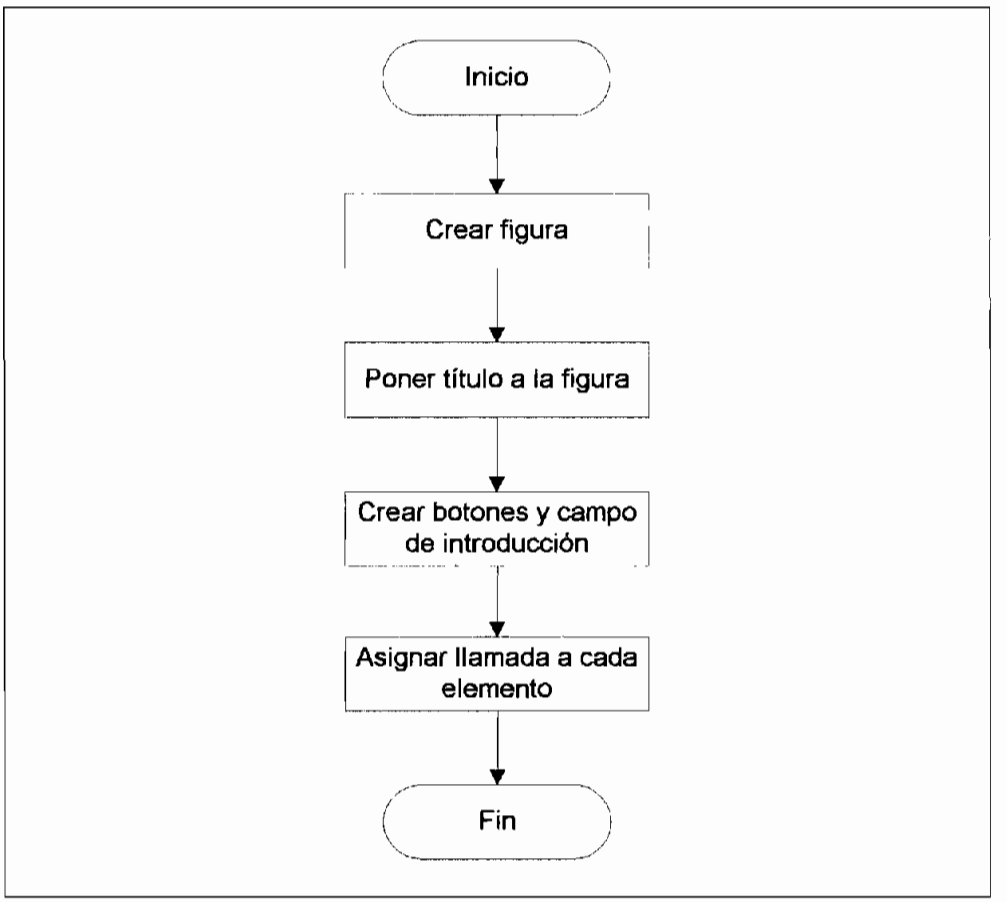


Figura 3. 13: Diagrama de flujo de la pantalla de control

Los nombres de las variables que se presentan en el diagrama de flujo son generales para todos los algoritmos, sin embargo, sus valores son exclusivos para cada uno de ellos. Estas variables son generadas por los archivos adqu*.m, de los cuales se habla más adelante.

Cada programa *op.m posee una función local denominada localDaqfft, la cual sirve para realizar el cálculo de la transformada rápida de Fourier para cada señal. El diagrama de flujo correspondiente a esta función se muestra en la figura 3.15

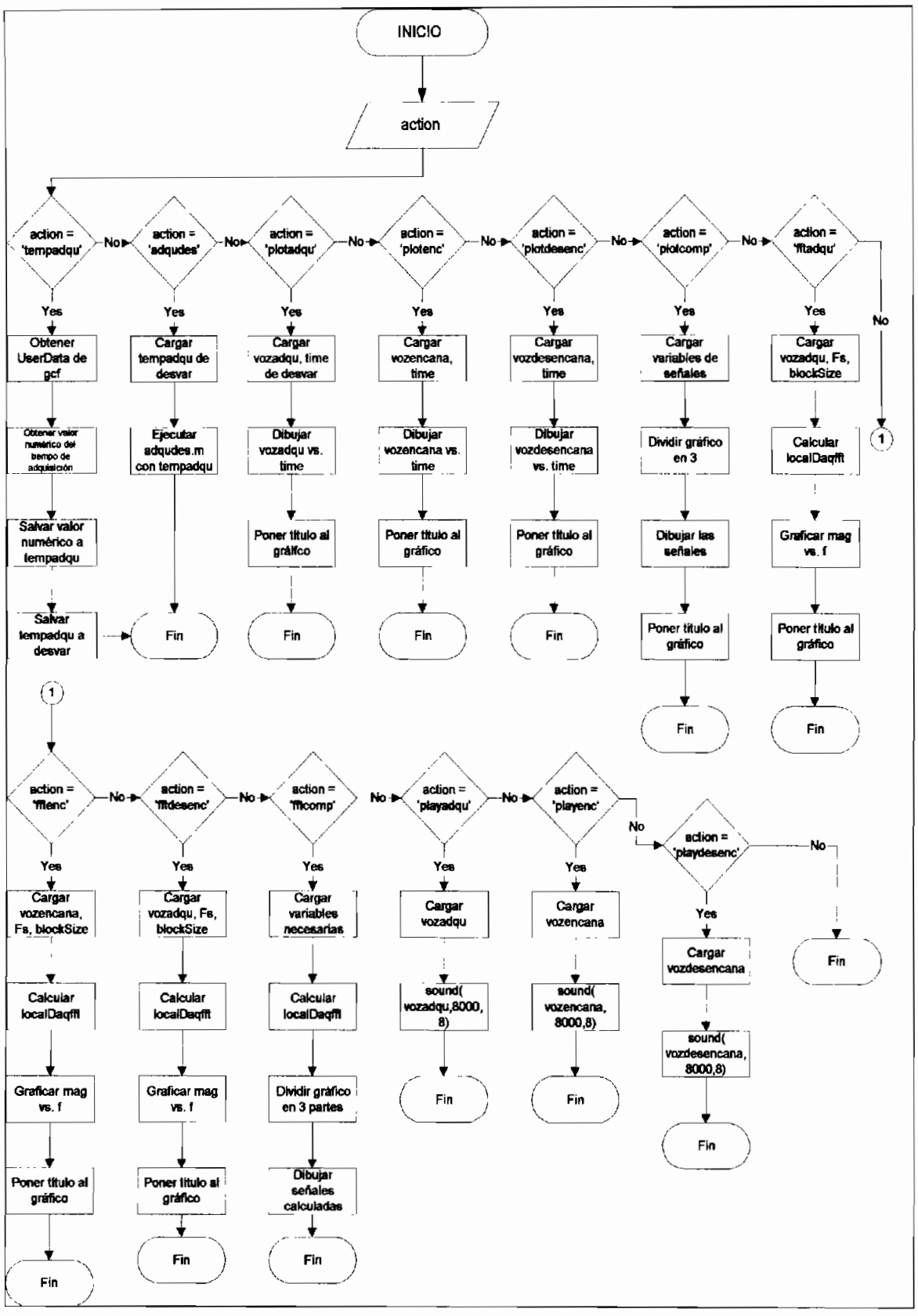


Figura 3. 14: Diagrama de flujo de *op.m

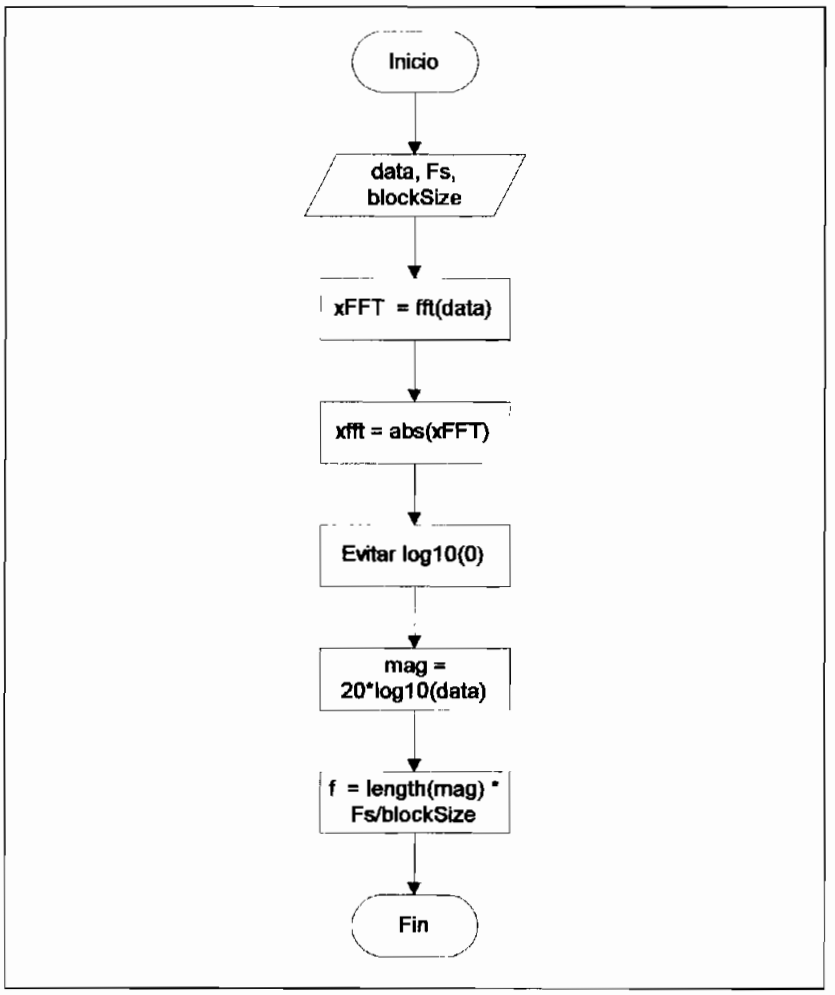


Figura 3. 15: Diagrama de flujo de la función local localDaqfft

3.3.1.4. Operación general del programa de adquisición de audio

Para realizar las funciones principales del sistema simulador, se crearon los programas adqu*.m (el significado de * se explica en el punto anterior). Los programas adqu*.m, en general, se encargan de calcular la clave para cada algoritmo, realizar la adquisición de audio, digitalizar la señal adquirida, encriptarla y desencriptarla, para, por último, crear un archivo propio de MATLAB denominado *var.mat, en el cual se guardan todas las variables necesarias para el funcionamiento del sistema simulador. Las variables, así como la función que cumplen, se listan en la tabla 3.10

NOMBRE DE LA VARIABLE	FUNCIÓN QUE CUMPLE
ActualRate	Número de muestras por disparo del objeto analoginput
Fs	Frecuencia de muestreo de la señal de audio
Key* (* = DES, IDEA, A5, RC4)	Clave del algoritmo
duracion	Tiempo de adquisición de la señal
tempadqu	Tiempo de adquisición almacenado por los programas *op.m
time	Escala de tiempo utilizada para los gráficos de las señales
vozadqu	Señal de audio adquirida
vozadqudig	Señal de audio adquirida y digitalizada
vozencdig	Señal de audio encriptada digital
vozencana	Señal de audio encriptada analógica
vozdesencdig	Señal de audio desencriptada digital
vozdesencana	Señal de audio desencriptada analógica

Tabla 3. 10: Variables relacionadas con los archivos *var.mat

En los programas de adquisición adqudes.m y adquidea.m, es necesario dividir a la señal de audio adquirida digital, vozadqudig en bloques de 64 bits para su procesamiento en los algoritmos DES e IDEA. En los algoritmos de flujo, esto no es necesario debido a la operación de los mismos (encriptación bit a bit o byte a byte). El diagrama de flujo general de los programas adqu*.m, se muestra en la figura 3.16.

3.3.2. PROGRAMACIÓN DE LOS ALGORITMOS

3.3.2.1. Implementación del algoritmo DES

El algoritmo DES se implementa mediante los archivos descip.m y desdec.m, los cuales cumplen con todo el algoritmo DES descrito en una sección precedente.

El archivo descip.m constituye la parte de encriptación del algoritmo DES. Este archivo utiliza como argumento de entrada un vector binario de longitud 64, el cual simula un bloque de texto llano de 64 bits.

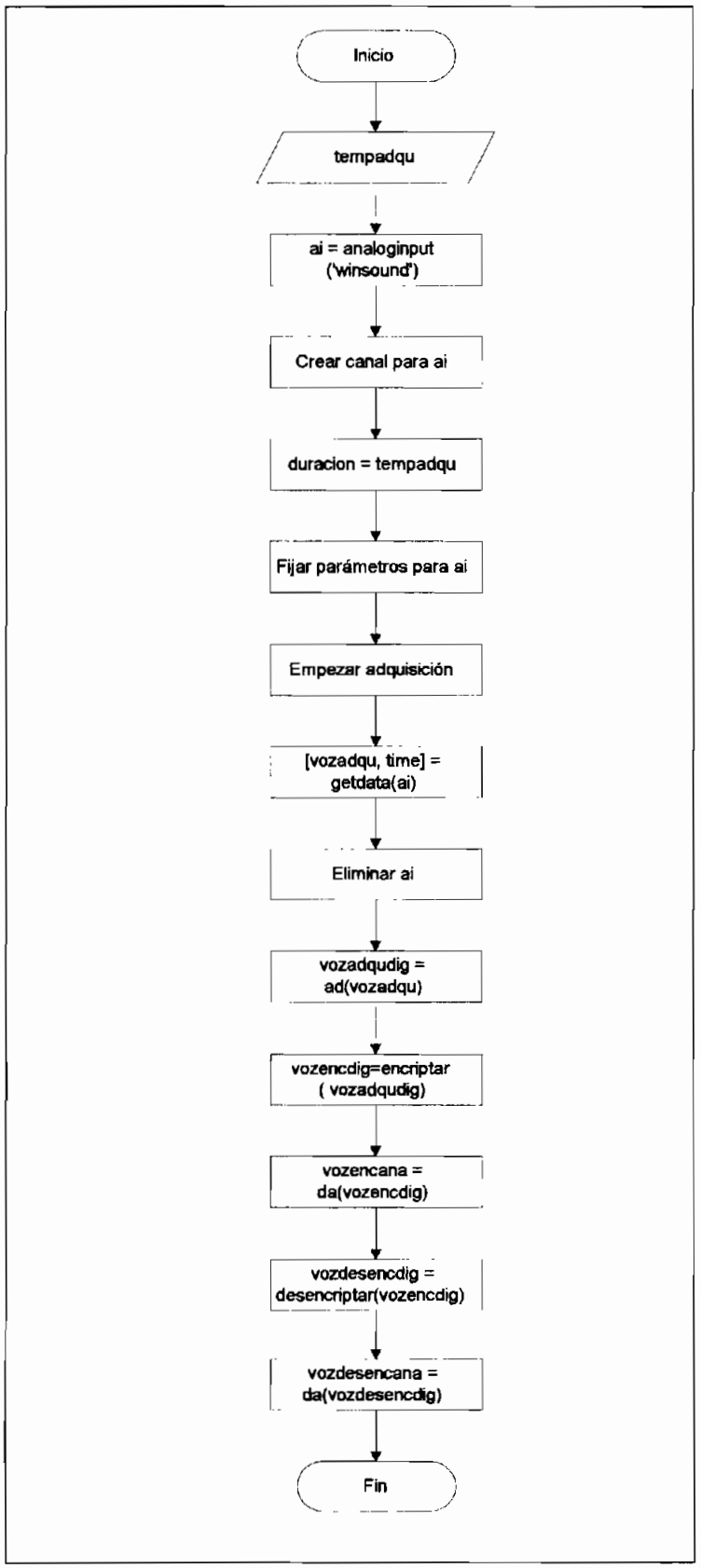


Figura 3. 16: Diagrama de flujo general para los archivos adqu*.m

La clave se calcula de forma externa, en el archivo adqudes.m; la longitud de la clave es estándar (64 bits), los bits que no se utilizan (8 bits) se desechan debido a que el sistema no simula un canal de comunicaciones real, es decir, no tiene que ejecutarse control de errores en la clave. Esta variable, al igual que las subclaves que se generan en cada ronda se declaran como variables globales; esto se hace debido a que tanto el algoritmo de encriptación como el de desencriptación necesitan la misma clave y las mismas subclaves (DES es un algoritmo de clave simétrica). Luego, se definen los vectores que realizarán todas las operaciones indicadas en el algoritmo. Al ser MATLAB un entorno especializado en vectores y matrices, la localización de cada elemento de un vector puede realizarse por medio de un vector externo, así por ejemplo, para la permutación inicial se define el vector p, y esta operación se cumple mediante la instrucción InPerm = PlText(:,p); esta propiedad resulta útil para expandir, comprimir y simular registros de desplazamiento en todos los vectores que se van formando en cada operación del algoritmo DES, y en los siguientes algoritmos.

Para la sustitución con cajas S, se crea el archivo dessbox.m. Al tratarse de una operación totalmente independiente del resto, se puede implementar como una función aparte; lo cual es recomendable debido al gran número de líneas de código que requiere esta función. Si se implementara en cada una de las rondas del algoritmo DES, haría que el mismo tenga un tiempo de ejecución mucho más grande. El diagrama de flujo de la figura 3.17 muestra el funcionamiento del archivo dessbox.m

El archivo desdec.m implementa el algoritmo de desencriptación DES, mediante las indicaciones presentadas en la sección correspondiente, es decir, utilizando las subclaves en orden inverso (esa es la razón para declararlas como variables globales en el programa descip.m), y haciendo que el desplazamiento de bits de la clave se realice de forma inversa al algoritmo de encriptación.

El anexo A provee el código fuente de los algoritmos de encriptación y desencriptación, así como la implementación de la sustitución con cajas S. La clave para este algoritmo se genera en el archivo adqudes.m

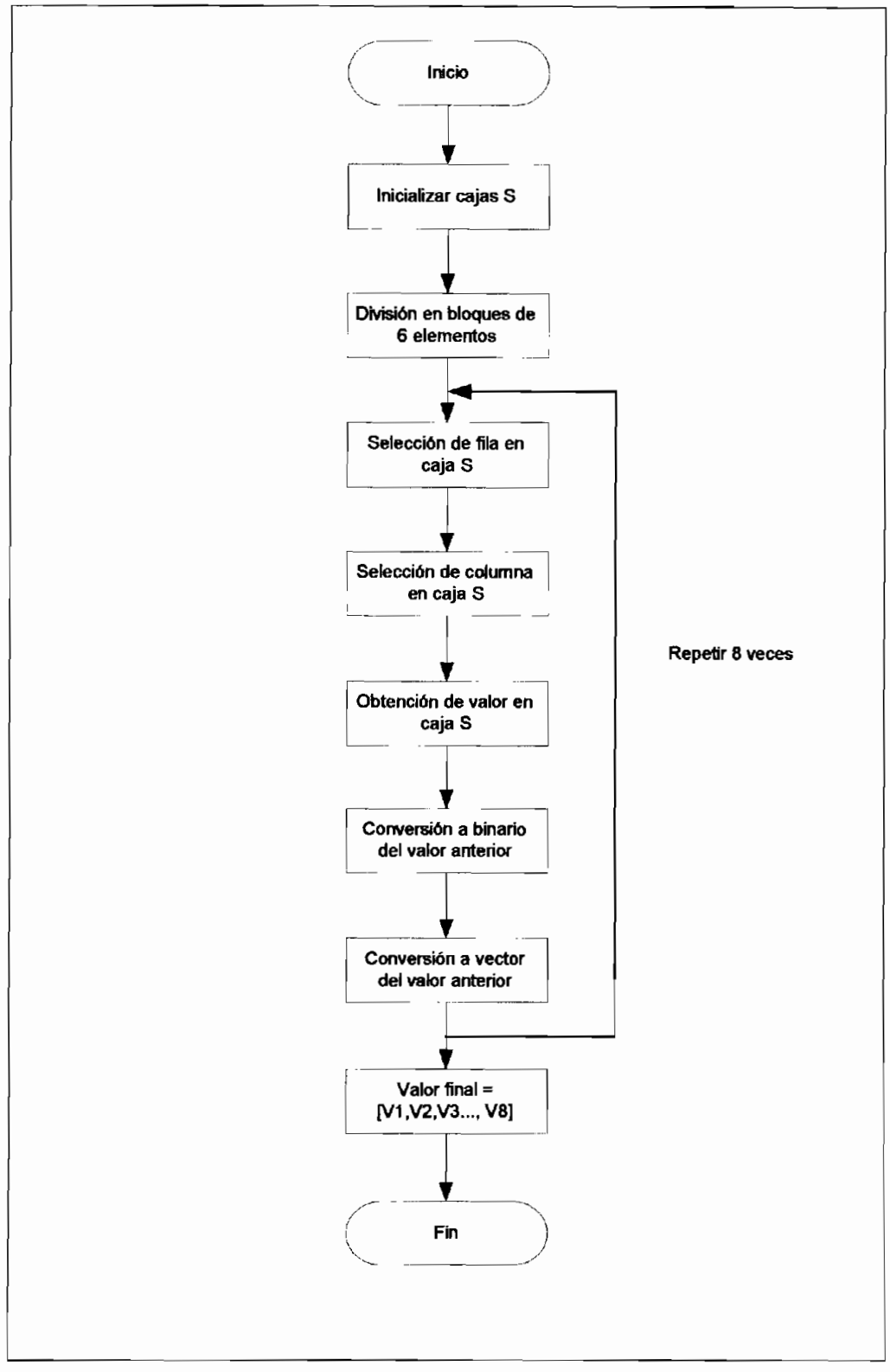


Figura 3. 17: Diagrama de flujo del programa dessbox.m

3.3.2.2. Implementación del algoritmo IDEA

De acuerdo con [1], el algoritmo IDEA no pierde su robustez si se implementan por lo menos cuatro de las ocho rondas. Tomando en cuenta este comentario, se ha implementado un modelo de algoritmo IDEA con cuatro rondas, realizando todas las operaciones indicadas en la definición original del algoritmo. El diagrama de flujo, así como la operación del algoritmo se indicaron en una sección precedente. El programa ideacip.m realiza la encriptación, tomando como argumento de entrada un vector binario de longitud 64, el cual simula un bloque de 64 bits. La clave de 128 bits es generada de forma externa al algoritmo y declarada como variable global, al igual que todas las subclaves generadas para cada ronda. Al igual que en el algoritmo DES, se define un vector que permita desplazar los bits de la clave en la forma indicada anteriormente. Sobre la base de estas claves desplazadas, se generan las subclaves para todas las rondas implementadas y la operación final. Después, se divide el bloque de texto llano en cuatro partes, y se lo convierte en un número decimal mediante las funciones auxiliares strg y bin2dec; este paso se realiza para llevar a cabo todas las operaciones matemáticas requeridas. Después de los cuatro primeros pasos, es necesario regresar los resultados decimales a vectores binarios, para lo cual se utilizan las funciones dec2bin y numer16. Una vez ejecutados los pasos 5) y 6) que se mencionan en la sección 3.2.2.2, se realiza la operación de vuelta, es decir convertir de nuevo los vectores binarios a valores decimales. Este procedimiento se realiza durante las cuatro rondas, y la operación final.

La razón principal para la reducción de ocho a cuatro rondas en la implementación de este algoritmo se debe a las pruebas preliminares de tiempo de ejecución ya que éste era excesivo (aproximadamente un minuto para encriptar un vector de entrada de 64 dígitos binarios). La implementación de cuatro rondas redujo el tiempo de ejecución drásticamente como se muestra en el siguiente capítulo. El algoritmo comprende ocho rondas en su estándar debido a que este número de rondas es lo que hace fuerte al algoritmo, evitando ataques criptoanalíticos no deseados. Aunque se indica que se ha demostrado

matemáticamente que el algoritmo no pierde su robustez si se implementan solo cuatro de las ocho rondas, no se puede asegurar eso en la práctica.

El algoritmo de desencriptación se implementa en el archivo ideadec.m. La característica principal de este algoritmo es el cálculo previo de los inversos aditivo y multiplicativo de las respectivas subclaves utilizadas en el algoritmo de encriptación. La tarea, en principio, parece muy fácil, sin embargo, no lo es. Para calcular el inverso aditivo, es necesario recurrir a la aritmética modular para demostrar que el inverso aditivo módulo *n* de un número *a* es –a mod n. En el caso del inverso multiplicativo, es necesario recurrir al algoritmo euclidiano extendido para calcular el máximo común divisor de dos números. El objetivo de este algoritmo es, dados dos números *x* e *y*, calcular el máximo común divisor de los mismos, *d*, y encontrar dos números, *a* y *b* tales que satisfagan la ecuación $a \cdot x + b \cdot y = d$. Para el caso del algoritmo IDEA, el número $2^{16} + 1$ (65537) es primo, lo cual es una condición esencial para que el algoritmo euclidiano extendido arroje como resultado el inverso multiplicativo módulo *n* necesario. La ecuación anterior se reduce a $a \cdot x + b \cdot y = 1$, y si se pone como argumentos de entrada $x = 2^{16} + 1$; *y* = subclave indicada en la tabla 3.9, se calculan los números *a* y *b* mediante la función interna de MATLAB gcd.m. El número *b* constituye el inverso multiplicativo módulo 65537 calculado para la operación de desencriptación. Existe una condición adicional para el cálculo: si *b* es negativo, se le suma 65537 y el nuevo número se constituye en el resultado deseado. A partir de ahí, el algoritmo es el mismo que el de encriptación.

El anexo A presenta el código fuente de los archivos ideacip.m e ideadec.m. La clave para este algoritmo se genera en el archivo adquidea.m

3.3.2.3. Implementación del algoritmo A5/1

El algoritmo A5/1 se implementa a través de los archivos a5cip.m y a5dec.m. A diferencia de los dos algoritmos anteriores, el algoritmo A5/1 trata con un elemento vectorial a la vez, es decir, es independiente de la longitud del texto llano, lo cual representa la propiedad principal de los cifrados de flujo. La clave es

un vector binario externo a los dos programas, el cual es declarado global, de longitud 64. Esta clave se distribuye entre los tres registros de desplazamiento con realimentación lineal, de la siguiente forma: en el R1, se ponen los primeros 19 elementos; en el R2, se ponen los siguientes 22 elementos; y en el R3, se ponen los últimos 23 elementos. La diferencia principal con el algoritmo A5/1 original es que no se implementa la regla de mayoría que se impone sobre los tres registros; a cambio, los tres registros son utilizados en cada operación del algoritmo.

El diagrama de flujo de la figura 3.18 detalla la operación del algoritmo A5/1

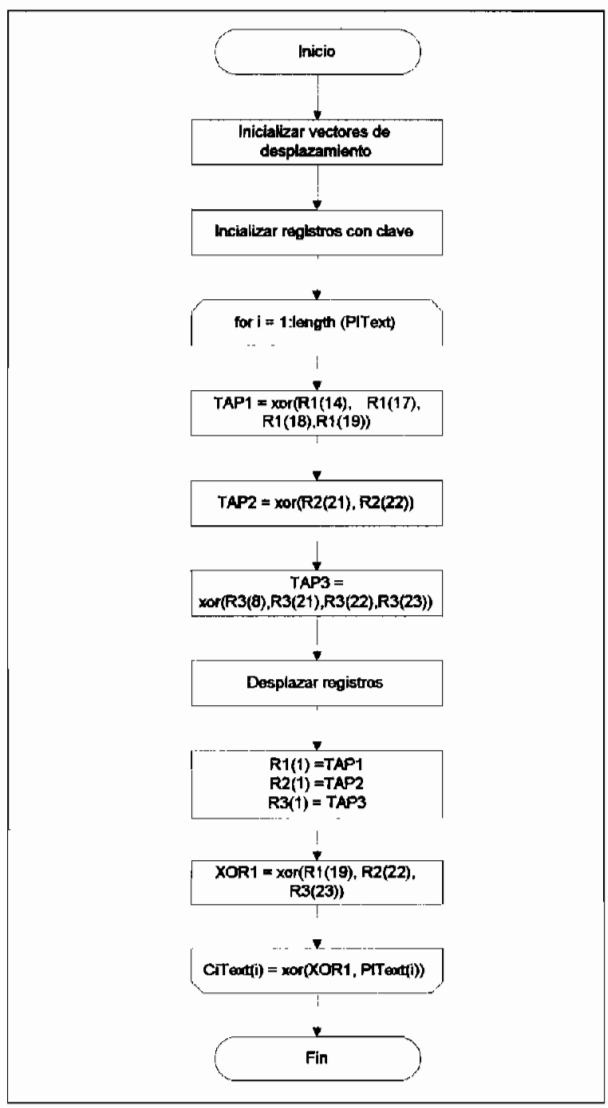


Figura 3. 18: Diagrama de flujo para el algoritmo A5/1

El anexo A muestra el código fuente de los archivos a5cip.m y a5dec.m La clave de este algoritmo se genera en el archivo adqua5.m

3.3.2.4. Implementación del algoritmo RC4

El algoritmo RC4 se implementa mediante los archivos rc4cip.m y rc4dec.m. En cierta forma, este algoritmo se asemeja a los algoritmos de bloque estudiados anteriormente, ya que utiliza como entrada un vector binario de ocho elementos; sin embargo, la operación de los algoritmos de flujo puede ser un bit o byte a la vez, así que se considera a este vector de entrada como un byte. La clave se genera externamente, se declara global, y se calcula con valores decimales, comprendidos entre 0 y 255. La longitud de la clave para este sistema simulador es de 256 bytes. La razón para operar con valores decimales se debe a que, como se sabe, un byte de información puede representar números decimales comprendidos entre 0 y 255; además, todas las operaciones aritméticas del algoritmo requieren números decimales. El algoritmo se implementa en su totalidad según la descripción que se realiza en una sección anterior. Para realizar el intercambio entre los elementos *i* y *j* de la caja S de este algoritmo, se emplea un vector auxiliar, Sax, lleno de unos, y que no interviene en ninguna otra operación. Una diferencia, aunque no esencial con el algoritmo original es que se calcula primero la corriente de clave total antes de la operación final de este algoritmo. Esto sólo se puede realizar con una longitud de corriente de datos conocida (para el caso de este proyecto, la longitud de los datos siempre es conocida). En caso de ser implementado en dispositivos reales, es necesario que la operación final se haga byte por byte.

El diagrama de flujo de la figura 3.19 muestra la operación del algoritmo RC4.

El anexo A muestra el código fuente de los archivos rc4cip.m y rc4dec.m. La clave para este algoritmo se genera en el archivo adqurc4.m.

