Indira Gandhi
National Open University
School of Sciences

MMTE-006
CRYPTOGRAPHY

Block

2

SYMMETRIC KEY CRYPTOSYSTEMS AND HASH FUNCTIONS

UNIT 4
Symmetric Key Block Ciphers **5**

UNIT 5
Stream Ciphers **29**

UNIT 6
Hash Functions **49**

Curriculum Design Committee

Dr. B.D. Acharya
Dept. of Science & Technology
New Delhi

Prof. Adimurthi
School of Mathematics
TIFR, Bangalore

Prof. Archana Aggarwal
CESP, School of Social Sciences
JNU, New Delhi

Prof. R. B. Bapat
Indian Statistical Institute, New Delhi

Prof. M.C. Bhandari
Dept. of Mathematics
IIT, Kanpur

Prof. R. Bhatia
Indian Statistical Institute, New Delhi

Prof. A. D. Dharmadhikari
Dept. of Statistics
University of Pune

Prof. O.P. Gupta
Dept. of Financial Studies
University of Delhi

Prof. S.D. Joshi
Dept. of Electrical Engineering
IIT, Delhi

Dr. R. K. Khanna
Scientific Analysis Group
DRDO, Delhi

Prof. Susheel Kumar
Dept. of Management Studies
IIT, Delhi

Prof. Veni Madhavan
Scientific Analysis Group
DRDO, Delhi

Prof. J.C. Mishra
Dept. of Mathematics
IIT, Kharagpur

Prof. C. Musili
Dept. of Mathematics and Statistics
University of Hyderabad

Prof. Sankar Pal
ISI, Kolkata

Prof. A.P. Singh
PG Dept. of Mathematics
University of Jammu

Faculty Members
School of Sciences, IGNOU
Dr. Deepika
Prof. Poornima Mital
Dr. Atul Razdan
Prof. Parvin Sinclair
Prof. Sujatha Varma
Dr. S. Venkataraman

Course Design Committee

Prof. C.A. Murthy
ISI, Kolkata

Prof. S.B. Pal
IIT, Kharagpur

Dr. B.S. Panda
IIT, Delhi

Prof. C.E. Veni Madhavan
IISC, Bangalore

Faculty Members
School of Sciences, IGNOU
Dr. Deepika
Prof. Poornima Mital
Dr. Atul Razdan
Prof. Parvin Sinclair
Dr. S. Venkataraman

Block Preparation Team

Dr. Sucheta Chakroborty(Editor)
Scientist 'E'
DRDO
Shri. Dhananjay Dey(Editor)
Scientist 'D'
DRDO

Dr. S. Venkataraman
School of Sciences
IGNOU

Course Coordinator: Dr. S. Venkataraman

September 2010

ISBN—978-81-266-4918-1

Further information on the Indira Gandhi National Open University courses may be obtained from the University's office at Maidan Garhi, New Delhi-110 068.

Printed and Published on behalf of Indira Gandhi National Open University, New Delhi, by Director, School of Sciences.

BLOCK INTRODUCTION

As you may be already aware, the cryptosystems are broadly divided into two types, namely symmetric key cryptosystems or private key cryptosystems and public key cryptosystems. The symmetric key cryptosystems are the subject of our study in this block.

Again, depending on the way they encrypt, the cryptosystems are divided into Block key ciphers and Stream ciphers. In unit 4 of this block, we will discuss block ciphers. In this Unit, after a discussion on general design principles of block ciphers, we will discuss the Data Encryption Standard, a block cipher that was considered very strong and was used for a long time. It became obsolete by the end of the 1990s and was replaced by the Advanced Encryption Standards(AES). In the first unit on block ciphers, we discuss this also.

In unit 5, we discuss stream ciphers. Stream cipher uses a pseudorandom sequence generated from a seed value for encryption. The seed itself is the key for the cipher. In this unit we discuss Linear Feedback Shift Registers(LFSR) which are used for generating pseudorandom sequences. For the encryption to be secure, the pseudorandom numbers have to be unpredictable, meaning that it should not be possible to predict the next bit in the pseudorandom sequence with probability greater than half. So, we discuss statistical tests for checking whether a pseudorandom sequence is sufficiently unpredictable. Many of the early stream ciphers were meant for software implementation. We also discuss RC4 cipher, a stream cipher invented by Ron Rivest for implementation in software.

In unit 6, we discuss hash functions. They are important tools for authentication. They are used in modification detection and digital signatures. In this unit, we first discuss compression functions the building blocks of hash functions. After this, we discuss the Merkle-Damgård method, a general method for constructing hash functions from compression functions. At the end of the unit, as examples of hash functions, we will discuss also discuss the MD5 algorithm and SHA-2.

NOTATIONS & SYMBOLS

$\mathcal{K}$	Key space
$\oplus$	Bitwise XOR
$\|\|$	Concatenation of strings
$\vee$	Logical OR
$\wedge$	Logical AND
$\neg$	Logical NOT
$\gg$	Right shift
$\ll$	Left shift
$\lll$	Left circular shift
$\ggg$	Right circular shift

UNIT 4 SYMMETRIC KEY BLOCK CIPHERS

Structure **Page No.**

4.1 Introduction 5
Objectives
4.2 Block Ciphers 6
4.3 The Data Encryption Standard 11
Modes of Operation
Cryptanalysis of DES
4.4 The Advanced Encryption Standard 20
The Basic Algorithm
The Layers
Decryption
4.5 Summary 25
4.6 Solutions/Answers 25

4.1 INTRODUCTION

In the first block, we had discussed some simple cryptosystems. In those cryptosystems, we encrypt and decrypt using pencil and paper. The cryptosystems were used mainly for military and diplomatic purposes. However, in the 1960s it was possible to connect computers in a network and communicate information from one computer to another through these networks. It also became possible to access computers from a remote network. Computers were also used for storing a lot of confidential and sensitive information. It became necessary to store and communicate information securely so that unauthorised persons had no access to them. For this, most of the encryption methods used earlier became useless. Methods for handling the challenges arising from the developments in the communication technology had to be evolved.

In response to a call for a cryptographic algorithm in 1973 from National Bureau of Standards(NBS), a US government agency, IBM submitted an algorithm called LUCIFER in 1974. The National Security Agency made some modifications and the modified algorithm, which is essentially the Data Encryption Standard(DES), was released by NBS with a free license in 1975. NBS made the algorithm the official standard for data encryption in 1977. The DES is an example of a **block cipher**. In Sec. 4.2, we will discuss block ciphers in general and the principles behind their design. We will discuss the DES in Sec. 4.3 of this unit.

NBS was renamed as National Institute of Standards and Technology(NIST) later.

DES remained the standard cryptographic system in 80s and 90s, but due to improvements in technology it became vulnerable. We discuss the weaknesses that were discovered in Sec. 4.3. In Sec. 4.4, we will discuss the Advanced Encryption Standard algorithm(AES) that was developed to replace the DES algorithm.

Objectives

After studying this unit, you should be able to

- define a block cipher;
- define a stream cipher;
- explain the general design principles behind block ciphers as enunciated by Claude Shannon;
- encrypt and decrypt using a toy block cipher;
- explain how DES encrypts and decrypts messages;
- explain the various modes of operation;

- explain why DES had to be replaced; and
- explain how the AES encrypts and decrypts messages.

4.2 BLOCK CIPHERS

The ciphers are broadly classified under two different types, the **block ciphers** and **stream ciphers**. Let us now formally define what they are.

Definition 1: A **block cipher** breaks a message P into blocks $P_1P_2\ldots$ of some fixed length n and encrypts each block P_i with the same key $k \in \mathcal{K}$, that is

$$E_k(P) = E_k(P_1)E_k(P_2)\ldots$$

A **stream cipher** breaks a message P into characters or bits $p_1p_2\ldots$ and encrypts with a **key stream** $k_1k_2\ldots$, that is

$$E_k(P) = E_{k_1}(p_1)E_{k_2}(p_2)\ldots$$

A **synchronous stream cipher** is one in which we generate the key stream independent of the plain text.