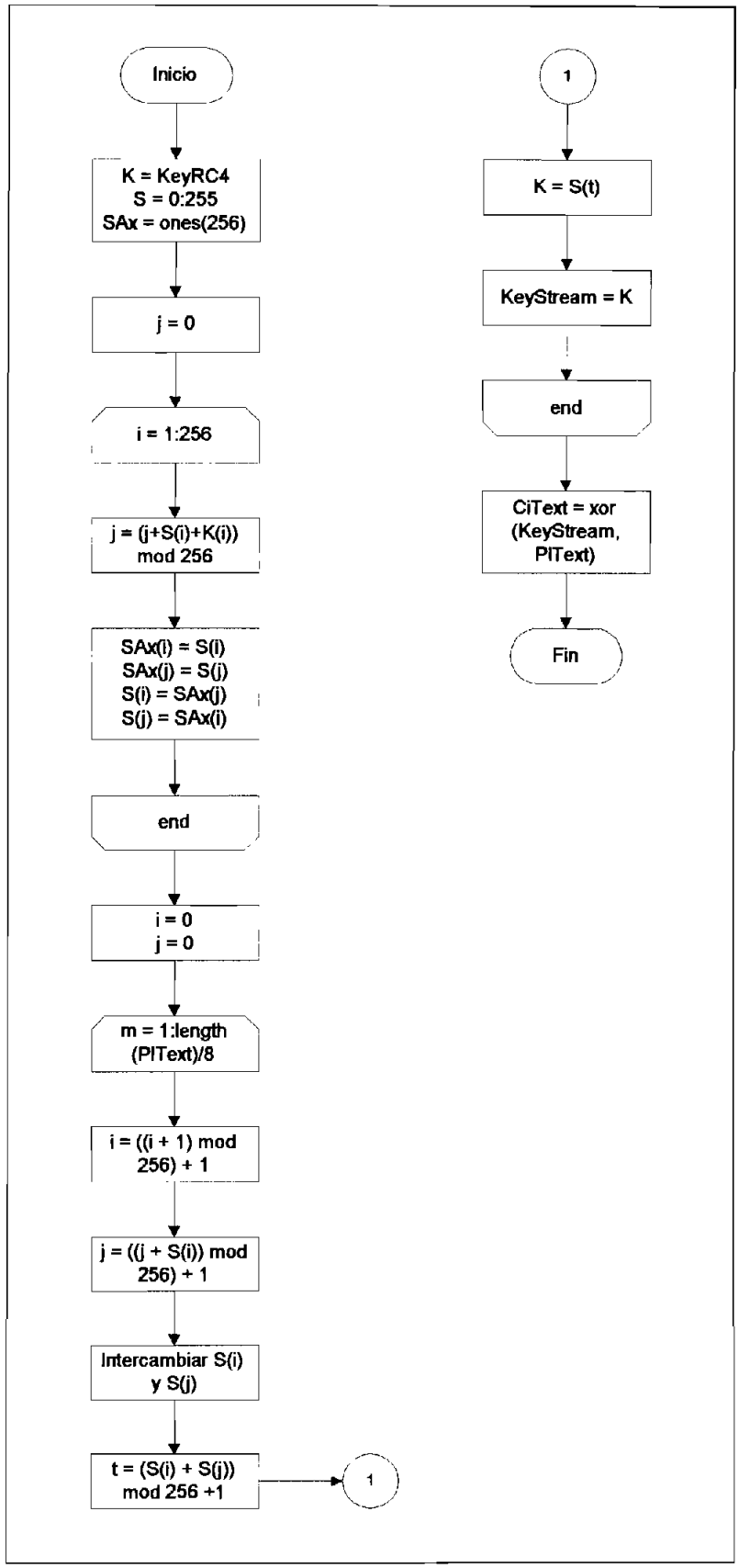


Figura 3. 19: Diagrama de flujo del algoritmo RC4

3.3.3. DESCRIPCIÓN DE LOS ARCHIVOS AUXILIARES PARA LA OPERACIÓN DEL SISTEMA SIMULADOR

3.3.3.1. Conversión analógica/digital

Para la conversión A/D se crea el archivo ad.m, el cual toma un vector de señal analógica, codificada a 8 bits, y lo convierte en un vector de valores binarios con resolución de 8 bits. Esta conversión es necesaria, debido a que los algoritmos de encriptación únicamente trabajan con datos digitales. La operación de este programa se puede ver mediante el diagrama de flujo de la figura 3.20

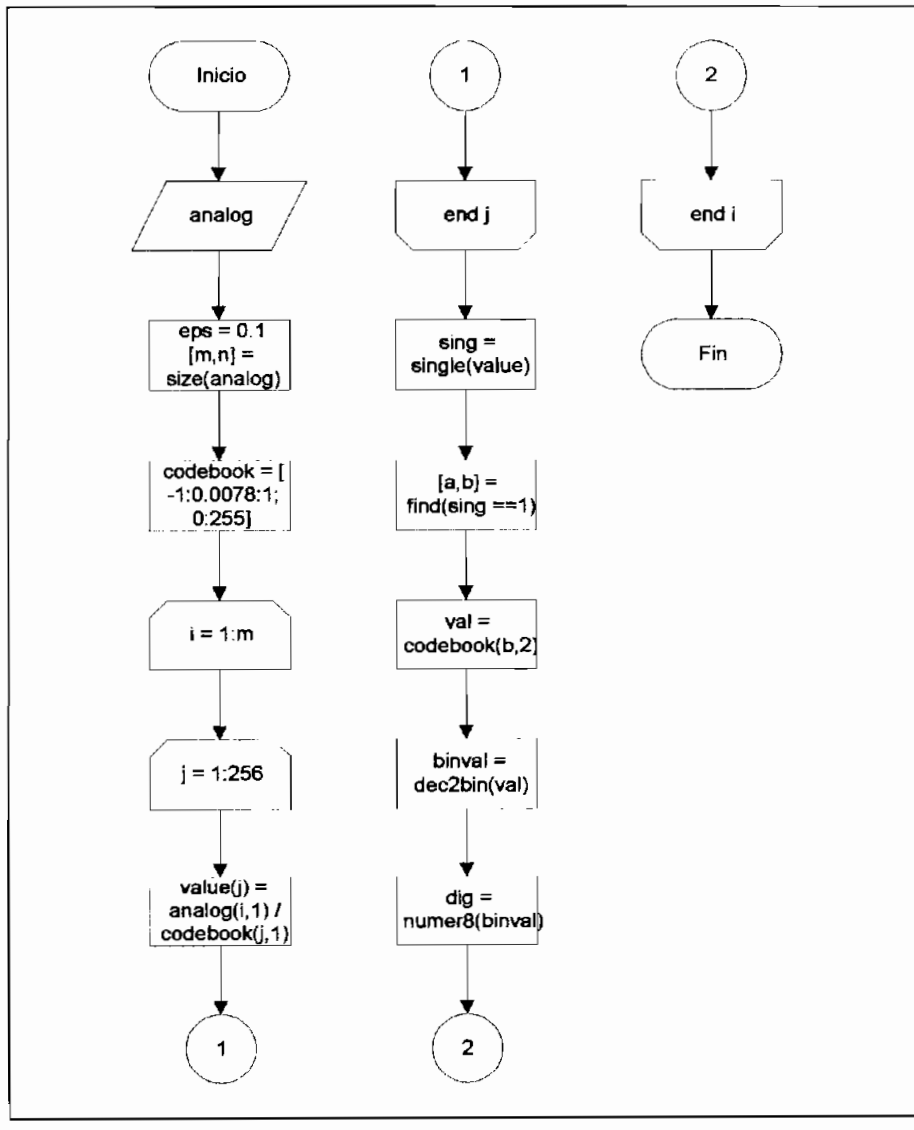


Figura 3. 20: Diagrama de flujo de ad.m

El método utilizado para la conversión analógica/digital se parece al método de aproximaciones sucesivas, aunque, para MATLAB es muy difícil implementar el método de aproximaciones sucesivas en su forma original.

3.3.3.2. Conversión digital/analógica

Para comprobar el funcionamiento perfecto del sistema simulador, es necesario que la señal de salida sea exactamente igual a la señal de entrada, lo cual obliga a convertir a la señal de salida de su formato digital a un vector de señal analógico. El archivo da.m permite convertir un vector de valores binarios a un vector de valores continuos, con resolución de ocho bits. EL diagrama de flujo de la figura 3.21 permite ver el funcionamiento de este programa.

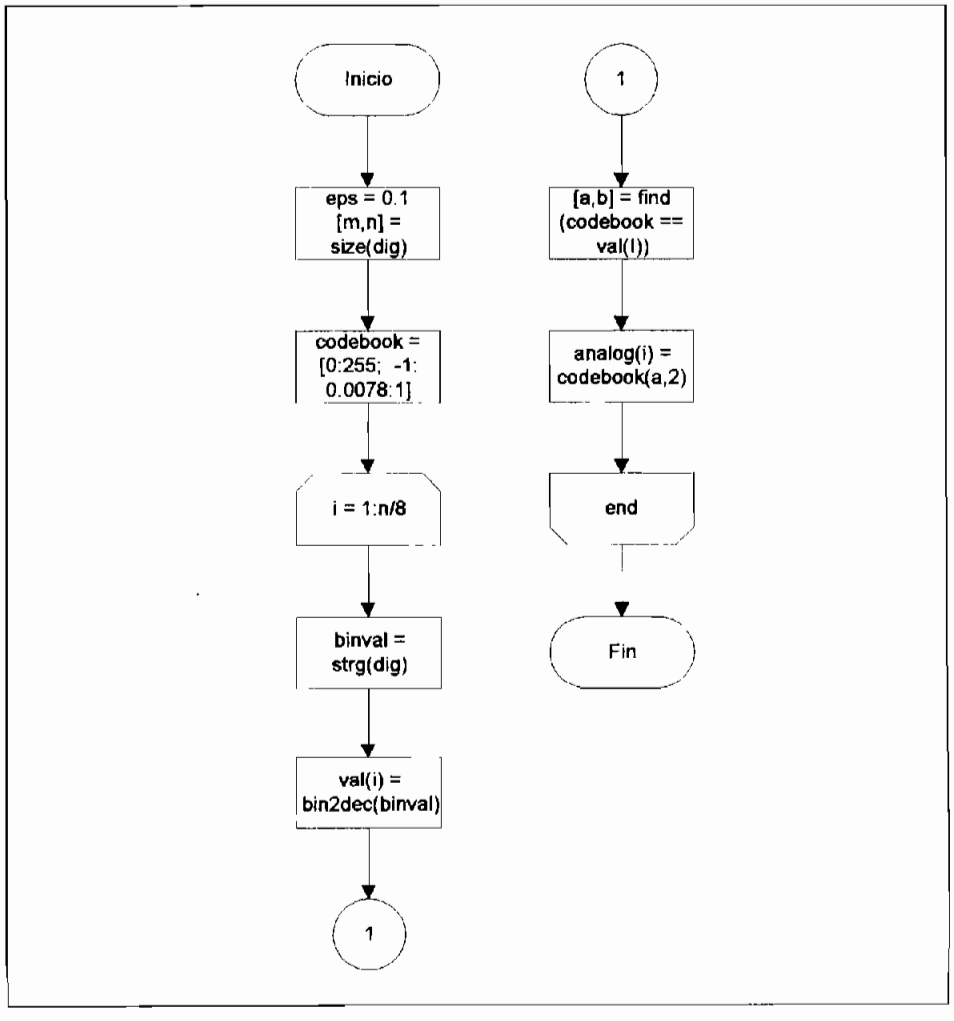


Figura 3. 21: Diagrama de flujo del programa da.m

3.3.3.3. Conversión de vectores a cadenas de caracteres

En varios algoritmos es necesario operar con valores decimales, pero el argumento de entrada a todos los algoritmos es un vector de valores binarios. Para poder convertir estos vectores a cadenas de caracteres, para su posterior conversión a valores decimales, se crea el archivo strg.m. El diagrama de flujo de la figura 3.22 muestra la forma de operación de este archivo.

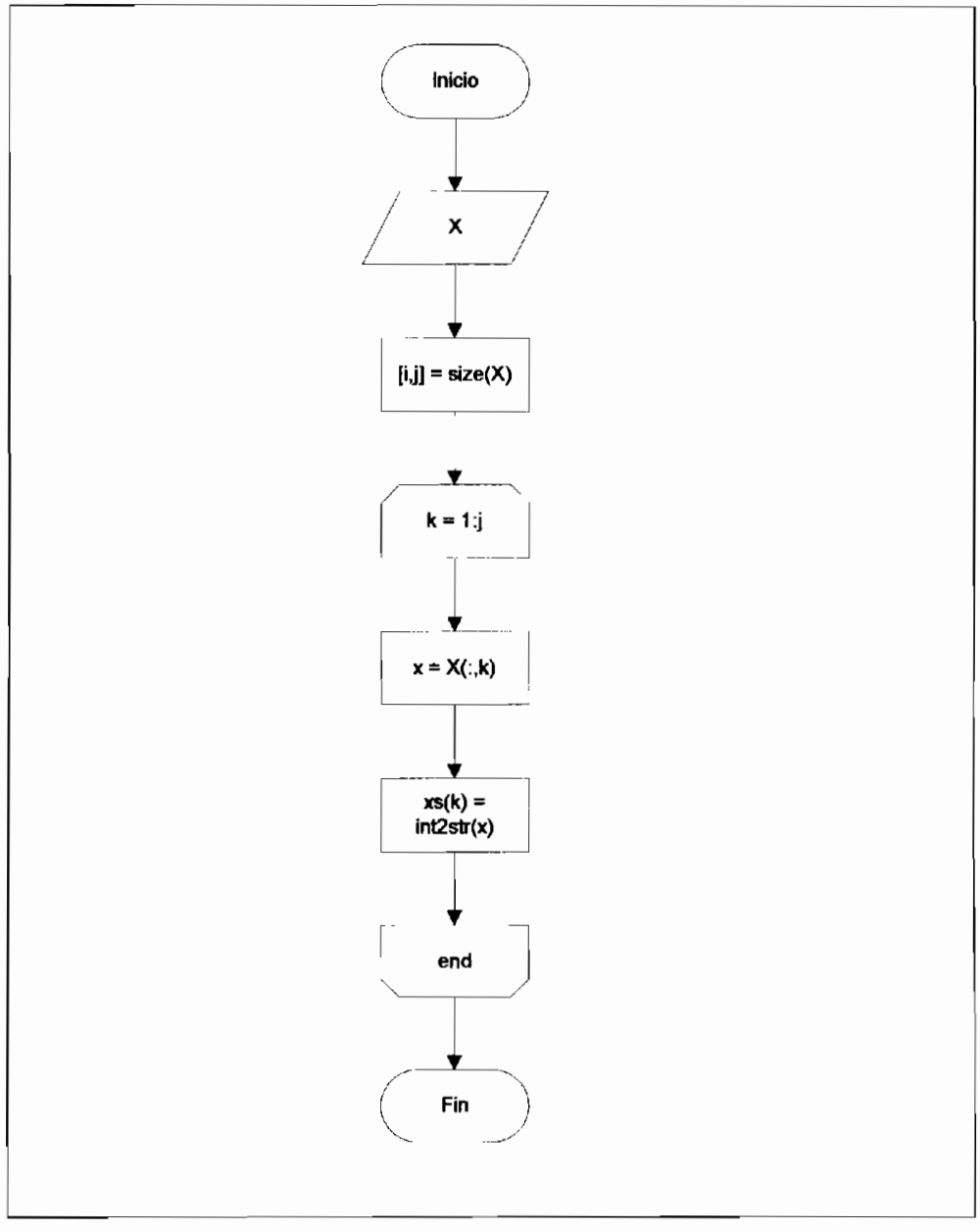


Figura 3. 22: Diagrama de flujo del archivo strg.m

3.3.3.4. Conversión de cadenas de caracteres a vectores

Existen muchas operaciones dentro de los algoritmos que necesitan vectores como operandos, es por eso que se crean los archivos numer4.m, numer8.m y numer16.m. Cada uno opera con un número de "bits" específico, en el caso de numer4.m, por ejemplo, opera con cadenas de caracteres binarias de longitud entre 1 y 4 elementos. Es necesario anteponer ceros en la mayoría de los casos, por cuanto la conversión decimal a binario no toma en cuenta los ceros que se producen a la izquierda, es decir, por ejemplo, si se convierte el valor decimal 2 a binario, siempre se convertirá como '10', no como '0010'. La filosofía de los otros dos programas es idéntica, así que sólo se presenta el diagrama de flujo del programa numer4.m en la figura 3.23

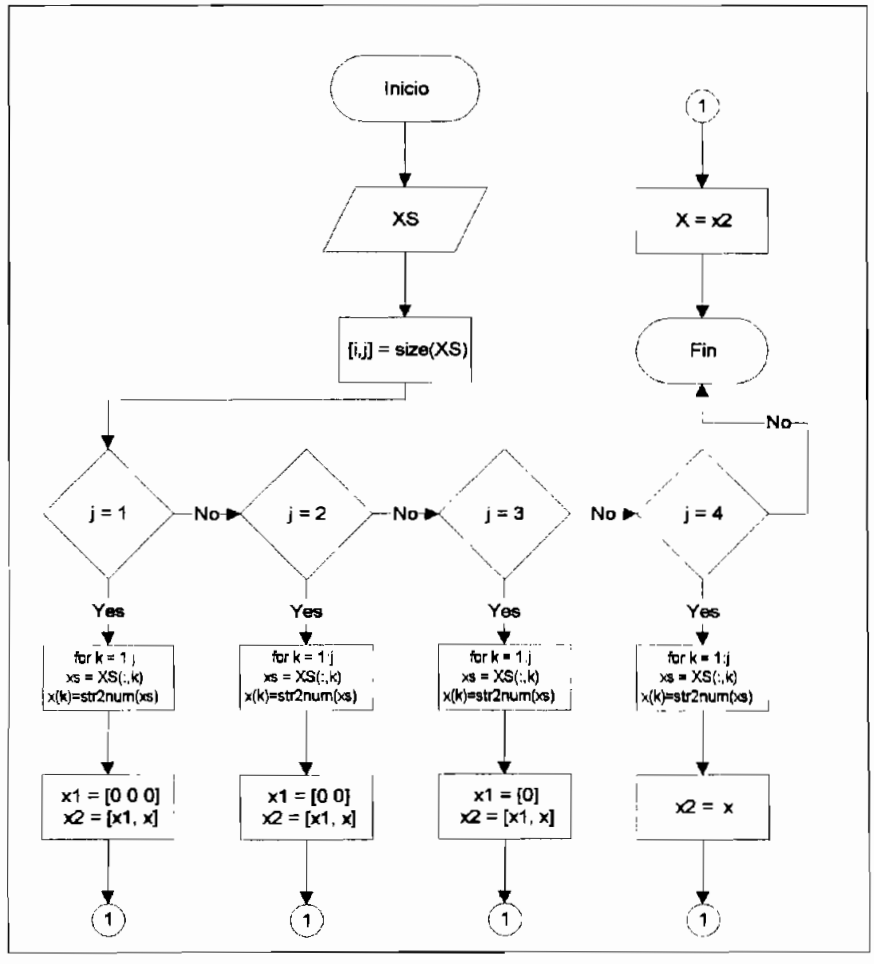


Figura 3. 23: Diagrama de flujo del programa numer4.m

CAPÍTULO IV

PRUEBAS REALIZADAS

4.1. INTRODUCCIÓN

El capítulo anterior presentó las características de cada uno de los algoritmos implementados en el sistema simulador. Para comprobar la potencialidad de cada uno, así como otras características tales como velocidad, ancho de banda utilizado, etc, se implementa un banco de pruebas comparativo que se utiliza tanto para los algoritmos, como para los programas auxiliares del sistema simulador.

4.2. PRUEBAS REALIZADAS AL SISTEMA SIMULADOR

4.2.1. CONDICIONES DE REALIZACIÓN DE PRUEBAS

Para todas las pruebas, las condiciones de operación fueron las siguientes:

- *Hardware:* Computador personal con las siguientes prestaciones:
  1. Procesador Intel Celeron a 333 MHz
  2. 124 MB de RAM
  3. 2 discos duros: C: de 20 GB y D: de 3.2 GB
  4. Tarjeta de video AGP SiS 620 con 4 MB de memoria de video
  5. Tarjeta de sonido C-Media 8738 compatible con SoundBlaster
  6. Micrófono y parlantes SoundBlaster
- *Software:* Entorno de programación MATLAB® ejecutándose bajo Windows Millenium Edition, junto con los siguientes programas en segundo plano:
  1. PC-Cillin Real-Time Scan (antivirus)
  2. Norton CrashGuard
  3. C-Media Mixer (control de volumen para la tarjeta de sonido)
  4. MSN Messenger (software de mensajería)

5. Barra de acceso directo de Microsoft Office 2000

Todas las mediciones se realizan mediante la función interna de MATLAB, *profile*, la cual mide el tiempo de ejecución de cada función invocada y genera un reporte en formato de página Web, el cual puede ser leído por cualquier navegador. Las páginas Web correspondientes a las mediciones efectuadas pueden ser consultadas en el anexo C.

4.2.2. PRUEBAS DE TIEMPO DE EJECUCIÓN

4.2.2.1. Tiempo de ejecución de la adquisición de audio

Para realizar la prueba de tiempo de ejecución de la adquisición de audio, se adquiere un segundo de audio a través del micrófono conectado a la tarjeta de sonido. La adquisición de audio se realiza mediante los programas adqu*.m, los cuales contienen las instrucciones necesarias para realizar esta operación. Los resultados tabulados son los siguientes:

Archivo de adquisición	Tiempo de ejecución [s]	Precisión del reloj [s]
adqudes.m	5.85	0.050
adquidea.m	5.55	0.060
adqua5.m	3.5	0.050
adqurc4.m	2.63	0.050

Tabla 4. 1: Tabulación del tiempo de ejecución de la adquisición de audio

Los resultados obtenidos no dependen en lo absoluto de los algoritmos de encriptación ni de cualquier otro programa que se ejecute dentro de los archivos adqu*.m.

4.2.2.2. Tiempo de ejecución de la conversión analógica/digital

Para esta prueba, se toma la misma señal adquirida por cada uno de los archivos adqu*.m, y se procede a aplicarle el archivo ad.m. Los resultados tabulados son los siguientes:

4.2.2.5. Tiempo de ejecución de la conversión digital/analógica

Para esta prueba se toma la señal encriptada y la señal desencriptada, puesto que ambas se presentan en el sistema simulador. Los resultados tabulados son los siguientes:

Archivo de adquisición	Tiempo de ejecución: señal encriptada [s]	Tiempo de ejecución: señal desencriptada [s]
adqudes.m	669.87	669.87
adquidea.m	733.01	733.01
adqua5.m	702.06	702.06
adqurc4.m	229.04	229.04

Tabla 4. 5: Tiempo de ejecución de la conversión digital/analógica

4.2.2.6. Tiempo de ejecución general

Para tabular el tiempo de ejecución general, se toman en cuenta todas las operaciones realizadas, tanto externas como internas de cada algoritmo. Los resultados tabulados son los siguientes:

Archivo de adquisición	Tiempo de ejecución [s]
adqudes.m	17091.55
adquidea.m	11789.70
adqua5.m	5891.58
adqurc4.m	2374.10

Tabla 4. 6: Tiempo de ejecución de los archivos de adquisición de audio

4.2.3. PRUEBAS DE DESEMPEÑO

4.2.3.1. Formas de onda obtenidas en el sistema simulador

En esta sección, se analizan las diversas formas de onda obtenidas al ejecutar el sistema simulador. Las condiciones de realización de estas pruebas, son idénticas a las anteriores, con la adición del paquete Winamp que permite reproducir todo tipo de archivos de audio (.wav, .mp3, .wma). Se escogió este

paquete debido a su facilidad de uso, y su excelente manejo del ambiente de audio en una PC.

4.2.3.1.1. Formas de onda obtenidas en el algoritmo DES

Para esta prueba, se procedió a adquirir una porción de cinco segundos de la canción "*Porqué No Ves Si Llueve en Tel Aviv*", del grupo ecuatoriano Contravía, a través del micrófono conectado a la tarjeta de sonido. No se produjeron problemas de realimentación, debido a que la salida del micrófono estaba apagada en C-Media Mixer. La forma de onda de la señal de entrada es la siguiente:

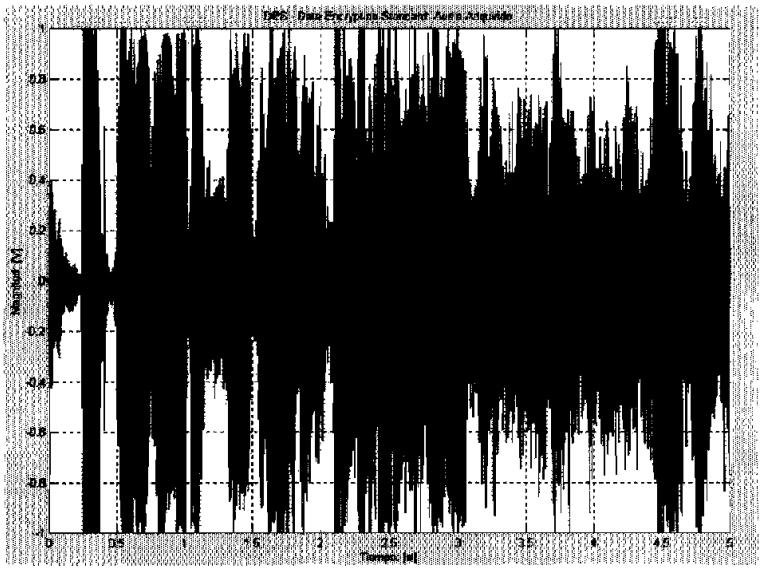


Figura 4. 1: Forma de onda de la señal adquirida

Para obtener la forma de onda de la señal encriptada, se procede a realizar una conversión D/A, con el objeto de poder analizarla de una forma más adecuada. La forma de onda de la señal encriptada es la siguiente:

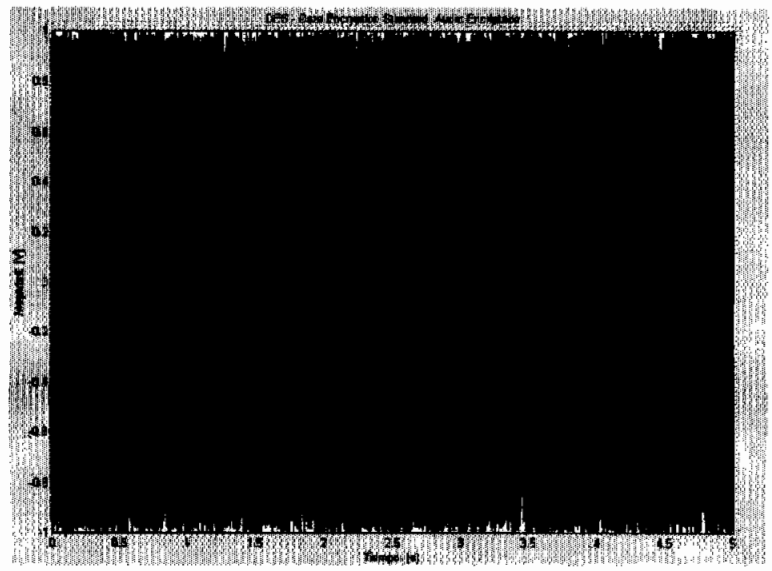


Figura 4. 2: Forma de onda de la señal encriptada mediante el algoritmo DES

Por último, se presenta la forma de onda de la señal desencriptada. Por definición, ésta debería ser idéntica punto a punto a la señal adquirida. En una prueba posterior, se comprobará esta propiedad. La forma de onda es la siguiente:

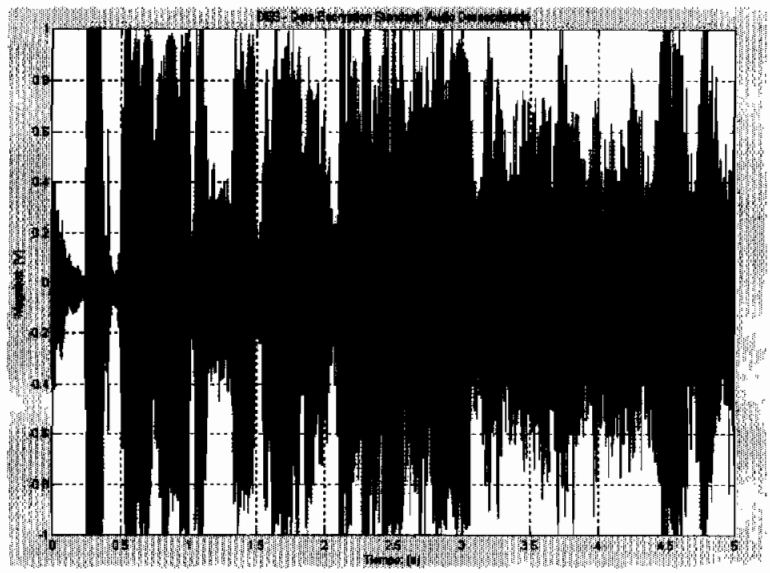


Figura 4. 3: Forma de onda de la señal desencriptada mediante el algoritmo DES

4.2.3.1.2. Formas de onda obtenidas en el algoritmo IDEA

Para esta prueba, se procedió a adquirir una porción de cinco segundos de la canción "*Devil Inside*" del grupo australiano INXS. Las condiciones de adquisición son las mismas que en al punto anterior. La forma de onda de la señal adquirida es la siguiente:

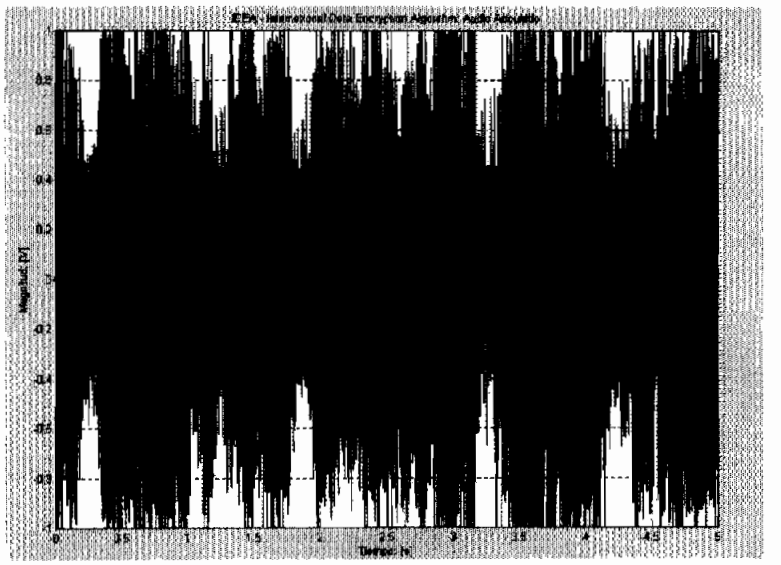


Figura 4. 4: Forma de onda de la señal adquirida

El procedimiento para obtener la señal encriptada es el mismo que en el punto anterior. La señal encriptada es la siguiente:

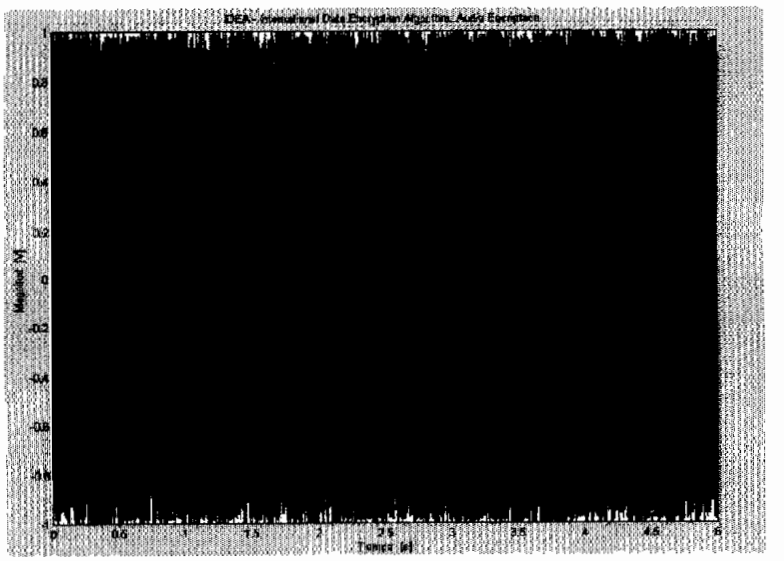


Figura 4. 5: Forma de onda de la señal encriptada mediante el algoritmo IDEA

La señal desencriptada es la siguiente:

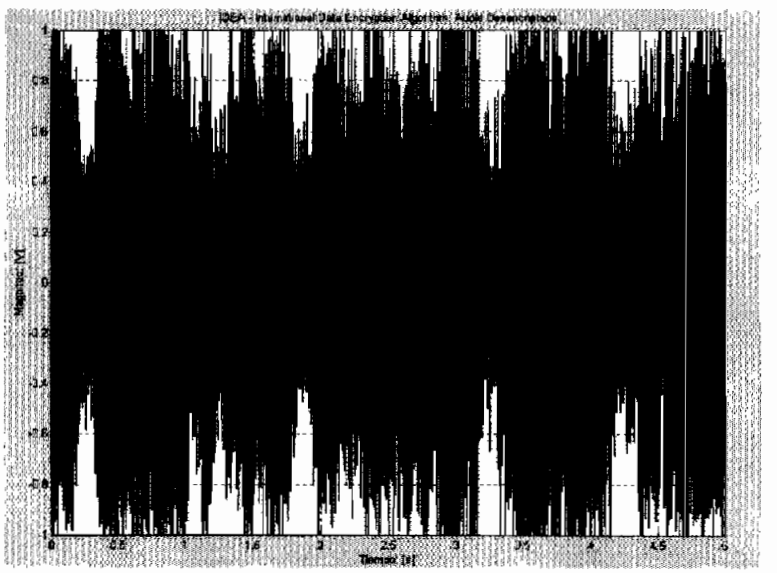


Figura 4. 6: Forma de onda de la señal desencriptada mediante el algoritmo IDEA

4.2.3.1.3. Formas de onda obtenidas en el algoritmo A5/1

Para esta prueba, se procedió a adquirir una porción de cinco segundos de la canción "*Everything I Do, I Do It For You*" del cantante canadiense Bryan Adams. La forma de onda de la señal adquirida es la siguiente:

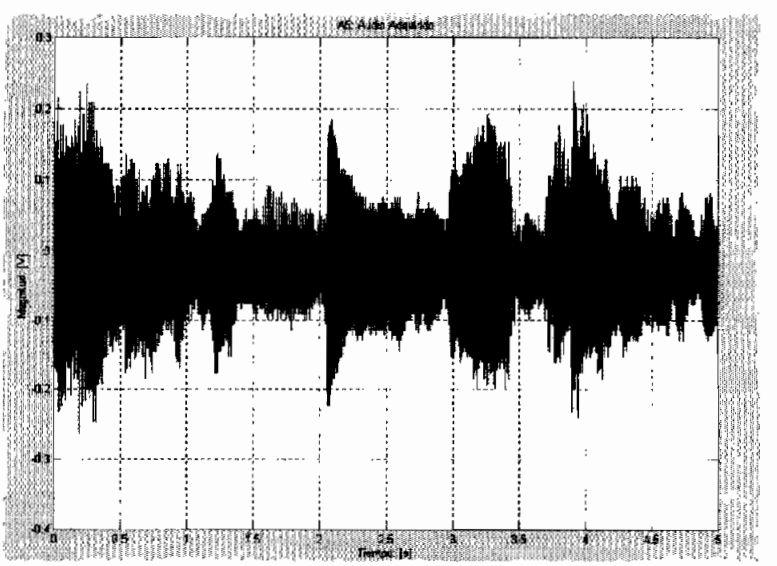


Figura 4. 7: Forma de onda de la señal adquirida

La forma de onda de la señal encriptada es la siguiente:

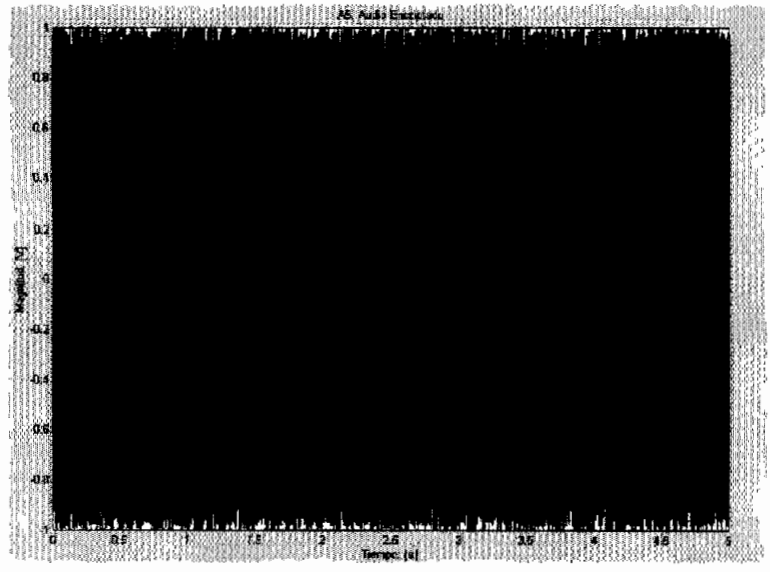


Figura 4. 8: Forma de onda de la señal encriptada mediante el algoritmo A5/1

La forma de onda de la señal desencriptada es la siguiente:

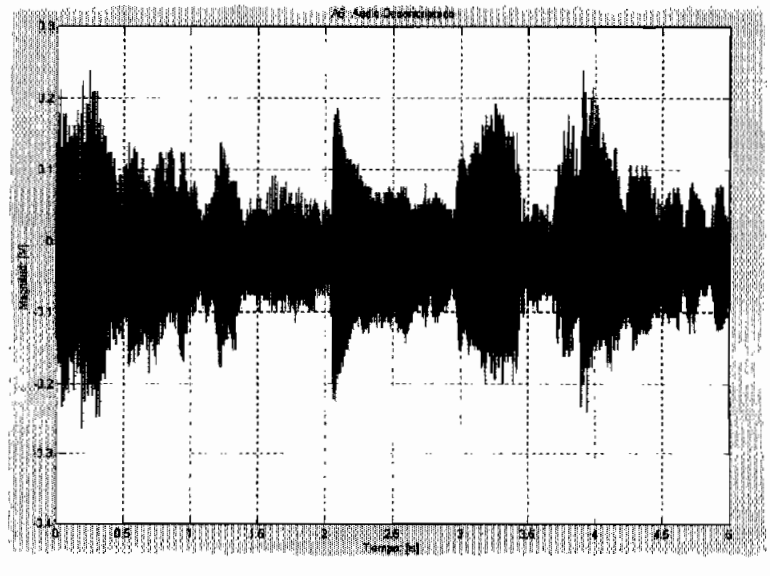


Figura 4. 9: Forma de onda de la señal desencriptada mediante el algoritmo A5/1

4.2.3.1.4. Formas de onda obtenidas en el algoritmo RC4

Para esta prueba, se procedió a adquirir una porción de cinco segundos de la canción "*Como Un Lobo*" del cantante español Miguel Bosé. La forma de onda de la señal adquirida es la siguiente:

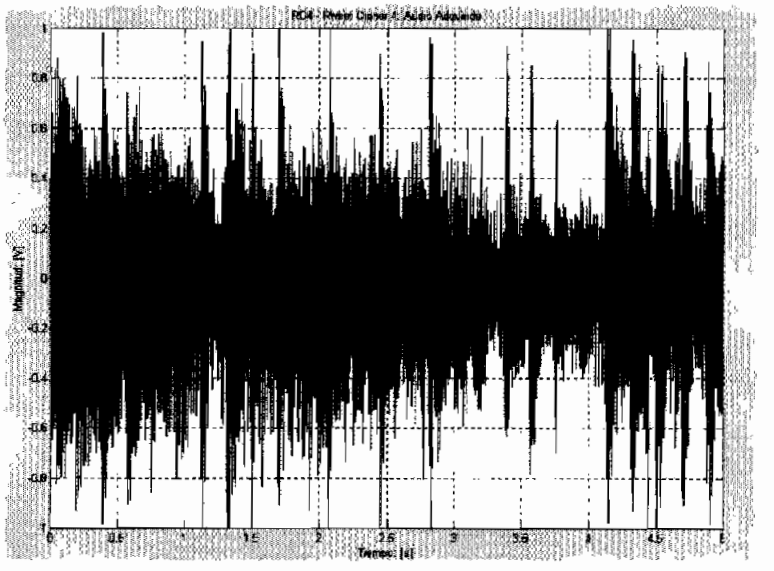


Figura 4. 10: Forma de onda de la señal adquirida

La forma de onda de la señal encriptada es la siguiente:

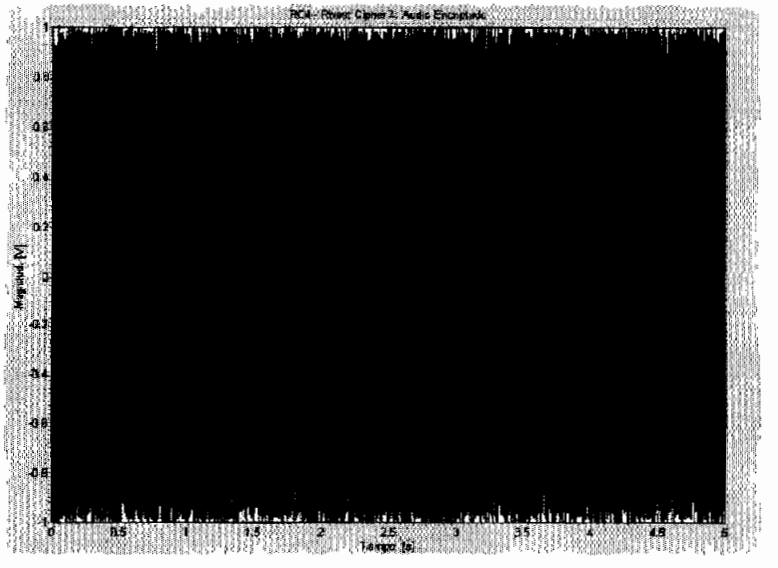


Figura 4. 11: Forma de onda de la señal encriptada mediante el algoritmo RC4

La forma de onda de la señal desencriptada es la siguiente:

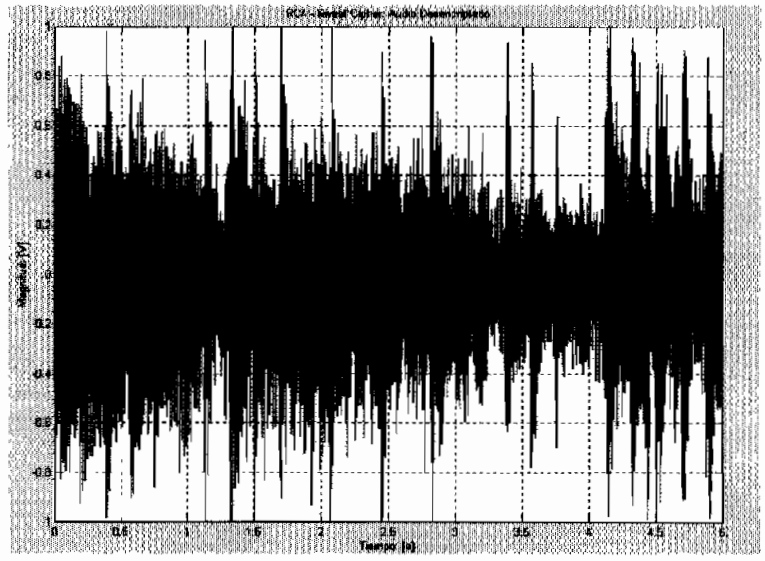


Figura 4. 12: Forma de onda de la señal desencriptada mediante el algoritmo RC4

4.2.3.2. Espectros de frecuencia obtenidos en el sistema simulador

Para calcular el espectro de frecuencia de cada una de las señales obtenidas, se aplica la transformada rápida de Fourier en su forma estándar (sin ventanas de ningún tipo) a cada una de las señales presentadas anteriormente, y se grafica los puntos obtenidos. Se clasifican los espectros de frecuencia de acuerdo al algoritmo aplicado.

4.2.3.2.1. Espectros de frecuencia obtenidos en el algoritmo DES

Los espectros de frecuencia obtenidos para las señales adquirida, encriptada y desencriptada se presentan a continuación. Para la señal adquirida, el contenido espectral más alto se encuentra entre los 200 y los 800 Hz, lo cual corresponde principalmente al espectro de la voz del vocalista. El resto del espectro corresponde al acompañamiento musical.

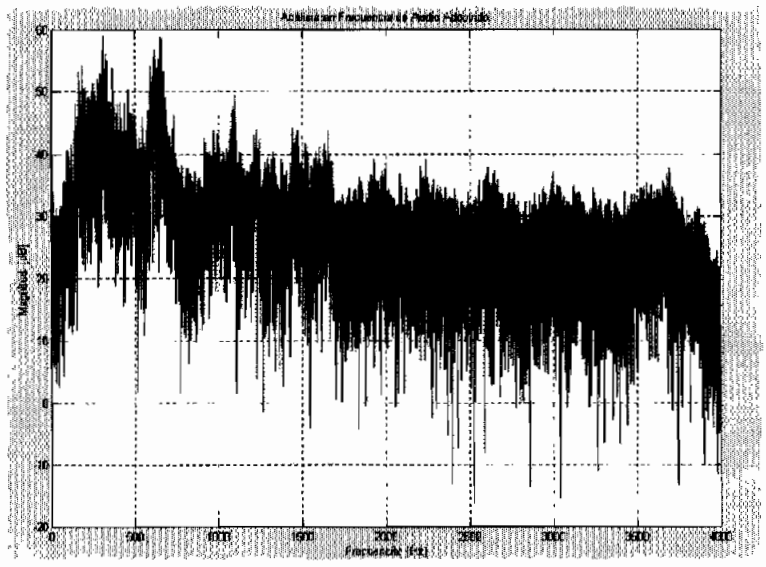


Figura 4. 13: Espectro de frecuencia de la señal adquirida

El siguiente espectro corresponde al de la señal encriptada. Como se puede observar en el gráfico, el espectro es prácticamente continuo en todo el ancho de banda, sin tener ninguna relación de traslación de amplitud con respecto al espectro de la señal adquirida. Este tipo de espectro corresponde a una señal en el dominio del tiempo totalmente ininteligible, lo cual corresponde muy bien al objetivo del algoritmo de encriptación. Si un atacante intentase un ataque, debería trabajar exclusivamente en la señal digital, sin tratar de convertirla.

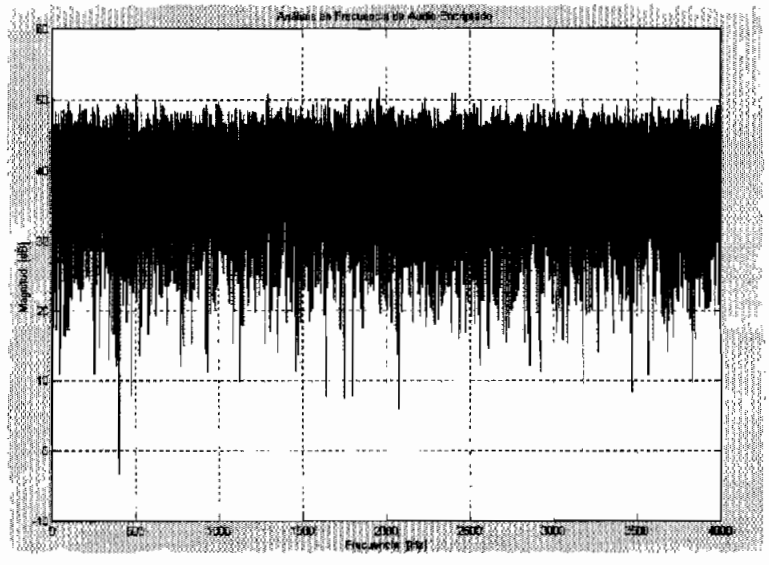


Figura 4. 14: Espectro de frecuencia de la señal encriptada mediante el algoritmo DES

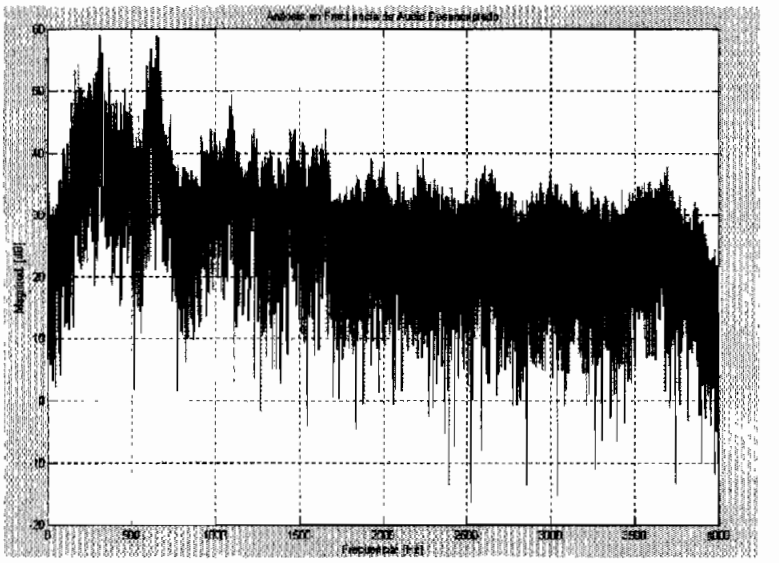


Figura 4. 15: Espectro de frecuencia de la señal desencriptada mediante el algoritmo DES

Como conclusión, el espectro de la señal desencriptada corresponde punto por punto al espectro de la señal adquirida, lo cual demuestra la efectividad del algoritmo de desencriptación.

4.2.3.2.2. Espectros de frecuencia obtenidos en el algoritmo IDEA

Los espectros de frecuencia obtenidos para las señales adquirida, encriptada y desencriptada son los siguientes:

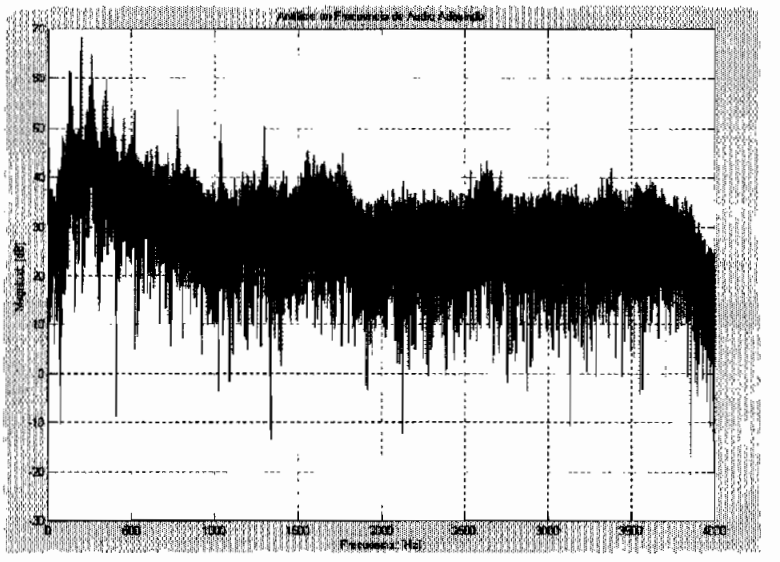


Figura 4. 16: Espectro de frecuencias de la señal adquirida

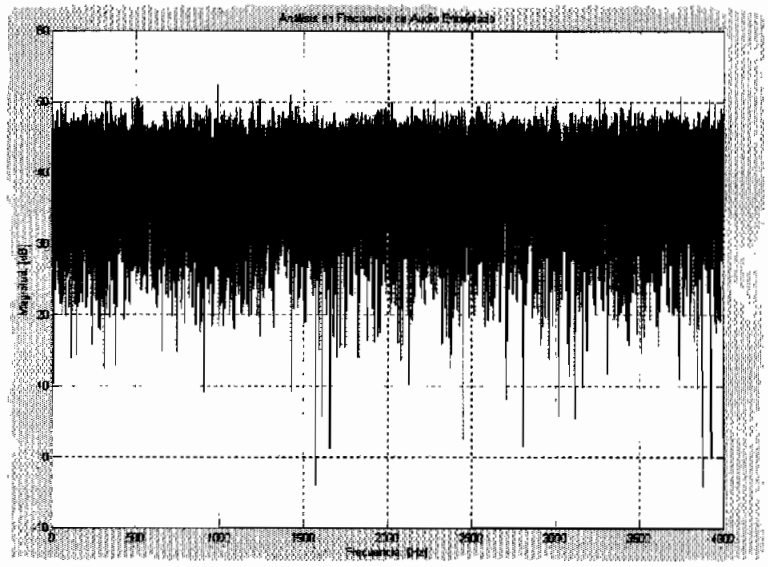


Figura 4. 17: Espectro de frecuencias de la señal encriptada mediante el algoritmo IDEA

Como se puede observar en los gráficos correspondientes a los espectros de las señales adquirida, encriptada y desencriptada, las características de los mismos no varían mucho en relación con los anteriores, lo cual sugiere un patrón para el resto del análisis en frecuencia.

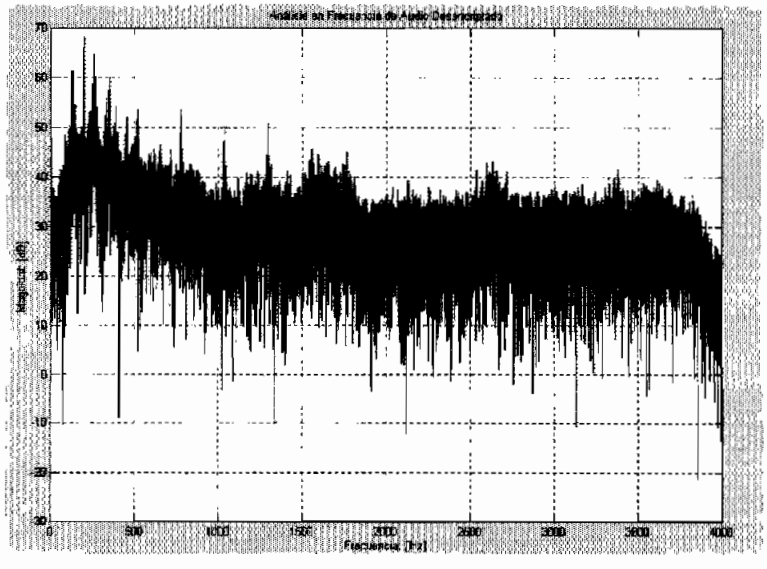


Figura 4. 18: Espectro de frecuencias de la señal desencriptada mediante el algoritmo IDEA

4.2.3.2.3. Espectros de frecuencia obtenidos en el algoritmo A5/1

Los espectros de frecuencia obtenidos para las señales adquirida, encriptada y desencriptada son los siguientes:

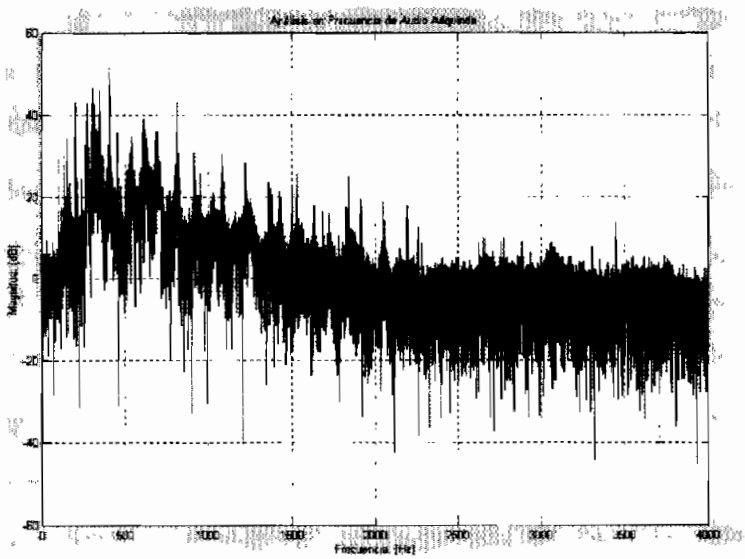


Figura 4. 19: Espectro de frecuencias de la señal adquirida

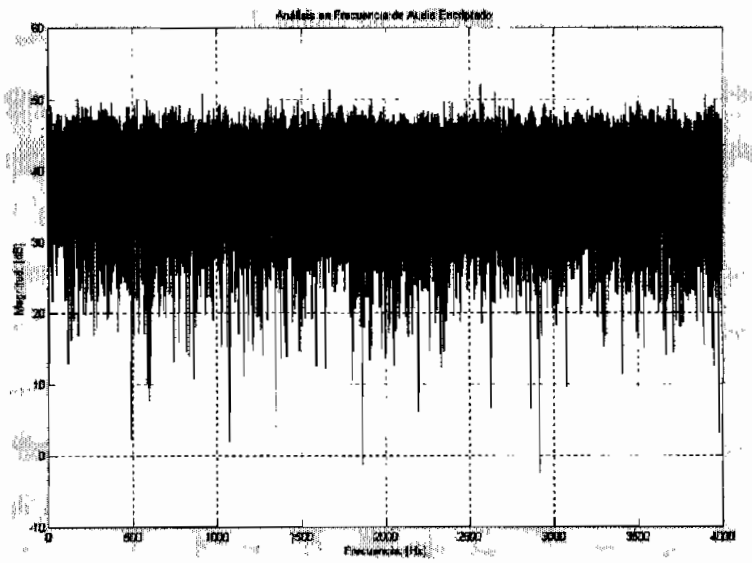


Figura 4. 20: Espectro de frecuencias de la señal encriptada mediante el algoritmo A5/1

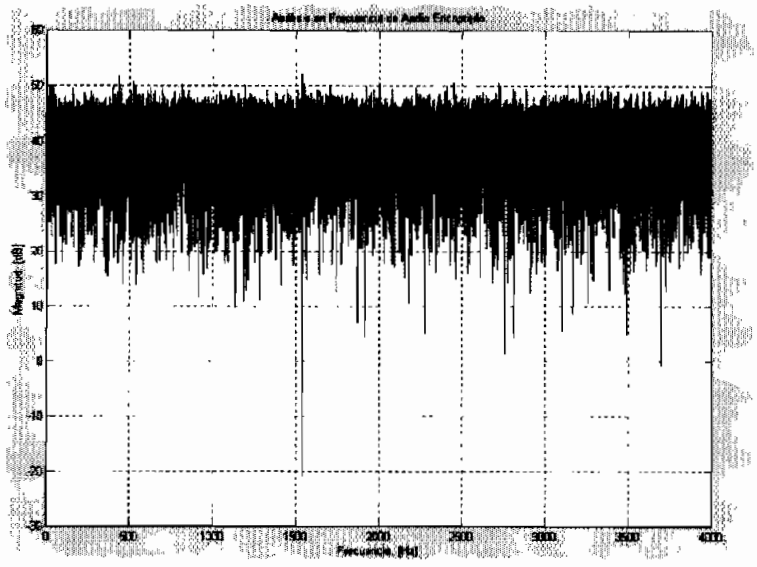


Figura 4. 23: Espectro de frecuencias de la señal encriptada mediante el algoritmo RC4

La gran similitud en los espectros de la señal encriptada sugiere que la eficiencia de encriptación de los algoritmos es muy parecida, aún cuando esto se debe juzgar de acuerdo a otros parámetros adicionales tales como ataques realizados, longitud de la clave, complejidad del algoritmo, etc.

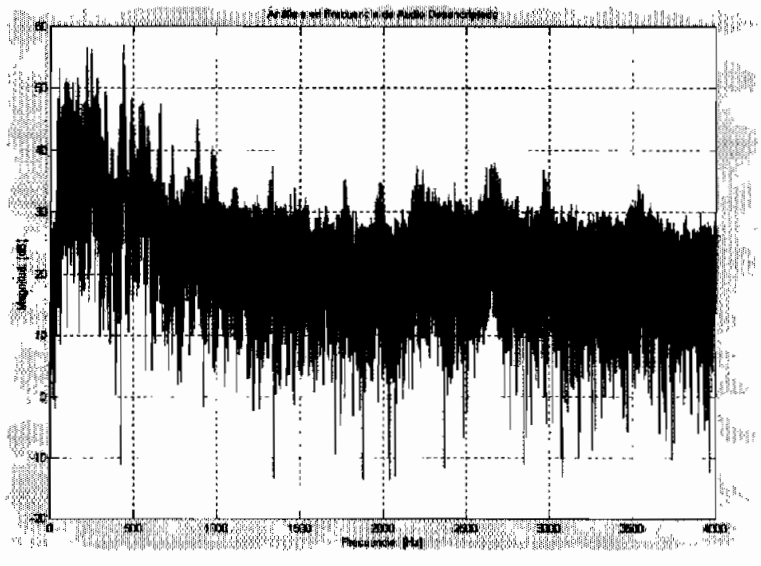


Figura 4. 24: Espectro de frecuencias de la señal desencriptada mediante el algoritmo RC4

4.2.3.3. Pruebas de escritorio realizadas a los algoritmos de encriptación

Para este banco de pruebas, se genera un vector de prueba de longitud 64, común a todos los algoritmos. Se analizan los resultados en la primera ronda, en el caso de los algoritmos de bloque, y en las primeras operaciones de los algoritmos de flujo. El vector de prueba es en notación hexadecimal: ACBCD2114DAE1577

4.2.3.3.1. Pruebas realizadas al algoritmo DES

En primer lugar, se genera una clave de sesión aleatoria de longitud 64, la cual en notación hexadecimal tiene el siguiente valor: C6DBF4C91A3CDA2F. La primera tarea a ejecutar es la permutación inicial, la cual reordena el texto llano para dejarlo en el valor: 94CEF3D827A333A4. Para comprobar la veracidad de la información, se realiza la siguiente prueba:

- Se toma el primer dígito hexadecimal del resultado de la permutación inicial: 9, cuyo valor binario es 1001.
- Los valores que conforman este dígito hexadecimal son los bits que están en las posiciones: 58, 50, 42 y 34, por lo tanto provienen de los siguientes dígitos hexadecimales del texto llano: 7, 1, A y 4, cuyas representaciones binarias son: 0111, 0001, 1010, 0100.
- En el caso del primer dígito hexadecimal, 7, comprende los bits que están en las posiciones 57, 58, 59 y 60, por lo tanto, el dígito binario correspondiente sería: 0111.
- En el caso del segundo dígito hexadecimal, 1, comprende los bits que están en las posiciones 49, 50, 51 y 52, por lo tanto, el dígito binario correspondiente sería: 0001.
- En el caso del tercer dígito hexadecimal, A, comprende los bits que están en las posiciones 41, 42, 43 y 44, por lo tanto, el dígito binario correspondiente sería: 1010.