A stream cipher is **periodic** of period d if the stream repeats after d characters.

The Vigenere cipher is an example of a synchronous, periodic stream cipher. This is because, if $c_1c_2\cdots c_n$ is the key, we use the key stream $k_1k_2\cdots$ where

$$k_i = \begin{cases} c_i & \text{if } 1 \le i \le n \\ c_m & 1 \le m \le n, i \equiv m \bmod n \end{cases}$$

In the previous block we discussed some simple substitution and transposition ciphers. They are not strong due to two attributes of the plain text. One is the **redundancy** in the plain text. For example, in English, the letter 'Q' is almost always followed by the letter 'U'. The redundancy is preserved by substitution ciphers. So, while decrypting a text that was encrypted with a substitution cipher, if a character decrypts to Q, then it is almost sure that the next character decrypts to U. However, this redundancy is destroyed by transposition ciphers.

Another attribute is the statistical characteristics of the plain text. For example, we know that the character that appears most frequently is the letter 'E'. The statistical characteristics of the text are preserved by a transposition cipher, but destroyed by a substitution cipher.

In his immensely influential paper, [20], Claude Shannon presented the principles of **confusion** and **diffusion**. Here, confusion is to make the relation between the key and the cipher text as complex as possible so that the redundancy is reduced; diffusion is to spread the influence of individual plain text characters over as much of the cipher text as possible so that the statistical properties of the plain text are hidden.

So, while designing a block cipher, we have to use the two principles so that the cipher is hard to break. In the same paper, Shannon also suggests a method for achieving confusion and diffusion, namely using a mixture of some simple transformations like transpositions and substitutions. We can get a **product cipher** by mixing different, suitable, combinations of some simpler ciphers.

Feistel was a part of the team at IBM which designed the LUCIFER cipher.

In the **iterated block ciphers**, we usually use a complex **round function** repeatedly, using the input from the previous round as the output for the next round. One such function is derived from the **Feistel cipher**. We will now present a simplified round function which is quite similar to the Feistel cipher. The design of many of the block

ciphers, including DES, are similar to the Feistel cipher. The discussion is based on [21].

Since using the DES manually is laborious and cumbersome—remember, it was meant to be used on computers as opposed to the other classical ciphers like the Vigenere cipher in which encryption was done by hand—we can't use it in our examples. To help you understand the basic ideas behind a block cipher, we will discuss a simpler cipher discussed in Sec. 4.2 of [21]. This cipher is similar to the DES, but much smaller. It has an S-box like many real world ciphers. We will call it 'Toy block cipher' since it can't be used in practice and yet we can understand how product block ciphers work using this. In this cipher, the message we have to encrypt consists of a single block of 12 bits and our key is nine bits long.

We will now discuss how to carry out one round of encryption using this block cipher with the help of an example. Suppose the message we want to encrypt is $m = 101011001101$ and the key is $k = 010100110$.

Step 1 We write the message in the form L_0R_0 where L_0 consists of the first six bits and R_0 consists of the remaining six bits. In our case $L_0 = 101011$ and $R_0 = 001101$.

Step 2 We obtain an eight-bit key k_1 from the nine-bit key for this round as follows: We use the first eight digits in the key from the left. Here, $k_1 = 01010011$. In general, in the i^{th} round of encryption, we use the eight digits of the key starting from i^{th} digit. If we reach the end of the nine digit key, we 'wrap around' and continue from the first digit. For example, in the third round, we use the seven digits of starting from the third digit as the first seven digits of k_3 and use the first digit of k as the last digit of k_3. For our key k, the key for the third round encryption is $k_3 = 01001100$.

We use a function $f\colon \{0,1\}^6 \times \{0,1\}^8 \to \{0,1\}^6$ to get a six bit output from (R_0, k_1).