- En el caso del cuarto dígito hexadecimal, 4, comprende los bits que están en las posiciones 33, 34, 35 y 36, por lo tanto, el dígito binario correspondiente sería: 0100.
- A partir de los cuatro dígitos binarios resaltados 1001, se reconstruye el primer dígito hexadecimal del resultado de la permutación inicial, cuyo valor es 9.

Como se observa, el resultado de la permutación inicial es exactamente igual al esperado.

A continuación, se realiza la permutación de la clave, es decir, reducir su valor de 64 bits a 56 bits mediante una permutación de posiciones. El valor hexadecimal de la clave permutada es el siguiente: 4F4FA47D3A5FA6. De nuevo, para comprobar la veracidad de la operación, se elige un dígito al azar, en este caso D, el cual contiene los bits que se encuentran en las posiciones 29, 30, 31 y 32, a los cuales corresponden respectivamente los bits 63, 55, 47 y 39 de la clave. Los dígitos hexadecimales que contienen estos dígitos binarios son respectivamente: F, A, C y A. Sus valores binarios son: 1111, 1010, 1100 y 1010. Los valores resaltados forman la cadena 1101, la cual representa el valor D.

A continuación se divide el texto llano en dos mitades, izquierda y derecha, las cuales tienen los valores hexadecimales: 94CEF3D8 y 27A333A4. Después se divide a la clave permutada en dos mitades, cuyos valores hexadecimales son: 4F4FA47 y D3A5FA6, para luego efectuar el desplazamiento de un bit de cada una de estas mitades; los resultados de este desplazamiento son los valores: A7A7D23 y 69D2FD3, los cuales generan la nueva clave A7A7D2369D2FD3. Se puede observar una ligera relación entre la clave original y la clave desplazada, lo cual confirma la operación de desplazamiento. Después de esta operación, se procede a comprimir la clave, lo cual da como resultado el valor hexadecimal: EBC87C7F752C. De nuevo, se comprueba la efectividad de la operación escogiendo un dígito hexadecimal al azar, en este caso F, el cual corresponde a los bits que se encuentran en las posiciones 29, 30, 31 y 32, correspondiendo respectivamente a los bits 47, 55, 30 y 40, los cuales se encuentran en los dígitos

hexadecimales de la clave desplazada F, 3, 6 y D, cuyo valor binario es, respectivamente: 1111, 0011, 0110 y 1101. El valor de cadena es 1111, que corresponde al dígito hexadecimal F.

Una vez realizada esta operación, se procede a expandir el bloque derecho del texto llano; el resultado en hexadecimal es: 10FD069A7D08. Se comprueba la operación eligiendo al azar un dígito, en este caso el segundo 0, el cual comprende los bits que se encuentran en las posiciones 17, 18, 19 y 20, que corresponden a los bits 12, 13, 12, 13 los mismos que se encuentran en los dígitos hexadecimales A y 3, cuyo valor binario es, respectivamente 1010 y 0011, y que, repetidos toman la forma 0000, lo cual corresponde al dígito hexadecimal 0. Una vez realizada esta operación, se procede a operar mediante or exclusivo a la clave comprimida y al texto llano permutado expandido. Utilizando la propiedad de $A \oplus 0 = A$, entonces el resultado de la operación en los bits 17, 18, 19 y 20, debe ser idéntico a los bits 17, 18, 19 y 20 de la clave comprimida. El resultado de esta operación en hexadecimal es: FB357AE50824. El resultado es idéntico, y por tanto la operación es válida. Es de notar que varios valores son iguales a los de la clave comprimida, producto de la aparición de varios 0 en el texto llano expandido.

Luego de las operaciones anteriores, se procede a entrar a las cajas S. Para realizar esta prueba, es necesario descomponer al último resultado en sus valores binarios, los cuales son en orden para cada caja S: 111110, 110011, 010101, 111010, 111001, 010000, 100000, 100100. Las posiciones de fila y columna respectivas son: (3,16), (4,10), (2,11), (3,14), (4,13), (1,9), (3,1), (3,3), cuyos valores decimales son: 0, 6, 5, 2, 10, 0, 2, 4. Se comprueba, al aplicar la operación dentro del algoritmo la veracidad de esta operación. El nuevo valor es: 0652A024.

Se procede a permutar el contenido del resultado de la sustitución por cajas S. El resultado de esta operación en hexadecimal es: 474100B0. Para comprobar la efectividad de esta operación, y de la anterior, se procede a realizar una prueba. De nuevo se escoge un dígito hexadecimal al azar, en este caso B.

El algoritmo de desencriptación no es tan redundante como en el caso del DES, debido al cálculo del inverso multiplicativo a través del algoritmo euclideano extendido. Para la primera ronda, es necesario calcular los valores de la ecuación $a \cdot x + b \cdot y = d$. De nuevo, se calculan los valores de las claves para cada ronda, de la forma indicada en la sección 3.3.2.2. Luego, a partir de estos valores, se calculan los valores de los inversos aditivos y multiplicativos con su correspondiente módulo. Los resultados indican que el inverso aditivo de las claves es igual al inverso aditivo normal. El inverso multiplicativo calculado es en cada caso $Z^{-1}_{11} = 56660$ y $Z^{-1}_{41} = 42218$. Al realizar la operación respectiva con la clave 1 de la ronda 5 y la clave 4 de la ronda 5, se obtiene: $(9007 \cdot 56660) \bmod 65537 = 1$ y $(14516 \cdot 42218) \bmod 65537 = 1$. El resto de operaciones son idénticas para cada ronda y la transformación final.

4.2.3.3.3. Pruebas realizadas al algoritmo A5/1

Para probar este algoritmo, se calcula los dos primeros bits de la clave de flujo en los algoritmos de encriptación y desencriptación. La clave de sesión generada aleatoriamente para el algoritmo de encriptación tiene el siguiente valor hexadecimal: 95D0306759C0F778. Se debe repartir este valor en tres registros, cuyos valores son:

1. R_1 = 1001010110110000001
2. R_2 = 10000011001110101100011
3. R_3 = 10000001111011101111000

Se calcula el primer bit de intervención de cada uno de los registros, así el primer bit de intervención de R_1 es igual a: $0 \oplus 0 \oplus 0 \oplus 1$, correspondientes a los bits 14, 17, 18 y 19 de R_1. El resultado es 1. Para R_2 se combinan los bits 21 y 22, cuyos valores son: $1 \oplus 1$. El resultado es igual a 0. Para R_3 se combinan los bits 8, 21, 22 y 23 de la siguiente forma: $1 \oplus 0 \oplus 0 \oplus 0$, lo que da como resultado 1. A continuación, se procede a desplazar los tres registros, cuyos nuevos valores son:

```
KeyPermRight2 = KeyShifted1(1,29:56);

%Desplazamiento de los bits de la clave
KeyShifLeft2 = KeyPermLeft2(:,kprs1);
KeyShiftRight2 = KeyPermRight2(:,kprs1);

%Reunión de la clave
KeyShifted2 = [KeyShifLeft2,KeyShiftRight2];

%Generación de la clave ronda 2
KeyCompr2 = KeyShifted2(:,kpr);

%Permutación de expansión
ExpPerm2 = RightBlockR2(:,xp);
XORExpPerm2 = XOR(KeyCompr2,ExpPerm2);

%Sustitución por cajas S
SBoxBlockVal2 = dessbox(XORExpPerm2);

%Permutación de caja P
PBoxVal2 = SBoxBlockVal2(:,pbox);

%Operación final
LeftBlockR3 = RightBlockR2;
RightBlockR3 = XOR(LeftBlockR2,PBoxVal2);

%Ronda 3

%Permutación de compresión de la clave

%División de la clave
KeyPermLeft3 = KeyShifted2(1,1:28);
KeyPermRight3 = KeyShifted2(1,29:56);

%Desplazamiento de los bits de la clave
KeyShifLeft3 = KeyPermLeft3(:,kprs2);
KeyShiftRight3 = KeyPermRight3(:,kprs2);

%Reunión de la clave
KeyShifted3 = [KeyShifLeft3,KeyShiftRight3];

%Generación de la clave ronda 3
KeyCompr3 = KeyShifted3(:,kpr);

%Permutación de expansión
ExpPerm3 = RightBlockR3(:,xp);
XORExpPerm3 = XOR(KeyCompr3,ExpPerm3);

%Sustitución por cajas S
SBoxBlockVal3 = dessbox(XORExpPerm3);

%Permutación de caja P
PBoxVal3 = SBoxBlockVal3(:,pbox);

%Operación final
LeftBlockR4 = RightBlockR3;
RightBlockR4 = XOR(LeftBlockR3,PBoxVal3);

%Ronda 4
```

```
%Permutación de compresión de la clave

%División de la clave
KeyPermLeft4 = KeyShifted3(1,1:28);
KeyPermRight4 = KeyShifted3(1,29:56);

%Desplazamiento de los bits de la clave
KeyShifLeft4 = KeyPermLeft4(:,kprs2);
KeyShiftRight4 = KeyPermRight4(:,kprs2);

%Reunión de la clave
KeyShifted4 = [KeyShifLeft4,KeyShiftRight4];

%Generación de la clave ronda 4
KeyCompr4 = KeyShifted4(:,kpr);

%Permutación de expansión
ExpPerm4 = RightBlockR4(:,xp);
XORExpPerm4 = XOR(KeyCompr4,ExpPerm4);

%Sustitución por cajas S

%Generación de las entradas a las cajas S
SBoxBlockVal4 = dessbox(XORExpPerm4);

%Permutación de caja P
PBoxVal4 = SBoxBlockVal4(:,pbox);

%Operación final
LeftBlockR5 = RightBlockR4;
RightBlockR5 = XOR(LeftBlockR4,PBoxVal4);

%Ronda 5

%Permutación de compresión de la clave

%División de la clave
KeyPermLeft5 = KeyShifted4(1,1:28);
KeyPermRight5 = KeyShifted4(1,29:56);

%Desplazamiento de los bits de la clave
KeyShifLeft5 = KeyPermLeft5(:,kprs2);
KeyShiftRight5 = KeyPermRight5(:,kprs2);

%Reunión de la clave
KeyShifted5 = [KeyShifLeft5,KeyShiftRight5];

%Generación de la clave ronda 4
KeyCompr5 = KeyShifted5(:,kpr);

%Permutación de expansión
ExpPerm5 = RightBlockR5(:,xp);
XORExpPerm5 = XOR(KeyCompr5,ExpPerm5);

%Sustitución por cajas S
SBoxBlockVal5 = dessbox(XORExpPerm5);

%Permutación de caja P
PBoxVal5 = SBoxBlockVal5(:,pbox);
```

```
%Operación final
LeftBlockR6 = RightBlockR5;
RightBlockR6 = XOR(LeftBlockR5,PBoxVal5);

%Ronda 6

%Permutación de compresión de la clave

%División de la clave
KeyPermLeft6 = KeyShifted5(1,1:28);
KeyPermRight6 = KeyShifted5(1,29:56);

%Desplazamiento de los bits de la clave
KeyShifLeft6 = KeyPermLeft6(:,kprs2);
KeyShiftRight6 = KeyPermRight6(:,kprs2);

%Reunión de la clave
KeyShifted6 = [KeyShifLeft6,KeyShiftRight6];

%Generación de la clave ronda 6
KeyCompr6 = KeyShifted6(:,kpr);

%Permutación de expansión
ExpPerm6 = RightBlockR6(:,xp);
XORExpPerm6 = XOR(KeyCompr6,ExpPerm6);

%Sustitución por cajas S
SBoxBlockVal6 = dessbox(XORExpPerm6);

%Permutación de caja P
PBoxVal6 = SBoxBlockVal6(:,pbox);

%Operación final
LeftBlockR7 = RightBlockR6;
RightBlockR7 = XOR(LeftBlockR6,PBoxVal6);

%Ronda 7

%Permutación de compresión de la clave

%División de la clave
KeyPermLeft7 = KeyShifted6(1,1:28);
KeyPermRight7 = KeyShifted6(1,29:56);

%Desplazamiento de los bits de la clave
KeyShifLeft7 = KeyPermLeft7(:,kprs2);
KeyShiftRight7 = KeyPermRight7(:,kprs2);

%Reunión de la clave
KeyShifted7 = [KeyShifLeft7,KeyShiftRight7];

%Generación de la clave ronda 7
KeyCompr7 = KeyShifted7(:,kpr);

%Permutación de expansión
ExpPerm7 = RightBlockR7(:,xp);
XORExpPerm7 = XOR(KeyCompr7,ExpPerm7);

%Sustitución por cajas S
SBoxBlockVal7 = dessbox(XORExpPerm7);
```

```
%Permutación de caja P
PBoxVal7 = SBoxBlockVal7(:,pbox);

%Operación final
LeftBlockR8 = RightBlockR7;
RightBlockR8 = XOR(LeftBlockR7,PBoxVal7);

%Ronda 8

%Permutación de compresión de la clave

%División de la clave
KeyPermLeft8 = KeyShifted7(1,1:28);
KeyPermRight8 = KeyShifted7(1,29:56);

%Desplazamiento de los bits de la clave
KeyShifLeft8 = KeyPermLeft8(:,kprs2);
KeyShiftRight8 = KeyPermRight8(:,kprs2);

%Reunión de la clave
KeyShifted8 = [KeyShifLeft8,KeyShiftRight8];

%Generación de la clave ronda 8
KeyCompr8 = KeyShifted8(:,kpr);

%Permutación de expansión
ExpPerm8 = RightBlockR8(:,xp);
XORExpPerm8 = XOR(KeyCompr8,ExpPerm8);

%Sustitución por cajas S
SBoxBlockVal8 = dessbox(XORExpPerm8);

%Permutación de caja P
PBoxVal8 = SBoxBlockVal8(:,pbox);

%Operación final
LeftBlockR9 = RightBlockR8;
RightBlockR9 = XOR(LeftBlockR8,PBoxVal8);

%Ronda 9

%Permutación de compresión de la clave

%División de la clave
KeyPermLeft9 = KeyShifted8(1,1:28);
KeyPermRight9 = KeyShifted8(1,29:56);

%Desplazamiento de los bits de la clave
KeyShifLeft9 = KeyPermLeft9(:,kprs1);
KeyShiftRight9 = KeyPermRight9(:,kprs1);

%Reunión de la clave
KeyShifted9 = [KeyShifLeft9,KeyShiftRight9];

%Generación de la clave ronda 9
KeyCompr9 = KeyShifted9(:,kpr);

%Permutación de expansión
ExpPerm9 = RightBlockR9(:,xp);
```

```
XORExpPerm9 = XOR(KeyCompr9,ExpPerm9);

%Sustitución por cajas S
SBoxBlockVal9 = dessbox(XORExpPerm9);

%Permutación de caja P
PBoxVal9 = SBoxBlockVal9(:,pbox);

%Operación final
LeftBlockR10 = RightBlockR9;
RightBlockR10 = XOR(LeftBlockR9,PBoxVal9);

%Ronda 10

%Permutación de compresión de la clave

%División de la clave
KeyPermLeft10 = KeyShifted9(1,1:28);
KeyPermRight10 = KeyShifted9(1,29:56);

%Desplazamiento de los bits de la clave
KeyShifLeft10 = KeyPermLeft10(:,kprs2);
KeyShiftRight10 = KeyPermRight10(:,kprs2);

%Reunión de la clave
KeyShifted10 = [KeyShifLeft10,KeyShiftRight10];

%Generación de la clave ronda 10
KeyCompr10 = KeyShifted10(:,kpr);

%Permutación de expansión
ExpPerm10 = RightBlockR10(:,xp);
XORExpPerm10 = XOR(KeyCompr10,ExpPerm10);

%Sustitución por cajas S
SBoxBlockVal10 = dessbox(XORExpPerm10);

%Permutación de caja P
PBoxVal10 = SBoxBlockVal10(:,pbox);

%Operación final
LeftBlockR11 = RightBlockR10;
RightBlockR11 = XOR(LeftBlockR10,PBoxVal10);

%Ronda 11

%Permutación de compresión de la clave

%División de la clave
KeyPermLeft11 = KeyShifted10(1,1:28);
KeyPermRight11 = KeyShifted10(1,29:56);

%Desplazamiento de los bits de la clave
KeyShifLeft11 = KeyPermLeft11(:,kprs2);
KeyShiftRight11 = KeyPermRight11(:,kprs2);

%Reunión de la clave
KeyShifted11 = [KeyShifLeft11,KeyShiftRight11];

%Generación de la clave ronda 11
```

```
KeyCompr11 = KeyShifted11(:,kpr);

%Permutación de expansión
ExpPerm11 = RightBlockR11(:,xp);
XORExpPerm11 = XOR(KeyCompr11,ExpPerm11);

%Sustitución por cajas S
SBoxBlockVal11 = dessbox(XORExpPerm11);

%Permutación de caja P
PBoxVal11 = SBoxBlockVal11(:,pbox);

%Operación final
LeftBlockR12 = RightBlockR11;
RightBlockR12 = XOR(LeftBlockR11,PBoxVal11);

%Ronda 12

%Permutación de compresión de la clave

%División de la clave
KeyPermLeft12 = KeyShifted11(1,1:28);
KeyPermRight12 = KeyShifted11(1,29:56);

%Desplazamiento de los bits de la clave
KeyShifLeft12 = KeyPermLeft12(:,kprs2);
KeyShiftRight12 = KeyPermRight12(:,kprs2);

%Reunión de la clave
KeyShifted12 = [KeyShifLeft12,KeyShiftRight12];

%Generación de la clave ronda 12
KeyCompr12 = KeyShifted12(:,kpr);

%Permutación de expansión
ExpPerm12 = RightBlockR12(:,xp);
XORExpPerm12 = XOR(KeyCompr12,ExpPerm12);

%Sustitución por cajas S
SBoxBlockVal12 = dessbox(XORExpPerm12);

%Permutación de caja P
PBoxVal12 = SBoxBlockVal12(:,pbox);

%Operación final
LeftBlockR13 = RightBlockR12;
RightBlockR13 = XOR(LeftBlockR12,PBoxVal12);

%Ronda 13

%Permutación de compresión de la clave

%División de la clave
KeyPermLeft13 = KeyShifted12(1,1:28);
KeyPermRight13 = KeyShifted12(1,29:56);

%Desplazamiento de los bits de la clave
KeyShifLeft13 = KeyPermLeft13(:,kprs2);
KeyShiftRight13 = KeyPermRight13(:,kprs2);
```

```
%Reunión de la clave
KeyShifted13 = [KeyShifLeft13,KeyShiftRight13];

%Generación de la clave ronda 13
KeyCompr13 = KeyShifted13(:,kpr);

%Permutación de expansión
ExpPerm13 = RightBlockR13(:,xp);
XORExpPerm13 = XOR(KeyCompr13,ExpPerm13);

%Sustitución por cajas S
SBoxBlockVal13 = dessbox(XORExpPerm13);

%Permutación de caja P
PBoxVal13 = SBoxBlockVal13(:,pbox);

%Operación final
LeftBlockR14 = RightBlockR13;
RightBlockR14 = XOR(LeftBlockR13,PBoxVal13);

%Ronda 14

%Permutación de compresión de la clave

%División de la clave
KeyPermLeft14 = KeyShifted13(1,1:28);
KeyPermRight14 = KeyShifted13(1,29:56);

%Desplazamiento de los bits de la clave
KeyShifLeft14 = KeyPermLeft14(:,kprs2);
KeyShiftRight14 = KeyPermRight14(:,kprs2);

%Reunión de la clave
KeyShifted14 = [KeyShifLeft14,KeyShiftRight14];

%Generación de la clave ronda 14
KeyCompr14 = KeyShifted14(:,kpr);

%Permutación de expansión
ExpPerm14 = RightBlockR14(:,xp);
XORExpPerm14 = XOR(KeyCompr14,ExpPerm14);

%Sustitución por cajas S
SBoxBlockVal14 = dessbox(XORExpPerm14);

%Permutación de caja P
PBoxVal14 = SBoxBlockVal14(:,pbox);

%Operación final
LeftBlockR15 = RightBlockR14;
RightBlockR15 = XOR(LeftBlockR14,PBoxVal14);

%Ronda 15

%Permutación de compresión de la clave

%División de la clave
KeyPermLeft15 = KeyShifted14(1,1:28);
KeyPermRight15 = KeyShifted14(1,29:56);
```

```
%Desplazamiento de los bits de la clave
KeyShifLeft15 = KeyPermLeft15(:,kprs2);
KeyShiftRight15 = KeyPermRight15(:,kprs2);

%Reunión de la clave
KeyShiftedl5 = [KeyShifLeft15,KeyShiftRight15];

%Generación de la clave ronda 15
KeyCompr15 = KeyShifted15(:,kpr);

%Permutación de expansión
ExpPerml5 = RightBlockR15(:,xp);
XORExpPerm15 = XOR(KeyCompr15,ExpPerm15);

%Sustitución por cajas S
SBoxBlockVal15 = dessbox(XORExpPerm15);

%Permutación de caja P
PBoxVal15 = SBoxBlockVal15(:,pbox);

%Operación final
LeftBlockR16 = RightBlockR15;
RightBlockR16 = XOR(LeftBlockR15,PBoxVal15);

%Ronda 16

%Permutación de compresión de la clave

%División de la clave
KeyPermLeft16 = KeyShiftedl5(1,1:28);
KeyPermRight16 = KeyShifted15(1,29:56);

%Desplazamiento de los bits de la clave
KeyShifLeft16 = KeyPermLeft16(:,kprs1);
KeyShiftRight16 = KeyPermRight11(:,kprs1);

%Reunión de la clave
KeyShifted16 = [KeyShifLeft16,KeyShiftRight16];

%Generación de la clave ronda 11
KeyCompr16 = KeyShifted16(:,kpr);

%Permutación de expansión
ExpPerm16 = RightBlockR16(:,xp);
XORExpPerml6 = XOR(KeyCompr16,ExpPerm16);

%Sustitución por cajas S
SBoxBlockVal16 = dessbox(XORExpPerm16);

%Permutación de caja P
PBoxVal16 = SBoxBlockVal16(:,pbox);

%Operación final
LeftBlockR17 = RightBlockR16;
RightBlockR17 = XOR(LeftBlockR16,PBoxVal16);

%Permutación final
PreCiText = [RightBlockR17,LeftBlockR17];
FinalPerm = PreCiText(:,fp);
CiText = FinalPerm;
```

A.1.2. ALGORITMO DE DESENCRIPTACIÓN DESDEC.M

```
function PlText = desdec(CiText)

%Desencriptación del algoritmo DES
%
%DESDEC(CiText) permite desencriptar un bloque
%encriptado con la función DESCIP.

%Tablas de permutación

%Tabla de permutación inicial
p = [58 50 42 34 26 18 10 2 60 52 44 36 28 20 12 4 ...
     62 54 46 38 30 22 14 6 64 56 48 40 32 24 16 8 ...
     57 49 41 33 25 17 9 1 59 51 43 35 27 19 11 3 ...
     61 53 45 37 29 21 13 5 63 55 47 39 31 23 15 7];

%Tabla de permutación de la clave
kp = [57 49 41 33 25 17 9 1 58 50 42 34 26 18 ...
      10 2 59 51 43 35 27 19 11 3 60 52 44 36 63 ...
      55 47 39 31 23 15 7 62 54 46 38 30 22 ...
      14 6 61 53 45 37 29 21 13 5 28 20 12 4];

%Tabla de permutación de compresión
kpr = [14 17 11 24 1 5 3 28 15 6 21 10 ...
       23 19 12 4 26 8 16 7 27 20 13 2 ...
       41 52 31 37 47 55 30 40 51 45 33 48 ...
       44 49 39 56 34 53 46 42 50 36 29 32];

%Tablas para los desplazamientos de clave
kprs1 = [28 1 2 3 4 5 6 7 8 9 10 11 12 13 ...
         14 15 16 17 18 19 20 21 22 23 24 25 26 27];

kprs2 = [27 28 1 2 3 4 5 6 7 8 9 10 11 12 ...
         13 14 15 16 17 18 19 20 21 22 23 24 25 26];

%Tabla de permutación de expansión
xp = [32 1 2 3 4 5 4 5 6 7 8 9 ...
      8 9 10 11 12 13 12 13 14 15 16 17 ...
      16 17 18 19 20 21 20 21 22 23 24 25 ...
      24 25 26 27 28 29 28 29 30 31 32 1];

%Caja P
pbox = [16 7 20 21 29 12 28 17 1 15 23 26 5 18 31 10 ...
      2 8 24 14 32 27 3 9 19 13 30 6 22 11 4 25];

%Tabla de permutación final
fp = [40 8 48 16 56 24 64 32 39 7 47 15 55 23 63 31 ...
      38 6 46 14 54 22 62 30 37 5 45 13 53 21 61 29 ...
      36 4 44 12 52 20 60 28 35 3 43 11 51 19 59 27 ...
      34 2 42 10 50 18 58 26 33 1 41 9 49 17 57 25];

%Permutación inicial
InPerm = CiText(:,p);

%Generación de la clave
global KeyCompr1 KeyCompr2 KeyCompr3 KeyCompr4 KeyCompr5 KeyCompr6 ...
       KeyCompr7 KeyCompr8 KeyCompr9 KeyComprl0 KeyComprl1 KeyComprl2 ...
       KeyComprl3 KeyComprl4 KeyComprl5 KeyComprl6;
```

```
%División del texto llano
InPermLeft1 = InPerm(:,1:32);
InPermRight1 = InPerm(:,33:64);

%Ronda 1

%Permutación de expansión
ExpPerm1 = InPermRight1(:,xp);
XORExpPerm1 = XOR(KeyCompr16,ExpPerm1);

%Sustitución por cajas S
SBoxBlockVal1 = dessbox(XORExpPerm1);

%Permutación de caja P
PBoxVal1 = SBoxBlockVal1(:,pbox);

%Operación final
LeftBlockR2 = InPermRight1;
RightBlockR2 = XOR(InPermLeft1,PBoxVal1);

%Ronda 2

%Permutación de expansión
ExpPerm2 = RightBlockR2(:,xp);
XORExpPerm2 = XOR(KeyCompr15,ExpPerm2);

%Sustitución por cajas S
SBoxBlockVal2 = dessbox(XORExpPerm2);

%Permutación de caja P
PBoxVal2 = SBoxBlockVal2(:,pbox);

%Operación final
LeftBlockR3 = RightBlockR2;
RightBlockR3 = XOR(LeftBlockR2,PBoxVal2);

%Ronda 3

%Permutación de expansión
ExpPerm3 = RightBlockR3(:,xp);
XORExpPerm3 = XOR(KeyCompr14,ExpPerm3);

%Sustitución por cajas S
SBoxBlockVal3 = dessbox(XORExpPerm3);

%Permutación de caja P
PBoxVal3 = SBoxBlockVal3(:,pbox);

%Operación final
LeftBlockR4 = RightBlockR3;
RightBlockR4 = XOR(LeftBlockR3,PBoxVal3);

%Ronda 4

%Permutación de expansión
ExpPerm4 = RightBlockR4(:,xp);
XORExpPerm4 = XOR(KeyCompr13,ExpPerm4);

%Sustitución por cajas S
```

```
%Permutación de expansión
ExpPerm8 = RightBlockR8(:,xp);
XORExpPerm8 = XOR(KeyCompr9,ExpPerm8);

%Sustitución por cajas S
SBoxBlockVal8 = dessbox(XORExpPerm8);

%Permutación de caja P
PBoxVal8 = SBoxBlockVal8(:,pbox);

%Operación final
LeftBlockR9 = RightBlockR8;
RightBlockR9 = XOR(LeftBlockR8,PBoxVal8);

%Ronda 9

%Permutación de expansión
ExpPerm9 = RightBlockR9(:,xp);
XORExpPerm9 = XOR(KeyCompr8,ExpPerm9);

%Sustitución por cajas S
SBoxBlockVal9 = dessbox(XORExpPerm9);

%Permutación de caja P
PBoxVal9 = SBoxBlockVal9(:,pbox);

%Operación final
LeftBlockR10 = RightBlockR9;
RightBlockR10 = XOR(LeftBlockR9,PBoxVal9);

%Ronda 10

%Permutación de expansión
ExpPerm10 = RightBlockR10(:,xp);
XORExpPerm10 = XOR(KeyCompr7,ExpPerm10);

%Sustitución por cajas S
SBoxBlockVal10 = dessbox(XORExpPerm10);

%Permutación de caja P
PBoxVal10 = SBoxBlockVal10(:,pbox);

%Operación final
LeftBlockR11 = RightBlockR10;
RightBlockR11 = XOR(LeftBlockR10,PBoxVal10);

%Ronda 11

%Permutación de expansión
ExpPerm11 = RightBlockR11(:,xp);
XORExpPerm11 = XOR(KeyCompr6,ExpPerm11);

%Sustitución por cajas S
SBoxBlockVal11 = dessbox(XORExpPerm11);

%Permutación de caja P
PBoxVal11 = SBoxBlockVal11(:,pbox);

%Operación final
LeftBlockR12 = RightBlockR11;
```

```
RightBlockR12 = XOR(LeftBlockR11,PBoxVal11);

%Ronda 12

%Permutación de expansión
ExpPerm12 = RightBlockR12(:,xp);
XORExpPerm12 = XOR(KeyCompr5,ExpPerm12);

%Sustitución por cajas S
SBoxBlockVal12 = dessbox(XORExpPerm12);

%Permutación de caja P
PBoxVal12 = SBoxBlockVal12(:,pbox);

%Operación final
LeftBlockR13 = RightBlockR12;
RightBlockR13 = XOR(LeftBlockR12,PBoxVal12);

%Ronda 13

%Permutación de expansión
ExpPerm13 = RightBlockR13(:,xp);
XORExpPerm13 = XOR(KeyCompr4,ExpPerm13);

%Sustitución por cajas S
SBoxBlockVal13 = dessbox(XORExpPerm13);

%Permutación de caja P
PBoxVal13 = SBoxBlockVal13(:,pbox);

%Operación final
LeftBlockR14 = RightBlockR13;
RightBlockR14 = XOR(LeftBlockR13,PBoxVal13);

%Ronda 14

%Permutación de expansión
ExpPerm14 = RightBlockR14(:,xp);
XORExpPerm14 = XOR(KeyCompr3,ExpPerm14);

%Sustitución por cajas S
SBoxBlockVal14 = dessbox(XORExpPerm14);

%Permutación de caja P
PBoxVal14 = SBoxBlockVal14(:,pbox);

%Operación final
LeftBlockR15 = RightBlockR14;
RightBlockR15 = XOR(LeftBlockR14,PBoxVal14);

%Ronda 15

%Permutación de expansión
ExpPerm15 = RightBlockR15(:,xp);
XORExpPerm15 = XOR(KeyCompr2,ExpPerm15);

%Sustitución por cajas S
SBoxBlockVal15 = dessbox(XORExpPerm15);

%Permutación de caja P
```

```
PBoxVal15 = SBoxBlockVal15(:,pbox);

%Operación final
LeftBlockR16 = RightBlockR15;
RightBlockR16 = XOR(LeftBlockR15,PBoxVal15);

%Ronda 16

%Permutación de expansión
ExpPerm16 = RightBlockR16(:,xp);
XORExpPerm16 = XOR(KeyCompr1,ExpPerm16);

%Sustitución por cajas S
SBoxBlockVal16 = dessbox(XORExpPerm16);

%Permutación de caja P
PBoxVal16 = SBoxBlockVal16(:,pbox);

%Operación final
LeftBlockR17 = RightBlockR16;
RightBlockR17 = XOR(LeftBlockR16,PBoxVal16);

%Permutación final
PrePlText = [RightBlockR17,LeftBlockR17];
FinalPerm = PrePlText(:,fp);
PlText = FinalPerm;
```

A.1.3. SUSTITUCIÓN CON CAJAS S DESSBOX.M

```
function SBoxBlockVal = dessbox(XORExpPerm)

% DESSBOX Cálculo de las cajas S para el algoritmo DES
%
%DESSBOX permite calcular el valor de las cajas S para
%el algoritmo DES. Se utiliza como programa auxiliar de
%DESCIP y DESDEC

%Cajas S
sbox1 = [14 4 13 1 2 15 11 8 3 10 6 12 5 9 0 7;
      0 15 7 4 14 2 13 1 10 6 12 11 9 5 3 8;
      4 1 14 8 13 6 2 11 15 12 9 7 3 10 5 0;
      15 12 8 2 4 9 1 7 5 11 3 14 10 0 6 13];

sbox2 = [15 1 8 14 6 11 3 4 9 7 2 13 12 0 5 10;
      3 13 4 7 15 2 8 14 12 0 1 10 6 9 11 5;
      0 14 7 11 10 4 13 1 5 8 12 6 9 3 2 15;
      13 8 10 1 3 15 4 2 11 6 7 12 0 5 14 9];

sbox3 = [10 0 9 14 6 3 15 5 1 13 12 7 11 4 2 8;
      13 7 0 9 3 4 6 10 2 8 5 14 12 11 15 1;
      13 6 4 9 8 15 3 0 11 1 2 12 5 10 14 7;
      1 10 13 0 6 9 8 7 4 15 14 3 11 5 2 12];

sbox4 = [7 13 14 3 0 6 9 10 1 2 8 5 11 12 4 15;
      13 8 11 5 6 15 0 3 4 7 2 12 1 10 14 9;
      10 6 9 0 12 11 7 13 15 1 3 14 5 2 8 4;
      3 15 0 6 10 1 13 8 9 4 5 11 12 7 2 14];

sbox5 = [2 12 4 1 7 10 11 6 8 5 3 15 13 0 14 9;
      14 11 2 12 4 7 13 1 5 0 15 10 3 9 8 6;
```

```
      4 2 1 11 10 13 7 8 15 9 12 5 6 3 0 14;
      11 8 12 7 1 14 2 13 6 15 0 9 10 4 5 3];

sbox6 = [12 1 10 15 9 2 6 8 0 13 3 4 14 7 5 11;
      10 15 4 2 7 12 9 5 6 1 13 14 0 11 3 8;
      9 14 15 5 2 8 12 3 7 0 4 10 1 13 11 6;
      4 3 2 12 9 5 15 10 11 14 1 7 6 0 8 13];

sbox7 = [4 11 2 14 15 0 8 13 3 12 9 7 5 10 6 1;
      13 0 11 7 4 9 1 10 14 3 5 12 2 15 8 6;
      1 4 11 13 12 3 7 14 10 15 6 8 0 5 9 2;
      6 11 13 8 1 4 10 7 9 5 0 15 14 2 3 12];

sbox8 = [13 2 8 4 6 15 11 1 10 9 3 14 5 0 2 7;
      1 15 13 8 10 3 7 4 12 5 6 11 0 14 9 2;
      7 11 4 1 9 12 14 2 0 6 10 13 15 3 5 8;
      2 1 14 7 4 10 8 13 15 12 9 0 3 5 6 11];
%Sustitución por cajas S

%Generación de las entradas a las cajas S
SubsBox1 = XORExpPerm(1,1:6); SubsBox2 = XORExpPerm(1,7:12);
SubsBox3 = XORExpPerm(1,13:18); SubsBox4 = XORExpPerm(1,19:24);
SubsBox5 = XORExpPerm(1,25:30); SubsBox6 = XORExpPerm(1,31:36);
SubsBox7 = XORExpPerm(1,37:42); SubsBox8 = XORExpPerm(1,43:48);

%Localización de entradas en las cajas S

%Bloque de entrada 1
%Localización de fila
CoefR11 = SubsBox1(1,1); CoefRStr11 = int2str(CoefR11);
CoefR12 = SubsBox1(1,6); CoefRStr12 = int2str(CoefR12);

BinRowS1 = strcat(CoefRStr11,CoefRStr12);
LocRowS1 = bin2dec(BinRowS1) + 1;

%Localización de columna
CoefC11 = SubsBox1(1,2); CoefCStr11 = int2str(CoefC11);
CoefC12 = SubsBox1(1,3); CoefCStr12 = int2str(CoefC12);
CoefC13 = SubsBox1(1,4); CoefCStr13 = int2str(CoefC13);
CoefC14 = SubsBox1(1,5); CoefCStr14 = int2str(CoefC14);

BinColS1 = strcat(CoefCStr11,CoefCStr12,CoefCStr13,CoefCStr14);
LocColS1 = bin2dec(BinColS1) + 1;

%Salida de la caja S
SBoxVal1 = sbox1(LocRowS1,LocColS1);
SBoxBinVal1 = dec2bin(SBoxVal1);
SBoxMatVal1 = numer4(SBoxBinVal1);

%Bloque de entrada 2
%Localización de fila
CoefR21 = SubsBox2(1,1); CoefRStr21 = int2str(CoefR21);
CoefR22 = SubsBox2(1,6); CoefRStr22 = int2str(CoefR22);

BinRowS2 = strcat(CoefRStr21,CoefRStr22);
LocRowS2 = bin2dec(BinRowS2) + 1;

%Localización de columna
CoefC21 = SubsBox2(1,2); CoefCStr21 = int2str(CoefC21);
CoefC22 = SubsBox2(1,3); CoefCStr22 = int2str(CoefC22);
```

```
CoefC23 = SubsBox2(1,4); CoefCStr23 = int2str(CoefC23);
CoefC24 = SubsBox2(1,5); CoefCStr24 = int2str(CoefC24);

BinColS2 = strcat(CoefCStr21,CoefCStr22,CoefCStr23,CoefCStr24);
LocColS2 = bin2dec(BinColS2) + 1;

%Salida de la caja S
SBoxVal2 = sbox2(LocRowS2,LocColS2);
SBoxBinVal2 = dec2bin(SBoxVal2);
SBoxMatVal2 = numer4(SBoxBinVal2);

%Bloque de entrada 3
%Localización de fila
CoefR31 = SubsBox3(1,1); CoefRStr31 = int2str(CoefR31);
CoefR32 = SubsBox3(1,6); CoefRStr32 = int2str(CoefR32);

BinRowS3 = strcat(CoefRStr31,CoefRStr32);
LocRowS3 = bin2dec(BinRowS3) + 1;

%Localización de columna
CoefC31 = SubsBox3(1,2); CoefCStr31 = int2str(CoefC31);
CoefC32 = SubsBox3(1,3); CoefCStr32 = int2str(CoefC32);
CoefC33 = SubsBox3(1,4); CoefCStr33 = int2str(CoefC33);
CoefC34 = SubsBox3(1,5); CoefCStr34 = int2str(CoefC34);

BinColS3 = strcat(CoefCStr31,CoefCStr32,CoefCStr33,CoefCStr34);
LocColS3 = bin2dec(BinColS3) + 1;

%Salida de la caja S
SBoxVal3 = sbox3(LocRowS3,LocColS3);
SBoxBinVal3 = dec2bin(SBoxVal3);
SBoxMatVal3 = numer4(SBoxBinVal3);

%Bloque de entrada 4
%Localización de fila
CoefR41 = SubsBox4(1,1); CoefRStr41 = int2str(CoefR41);
CoefR42 = SubsBox4(1,6); CoefRStr42 = int2str(CoefR42);

BinRowS4 = strcat(CoefRStr41,CoefRStr42);
LocRowS4 = bin2dec(BinRowS4) + 1;

%Localización de columna
CoefC41 = SubsBox4(1,2); CoefCStr41 = int2str(CoefC41);
CoefC42 = SubsBox4(1,3); CoefCStr42 = int2str(CoefC42);
CoefC43 = SubsBox4(1,4); CoefCStr43 = int2str(CoefC43);
CoefC44 = SubsBox4(1,5); CoefCStr44 = int2str(CoefC44);

BinColS4 = strcat(CoefCStr41,CoefCStr42,CoefCStr43,CoefCStr44);
LocColS4 = bin2dec(BinColS4) + 1;

%Salida de la caja S
SBoxVal4 = sbox4(LocRowS4,LocColS4);
SBoxBinVal4 = dec2bin(SBoxVal4);
SBoxMatVal4 = numer4(SBoxBinVal4);

%Bloque de entrada 5
%Localización de fila
CoefR51 = SubsBox5(1,1); CoefRStr51 = int2str(CoefR51);
CoefR52 = SubsBox5(1,6); CoefRStr52 = int2str(CoefR52);
```

```
%Bloque de entrada 8
%Localización de fila
CoefR81 = SubsBox8(1,1); CoefRStr81 = int2str(CoefR81);
CoefR82 = SubsBox8(1,6); CoefRStr82 = int2str(CoefR82);

BinRowS8 = strcat(CoefRStr81,CoefRStr82);
LocRowS8 = bin2dec(BinRowS8) + 1;

%Localización de columna
CoefC81 = SubsBox8(1,2); CoefCStr81 = int2str(CoefC81);
CoefC82 = SubsBox8(1,3); CoefCStr82 = int2str(CoefC82);
CoefC83 = SubsBox8(1,4); CoefCStr83 = int2str(CoefC83);
CoefC84 = SubsBox8(1,5); CoefCStr84 = int2str(CoefC84);

BinColS8 = strcat(CoefCStr81,CoefCStr82,CoefCStr83,CoefCStr84);
LocColS8 = bin2dec(BinColS8) + 1;

%Salida de la caja S
SBoxVal8 = sbox8(LocRowS8,LocColS8);
SBoxBinVal8 = dec2bin(SBoxVal8);
SBoxMatVal8 = numer4(SBoxBinVal8);

%Salida de la caja S
SBoxBlockVal = [SBoxMatVal1,SBoxMatVal2,SBoxMatVal3,SBoxMatVal4, ...
                SBoxMatVal5,SBoxMatVal6,SBoxMatVal7,SBoxMatVal8];
```

A.2. CÓDIGO FUENTE DEL ALGORITMO IDEA

A.2.1. ALGORITMO DE ENCRIPTACIÓN IDEACIP.M

```
function CiText = ideacip(PlText)

%IDEACIP Implementación de encriptación por algotirmo IDEA
%
%IDEACIP(PlText) permite encriptar un bloque de 64
%bits mediante el algoritmo IDEA

%Tabla de desplazamiento de la clave

lsh = [26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 ...
       42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 ...
       58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 ...
       74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 ...
       90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 ...
       106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121
...
       122 123 124 125 126 127 128 1 2 3 4 5 6 7 8 9 ...
       10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25];

%Generación de las claves
global KeyIDEA
KeyLeft1 = KeyIDEA(:,lsh);
KeyLeft2 = KeyLeft1(:,lsh);
KeyLeft3 = KeyLeft2(:,lsh);
KeyLeft4 = KeyLeft3(:,lsh);
KeyLeft5 = KeyLeft4(:,lsh);
KeyLeft6 = KeyLeft5(:,lsh);
```

```
%Claves para la ronda 1
Z11 = KeyIDEA(:,1:16); Z21 = KeyIDEA(:,17:32); Z31 = KeyIDEA(:,33:48);
Z41 = KeyIDEA(:,49:64); Z51 = KeyIDEA(:,65:80); Z61 = KeyIDEA(:,81:96);

%Claves para la ronda 2
Z12 = KeyIDEA(:,97:112); Z22 = KeyIDEA(:,113:128); Z32 =
KeyLeft1(:,1:16);
Z42 = KeyLeft1(:,17:32); Z52 = KeyLeft1(:,33:48); Z62 =
KeyLeft1(:,49:64);

%Claves para la ronda 3
Z13 = KeyLeft1(:,65:80); Z23 = KeyLeft1(:,81:96); Z33 =
KeyLeft1(:,97:112);
Z43 = KeyLeft1(:,113:128); Z53 = KeyLeft2(:,1:16); Z63 =
KeyLeft2(:,17:32);

%Claves para la ronda 4
Z14 = KeyLeft2(:,33:48); Z24 = KeyLeft2(:,49:64); Z34 =
KeyLeft2(:,65:80);
Z44 = KeyLeft2(:,81:96); Z54 = KeyLeft2(:,97:112); Z64 =
KeyLeft2(:,113:128);

%Claves para la ronda 5
Z15 = KeyLeft3(:,1:16); Z25 = KeyLeft3(:,17:32); Z35 = KeyLeft3(:,33:48);
Z45 = KeyLeft3(:,49:64); Z55 = KeyLeft3(:,65:80); Z65 =
KeyLeft3(:,81:96);

%Claves para la ronda 6
Z16 = KeyLeft3(:,97:112); Z26 = KeyLeft3(:,113:128); Z36 =
KeyLeft4(:,1:16);
Z46 = KeyLeft4(:,17:32); Z56 = KeyLeft4(:,33:48); Z66 =
KeyLeft4(:,49:64);

%Claves para la ronda 7
Z17 = KeyLeft4(:,65:80); Z27 = KeyLeft4(:,81:96); Z37 =
KeyLeft4(:,97:112);
Z47 = KeyLeft4(:,113:128); Z57 = KeyLeft5(:,1:16); Z67 =
KeyLeft5(:,17:32);

%Claves para la ronda 8
Z18 = KeyLeft5(:,33:48); Z28 = KeyLeft5(:,49:64); Z38 =
KeyLeft5(:,65:80);
Z48 = KeyLeft5(:,81:96); Z58 = KeyLeft5(:,97:112); Z68 =
KeyLeft5(:,113:128);

%Claves para la transformación final
Z19 = KeyLeft6(:,1:16); Z29 = KeyLeft6(:,17:32); Z39 = KeyLeft6(:,33:48);
Z49 = KeyLeft6(:,49:64);

%Ronda 1

%Conversión binario a decimal
X1 = PlText(:,1:16); X2 = PlText(:,17:32);
X3 = PlText(:,33:48); X4 = PlText(:,49:64);

X1Bin = strg(X1); X2Bin = strg(X2);
X3Bin = strg(X3); X4Bin = strg(X4);

X1Val = bin2dec(X1Bin); X2Val = bin2dec(X2Bin);
```

```
X3Val = bin2dec(X3Bin); X4Val = bin2dec(X4Bin);

%Adecuación de las claves
Z11Bin = strg(Z11); Z21Bin = strg(Z21); Z31Bin = strg(Z31);
Z41Bin = strg(Z41); Z51Bin = strg(Z51); Z61Bin = strg(Z61);

Z11Val = bin2dec(Z11Bin); Z21Val = bin2dec(Z21Bin); Z31Val =
bin2dec(Z31Bin);
Z41Val = bin2dec(Z41Bin); Z51Val = bin2dec(Z51Bin); Z61Val =
bin2dec(Z61Bin);

%Operaciones de la ronda 1
ModMul11 = mod((X1Val * Z11Val),(2^16 + 1)); %Paso 1
ModSum11 = mod((X2Val + Z21Val),2^16);       %Paso 2
ModSum21 = mod((X3Val + Z31Val),2^16);       %Paso 3
ModMul21 = mod((X4Val * Z41Val),(2^16 + 1)); %Paso 4

ModMul11Bin = dec2bin(ModMul11); ModSum11Bin = dec2bin(ModSum11);
ModSum21Bin = dec2bin(ModSum21); ModMul21Bin = dec2bin(ModMul21);

ModMul11Mat = numer16(ModMul11Bin); ModSum11Mat = numer16(ModSum11Bin);
ModSum21Mat = numer16(ModSum21Bin); ModMul21Mat = numer16(ModMul21Bin);

XOR11 = xor(ModMul11Mat, ModSum21Mat);    %Paso 5
XOR21 = xor(ModSum11Mat, ModMul21Mat);    %Paso 6

XOR11Bin = strg(XOR11); XOR11Val = bin2dec(XOR11Bin);
XOR21Bin = strg(XOR21); XOR21Val = bin2dec(XOR21Bin);

ModMul31 = mod((XOR11Val * Z51Val),(2^16 + 1)); %Paso 7
ModSum31 = mod((XOR21Val + ModMul31),2^16);     %Paso 8
ModMul41 = mod((ModSum31 * Z61Val),(2^16 + 1)); %Paso 9
ModSum41 = mod((ModMul31 + ModMul41),2^16);     %Paso 10

ModMul31Bin = dec2bin(ModMul31); ModSum31Bin = dec2bin(ModSum31);
ModSum41Bin = dec2bin(ModSum41); ModMul41Bin = dec2bin(ModMul41);

ModMul31Mat = numer16(ModMul31Bin); ModSum31Mat = numer16(ModSum31Bin);
ModSum41Mat = numer16(ModSum41Bin); ModMul41Mat = numer16(ModMul41Bin);

XOR31 = xor(ModMul11Mat,ModMul41Mat);     %Paso 11
XOR41 = xor(ModSum21Mat,ModMul41Mat);     %Paso 12
XOR51 = xor(ModSum11Mat,ModSum41Mat);     %Paso 13
XOR61 = xor(ModMul21Mat,ModSum41Mat);     %Paso 14

%Ronda 2

%Conversión binario a decimal
X1 = XOR31; X2 = XOR51;
X3 = XOR41; X4 = XOR61;

X1Bin = strg(X1); X2Bin = strg(X2);
X3Bin = strg(X3); X4Bin = strg(X4);

X1Val = bin2dec(X1Bin); X2Val = bin2dec(X2Bin);
X3Val = bin2dec(X3Bin); X4Val = bin2dec(X4Bin);

%Adecuación de las claves
Z12Bin = strg(Z12); Z22Bin = strg(Z22); Z32Bin = strg(Z32);
Z42Bin = strg(Z42); Z52Bin = strg(Z52); Z62Bin = strg(Z62);
```

```
Z12Val = bin2dec(Z12Bin); Z22Val = bin2dec(Z22Bin); Z32Val =
bin2dec(Z32Bin);
Z42Val = bin2dec(Z42Bin); Z52Val = bin2dec(Z52Bin); Z62Val =
bin2dec(Z62Bin);

%Operaciones de la ronda 2
ModMul12 = mod((X1Val * Z12Val),(2^16 + 1)); %Paso 1
ModSum12 = mod((X2Val + Z22Val),2^16);       %Paso 2
ModSum22 = mod((X3Val + Z32Val),2^16);       %Paso 3
ModMul22 = mod((X4Val * Z42Val),(2^16 + 1)); %Paso 4

ModMul12Bin = dec2bin(ModMul12); ModSum12Bin = dec2bin(ModSum12);
ModSum22Bin = dec2bin(ModSum22); ModMul22Bin = dec2bin(ModMul22);

ModMul12Mat = numer16(ModMul12Bin); ModSum12Mat = numer16(ModSum12Bin);
ModSum22Mat = numer16(ModSum22Bin); ModMul22Mat = numer16(ModMul22Bin);

XOR12 = xor(ModMul12Mat, ModSum22Mat);     %Paso 5
XOR22 = xor(ModSum12Mat, ModMul22Mat);     %Paso 6

XOR12Bin = strg(XOR12); XOR12Val = bin2dec(XOR12Bin);
XOR22Bin = strg(XOR22); XOR22Val = bin2dec(XOR22Bin);

ModMul32 = mod((XOR12Val * Z52Val),(2^16 + 1)); %Paso 7
ModSum32 = mod((XOR22Val + ModMul32),2^16);     %Paso 8
ModMul42 = mod((ModSum32 * Z62Val),(2^16 + 1)); %Paso 9
ModSum42 = mod((ModMul32 + ModMul42),2^16);     %Paso 10

ModMul32Bin = dec2bin(ModMul32); ModSum32Bin = dec2bin(ModSum32);
ModSum42Bin = dec2bin(ModSum42); ModMul42Bin = dec2bin(ModMul42);

ModMul32Mat = numer16(ModMul32Bin); ModSum32Mat = numer16(ModSum32Bin);
ModSum42Mat = numer16(ModSum42Bin); ModMul42Mat = numer16(ModMul42Bin);

XOR32 = xor(ModMul12Mat,ModMul42Mat);      %Paso 11
XOR42 = xor(ModSum22Mat,ModMul42Mat);      %Paso 12
XOR52 = xor(ModSum12Mat,ModSum42Mat);      %Paso 13
XOR62 = xor(ModMul22Mat,ModSum42Mat);      %Paso 14

%Ronda 3

%Conversión binario a decimal
X1 = XOR32; X2 = XOR52;
X3 = XOR42; X4 = XOR62;

X1Bin = strg(X1); X2Bin = strg(X2);
X3Bin = strg(X3); X4Bin = strg(X4);

X1Val = bin2dec(X1Bin); X2Val = bin2dec(X2Bin);
X3Val = bin2dec(X3Bin); X4Val = bin2dec(X4Bin);

%Adecuación de las claves
Z13Bin = strg(Z13); Z23Bin = strg(Z23); Z33Bin = strg(Z33);
Z43Bin = strg(Z43); Z53Bin = strg(Z53); Z63Bin = strg(Z63);

Z13Val = bin2dec(Z13Bin); Z23Val = bin2dec(Z23Bin); Z33Val =
bin2dec(Z33Bin);
Z43Val = bin2dec(Z43Bin); Z53Val = bin2dec(Z53Bin); Z63Val =
bin2dec(Z63Bin);
```

```
%Operaciones de la ronda 3
ModMul13 = mod((X1Val * Z13Val),(2^16 + 1)); %Paso 1
ModSum13 = mod((X2Val + Z23Val),2^16);       %Paso 2
ModSum23 = mod((X3Val + Z33Val),2^16);       %Paso 3
ModMul23 = mod((X4Val * Z43Val),(2^16 + 1)); %Paso 4

ModMul13Bin = dec2bin(ModMul13); ModSum13Bin = dec2bin(ModSum13);
ModSum23Bin = dec2bin(ModSum23); ModMul23Bin = dec2bin(ModMul23);

ModMul13Mat = numer16(ModMul13Bin); ModSum13Mat = numer16(ModSum13Bin);
ModSum23Mat = numer16(ModSum23Bin); ModMul23Mat = numer16(ModMul23Bin);

XOR13 = xor(ModMul13Mat, ModSum23Mat);    %Paso 5
XOR23 = xor(ModSum13Mat, ModMul23Mat);    %Paso 6

XOR13Bin = strg(XOR13); XOR13Val = bin2dec(XOR13Bin);
XOR23Bin = strg(XOR23); XOR23Val = bin2dec(XOR23Bin);

ModMul33 = mod((XOR13Val * Z53Val),(2^16 + 1)); %Paso 7
ModSum33 = mod((XOR23Val + ModMul33),2^16);     %Paso 8
ModMul43 = mod((ModSum33 * Z63Val),(2^16 + 1)); %Paso 9
ModSum43 = mod((ModMul33 + ModMul43),2^16);     %Paso 10

ModMul33Bin = dec2bin(ModMul33); ModSum33Bin = dec2bin(ModSum33);
ModSum43Bin = dec2bin(ModSum43); ModMul43Bin = dec2bin(ModMul43);

ModMul33Mat = numer16(ModMul33Bin); ModSum33Mat = numer16(ModSum33Bin);
ModSum43Mat = numer16(ModSum43Bin); ModMul43Mat = numer16(ModMul43Bin);

XOR33 = xor(ModMul13Mat,ModMul43Mat);     %Paso 11
XOR43 = xor(ModSum23Mat,ModMul43Mat);     %Paso 12
XOR53 = xor(ModSum13Mat,ModSum43Mat);     %Paso 13
XOR63 = xor(ModMul23Mat,ModSum43Mat);     %Paso 14

%Ronda 4

%Conversión binario a decimal
X1 = XOR33; X2 = XOR53;
X3 = XOR43; X4 = XOR63;

X1Bin = strg(X1); X2Bin = strg(X2);
X3Bin = strg(X3); X4Bin = strg(X4);

X1Val = bin2dec(X1Bin); X2Val = bin2dec(X2Bin);
X3Val = bin2dec(X3Bin); X4Val = bin2dec(X4Bin);

%Adecuación de las claves
Z14Bin = strg(Z14); Z24Bin = strg(Z24); Z34Bin = strg(Z34);
Z44Bin = strg(Z44); Z54Bin = strg(Z54); Z64Bin = strg(Z64);

Z14Val = bin2dec(Z14Bin); Z24Val = bin2dec(Z24Bin); Z34Val =
bin2dec(Z34Bin);
Z44Val = bin2dec(Z44Bin); Z54Val = bin2dec(Z54Bin); Z64Val =
bin2dec(Z64Bin);

%Operaciones de la ronda 4
ModMul14 = mod((X1Val * Z14Val),(2^16 + 1)); %Paso 1
ModSum14 = mod((X2Val + Z24Val),2^16);       %Paso 2
ModSum24 = mod((X3Val + Z34Val),2^16);       %Paso 3
```

```
ModMul24 = mod((X4Val * Z44Val),(2^16 + 1)); %Paso 4

ModMul14Bin = dec2bin(ModMul14); ModSum14Bin = dec2bin(ModSum14);
ModSum24Bin = dec2bin(ModSum24); ModMul24Bin = dec2bin(ModMul24);

ModMul14Mat = numer16(ModMul14Bin); ModSum14Mat = numer16(ModSum14Bin);
ModSum24Mat = numer16(ModSum24Bin); ModMul24Mat = numer16(ModMul24Bin);

XOR14 = xor(ModMul14Mat, ModSum24Mat);     %Paso 5
XOR24 = xor(ModSum14Mat, ModMul24Mat);     %Paso 6

XOR14Bin = strg(XOR14); XOR14Val = bin2dec(XOR14Bin);
XOR24Bin = strg(XOR24); XOR24Val = bin2dec(XOR24Bin);

ModMul34 = mod((XOR14Val * Z54Val),(2^16 + 1)); %Paso 7
ModSum34 = mod((XOR24Val + ModMul34),2^16);     %Paso 8
ModMul44 = mod((ModSum34 * Z64Val),(2^16 + 1)); %Paso 9
ModSum44 = mod((ModMul34 + ModMul44),2^16);     %Paso 10

ModMul34Bin = dec2bin(ModMul34); ModSum34Bin = dec2bin(ModSum34);
ModSum44Bin = dec2bin(ModSum44); ModMul44Bin = dec2bin(ModMul44);

ModMul34Mat = numer16(ModMul34Bin); ModSum34Mat = numer16(ModSum34Bin);
ModSum44Mat = numer16(ModSum44Bin); ModMul44Mat = numer16(ModMul44Bin);

XOR34 = xor(ModMul14Mat,ModMul44Mat);     %Paso 11
XOR44 = xor(ModSum24Mat,ModMul44Mat);     %Paso 12
XOR54 = xor(ModSum14Mat,ModSum44Mat);     %Paso 13
XOR64 = xor(ModMul24Mat,ModSum44Mat);     %Paso 14

%Transformación final
%Conversión binario a decimal
Y1F = XOR34; Y2F = XOR54;
Y3F = XOR44; Y4F = XOR64;

Y1FBin = strg(Y1F); Y2FBin = strg(Y2F);
Y3FBin = strg(Y3F); Y4FBin = strg(Y4F);

Y1FVal = bin2dec(Y1FBin); Y2FVal = bin2dec(Y2FBin);
Y3FVal = bin2dec(Y3FBin); Y4FVal = bin2dec(Y4FBin);

%Adecuación de las claves
Z1F = Z15; Z2F = Z25;
Z3F = Z35; Z4F = Z45;

Z1FBin = strg(Z1F); Z2FBin = strg(Z2F);
Z3FBin = strg(Z3F); Z4FBin = strg(Z4F);

Z1FVal = bin2dec(Z1FBin); Z2FVal = bin2dec(Z2FBin);
Z3FVal = bin2dec(Z3FBin); Z4FVal = bin2dec(Z4FBin);

%Operaciones de la transformación final
ModY1 = mod((Y1FVal * Z1FVal),(2^16 + 1));     %Paso 1
ModY2 = mod((Y2FVal + Z2FVal),2^16);           %Paso 2
ModY3 = mod((Y3FVal + Z3FVal),2^16);           %Paso 3
ModY4 = mod((Y4FVal * Z4FVal),(2^16 + 1));     %Paso 4

ModY1Bin = dec2bin(ModY1); ModY2Bin = dec2bin(ModY2);
ModY3Bin = dec2bin(ModY3); ModY4Bin = dec2bin(ModY4);
```

```
Y1 = numer16(ModY1Bin); Y2 = numer16(ModY2Bin);
Y3 = numer16(ModY3Bin); Y4 = numer16(ModY4Bin);

CiText = [Y1,Y2,Y3,Y4];
```