Let us see how this function works. First, it expands the given six bit number to eight bits as follows:

- The first and the second bits of the output are the same as the first and second bits of the input, respectively.
- The third bit and the fifth bit of the output are the same as the fourth bit of the input.
- The fourth and sixth bit of the output are, respectively, the same as the third bit of the input.
- The seventh and the eighth bit of the output are, respectively, the fifth and sixth bit of the output.

To summarise, if the six-bit input is $a_1a_2a_3a_4a_5a_6$, the eight-bit output is $a_1a_2a_4a_3a_4a_3a_5a_6$. In this example, $R_0 = 101011$, so $a_1 = 1$, $a_2 = 0$, $a_3 = 1$, $a_4 = 0$, $a_5 = 1$ and $a_6 = 1$. So, from $R_0 = 101011$, we get the output 10010111

We now XOR the eight bit output with the key k_1 for the first round. We have

Recall the exclusive OR operator ^ in the C language from the programming course.

$$\begin{array}{r} 10010111 \\ 01010011 \\ \hline 11000100 \\ \hline \end{array}$$

Note that XORing is just bit by bit addition modulo two of the digits of the two numbers.

Next, we split the eight-bit output into two halves and use the **S-boxes** to get two numbers of three bits each, one from each S-box. Let us see how to do this. Splitting 11000100 into two halves, we get 1100 and 0100. We use the

following two S-boxes:

$$S_1 \begin{bmatrix} 101 & 010 & 001 & 110 & 011 & 100 & 111 & 000 \\ 001 & 100 & 110 & 010 & 000 & 111 & 101 & 011 \end{bmatrix}$$

$$S_2 \begin{bmatrix} 100 & 000 & 110 & 101 & 111 & 001 & 011 & 010 \\ 101 & 011 & 000 & 111 & 110 & 010 & 001 & 100 \end{bmatrix}$$

Let us find the three-bit output corresponding to the left half 1100. For this we use the first S-box, namely S_1. Here, the first digit is 1, so we use the second row of S_1. The next three digits are 100 whose decimal value is four. We choose the entry in the fifth column in the second row of S_1 which is 000. In general, if the decimal value of the last three digits is k, we choose the entry in the $k+1$st column. Note that $k \leq 7$ and the number of columns in the S-boxes is eight.

Similarly, for the right half 0100 we use S_2 and choose the entry in the first row, fifth column of S_2 since the first digit is 0 and the next three digits have decimal value 4. The three bit number we get in this case is 111.

Concatenating the two outputs, we get 000111. This is our $f(R_0, k_1)$.

Step 3 Next, we XOR the $f(R_0, k_1)$ with L_0. We get

$$\begin{array}{r} 000111 \\ 101011 \\ \hline 101100 \end{array}$$

Step 4 The answer 101100 is our R_1 and we take R_0 as L_1, i.e. the output is 001101101100. This is the input for the second round and we use the steps 1 to 4 to get L_2R_2 the output for the second round. We can carry out the procedure many times, i.e. carry out many rounds of encryption.

Let us now discuss the method of encryption in general terms. We assume that our key k is nine bits long. In the ith round of the algorithm, we transform an input $L_{i-1}R_{i-1}$ to the output L_iR_i using a key k_i of size eight bits that is derived from the key k.

We use a function $f(R_{i-1}, k_i)$ that takes a six-bit input R_{i-1} and an eight-bit input k_i and produces an output of size six. As we saw, the function is actually a composition of many functions. The first function is the **expander** function. It takes an input of size six bits and 'expands' it to eight bits. Given a six-bit input $a_1a_2a_3a_4a_5a_6$ it outputs the eight-bit number $a_1a_2a_4a_3a_4a_3a_5a_6$.

Now, the key used for encryption enters the picture. The key k consists of nine bits. The k_i for the i^{th} round of encryption is obtained by using eight bits of k, starting with i^{th} bit. If $k = 001100111$, $k_5 = 00111001$. Here, after four bits, we reach the end of the key, so we 'wrap around', start from the beginning and use the first four bits of k.