A.2.2. ALGORITMO DE DESENCRIPTACIÓN IDEADEC.M

```
function PlText = ideadec(CiText)

%Implementación de desencriptación por algoritmo IDEA
%
%IDEADEC(CiText) permite desencriptar un bloque de 64
%bits mediante el algoritmo IDEA

%Tabla de desplazamiento de la clave

lsh = [26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 ...
       42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 ...
       58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 ...
       74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 ...
       90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 ...
       106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121
...
       122 123 124 125 126 127 128 1 2 3 4 5 6 7 8 9 ...
       10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25];

%Generación de las claves
global KeyIDEA
KeyLeft1 = KeyIDEA(:,lsh);
KeyLeft2 = KeyLeft1(:,lsh);
KeyLeft3 = KeyLeft2(:,lsh);
KeyLeft4 = KeyLeft3(:,lsh);
KeyLeft5 = KeyLeft4(:,lsh);
KeyLeft6 = KeyLeft5(:,lsh);

%Claves para la ronda 1
Z11 = KeyIDEA(:,1:16); Z21 = KeyIDEA(:,17:32); Z31 = KeyIDEA(:,33:48);
Z41 = KeyIDEA(:,49:64); Z51 = KeyIDEA(:,65:80); Z61 = KeyIDEA(:,81:96);

%Claves para la ronda 2
Z12 = KeyIDEA(:,97:112); Z22 = KeyIDEA(:,113:128); Z32 =
KeyLeft1(:,1:16);
Z42 = KeyLeft1(:,17:32); Z52 = KeyLeft1(:,33:48); Z62 =
KeyLeft1(:,49:64);

%Claves para la ronda 3
Z13 = KeyLeft1(:,65:80); Z23 = KeyLeft1(:,81:96); Z33 =
KeyLeft1(:,97:112);
Z43 = KeyLeft1(:,113:128); Z53 = KeyLeft2(:,1:16); Z63 =
KeyLeft2(:,17:32);

%Claves para la ronda 4
Z14 = KeyLeft2(:,33:48); Z24 = KeyLeft2(:,49:64); Z34 =
KeyLeft2(:,65:80);
Z44 = KeyLeft2(:,81:96); Z54 = KeyLeft2(:,97:112); Z64 =
KeyLeft2(:,113:128);

%Claves para la ronda 5
Z15 = KeyLeft3(:,1:16); Z25 = KeyLeft3(:,17:32); Z35 = KeyLeft3(:,33:48);
```

```
Z45 = KeyLeft3(:,49:64); Z55 = KeyLeft3(:,65:80); Z65 =
KeyLeft3(:,81:96);

%Claves para la ronda 6
Z16 = KeyLeft3(:,97:112); Z26 = KeyLeft3(:,113:128); Z36 =
KeyLeft4(:,1:16);
Z46 = KeyLeft4(:,17:32); Z56 = KeyLeft4(:,33:48); Z66 =
KeyLeft4(:,49:64);

%Claves para la ronda 7
Z17 = KeyLeft4(:,65:80); Z27 = KeyLeft4(:,81:96); Z37 =
KeyLeft4(:,97:112);
Z47 = KeyLeft4(:,113:128); Z57 = KeyLeft5(:,1:16); Z67 =
KeyLeft5(:,17:32);

%Claves para la ronda 8
Z18 = KeyLeft5(:,33:48); Z28 = KeyLeft5(:,49:64); Z38 =
KeyLeft5(:,65:80);
Z48 = KeyLeft5(:,81:96); Z58 = KeyLeft5(:,97:112); Z68 =
KeyLeft5(:,113:128);

%Claves para la transformación final
Z19 = KeyLeft6(:,1:16); Z29 = KeyLeft6(:,17:32); Z39 = KeyLeft6(:,33:48);
Z49 = KeyLeft6(:,49:64);

%Ronda 1

%Conversión binario a decimal
X1 = CiText(:,1:16); X2 = CiText(:,17:32);
X3 = CiText(:,33:48); X4 = CiText(:,49:64);

X1Bin = strg(X1); X2Bin = strg(X2);
X3Bin = strg(X3); X4Bin = strg(X4);

X1Val = bin2dec(X1Bin); X2Val = bin2dec(X2Bin);
X3Val = bin2dec(X3Bin); X4Val = bin2dec(X4Bin);

%Adecuación de las claves
Z11Bin = strg(Z15); Z21Bin = strg(Z25); Z31Bin = strg(Z35);
Z41Bin = strg(Z45); Z51Bin = strg(Z54); Z61Bin = strg(Z64);

Z11Num = bin2dec(Z11Bin); Z21Num = bin2dec(Z21Bin);
Z31Num = bin2dec(Z31Bin); Z41Num = bin2dec(Z41Bin);

[G,C,D1] = gcd((2^16 + 1),Z11Num); [G,C,D2] = gcd((2^16 + 1),Z41Num);

if D1 < 0, D1 = 2^16 + 1 + D1; elseif D1 == 0, D1 = 0; end
if D2 < 0, D2 = 2^16 + 1 + D2; elseif D2 == 0, D2 = 0; end

Z11Val = D1; Z21Val = mod(-Z21Num,2^16); Z51Val = bin2dec(Z51Bin);
Z41Val = D2; Z31Val = mod(-Z31Num,2^16); Z61Val = bin2dec(Z61Bin);

%Operaciones de la ronda 1
ModMul11 = mod((X1Val * Z11Val),(2^16 + 1)); %Paso 1
ModSum11 = mod((X2Val + Z21Val),2^16);       %Paso 2
ModSum21 = mod((X3Val + Z31Val),2^16);       %Paso 3
ModMul21 = mod((X4Val * Z41Val),(2^16 + 1)); %Paso 4

ModMul11Bin = dec2bin(ModMul11); ModSum11Bin = dec2bin(ModSum11);
ModSum21Bin = dec2bin(ModSum21); ModMul21Bin = dec2bin(ModMul21);
```

```
ModMul11Mat = numer16(ModMul11Bin); ModSum11Mat = numer16(ModSum11Bin);
ModSum21Mat = numer16(ModSum21Bin); ModMul21Mat = numer16(ModMul21Bin);

XOR11 = xor(ModMul11Mat, ModSum21Mat);     %Paso 5
XOR21 = xor(ModSum11Mat, ModMul21Mat);     %Paso 6

XOR11Bin = strg(XOR11); XOR11Val = bin2dec(XOR11Bin);
XOR21Bin = strg(XOR21); XOR21Val = bin2dec(XOR21Bin);

ModMul31 = mod((XOR11Val * Z51Val),(2^16 + 1)); %Paso 7
ModSum31 = mod((XOR21Val + ModMul31),2^16);     %Paso 8
ModMul41 = mod((ModSum31 * Z61Val),(2^16 + 1)); %Paso 9
ModSum41 = mod((ModMul31 + ModMul41),2^16);     %Paso 10

ModMul31Bin = dec2bin(ModMul31); ModSum31Bin = dec2bin(ModSum31);
ModSum41Bin = dec2bin(ModSum41); ModMul41Bin = dec2bin(ModMul41);

ModMul31Mat = numer16(ModMul31Bin); ModSum31Mat = numer16(ModSum31Bin);
ModSum41Mat = numer16(ModSum41Bin); ModMul41Mat = numer16(ModMul41Bin);

XOR31 = xor(ModMul11Mat,ModMul41Mat);      %Paso 11
XOR41 = xor(ModSum21Mat,ModMul41Mat);      %Paso 12
XOR51 = xor(ModSum11Mat,ModSum41Mat);      %Paso 13
XOR61 = xor(ModMul21Mat,ModSum41Mat);      %Paso 14

%Ronda 2

%Conversión binario a decimal
X1 = XOR31; X2 = XOR51;
X3 = XOR41; X4 = XOR61;

X1Bin = strg(X1); X2Bin = strg(X2);
X3Bin = strg(X3); X4Bin = strg(X4);

X1Val = bin2dec(X1Bin); X2Val = bin2dec(X2Bin);
X3Val = bin2dec(X3Bin); X4Val = bin2dec(X4Bin);

%Adecuación de las claves
Z12Bin = strg(Z14); Z22Bin = strg(Z24); Z32Bin = strg(Z34);
Z42Bin = strg(Z44); Z52Bin = strg(Z53); Z62Bin = strg(Z63);

Z12Num = bin2dec(Z12Bin); Z22Num = bin2dec(Z22Bin);
Z32Num = bin2dec(Z32Bin); Z42Num = bin2dec(Z42Bin);

[G,C,D3] = gcd((2^16 + 1),Z12Num); [G,C,D4] = gcd((2^16 + 1),Z42Num);

if D3 < 0, D3 = 2^16 + 1 + D3; elseif D3 == 0, D3 = 0; end
if D4 < 0, D4 = 2^16 + 1 + D4; elseif D4 == 0, D4 = 0; end

Z12Val = D3; Z22Val = mod(-Z22Num,2^16); Z52Val = bin2dec(Z52Bin);
Z42Val = D4; Z32Val = mod(-Z32Num,2^16); Z62Val = bin2dec(Z62Bin);

%Operaciones de la ronda 2
ModMul12 = mod((X1Val * Z12Val),(2^16 + 1)); %Paso 1
ModSum12 = mod((X2Val + Z22Val),2^16);       %Paso 2
ModSum22 = mod((X3Val + Z32Val),2^16);       %Paso 3
ModMul22 = mod((X4Val * Z42Val),(2^16 + 1)); %Paso 4

ModMul12Bin = dec2bin(ModMul12); ModSum12Bin = dec2bin(ModSum12);
```

```
ModMul13Bin = dec2bin(ModMul13); ModSum13Bin = dec2bin(ModSum13);
ModSum23Bin = dec2bin(ModSum23); ModMul23Bin = dec2bin(ModMul23);

ModMul13Mat = numer16(ModMul13Bin); ModSum13Mat = numer16(ModSum13Bin);
ModSum23Mat = numer16(ModSum23Bin); ModMul23Mat = numer16(ModMul23Bin);

XOR13 = xor(ModMul13Mat, ModSum23Mat);     %Paso 5
XOR23 = xor(ModSum13Mat, ModMul23Mat);     %Paso 6

XOR13Bin = strg(XOR13); XOR13Val = bin2dec(XOR13Bin);
XOR23Bin = strg(XOR23); XOR23Val = bin2dec(XOR23Bin);

ModMul33 = mod((XOR13Val * Z53Val),(2^16 + 1)); %Paso 7
ModSum33 = mod((XOR23Val + ModMul33),2^16);     %Paso 8
ModMul43 = mod((ModSum33 * Z63Val),(2^16 + 1)); %Paso 9
ModSum43 = mod((ModMul33 + ModMul43),2^16);     %Paso 10

ModMul33Bin = dec2bin(ModMul33); ModSum33Bin = dec2bin(ModSum33);
ModSum43Bin = dec2bin(ModSum43); ModMul43Bin = dec2bin(ModMul43);

ModMul33Mat = numer16(ModMul33Bin); ModSum33Mat = numer16(ModSum33Bin);
ModSum43Mat = numer16(ModSum43Bin); ModMul43Mat = numer16(ModMul43Bin);

XOR33 = xor(ModMul13Mat,ModMul43Mat);      %Paso 11
XOR43 = xor(ModSum23Mat,ModMul43Mat);      %Paso 12
XOR53 = xor(ModSum13Mat,ModSum43Mat);      %Paso 13
XOR63 = xor(ModMul23Mat,ModSum43Mat);      %Paso 14

%Ronda 4

%Conversión binario a decimal
X1 = XOR33; X2 = XOR53;
X3 = XOR43; X4 = XOR63;

X1Bin = strg(X1); X2Bin = strg(X2);
X3Bin = strg(X3); X4Bin = strg(X4);

X1Val = bin2dec(X1Bin); X2Val = bin2dec(X2Bin);
X3Val = bin2dec(X3Bin); X4Val = bin2dec(X4Bin);

%Adecuación de las claves
Z14Bin = strg(Z12); Z24Bin = strg(Z22); Z34Bin = strg(Z32);
Z44Bin = strg(Z42); Z54Bin = strg(Z51); Z64Bin = strg(Z61);

Z14Num = bin2dec(Z14Bin); Z24Num = bin2dec(Z24Bin);
Z34Num = bin2dec(Z34Bin); Z44Num = bin2dec(Z44Bin);

[G,C,D7] = gcd((2^16 + 1),Z14Num); [G,C,D8] = gcd((2^16 + 1),Z44Num);

if D7 < 0, D7 = 2^16 + 1 + D7; elseif D7 == 0, D7 = 0; end
if D8 < 0, D8 = 2^16 + 1 + D8; elseif D8 == 0, D8 = 0; end

Z14Val = D7; Z24Val = mod(-Z24Num,2^16); Z54Val = bin2dec(Z54Bin);
Z44Val = D8; Z34Val = mod(-Z34Num,2^16); Z64Val = bin2dec(Z64Bin);

%Operaciones de la ronda 4
ModMul14 = mod((X1Val * Z14Val),(2^16 + 1)); %Paso 1
ModSum14 = mod((X2Val + Z24Val),2^16);       %Paso 2
ModSum24 = mod((X3Val + Z34Val),2^16);       %Paso 3
ModMul24 = mod((X4Val * Z44Val),(2^16 + 1)); %Paso 4
```

```
ModMul14Bin = dec2bin(ModMul14); ModSum14Bin = dec2bin(ModSum14);
ModSum24Bin = dec2bin(ModSum24); ModMul24Bin = dec2bin(ModMul24);

ModMul14Mat = numer16(ModMul14Bin); ModSum14Mat = numer16(ModSum14Bin);
ModSum24Mat = numer16(ModSum24Bin); ModMul24Mat = numer16(ModMul24Bin);

XOR14 = xor(ModMul14Mat, ModSum24Mat);     %Paso 5
XOR24 = xor(ModSum14Mat, ModMul24Mat);     %Paso 6

XOR14Bin = strg(XOR14); XOR14Val = bin2dec(XOR14Bin);
XOR24Bin = strg(XOR24); XOR24Val = bin2dec(XOR24Bin);

ModMul34 = mod((XOR14Val * Z54Val),(2^16 + 1)); %Paso 7
ModSum34 = mod((XOR24Val + ModMul34),2^16);     %Paso 8
ModMul44 = mod((ModSum34 * Z64Val),(2^16 + 1)); %Paso 9
ModSum44 = mod((ModMul34 + ModMul44),2^16);     %Paso 10

ModMul34Bin = dec2bin(ModMul34); ModSum34Bin = dec2bin(ModSum34);
ModSum44Bin = dec2bin(ModSum44); ModMul44Bin = dec2bin(ModMul44);

ModMul34Mat = numer16(ModMul34Bin); ModSum34Mat = numer16(ModSum34Bin);
ModSum44Mat = numer16(ModSum44Bin); ModMul44Mat = numer16(ModMul44Bin);

XOR34 = xor(ModMul14Mat,ModMul44Mat);      %Paso 11
XOR44 = xor(ModSum24Mat,ModMul44Mat);      %Paso 12
XOR54 = xor(ModSum14Mat,ModSum44Mat);      %Paso 13
XOR64 = xor(ModMul24Mat,ModSum44Mat);      %Paso 14

%Transformación final
%Conversión binario a decimal
Y1F = XOR34; Y2F = XOR54;
Y3F = XOR44; Y4F = XOR64;

Y1FBin = strg(Y1F); Y2FBin = strg(Y2F);
Y3FBin = strg(Y3F); Y4FBin = strg(Y4F);

Y1FVal = bin2dec(Y1FBin); Y2FVal = bin2dec(Y2FBin);
Y3FVal = bin2dec(Y3FBin); Y4FVal = bin2dec(Y4FBin);

%Adecuación de las claves
Z1F = Z11; Z2F = Z21;
Z3F = Z31; Z4F = Z41;

Z1FBin = strg(Z1F); Z2FBin = strg(Z2F);
Z3FBin = strg(Z3F); Z4FBin = strg(Z4F);

Z1FNum = bin2dec(Z1FBin); Z2FNum = bin2dec(Z2FBin);
Z3FNum = bin2dec(Z3FBin); Z4FNum = bin2dec(Z4FBin);

[G,C,D3] = gcd((2^16 + 1),Z1FNum); [G,C,D4] = gcd((2^16 + 1),Z4FNum);

if D3 < 0, D3 = 2^16 + 1 + D3; elseif D3 == 0, D3 = 0; end
if D4 < 0, D4 = 2^16 + 1 + D4; elseif D4 == 0, D4 = 0; end

Z1FVal = D3; Z2FVal = mod(-Z2FNum,2^16);
Z4FVal = D4; Z3FVal = mod(-Z3FNum,2^16);

%Operaciones de la transformación final
ModY1 = mod((Y1FVal * Z1FVal),(2^16 + 1));     %Paso 1
```

```
ModY2 = mod((Y2FVal + Z2FVal),2^16);          %Paso 2
ModY3 = mod((Y3FVal + Z3FVal),2^16);          %Paso 3
ModY4 = mod((Y4FVal * Z4FVal),(2^16 + 1));    %Paso 4

ModY1Bin = dec2bin(ModY1); ModY2Bin = dec2bin(ModY2);
ModY3Bin = dec2bin(ModY3); ModY4Bin = dec2bin(ModY4);

Y1 = numer16(ModY1Bin); Y2 = numer16(ModY2Bin);
Y3 = numer16(ModY3Bin); Y4 = numer16(ModY4Bin);

PlText = [Y1,Y2,Y3,Y4];
```

A.3. CÓDIGO FUENTE DEL ALGORITMO A5/1

A.3.1. ALGORITMO DE ENCRIPTACIÓN A5CIP.M

```
function CiText = a5cip(PlText)

%A5CIP Implementación del algoritmo A5
%
%A5CIP(PlText) permite encriptar una corriente
%de bits mediante el algoritmo A5

%Vectores para el registro de desplazamiento
lsf1 = [19 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18];
lsf2 = [22 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21];
lsf3 = [23 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22];

global KeyA5

%Inicializar registros
R1 = KeyA5(:,1:19);
R2 = KeyA5(:,20:41);
R3 = KeyA5(:,42:64);

%Algoritmo de encriptación
for i = 1:length(PlText)
   TAP1 = xor(xor(xor(R1(14),R1(17)),R1(18)),R1(19));
   TAP2 = xor(R2(21),R2(22));
   TAP3 = xor(xor(xor(R3(8),R3(21)),R3(22)),R3(23));
   R1 = R1(:,lsf1);
   R2 = R2(:,lsf2);
   R3 = R3(:,lsf3);
   R1(1) = TAP1;
   R2(1) = TAP2;
   R3(1) = TAP3;
   XOR1 = xor(xor(R1(19),R2(22)),R3(23));
   CiText(i) = xor(XOR1,PlText(i));
end
```

A.3.2. ALGORITMO DE DESENCRIPTACIÓN A5DEC.M

```
function PlText = a5dec(CiText)

%A5DEC Implementación del algoritmo A5
%
%A5DEC(CiText) permite desencriptar una corriente
%de bits mediante el algoritmo A5
```

```
%Vectores para los registros de desplazamiento
lsf1 = [19 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18];
lsf2 = [22 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21];
lsf3 = [23 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22];

global KeyA5

%Inicializar registros
R1 = KeyA5(:,1:19);
R2 = KeyA5(:,20:41);
R3 = KeyA5(:,42:64);

%Algoritmo de desencriptación
for i = 1:length(CiText)
   TAP1 = xor(xor(xor(R1(14),R1(17)),R1(18)),R1(19));
   TAP2 = xor(R2(21),R2(22));
   TAP3 = xor(xor(xor(R3(8),R3(21)),R3(22)),R3(23));
   R1 = R1(:,lsf1);
   R2 = R2(:,lsf2);
   R3 = R3(:,lsf3);
   R1(1) = TAP1;
   R2(1) = TAP2;
   R3(1) = TAP3;
   XOR1 = xor(xor(R1(19),R2(22)),R3(23));
   PlText(i) = xor(XOR1,CiText(i));
end
```

A.4. CÓDIGO FUENTE DEL ALGORITMO RC4

A.4.1. ALGORITMO DE ENCRIPTACIÓN RC4CIP.M

```
function CiText = rc4cip(PlText)

%RC4CIP Implementación del algoritmo RC4
%
%RC4CIP(PlText) permite encriptar una corriente de bits
%mediante el algoritmo RC4


[k,l] = size(PlText);
global KeyRC4

%Inicializar caja S
K = KeyRC4;
S = 0:255;
SAx = ones(1,256);
j = 0;

for i = 1:256
      j = mod((j + S(i) + K(i)),256) + 1;
      SAx(i) = S(i);
      SAx(j) = S(j);
      S(j) = SAx(i);
      S(i) = SAx(j);
end

%Algoritmo de encriptación
i = 0; j = 0;
for m = 1:(1/8)
```

```
    i = mod((i + 1),256) + 1;
    j = mod((j + S(i)),256) + 1;
    SAx(i) = S(i);
    SAx(j) = S(j);
    S(j) = SAx(i);
    S(i) = SAx(j);
    t = mod((S(i) + S(j)),256) + 1;
    KS = S(t);
    KSBin = dec2bin(KS);
    KSMat = numer8(KSBin);
    KeyStream(8*m - 7:8*m) = KSMat;
end

CiText = xor(KeyStream,PlText);
```

A.4.2. ALGORITMO DE DESENCRIPTACIÓN RC4DEC.M

```
function PlText = rc4dec(CiText)

%RC4DEC Implementación del algoritmo RC4
%
%RC4DEC(CiText) permite desencriptar una corriente de bits
%mediante el algoritmo RC4

[k,l] = size(CiText);
global KeyRC4


%Inicializar caja S
K = KeyRC4;
j = 0;
S = 0:255;
SAx = ones(1,256);
for i = 1:256
      j = mod((j + S(i) + K(i)),256) + 1;
      SAx(i) = S(i);
      SAx(j) = S(j);
      S(j) = SAx(i);
      S(i) = SAx(j);
end

%Algoritmo de desencriptación
i = 0; j = 0;
for m = 1:(l/8)
    i = mod((i+1),256) + 1;
    j = mod((j+S(i)),256) + 1;
    SAx(i) = S(i);
    SAx(j) = S(j);
    S(j) = SAx(i);
    S(i) = SAx(j);
    t = mod((S(i) + S(j)),256) + 1;
    KS = S(t);
    KSBin = dec2bin(KS);
    KSMat = numer8(KSBin);
    KeyStream(8*m - 7:8*m) = KSMat;
end

PlText = xor(KeyStream, CiText);
```

ANEXO B

"REAL TIME CRYPTANALYSIS OF A5/1 ON A PC"

El *paper* que se reproduce a continuación presenta formas de romper el algoritmo A5/1 que se implementa en este trabajo. Los puntos más relevantes de este *paper* son los siguientes:

- El ataque descrito no es de fuerza bruta, utiliza un ataque de cumpleaños modificado, por lo que el número de operaciones es mucho menor (2^{45}) que lo que sería un ataque de fuerza bruta (2^{64}).
- El costo de los recursos empleados es mínimo, en comparación con métodos empleados para otros algoritmos. Un PC equipado con un procesador Pentium III, 128 MB de RAM, y 4 discos duros de 73 GB costaría aproximadamente USD 3000. Como se puede ver, la NSA (National Security Agency, la agencia estadounidense que se encarga de la criptografía y criptoanálisis de todos los algoritmos creados), puede crear millones de ataques en este estilo, ya que su presupuesto anual es de USD 21 billones.
- La facilidad de implementación con ataques de sólo texto cifrado, junto con el enfoque hacia el sistema de comunicaciones móviles celulares GSM, hacen que el proceso de romper el algoritmo A5/1 sea fácil en otros entornos.

El *paper* se reproduce en su totalidad, tal y como fue presentado en la conferencia FES 2000.

Real Time Cryptanalysis of A5/1 on a PC

Alex Biryukov * Adi Shamir ** David Wagner ***

Abstract. A5/1 is the strong version of the encryption algorithm used by about 130 million GSM customers in Europe to protect the over-the-air privacy of their cellular voice and data communication. The best published attacks against it require between 2^{40} and 2^{45} steps. This level of security makes it vulnerable to hardware-based attacks by large organizations, but not to software-based attacks on multiple targets by hackers.

In this paper we describe new attacks on A5/1, which are based on subtle flaws in the tap structure of the registers, their noninvertible clocking mechanism, and their frequent resets. After a 2^{48} parallelizable data preparation stage (which has to be carried out only once), the actual attacks can be carried out in real time on a single PC.

The first attack requires the output of the A5/1 algorithm during the first two minutes of the conversation, and computes the key in about one second. The second attack requires the output of the A5/1 algorithm during about two seconds of the conversation, and computes the key in several minutes. The two attacks are related, but use different types of time-memory tradeoffs. The attacks were verified with actual implementations, except for the preprocessing stage which was extensively sampled rather than completely executed.

REMARK: We based our attack on the version of the algorithm which was derived by reverse engineering an actual GSM telephone and published at `http://www.scard.org`. We would like to thank the GSM organization for graciously confirming to us the correctness of this unofficial description. In addition, we would like to stress that this paper considers the narrow issue of the cryptographic strength of A5/1, and not the broader issue of the practical security of fielded GSM systems, about which we make no claims.

1 Introduction

The over-the-air privacy of GSM telephone conversations is protected by the A5 stream cipher. This algorithm has two main variants: The stronger A5/1 version is used by about 130 million customers in Europe, while the weaker A5/2 version is used by another 100 million customers in other markets. The approximate design of A5/1 was leaked in 1994, and the exact design of both A5/1 and A5/2 was reverse engineered by Briceno from an actual GSM telephone in 1999 (see [3]).

* Computer Science department, The Weizmann Institute, Rehovot 76100, Israel.
** Computer Science department, The Weizmann Institute, Rehovot 76100, Israel.
*** Computer Science department, University of California, Berkeley CA 94720, USA.

In this paper we develop two new cryptanalytic attacks on A5/1, in which a single PC can extract the conversation key in real time from a small amount of generated output. The attacks are related, but each one of them optimizes a different parameter: The first attack (called **the biased birthday attack**) requires two minutes of data and one second of processing time, whereas the second attack (called the **the random subgraph attack**) requires two seconds of data and several minutes of processing time. There are many possible choices of tradeoff parameters in these attacks, and three of them are summarized in Table 1.

Attack Type	Preprocessing steps	Available data	Number of 73GB disks	Attack time
Biased Birthday attack (1)	2^{42}	2 minutes	4	1 second
Biased Birthday attack (2)	2^{48}	2 minutes	2	1 second
Random Subgraph attack	2^{48}	2 seconds	4	minutes

Table 1. Three possible tradeoff points in the attacks on A5/1.

Many of the ideas in these two new attacks are applicable to other stream ciphers as well, and define new quantifiable measures of security.

The paper is organized in the following way: Section 2 contains a full description of the A5/1 algorithm. Previous attacks on A5/1 are surveyed in Section 3, and an informal description of the new attacks is contained in Section 4. Finally, Section 5 contains various implementation details and an analysis of the expected success rate of the attacks, based on large scale sampling with actual implementations.

2 Description of the A5/1 stream cipher

A GSM conversation is sent as a sequence of frames every 4.6 millisecond. Each frame contains 114 bits representing the digitized A to B communication, and 114 bits representing the digitized B to A communication. Each conversation can be encrypted by a new session key K. For each frame, K is mixed with a publicly known frame counter F_n, and the result serves as the initial state of a generator which produces 228 pseudo random bits. These bits are XOR'ed by the two parties with the 114+114 bits of the plaintext to produce the 114+114 bits of the ciphertext.

A5/1 is built from three short linear feedback shift registers (LFSR) of lengths 19, 22, and 23 bits, which are denoted by $R1, R2$ and $R3$ respectively. The rightmost bit in each register is labelled as bit zero. The taps of $R1$ are at bit positions 13,16,17,18; the taps of $R2$ are at bit positions 20,21; and the taps of $R3$ are at bit positions 7, 20,21,22 (see Figure 1). When a register is clocked,

its taps are XORed together, and the result is stored in the rightmost bit of the left-shifted register. The three registers are maximal length LFSR's with periods $2^{19}-1$, $2^{22}-1$, and $2^{23}-1$, respectively. They are clocked in a stop/go fashion using the following majority rule: Each register has a single "clocking" tap (bit 8 for $R1$, bit 10 for $R2$, and bit 10 for for $R3$); each clock cycle, the majority function of the clocking taps is calculated and only those registers whose clocking taps agree with the majority bit are actually clocked. Note that at each step either two or three registers are clocked, and that each register moves with probability 3/4 and stops with probability 1/4.

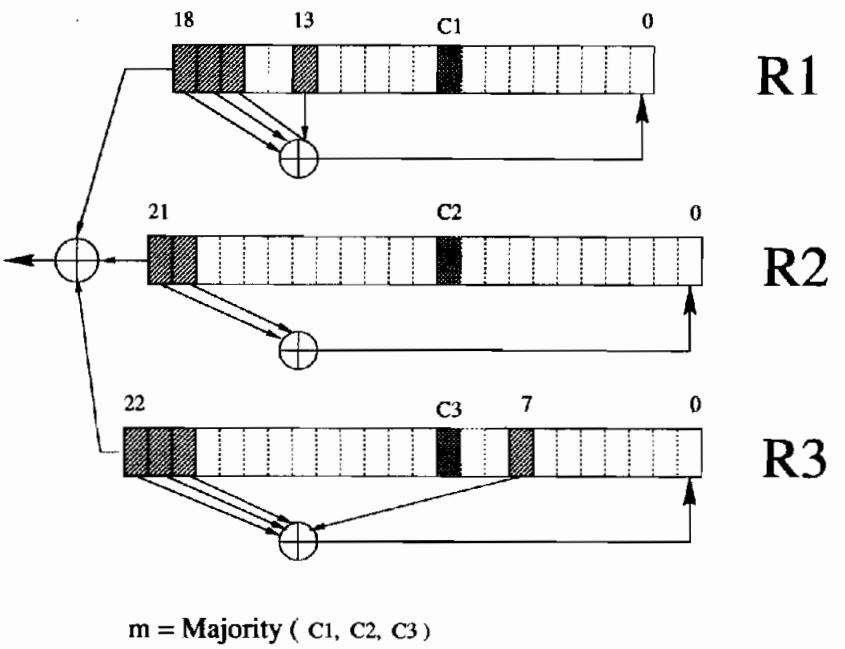


Fig. 1. The A5/1 stream cipher.

The process of generating pseudo random bits from the session key K and the frame counter F_n is carried out in four steps:

- The three registers are zeroed, and then clocked for 64 cycles (ignoring the stop/go clock control). During this period each bit of K (from lsb to msb) is XOR'ed in parallel into the lsb's of the three registers.
- The three registers are clocked for 22 additional cycles (ignoring the stop/go clock control). During this period the successive bits of F_n (from lsb to msb) are again XOR'ed in parallel into the lsb's of the three registers. The contents of the three registers at the end of this step is called the **initial state** of the

frame.
- The three registers are clocked for 100 additional clock cycles with the stop/go clock control but without producing any outputs.
- The three registers are clocked for 228 additional clock cycles with the stop/go clock control in order to produce the 228 output bits. At each clock cycle, one output bit is produced as the XOR of the msb's of the three registers.

3 Previous attacks

The attacker is assumed to know some pseudo random bits generated by A5/1 in some of the frames. This is the standard assumption in the cryptanalysis of stream ciphers, and we do not consider in this paper the crucial issue of how one can obtain these bits in fielded GSM systems. For the sake of simplicity, we assume that the attacker has complete knowledge of the outputs of the A5/1 algorithm during some initial period of the conversation, and his goal is to find the key in order to decrypt the remaining part of the conversation. Since GSM telephones send a new frame every 4.6 milliseconds, each second of the conversation contains about 2^8 frames.

At the rump session of Crypto 99, Ian Goldberg and David Wagner announced an attack on A5/2 which requires very few pseudo random bits and just $O(2^{16})$ steps. This demonstrated that the "export version" A5/2 is totally insecure.

The security of the A5/1 encryption algorithm was analyzed in several papers. Some of them are based on the early imprecise description of this algorithm, and thus their details have to be slightly modified. The known attacks can be summarized in the following way:

- Briceno[3] found out that in all the deployed versions of the A5/1 algorithm, the 10 least significant of the 64 key bits were always set to zero. The complexity of exhaustive search is thus reduced to $O(2^{54})$. [4]
- Anderson and Roe[1] proposed an attack based on guessing the 41 bits in the shorter R_1 and R_2 registers, and deriving the 23 bits of the longer R_3 register from the output. However, they occasionally have to guess additional bits to determine the majority-based clocking sequence, and thus the total complexity of the attack is about $O(2^{45})$. Assuming that a standard PC can test ten million guesses per second, this attack needs more than a month to find one key.
- Golic[4] described an improved attack which requires $O(2^{40})$ steps. However, each operation in this attack is much more complicated, since it is based on the solution of a system of linear equations. In practice, this algorithm is not likely to be faster than the previous attack on a PC.

[4] Our new attack is not based on this assumption, and is thus applicable to A5/1 implementations with full 64 bit keys. It is an interesting open problem whether we can speed it up by assuming that 10 key bits are zero.

– Golic[4] describes a general time-memory tradeoff attack on stream ciphers (which was independently discovered by Babbage [2] two years earlier), and concludes that it is possible to find the A5/1 key in 2^{22} probes into random locations in a precomputed table with 2^{42} 128 bit entries. Since such a table requires a 64 terabyte hard disk, the space requirement is unrealistic. Alternatively, it is possible to reduce the space requirement to 862 gigabytes, but then the number of probes increases to $O(2^{28})$. Since random access to the fastest commercially available PC disks requires about 6 milliseconds, the total probing time is almost three weeks. In addition, this tradeoff point can only be used to attack GSM phone conversations which last more than 3 hours, which again makes it unrealistic.

4 Informal Description of the New Attacks

We start with an executive summary of the key ideas of the two attacks. More technical descriptions of the various steps will be provided in the next section.

Key idea 1: Use the Golic time-memory tradeoff. The starting point for the new attacks is the time-memory tradeoff described in Golic[3], which is applicable to any cryptosystem with a relatively small number of internal states. A5/1 has this weakness, since it has $n = 2^{64}$ states defined by the 19+22+23 = 64 bits in its three shift registers. The basic idea of the Golic time-memory tradeoff is to keep a large set A of precomputed states on a hard disk, and to consider the large set B of states through which the algorithm progresses during the actual generation of output bits. Any intersection between A and B will enable us to identify an actual state of the algorithm from stored information.

Key idea 2: Identify states by prefixes of their output sequences. Each state defines an infinite sequence of output bits produced when we start clocking the algorithm from that state. In the other direction, states are usually uniquely defined by the first $log(n)$ bits in their output sequences, and thus we can look for equality between unknown states by comparing such prefixes of their output sequences. During precomputation, pick a subset A of states, compute their output prefixes, and store the (prefix, state) pairs sorted into increasing prefix values. Given actual outputs of the A5/1 algorithm, extract all their (partially overlapping) prefixes, and define B as the set of their corresponding (unknown) states. Searching for common states in A and B can be efficiently done by probing the sorted data A on the hard disk with prefix queries from B.

Key idea 3: A5/1 can be efficiently inverted. As observed by Golic, the state transition function of A5/1 is not uniquely invertible: The majority clock control rule implies that up to 4 states can converge to a common state in one clock cycle, and some states have no predecessors. We can run A5/1 backwards by exploring the tree of possible predecessor states, and backtracking from dead ends. The average number of predecessors of each node is 1, and thus the expected number of vertices in the first k levels of each tree grows only linearly in k (see[3]). As a result, if we find a common state in the disk and data, we can obtain a small number of candidates for the initial state of the frame.

The weakness we exploit here is that due to the frequent reinitializations there is a very short distance from intermediate states to initial states.

Key idea 4: The key can be extracted from the initial state of any frame. Here we exploit the weakness of the A5/1 key setup routine. Assume that we know the state of A5/1 immediately after the key and frame counter were used, and before the 100 mixing steps. By running backwards, we can eliminate the effect of the known frame counter in a unique way, and obtain 64 linear combinations of the 64 key bits. Since the tree exploration may suggest several keys, we can choose the correct one by mixing it with the next frame counter, running A5/1 forward for more than 100 steps, and comparing the results with the actual data in the next frame.

Key idea 5: The Golic attack on A5/1 is marginally impractical. By the well known birthday paradox, A and B are likely to have a common state when their sizes a and b satisfy $a * b \approx n$. We would like a to be bounded by the size of commercially available PC hard disks, and b to be bounded by the number of overlapping prefixes in a typical GSM telephone conversation. Reasonable bounds on these values (justified later in this paper) are $a \approx 2^{35}$ and $b \approx 2^{22}$. Their product is 2^{57}, which is about 100 times smaller than $n = 2^{64}$. To make the intersection likely, we either have to increase the storage requirement from 150 gigabytes to 15 terabytes, or to increase the length of the conversation from two minutes to three hours. Neither approach seems to be practical, but the gap is not huge and a relatively modest improvement by two orders of magnitude is all we need to make it practical.

Key idea 6: Use special states. An important consideration in implementing time-memory tradeoff attacks is that access to disk is about a million times slower than a computational step, and thus it is crucial to minimize the number of times we look for data on the hard disk. An old idea due to Ron Rivest is to keep on the disk only special states which are guaranteed to produce output bits starting with a particular pattern α of length k, and to access the disk only when we encounter such a prefix in the data. This reduces the number b of disk probes by a factor of about 2^k. The number of points a we have to memorize remains unchanged, since in the formula $a * b \approx n$ both b and n are reduced by the same factor 2^k. The downside is that we have to work 2^k times harder during the preprocessing stage, since only 2^{-k} of the random states we try produce outputs with such a k bit prefix. If we try to reduce the number of disk access steps in the time memory attack on A5/1 from 2^{22} to 2^6, we have to increase the preprocessing time by a factor of about 64,000, which makes it impractically long.

Key idea 7: Special states can be efficiently sampled in A5/1. A major weakness of A5/1 which we exploit in both attacks is that it is easy to generate all the states which produce output sequences that start with a particular k-bit pattern α with $k = 16$ without trying and discarding other states. This is due to a poor choice of the clocking taps, which makes the register bits that affect the clock control and the register bits that affect the output unrelated for about 16 clock cycles, so we can choose them independently. This easy access

to special states does not happen in good block ciphers, but can happen in stream ciphers due to their simpler transition functions. In fact, the maximal value of k for which special states can be sampled without trial and error can serve as a new security measure for stream ciphers, which we call its **sampling resistance**. As demonstrated in this paper, high values of k can have a big impact on the efficiency of time-memory tradeoff attacks on such cryptosystems.

Key idea 8: Use biased birthday attacks. The main idea of the first attack is to consider sets A and B which are not chosen with uniform probability distribution among all the possible states. Assume that each state s is chosen for A with probability $P_A(s)$, and is chosen for B with probability $P_B(s)$. If the means of these probability distributions are a/n and b/n, respectively, then the expected size of A is a, and the expected size of B is b.

The birthday threshold happens when $\sum_s P_A(s)P_B(s) \approx 1$. For independent uniform distributions, this evaluates to the standard condition $a*b \approx n$. However, in the new attack we choose states for the disk and states in the data with two non-uniform probability distributions which have strong positive correlation. This makes our time memory tradeoff much more efficient than the one used by Golic. This is made possible by the fact that in A5/1, the initial state of each new frame is rerandomized very frequently with different frame counters.

Key idea 9: Use Hellman's time-memory tradeoff on a subgraph of special states. The main idea of the second attack (called the random subgraph attack) is to make most of the special states accessible by simple computations from the subset of special states which are actually stored in the hard disk. The first occurrence of a special state in the data is likely to happen in the first two seconds of the conversation, and this single occurrence suffices in order to locate a related special state in the disk even though we are well below the threshold of either the normal or the biased birthday attack. The attack is based on a new function f which maps one special state into another special state in an easily computable way. This f can be viewed as a random function over the subspace of 2^{48} special states, and thus we can use Hellman's time-memory tradeoff[4] in order to invert it efficiently. The inverse function enables us to compute special states from output prefixes even when they are not actually stored on the hard disk, with various combinations of time T and memory M satisfying $M\sqrt{T} = 2^{48}$. If we choose $M = 2^{36}$, we get $T = 2^{24}$, and thus we can carry out the attack in a few minutes, after a 2^{48} preprocessing stage which explores the structure of this function f.

Key idea 10: A5/1 is very efficient on a PC. The A5/1 algorithm was designed to be efficient in hardware, and its straightforward software implementation is quite slow. To execute the preprocessing stage, we have to run it on a distributed network of PC's up to 2^{48} times, and thus we need an extremely efficient way to compute the effect of one clock cycle on the three registers.

We exploit the following weakness in the design of A5/1: Each one of the three shift registers is so small that we can precompute all its possible states, and keep them in RAM as three cyclic arrays, where successive locations in each array represent successive states of the corresponding shift register. In fact, we

don't have to keep the full states in the arrays, since the only information we have to know about a state is its clocking tap and its output tap. A state can thus be viewed as a triplet of indices (i, j, k) into three large single bit arrays (see Figure 2). $A_1(i), A_2(j), A_3(k)$ are the clocking taps of the current state, and $A_1(i-11), A_2(j-12), A_3(k-13)$ are the output taps of the current state (since these are the corresponding delays in the movement of clocking taps to output taps when each one of the three registers is clocked). Since there is no mixing of the values of the three registers, their only interaction is in determining which of the three indices should be incremented by 1. This can be determined by a precomputed table with three input bits (the clocking taps) and three output bits (the increments of the three registers). When we clock A5/1 in our software implementation, we don't shift registers or compute feedbacks - we just add a 0/1 vector to the current triplet of indices. A typical two dimensional variant of such movement vectors in triplet space is described in Figure 3. Note the local tree structure determined by the deterministic forward evaluation and the nondeterministic backward exploration in this triplet representation.

Since the increment table is so small, we can expand the A tables from bits to bytes, and use a larger precomputed table with 2^{24} entries, whose inputs are the three bytes to the right of the clocking taps in the three registers, and outputs are the three increments to the indices which allow us to jump directly to the state which is 8 clock cycles away. The total amount of RAM needed for the state arrays and precomputed movement tables is less than 128 MB, and the total cost of advancing the three registers for 8 clock cycles is one table lookup and three integer additions! A similar table lookup technique can be used to compute in a single step output bytes instead of output bits, and to speed up the process of running A5/1 backwards.

5 Detailed Description of the Attacks

In this section we fill in the missing details, and analyse the success rate of the new attacks.

5.1 Efficient Sampling of Special States

Let α be any 16 bit pattern of bits. To simplify the analysis, we prefer to use an α which does not coincide with shifted versions of itself (such as $\alpha = 1000...0$) since this makes it very unlikely that a single 228-bit frame contains more than one occurrence of α.

The total number of states which generate an output prefix of α is about $2^{64} * 2^{-16} = 2^{48}$. We would like to generate all of them in a (barely doable) 2^{48} preprocessing stage, without trying all the 2^{64} possible states and discarding the vast majority which fail the test. The low sampling resistance of A5/1 is made possible by several flaws in its design, which are exploited in the following algorithm:

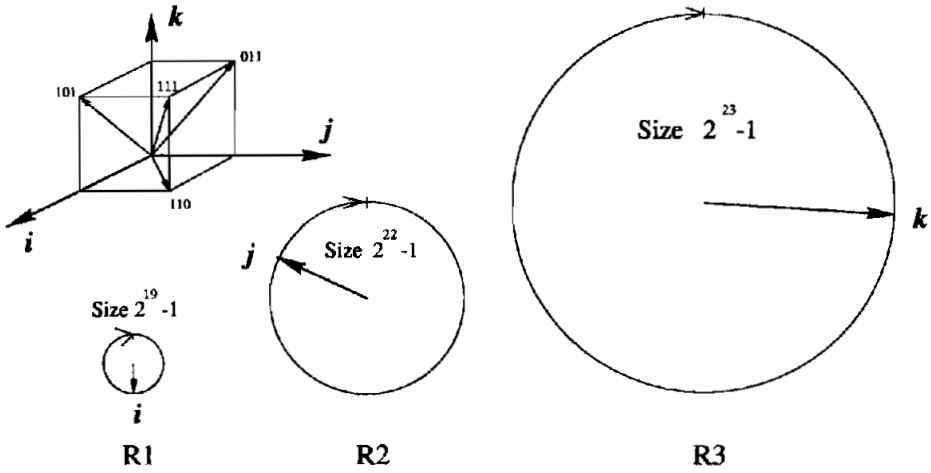


Fig. 2. Triplet representation of a state.

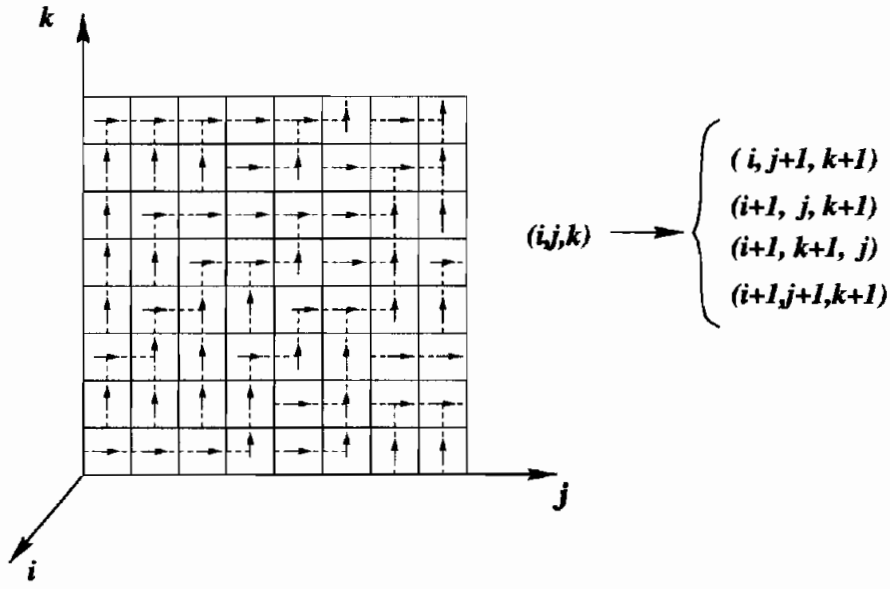


Fig. 3. The state-transition graph in the triplet representation of A5/1.

- Pick an arbitrary 19-bit value for the shortest register $R1$. Pick arbitrary values for the rightmost 11 bits in $R2$ and $R3$ which will enter the clock control taps in the next few cycles. We can thus define $2^{19+11+11} = 2^{41}$ partial states.
- For each partial state we can uniquely determine the clock control of the three registers for the next few cycles, and thus determine the identity of the bits that enter their msb's and affect the output.
- Due to the majority clock control, at least one of $R2$ and $R3$ shifts a new (still unspecified) bit into its msb at each clock cycle, and thus we can make sure that the computed output bit has the desired value. Note that about half the time only one new bit is shifted (and then its choice is forced), and about half the time two new bits are shifted (and then we can choose them in two possible ways). We can keep this process alive without time consuming trial and error as long as the clock control taps contain only known bits whereas the output taps contain at least one unknown bit. A5/1 makes this very easy, by using a single clocking tap and placing it in the middle of each register: We can place in $R2$ and $R3$ 11 specified bits to the right of the clock control tap, and 11-12 unspecified bits to the right of the output tap. Since each register moves only 3/4 of the time, we can keep this process alive for about 16 clock cycles, as desired.
- This process generates only special states, and cannot miss any special state (if we start the process with its partial specification, we cannot get into an early contradiction). We can similarly generate any number $c < 2^{48}$ of randomly chosen special states in time proportional to c. As explained later in the paper, this can make the preprocessing faster, at the expense of other parameters in our attack.

5.2 Efficient Disk Probing

To leave room for a sufficiently long identifying prefix of 35 bits after the 16-bit α, we allow it to start only at bit positions 1 to 177 in each one of the given frames (i.e., at a distance of 101 to 277 from the initial state). The expected number of occurrences of α in the data produced by A5/1 during a two minute conversation is thus $2^{-16} * 177 * 120 * 1000/4.6 \approx 71$. This is the expected number of times b we access the hard disk. Since each random access takes about 6 milliseconds, the total disk access time becomes negligible (about 0.4 seconds).

5.3 Efficient Disk Storage

The data items we store on the disk are (prefix, state) pairs. The state of A5/1 contains 64 bits, but we keep only special states and thus we can encode them efficiently with shorter 48 bit names, by specifying the 41 bits of the partial state and the ≈ 7 choice bits in the sampling procedure. We can further reduce the state to less than 40 bits (5 bytes) by leaving some of the 48 bits unspecified. This saves a considerable fraction of the disk space prepared during preprocessing, and the only penalty is that we have to try a small number of candidate states

instead of one candidate state for each one of the 71 relevant frames. Since this part is so fast, even in its slowed down version it takes less than a second.

The output prefix produced from each special state is nominally of length 16+35=51 bits. However, the first 16 bits are always the constant α, and the next 35 bits are stored in sorted order on the disk. We can thus store the full value of these 35 bits only once per sector, and encode on the disk only their small increments (with a default value of 1). Other possible implementations are to use the top parts of the prefixes as direct sector addresses or as file names. With these optimizations, we can store each one of the sorted (prefix, state) pairs in just 5 bytes. The largest commercially available PC hard disks (such as IBM Ultrastar 72 ZX or Seagate Cheetah 73) have 73 gigabytes. By using two such disks, we can store $146 * 2^{30}/5 \approx 2^{35}$ pairs during the preprocessing stage, and characterize each one of them by the (usually unique) 35-bit output prefix which follows α.

5.4 Efficient Tree Exploration

The forward state-transition function of A5/1 is deterministic, but in the reverse direction we have to consider four possible predecessors. About 3/8 of the states have no predecessors, 13/32 of the states have one predecessor, 3/32 of the states have two predecessors, 3/32 of the states have three predecessors, and 1/32 of the states have four predecessors.

Since the average number of predecessors is 1, Golic assumed that a good statistical model for the generated trees of predecessors is the critical branching process (see [3]). We were surprised to discover that in the case of A5/1, there was a very significant difference between the predictions of this model and our experimental data. For example, the theory predicted that only 2% of the states would have some predecessor at depth 100, whereas in a large sample of 100,000,000 trees we generated from random A5/1 states the percentage was close to 15%. Another major difference was found in the tail distributions of the number of sons at depth 100: Theory predicted that in our sample we should see some cases with close to 1000 sons, whereas in our sample we never saw trees with more than 120 sons at depth 100.

5.5 The Biased Birthday Attack.

To analyse the performance of our biased birthday attack, we introduce the following notation:

Definition 1 *A state s is coloured* **red**, *if the sequence of output bits produced from state s starts with α (i.e., it is a special state). The subspace of all the red states is denoted by R.*

Definition 2 *A state is coloured* **green**, *if the sequence of output bits produced from state s contains an occurrence of α which starts somewhere between bit positions 101 and 277. The subspace of all the green states is denoted by G.*

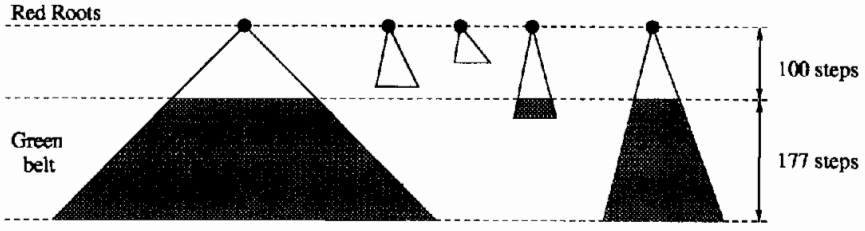


Fig. 4. Trees of different sizes.

The red states are the states that we keep in the disk, look for in the data, and try to collide by comparing their prefixes. The green states are all the states that could serve as initial states in frames that contain α. Non-green initial states are of no interest to us, since we discard the frames they generate from the actual data.

The size of R is approximately 2^{48}, since there are 2^{64} possible states, and the probability that α occurs right at the beginning of the output sequence is 2^{-16}. Since the redness of a state is not directly related to its separate coordinates i, j, k in the triplet space, the red states can be viewed as randomly and sparsely located in this representation. The size of G is approximately $177 * 2^{48}$ (which is still a small fraction of the state space) since α has 177 opportunities to occur along the output sequence.

Since a short path of length 277 in the output sequence is very unlikely to contain two occurrences of α, the relationship between green and red states is essentially many to one: The set of all the relevant states we consider can be viewed as a collection of disjoint trees of various sizes, where each tree has a red state as its root and a "belt" of green states at levels 101 to 277 below it (see Figure 4). The weight $W(s)$ of a tree whose root is the red state s is defined as the number of green states in its belt, and s is called k-heavy if $W(s) \geq k$.

The crucial observation which makes our biased birthday attack efficient is that in A5/1 there is a huge variance in the weights of the various red states. We ran the tree exploration algorithm on 100,000,000 random states and computed their weights. We found out that the weight of about 85% of the states was zero, because their trees died out before reaching depth 100. Other weights ranged all the way from 1 to more than 26,000.

The leftmost graph of Figure 5 describes for each x which is a multiple of 100 the value y which is the total weight of all the trees whose weights were between x and $x + 100$. The total area under the graph to the right of $x = k$ represents the total number of green states in all the k-heavy trees in our sample.

The initial mixing of the key and frame number, which ignores the usual clock control and flips the least significant bits of the registers about half the time before shifting them, can be viewed as random jumps with uniform probability

distribution into new initial states: even a pair of frame counters with Hamming distance 1 can lead to far away initial states in the triplet space. When we restrict our attention to the frames that contain α, we get a uniform probability distribution over the green states, since only green states can serve as initial states in such frames.

The red states, on the other hand, are not encountered with uniform probability distribution in the actual data. For example, a red state whose tree has no green belt will never be seen in the data. On the other hand, a red state with a huge green belt has a huge number of chances to be reached when the green initial state is chosen with uniform probability distribution. In fact the probability of encountering a particular red state s in a particular frame which is known to contain α is the ratio of its weight $W(s)$ and the total number of green states $177 * 2^{48}$, and the probability of encountering it in one of the 71 relevant frames is $P_B(s) = 71 * W(s)/(177 * 2^{48})$.

Since $P_B(s)$ has a huge variance, we can maximize the expected number of collisions $\sum_s P_A(s) * P_B(s)$ by choosing red points for the hard disk not with uniform probability distribution, but with a biased probability $P_A(s)$ which maximizes the correlation between these distributions, while minimizing the expected size of A. The best way to do this is to keep on the disk only the heaviest trees. In other words, we choose a threshold number k, and define $P_A(s) = 0$ if $W(s) < k$, and $P_A(s) = 1$ if $W(s) \geq k$. We can now easily compute the expected number of collisions by the formula:

$$\sum_s P_A(s) * P_B(s) = \sum_{s|W(s)\geq k} 71 * W(s)/(177 * 2^{48})$$

which is just the number of red states we keep on the disk, times the average weight of their trees, times $71/(177 * 2^{48})$.

In our actual attack, we keep 2^{35} red states on the disk. This is a 2^{-13} fraction of the 2^{48} red states. With such a tiny fraction, we can choose particularly heavy trees with an average weight of 12,500. The expected number of colliding red states in the disk and the actual data is $2^{35} * 12,500 * 71/(177 * 2^{48}) \approx 0.61$. This expected value makes it quite likely that a collision will actually exist. [5]

The intuition behind the biased time memory tradeoff attack is very simple. We store red states, but what we really want to collide are the green states in their belts (which are accessible from the red roots by an easy computation). The 71 green states in the actual data are uniformly distributed, and thus we want to cover about 1% of the green area under the curve in the right side of Figure 5. Standard time memory tradeoff attacks store random red states, but each stored state increases the coverage by just 177 green states on average. With our optimized choice in the preprocessing stage, each stored state increases the coverage by 12,500 green states on average, which improves the efficiency of the attack by almost two orders of magnitude.

[5] Note that in time memory tradeoff attacks, it becomes increasingly expensive to push this probability towards 1, since the only way to guarantee success is to memorize the whole state space.

5.6 Efficient Determination of Initial States

One possible disadvantage of storing heavy trees is that once we find a collision, we have to try a large number of candidate states in the green belt of the colliding red state. Since each green state is only partially specified in our compact 5-byte representation, the total number of candidate green states can be hundreds of thousands, and the real time part of the attack can be relatively slow.

However, this simple estimate is misleading. The parasitic red states obtained from the partial specification can be quickly discarded by evaluating their outputs beyond the guaranteed occurrence of α and comparing it to the bits in the given frame. In addition, we know the exact location of α in this frame, and thus we know the exact depth of the initial state we are interested in within the green belt. As a result, we have to try only about 70 states in a cut through the green belt, and not the 12,500 states in the full belt.

5.7 Reducing the Preprocessing Time of the Biased Birthday Attack

The 2^{48} complexity of the preprocessing stage of this attack can make it too time consuming for a small network of PC's. In this section we show how to reduce this complexity by any factor of up to 1000, by slightly increasing either the space complexity or the length of the attacked conversation.

The efficient sampling procedure makes it possible to generate any number $c < 2^{48}$ of random red states in time proportional to c. To store the same number of states in the disk, we have to choose a larger fraction of the tested trees, which have a lower average weight, and thus a less efficient coverage of the green states. Table 2 describes the average weight of the heaviest trees for various fractions of the red states, which was experimentally derived from our sample of 100,000,000 A5/1 trees. This table can be used to choose the appropriate value of k in the

Average Weights							
2^{-4}	2432	2^{-5}	3624	2^{-6}	4719	2^{-7}	5813
2^{-8}	6910	2^{-9}	7991	2^{-10}	9181	2^{-11}	10277
2^{-12}	11369	2^{-13}	12456	2^{-14}	13471	2^{-15}	14581
2^{-16}	15686	2^{-17}	16839	2^{-18}	17925	2^{-19}	19012
2^{-20}	20152	2^{-21}	21227	2^{-22}	22209	2^{-23}	23515
2^{-24}	24597	2^{-25}	25690	2^{-26}	26234		

Table 2. The average weight of the heaviest trees for various fractions of R.

definition the k-heavy trees for various choices of c. The implied tradeoff is very favorable: If we increase the fraction from 2^{-13} to 2^{-7}, we can reduce the preprocessing time by a factor of 64 (from 2^{48} to 2^{42}), and compensate by either doubling the length of the attacked conversation from 2 minutes to 4 minutes,

or doubling the number of hard disks from 2 to 4. The extreme point in this tradeoff is to store in the disk all the sampled red states with nonzero weights (the other sampled red states are just a waste of space, since they will never be seen in the actual data). In A5/1 about 15% of the red states have nonzero weights, and thus we have to sample about 2^{38} red states in the preprocessing stage in order to find the 15% among them (about 2^{35} states) which we want to store, with an average tree weight of 1180. To keep the same probability of success, we have to attack conversations which last about half an hour.

A further reduction in the complexity of the preprocessing stage can be obtained by the early abort strategy: Explore each red state to a shallow depth, and continue to explore only the most promising candidates which have a large number of sons at that depth. This heuristic does not guarantee the existence of a large belt, but there is a clear correlation between these events.

To check whether the efficiency of our biased birthday attack depends on the details of the stream cipher, we ran several experiments with modified variants of A5/1. In particular, we concentrated on the effect of the clock control rule, which determines the noninvertibility of the model. For example, we hashed the full state of the three registers and used the result to choose among the four possible majority-like movements (+1,+1,+1), (+1,+1,0), (+1,0,+1), (0,+1,+1) in the triplet space. The results were very different from the real majority rule. We then replaced the majority rule by a minority rule (if all the clocking taps agree, all the registers move, otherwise only the minority register moves). The results of this minority rule were very similar to the majority-like hashing case, and very different from the real majority case (see Figure 5). It turns out that in this sense A5/1 is actually stronger than its modified versions, but we do not currently understand the reason for this strikingly different behavior. We believe that the type of data in Table 2, which we call the **tail coverage** of the cryptosystem, can serve as a new security measure for stream ciphers with noninvertible state transition functions.

5.8 Extracting the Key From a Single Red State

The biased birthday attack was based on a direct collision between a state in the disk and a state in the data, and required ≈ 71 red states from a relatively long (≈ 2 minute) prefix of the conversation. In the random subgraph attack we use indirect collisions, which make it possible to find the key with reasonable probability from the very first red state we encounter in the data, even though it is unlikely to be stored in the disk. This makes it possible to attack A5/1 with less than two seconds of available data. The actual attack requires several minutes instead of one second, but this is still a real time attack on normal telephone conversations.

The attack is based on Hellman's original time-memory tradeoff for block ciphers, described in [4]. Let E be an arbitrary block cipher, and let P be some fixed plaintext. Define the function f from keys K to ciphertexts C by $f(K) = E_K(P)$. Assuming that all the plaintexts, ciphertexts and keys have the same binary size, we can consider f as a random function (which is not necessarily

one-to-one) over a common space U. This function is easy to evaluate and to iterate but difficult to invert, since computing the key K from the ciphertext $f(K) = E_K(P)$ is essentially the problem of chosen message cryptanalysis.

Hellman's idea was to perform a precomputation in which we choose a large number m of random start points in U, and iterate f on each one of them t times. We store the m (start point, end point) pairs on a large disk, sorted into increasing endpoint order. If we are given $f(K)$ for some unknown K which is located somewhere along one of the covered paths, we can recover K by repeatedly applying f in the easy forward direction until we hit a stored end point, jump to its corresponding start point, and continue to apply f from there. The last point before we hit $f(K)$ again is likely to be the key K which corresponds to the given ciphertext $f(K)$.

Since it is difficult to cover a random graph with random paths in an efficient way, Hellman proposed a rerandomization technique which creates multiple variants of f (e.g., by permuting the order of the output bits of f). We use t variants f_i, and iterate each one of them t times on m random start points to get m corresponding end points. If the parameters m and t satisfy $mt^2 = |U|$, then each state is likely to be covered by one of the variants of f. Since we have to handle each variant separately (both in the preprocessing and in the actual attack), the total memory becomes $M = mt$ and the total running time becomes $T = t^2$, where M and T can be anywhere along the tradeoff curve $M\sqrt{T} = |U|$. In particular, Hellman suggests using $M = T = |U|^{2/3}$.

A straightforward application of this $M\sqrt{T} = |U|$ tradeoff to the $|U| = 2^{64}$ states of A5/1 with the maximal memory $M = 2^{36}$ requires time $T = 2^{56}$, which is much worse than previously known attacks. The basic idea of the new random subgraph attack is to apply the time-memory tradeoff to the subspace R of 2^{48} red states, which is made possible by the fact that it can be efficiently sampled. Since T occurs in the tradeoff formula $M\sqrt{T} = |U|$ with a square root, reducing the size of the graph by a modest 2^{16} (from $|U| = 2^{64}$ to $|R| = 2^{48}$) and using the same memory ($M = 2^{36}$), reduces the time by a huge factor of 2^{32} (from $T = 2^{56}$ to just $T = 2^{24}$). This number of steps can be carried out in several minutes on a fast PC.

What is left is to design a random function f over R whose output-permuted variants are easy to evaluate, and for which the inversion of any variant yields the desired key. Each state has a "full name" of 64 bits which describes the contents of its three registers. However, our efficient sampling technique enables us to give each red state a "short name" of 48 bits (which consists of the partial contents of the registers and the random choices made during the sampling process), and to quickly translate short names to full names. In addition, red states are characterized (almost uniquely) by their "output names" defined as the 48 bits which occur after α in their output sequences. We can now define the desired function f over 48-bit strings as the mapping from short names to output names of red states: Given a 48-bit short name x, we expand it to the full name of a red state, clock this state 64 times, delete the initial 16-bit α, and define $f(x)$ as the remaining 48 output bits. The computation of $f(x)$ from

x can be efficiently done by using the previously described precomputed tables, but the computation of x from $f(x)$ is exactly the problem of computing the (short) name of an unknown red state from the 48 output bits it produces after α. When we consider some output-permuted variant f_i of f, we obviously have to apply the same permutation to the given output sequence before we try to invert f_i over it.

The recommended preprocessing stage stores 2^{12} tables on the hard disk. Each table is defined by iterating one of the variants f_i 2^{12} times on 2^{24} randomly chosen 48-bit strings. Each table contains 2^{24} (start point, end point) pairs, but implicitly covers about 2^{36} intermediate states. The collection of all the 2^{12} tables requires 2^{36} disk space, but implicitly covers about 2^{48} red states.

The simplest implementation of the actual attack iterates each one of the 2^{12} variants of f separately 2^{12} times on appropriately permuted versions of the single red state we expect to find in the 2 seconds of data. After each step we have to check whether the result is recorded as an end point in the corresponding table, and thus we need $T = 2^{24}$ probes to random disk locations. At 6 ms per probe, this requires more than a day. However, we can again use Rivest's idea of special points: We say that a red state is bright if the first 28 bits of its output sequence contain the 16-bit α extended by 12 additional zero bits. During preprocessing, we pick a random red start point, and use f_i to quickly jump from one red state to another. After approximately 2^{12} jumps, we expect to encounter another bright red state, at which we stop and store the pair of (start point, end point) in the hard disk. In fact, each end point consists of a 28 bit fixed prefix followed by 36 additional bits. As explained in the previous attack, we do not have to store either the prefix (which is predictable) or the suffix (which is used as an index) on the hard disk, and thus we need only half the expected storage. We can further reduce the required storage by using the fact that the bright red states have even shorter short names than red states (36 instead of 48 bits), and thus we can save 25% of the space by using bright red instead of red start points in the table. [6] During the actual attack, we find the first red state in the data, iterate each one of the 2^{12} variants of f over it until we encounter a bright red state, and only then search this state among the pairs stored in the disk. We thus have to probe the disk only once in each one of the $t = 2^{12}$ tables, and the total probing time is reduced to 24 seconds.

There are many additional improvement ideas and implementation details which will be described in the final version of this paper.

Acknowledgements: We would like to thank Ross Anderson, Mike Roe, Jovan Golic, Marc Briceno, and Ian Goldberg for their pioneering contributions

[6] Note that we do not know how to jump in a direct way from one bright red state to another, since we do not know how to sample them in an efficient way. We have to try about 2^{12} red states in order to find one bright red start point, but the total time needed to find the 2^{36} bright red start points in all the tables is less than the 2^{48} complexity of the path evaluations during the preprocessing stage.

to the analysis of A5/1, which made this paper possible. We would like to thank Marc Briceno, Ian Goldberg, Ron Rivest and Nicko van Someren for sending very useful comments on the early version of this paper.

References

1. R. Anderson, M. Roe, *A5*, `http://jya.com/crack-a5.htm`, 1994.
2. S. Babbage, *A Space/Time Tradeoff in Exhaustive Search Attacks on Stream Ciphers*, European Convention on Security and Detection, IEE Conference publication, No. 408, May 1995.
3. M. Briceno, I. Goldberg, D. Wagner, A pedagogical implementation of A5/1, `http://www.scard.org`, May 1999.
4. J. Golic, *Cryptanalysis of Alleged A5 Stream Cipher*, proceedings of EUROCRYPT'97, LNCS 1233,pp.239–255, Springer-Verlag 1997.
5. M. E. Hellman, *A Cryptanalytic Time-Memory Trade-Off*, IEEE Transactions on Information Theory, Vol. IT-26, N 4, pp.401–406, July 1980.

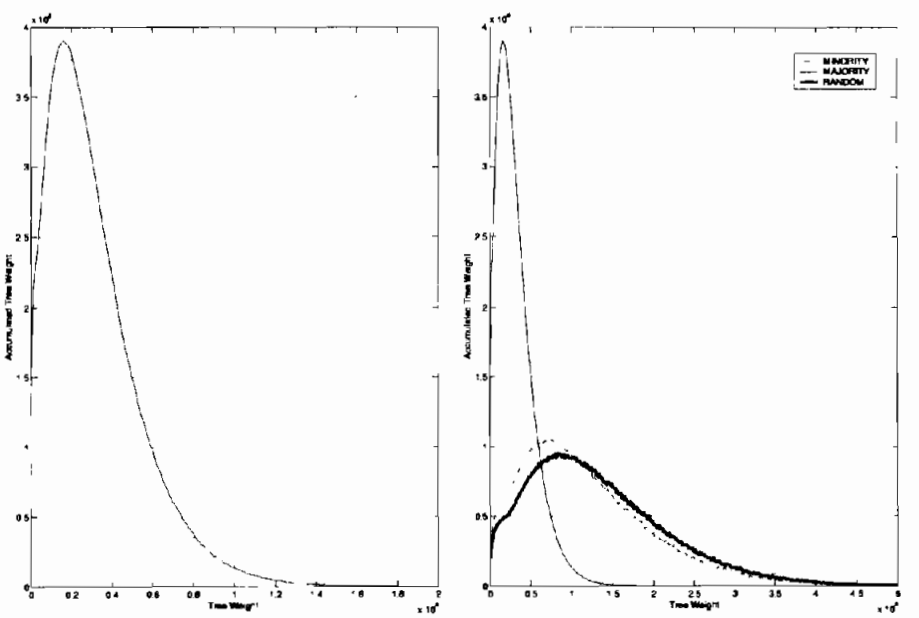


Fig. 5. Weight distributions. The graph on the left shows weight distribution for the majority function; the graph on the right compares the weight distributions of several clock-control functions.

This article was processed using the LaTeX macro package with LLNCS style

ANEXO C

REPORTES DE PRUEBAS GENERADOS POR MATLAB

C.1. INTRODUCCIÓN

En este anexo se incluyen los reportes de pruebas generados por MATLAB. Para una mejor lectura de los mismos, se han editado, sin alterar de ninguna forma los valores obtenidos.

C.2. REPORTE DE PRUEBAS EN EL ALGORITMO DES

MATLAB PROFILE REPORT: SUMMARY

Report generated 04-Mar-2001 09:07:35

Total recorded time: 17091.60 s
Number of M-functions: 49
Number of M-subfunctions: 2
Number of MEX-functions: 1
Clock precision: 0.050 s

FUNCTION LIST

Name	Time		Calls	Time/call	Self time	
adqudes	17091.55	100.0%	1	17091.550000000	48.91	0.3%
dessbox	14699.70	86.0%	32000	0.459365625	1305.32	7.6%
descip	7425.24	43.4%	1000	7.425240000	71.64	0.4%
desdec	7422.87	43.4%	1000	7.422870000	41.96	0.2%
strcat	6296.61	36.8%	512000	0.012298066	3415.97	20.0%
numer4	2944.12	17.2%	256000	0.011500469	797.79	4.7%
int2str	2589.88	15.2%	1664000	0.001556418	2589.88	15.2%
str2num	2293.72	13.4%	849875	0.002698891	1421.14	8.3%
deblank	2165.18	12.7%	1536001	0.001409621	2165.18	12.7%
bin2dec	1448.05	8.5%	528000	0.002742519	1086.36	6.4%
da	1339.74	7.8%	2	669.870000000	1005.51	5.9%
str2num/protected_conversion	872.58	5.1%	849875	0.001026716	872.58	5.1%
ad	851.56	5.0%	1	851.560000000	641.11	3.8%
strvcat	715.46	4.2%	512001	0.001397380	715.46	4.2%

dec2bin	378.91	2.2%	264000	0.001435265	378.91	2.2%
iscellstr	361.69	2.1%	528000	0.000685019	361.69	2.1%
strg	290.48	1.7%	16000	0.018155000	85.06	0.5%
numer8	196.43	1.1%	8000	0.024553750	49.04	0.3%
xor	34.81	0.2%	64000	0.000543906	34.81	0.2%
daqmex	1.92	0.0%	33	0.058181818	1.87	0.0%
analoginput/analoginput	1 43	0.0%	2	0.715000000	0.66	0.0%
analoginput/getdata	1.04	0.0%	1	1.040000000	0.00	0.0%
daqdevice/addchannel	0.39	0.0%	1	0.390000000	0.22	0.0%
daqgate	0.22	0.0%	7	0.031428571	0.05	0.0%
daqdevice/delete	0.16	0.0%	1	0.160000000	0.00	0.0%
daqdevice/set	0.16	0.0%	3	0.053333333	0.05	0.0%
num2str	0.16	0.0%	1	0.160000000	0.11	0.0%
privatechecksetinput	0.11	0.0%	3	0.036666667	0.06	0.0%
aichannel/horzcat	0.11	0.0%	1	0.110000000	0.00	0.0%
privatecheckhandle	0.06	0.0%	2	0.030000000	0.06	0.0%
aichannel/loadobj	0.05	0.0%	1	0.050000000	0.05	0.0%
iscell	0.05	0.0%	7	0.007142857	0.05	0.0%
datenum	0.05	0.0%	2	0.025000000	0.05	0.0%
now	0.05	0.0%	2	0.025000000	0.00	0.0%
daqdevice/start	0.05	0.0%	1	0.050000000	0.00	0.0%
log10	0.05	0.0%	1	0.050000000	0.05	0.0%
profile	0.00	0.0%	1	0.000000000	0.00	0.0%
isnumeric	0.00	0.0%	1	0.000000000	0.00	0.0%
daqchild/getsetstore	0.00	0.0%	2	0.000000000	0.00	0.0%
aichannel/saveobj	0.00	0.0%	1	0.000000000	0.00	0.0%
isstruct	0.00	0.0%	6	0.000000000	0.00	0.0%
aichannel/vertcat	0.00	0.0%	1	0.000000000	0.00	0.0%
privatecheckparent	0.00	0.0%	2	0.000000000	0.00	0.0%
daqchild/size	0.00	0.0%	3	0.000000000	0.00	0.0%
daqchild/isvalid	0.00	0.0%	3	0.000000000	0.00	0.0%
daqchild/daqchild	0.00	0.0%	2	0.000000000	0.00	0.0%
aichannel/aichannel	0.00	0.0%	2	0.000000000	0.00	0.0%
daqdevice/get	0.00	0.0%	2	0.000000000	0.00	0.0%
daqdevice/isvalid	0.00	0.0%	9	0.000000000	0.00	0.0%
nargchk	0.00	0.0%	4	0.000000000	0.00	0.0%
daqdevice/daqdevice	0.00	0.0%	1	0.000000000	0.00	0.0%
num2str/lefttrim	0.00	0.0%	1	0.000000000	0 00	0.0%

C.3. REPORTE DE PRUEBAS EN EL ALGORITMO IDEA

MATLAB PROFILE REPORT: SUMMARY

Report generated 06-Mar-2001 14:21:35

Total recorded time: 11789.70 s
Number of M-functions: 47
Number of M-subfunctions: 2
Number of MEX-functions: 1
Clock precision: 0.060 s

FUNCTION LIST

Name	Time		Calls	Time/call	Self time	
adquidea	11789.70	100.0%	1	11789.700000000	54.49	0.5%
ideadec	5084.73	43.1%	1000	5.084730000	124.50	1.1%
strg	4566.41	38.7%	128000	0.035675078	1359.71	11.5%
ideacip	4312.57	36.6%	1000	4.312570000	99.29	0.8%
numer16	4080.44	34.6%	72000	0.056672778	990.04	8.4%
str2num	3240.39	27.5%	1133565	0.002858583	1997.80	16.9%
int2str	3206.70	27.2%	1920000	0.001670156	3206.70	27.2%
da	1466.02	12.4%	2	733.010000000	1084.98	9.2%
str2num/protected_conversion	1242.59	10.5%	1133565	0.001096179	1242.59	10.5%
ad	868.70	7.4%	1	868.700000000	653.76	5.5%
bin2dec	395.23	3.4%	128000	0.003087734	307.40	2.6%
mod	270.40	2.3%	82000	0.003297561	270.40	2.3%
numer8	198.25	1.7%	8000	0.024781250	48.26	0.4%
dec2bin	130.93	1.1%	80000	0.001636625	130.93	1.1%
gcd	106.89	0.9%	10000	0.010689000	106.89	0.9%
iscellstr	87.83	0.7%	128000	0.000686172	87.83	0.7%
xor	20.94	0.2%	48000	0.000436250	20.94	0.2%
daqmex	1.76	0.0%	37	0.047567568	1.65	0.0%
analoginput/analoginput	1.31	0.0%	2	0.655000000	0.65	0.0%
analoginput/getdata	1.10	0.0%	1	1.100000000	0.11	0.0%
daqdevice/addchannel	0.38	0.0%	1	0.380000000	0.17	0.0%
daqgate	0.22	0.0%	7	0.031428571	0.11	0.0%
daqdevice/delete	0.17	0.0%	1	0.170000000	0.00	0.0%
daqdevice/set	0.17	0.0%	3	0.056666667	0.00	0.0%
num2str	0.17	0.0%	1	0.170000000	0.17	0.0%
aichannel/horzcat	0.16	0.0%	1	0.160000000	0.05	0.0%
aichannel/saveobj	0.06	0.0%	1	0.060000000	0.06	0.0%
daqdevice/isvalid	0.06	0.0%	11	0.005454545	0.06	0.0%

privatecheckhandle	0.06	0.0%	2	0.030000000	0.00	0.0%
daqchild/isvalid	0.06	0.0%	3	0.020000000	0.06	0 0%
privatechecksetinput	0.05	0.0%	3	0.016666667	0.05	0.0%
now	0.05	0.0%	2	0.025000000	0.05	0.0%
profile	0.00	0 0%	1	0.000000000	0.00	0.0%
daqchild/getsetstore	0 00	0.0%	1	0.000000000	0.00	0.0%
daqdevice/start	0.00	0.0%	1	0 000000000	0.00	0.0%
isstruct	0.00	0.0%	6	0.000000000	0.00	0.0%
aichannel/vertcat	0 00	0.0%	1	0.000000000	0.00	0.0%
privatecheckparent	0.00	0.0%	2	0.000000000	0 00	0.0%
daqchild/size	0.00	0.0%	3	0.000000000	0.00	0.0%
daqchild/daqchild	0.00	0.0%	1	0.000000000	0.00	0.0%
aichannel/aichannel	0.00	0.0%	1	0.000000000	0.00	0.0%
iscell	0.00	0.0%	7	0.000000000	0.00	0.0%
daqdevice/get	0.00	0.0%	4	0.000000000	0.00	0.0%
nargchk	0.00	0.0%	6	0.000000000	0.00	0.0%
daqdevice/daqdevice	0.00	0.0%	1	0.000000000	0.00	0 0%
datenum	0.00	0.0%	2	0.000000000	0.00	0.0%
deblank	0.00	0.0%	1	0.000000000	0.00	0.0%
num2str/lefttrim	0.00	0.0%	1	0.000000000	0.00	0.0%
strvcat	0.00	0.0%	1	0.000000000	0.00	0.0%
log10	0.00	0.0%	1	0.000000000	0.00	0.0%

C.4. REPORTE DE PRUEBAS EN EL ALGORITMO A5/1

MATLAB PROFILE REPORT: SUMMARY

Report generated 06-Mar-2001 23:13:48

Total recorded time: 5891.58 s

Number of M-functions: 43

Number of M-subfunctions: 2

Number of MEX-functions: 2

Clock precision: 0.050 s

FUNCTION LIST

Name	Time		Calls	Time/call	Self time	
adqua5	5891.58	100.0%	1	5891.580000000	10.68	0.2%
a5cip	1845.72	31.3%	1	1845.720000000	1526.53	25 9%
a5dec	1729.82	29.4%	1	1729.820000000	1473.62	25.0%
da	1404.12	23.8%	2	702.060000000	1041 21	17.7%

ANEXO D

MANUAL DE USUARIO

D.1. INTRODUCCIÓN

El Sistema Simulador de Encriptación de Audio es un programa que permite encriptar y desencriptar muestras de audio a través de cuatro algoritmos: DES, IDEA, A5/1 y RC4. Este programa forma parte del proyecto de titulación denominado: "Implementación de un Sistema Simulador de Encriptación de Audio utilizando el entorno de programación MATLAB". Para ejecutar este programa se requiere un conocimiento básico de la operación de MATLAB.

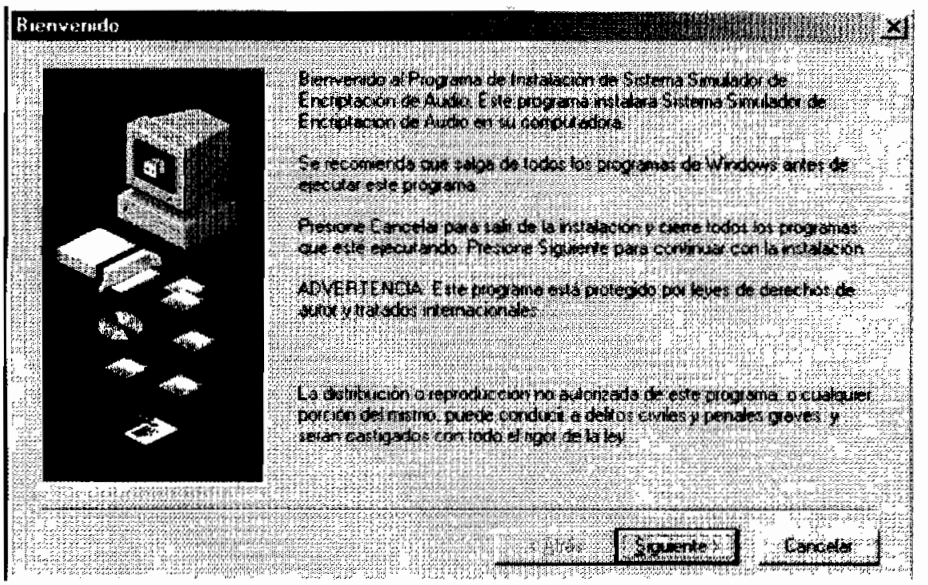
D.2. REQUERIMIENTOS DEL SISTEMA

El Sistema Simulador de Encriptación de Audio necesita los siguientes recursos para poder instalarse:

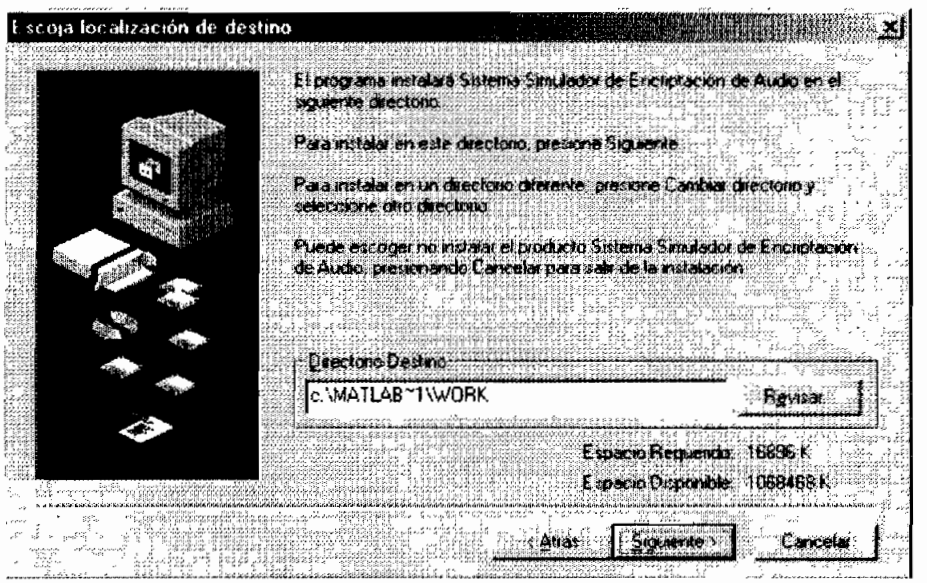
- Procesador Intel 80486 o superior
- Microsoft Windows 95/98/ME/NT/2000
- 16 MB de RAM mínimo
- MATLAB R11 o superior
- 12 MB de espacio en disco duro
- CD-ROM 4x o superior
- Tarjeta de sonido compatible con SoundBlaster
- Resolución de video de 800x600 o superior
- Micrófono y parlantes conectados a la tarjeta de sonido

D.3. INSTALACIÓN

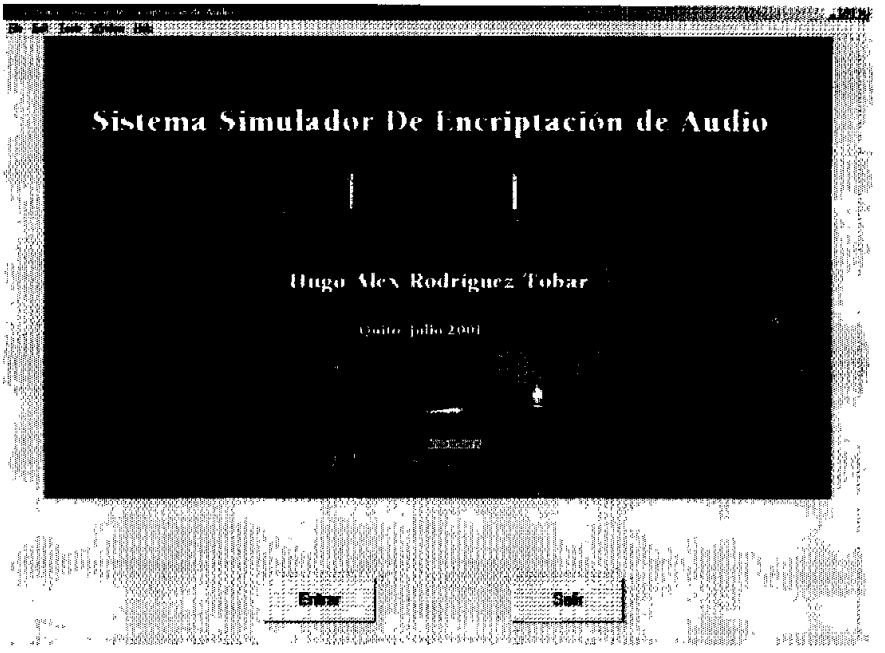
Para instalar el Sistema Simulador de Encriptación, es necesario ejecutar el archivo SSEA.exe incluido en el CD de instalación. Al ejecutarlo, aparecerá la siguiente pantalla:



SI se desea instalar el Sistema Simulador de Encriptación de Audio, se presiona el botón Siguiente; si no, se presiona el botón Cancelar. Si se presiona Siguiente, aparece la siguiente pantalla:

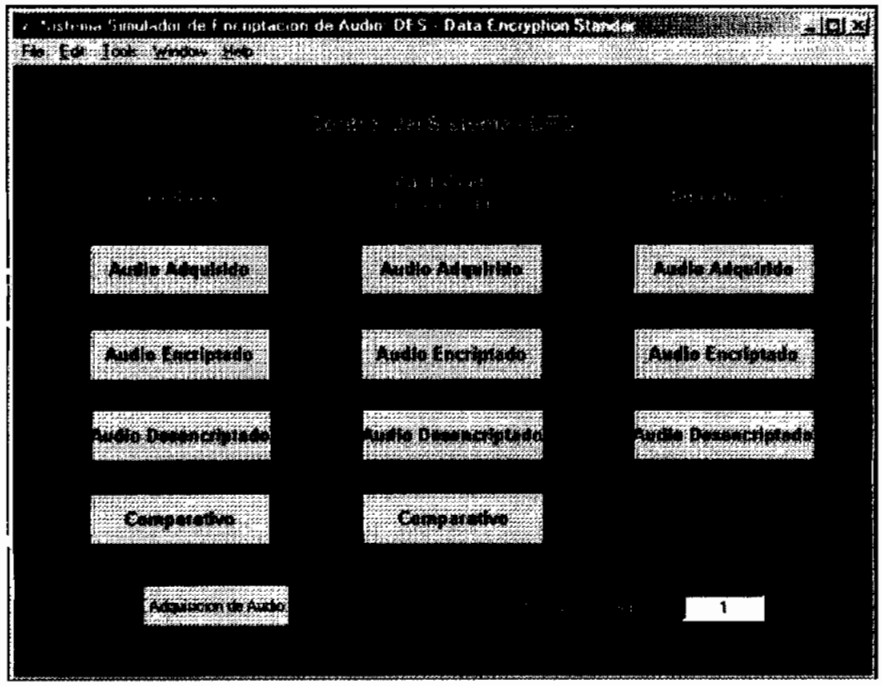


Una vez realizadas estas tareas, se ejecutará el programa MATLAB, el cual cargará el Sistema Simulador de Encriptación de Audio automáticamente. La pantalla de presentación tiene la siguiente forma:



Para seguir ejecutando el sistema, se presiona "Entrar". Para abandonar la ejecución, se presiona "Salir". Una vez que se presiona "Entrar", se presenta la siguiente pantalla:

En esta pantalla se elige el algoritmo que se desea simular, presionando cualquiera de los cuatro botones que tienen el nombre de cada algoritmo. Una vez que se presiona un botón, aparece la siguiente pantalla:



Cada una de las opciones mostradas en esta pantalla se explican abajo:

- *Gráficos:* Permite visualizar las formas de onda de audio adquirido, encriptado y desencriptado, así como un gráfico comparativo de las tres señales. Al presionar cada botón, aparece una pantalla mostrando las formas de onda descritas arriba.
- *Análisis en Frecuencia:* Permite visualizar el espectro de frecuencia de las señales indicadas en el punto anterior. Al presionar cada botón se muestra el espectro de frecuencia respectivo.
- *Reproducción:* Permite reproducir las señales de audio a través de la tarjeta de sonido y los parlantes conectados a la misma.
- *Adquisición de Audio:* Esta opción permite adquirir una muestra de audio, encriptarla y desencriptarla de acuerdo al algoritmo escogido, y al tiempo de adquisición que se muestra a la derecha de esta opción. Es importante

recalcar que si se necesita el tiempo de adquisición señalado, primero se debe especificar éste antes de ejecutar la adquisición de audio.

- *Tiempo de adquisición:* Es el tiempo que el usuario puede especificar para adquirir una muestra de audio. La unidad de este tiempo está en segundos. Cuando se introduce un valor, se debe presionar la tecla Enter para que el nuevo valor sea reconocido.

El programa de adquisición de audio dispone de varias paradas que hacen que el usuario pueda controlar los procesos incluidos en el programa. Estas paradas se pueden verificar en la ventana de comandos de MATLAB. Los mensajes que se muestran son los siguientes:

- *Listo para adquirir muestra de audio. Presione Enter para continuar:* Este mensaje se presenta para prevenir al usuario que el sistema está listo para adquirir la muestra de audio de acuerdo al tiempo indicado. El usuario en este momento deberá disponer de una fuente de audio, cualquiera que ésta sea (un micrófono, un programa reproductor, etc.)
- *Listo para realizar conversión analógica/digital. Presione Enter para continuar:* Este mensaje indica que se producirá la conversión de la señal adquirida de su formato analógico a uno digital. Simplemente presione Enter cuando esté listo para continuar.
- *Listo para realizar encriptación. Presione Enter para continuar:* Este mensaje indica que está a punto de realizarse la encriptación. Cuando el usuario esté listo, debe presionar Enter.
- *Listo para realizar desencriptación. Presione Enter para continuar:* Este mensaje indica que está a punto de realizarse la desencriptación. El usuario debe presionar Enter para continuar.
- *Proceso terminado. Presione Enter para continuar:* Este mensaje indica que las variables del proceso están calculadas y listas para ser salvadas y utilizadas por el Sistema Simulador. El usuario debe presionar Enter para continuar.

D.5. RECOMENDACIONES DE USO

Para que el programa funcione de forma óptima, se necesitan ciertos parámetros que se exponen a continuación:

- No ejecutar otros programas aparte de MATLAB y del Sistema Simulador. La cantidad de memoria RAM libre es importante para la correcta ejecución de los algoritmos de encriptación
- Adquirir pocos segundos de audio para realizar la encriptación y desencriptación. Esto se realiza debido a que el tiempo de encriptación y desencriptación es grande, y por ende, el usuario puede cansarse, o pensar que el programa ejecutó una operación no válida. Si el usuario dispone de tiempo y paciencia, puede pasar de 10 s de adquisición de audio.
- Los menús que aparecen en todas las ventanas del Sistema Simulador no tienen relación con el mismo. Se trata de menús por defecto que provee MATLAB en todas sus ventanas gráficas.
- Para cerrar las ventanas del Sistema Simulador se puede recurrir al botón Salir estándar de las ventanas de Windows, al comando Close en el menú File de la misma ventana, o a presionar las teclas Ctrl-W o Alt-F4.
- Se recomienda maximizar la ventana cuando se hace uso de la opción Comparativo, tanto en Gráficos como en Análisis en Frecuencia.

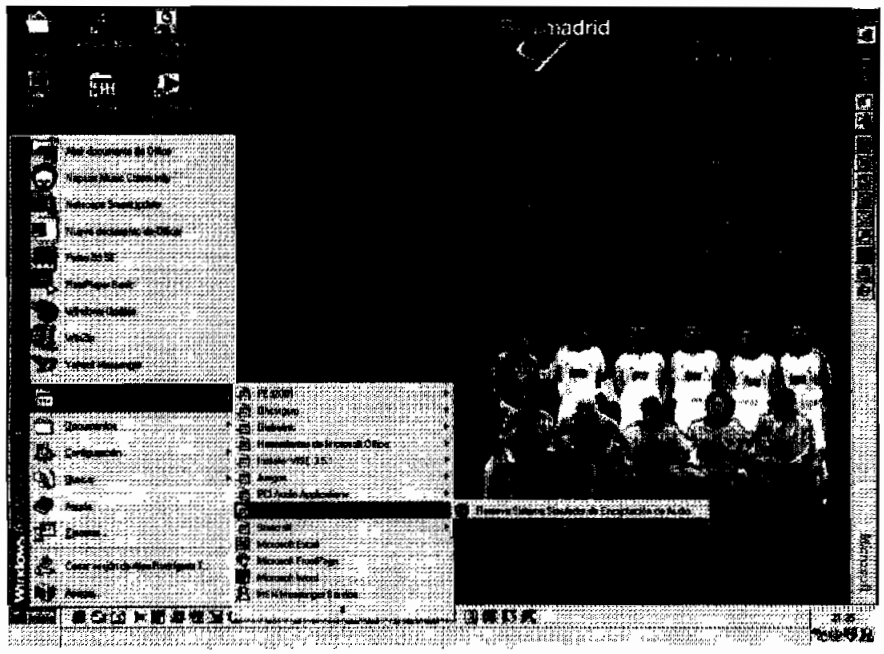
D.6. EJEMPLO DE OPERACIÓN

Para finalizar el manual de usuario se incluye un ejemplo de la operación del Sistema Simulador de Encriptación de Audio.

D.6.1. EJECUCIÓN DEL SISTEMA SIMULADOR

Para ejecutar el Sistema Simulador de Encriptación de Audio, se hace clic sobre el icono "Sistema Simulador de Encriptación de Audio", que se incluye en el

grupo del mismo nombre, el cual se encuentra en el menú Inicio. La siguiente figura muestra esta operación:



Una vez que se realiza lo anteriormente explicado, se ejecutará el paquete MATLAB, el cual cargará automáticamente el Sistema Simulador de Encriptación de Audio. Después de pasar la pantalla de presentación, y de haber escogido el algoritmo en la pantalla de selección, se procede a la pantalla de control. Para este ejemplo, se escoge el algoritmo RC4. Si se hace clic sobre el botón Audio Adquirido de la opción Gráficos, aparecerá la forma de onda adquirida, y así en adelante.

La parte crítica del Sistema Simulador es el programa de Adquisición de Audio, el cual funciona en la pantalla principal de MATLAB. Lo único que el usuario tiene que hacer es seguir las instrucciones indicadas en la pantalla.