We then XOR the eight-bit output from the expander part with the key to get another eight-bit number. We split the eight-bit output we get into two halves. We use the left half as the input for the first S-box. Of the four bits, the first bit specifies the row. If the first bit is zero, we choose the first row, if the first digit is one we choose the second row. If the decimal value of the last three bits is k, we choose $k+1$th column. For example, if the last three bits are 011, we choose the fourth column since the decimal value of 011 is three. Similarly, we get the output for the second half using the second S-box. We concatenate the two outputs to get the 6-bit output. We then XOR the six-bit output with L_{i-1} to get R_i.

We define the output for the i^{th} round by $L_i = R_{i-1}$, $R_i = L_{i-1} \oplus f(R_{i-1}, k_i)$. Here $\oplus$ denotes XOR operation. We have shown the encryption operation in Fig. 1 on the facing page. We simply apply this operation a specified number of times, say n to get the output L_nR_n.

Let us now look at an example to understand this.

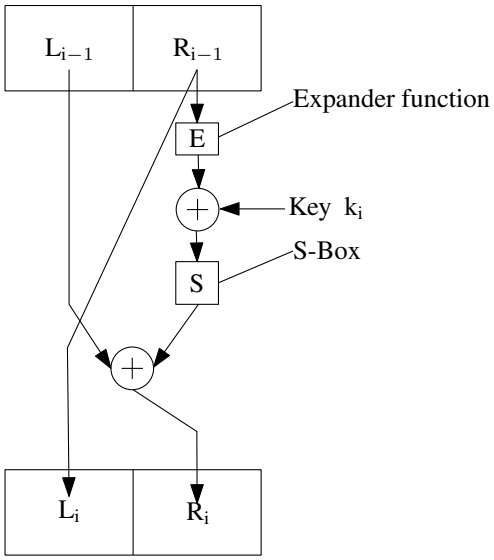


Fig. 1: One round of the encryption procedure.

Example 1: Suppose the key is 110101110 and the input is 111000101011. Carry out two rounds of encryption by using the function f.

Solution:

Step 1 We split the 12 bit input into two halves, R_0 and L_0. Here $L_0 = 111000$ and $R_0 = 101011$.

Step 2 If $a_1a_2a_3a_4a_5a_6 = 101011$, by expanding, we get the number $a_1a_2a_4a_3a_4a_3a_5a_6 = 10010111$.

Step 3 The key for the first round is simply the first eight digits of the key, which is 11010111. So, $k_1 = 11010111$. We XOR the eight-bit number we got by expanding R_0, viz. 10010111, with k_1 to get 01000000. We divide the output into two parts and use the S-boxes. From the left half 0100, we get 011 from the first S-box. From the right half 0000, we get 100 from the second S-box. Concatenating, we get the six-bit number $f(R_0, k_1) = 011100$.

Step 4 To get R_1, we XOR L_0 with $f(R_0, k_1)$:

$$R_1 = L_0 \oplus f(R_0, k_1) = 111000 \oplus 011100 = 100100$$

Since we take $L_1 = R_0$, the output L_1R_1 is 101011100100.

For the second round, we start from the second digit and read off eight-digits to get $k_2 = 10101110$. We leave the remaining details as an exercise to you.

* * *

Let us now summarise the procedure for encryption:

Encryption using Toy Cipher in the ith round

1) Split the 12 bit input into left and right halves, L_i, R_i of size six each. Expand the right half to eight bits according to the rule $a_1a_2a_3a_4a_5a_6 \rightarrow a_1a_2a_4a_3a_4a_3a_5a_6$ where a_i are the bits in R_i.
2) After expanding, XOR the result with the key for the ith round.
3) After XORing split the result into two halves of four bits each. Use the S-boxes on each of the two halves to get two strings of length three each and concatenate them to get a six bit string.
4) Get R_{i+1} by XORing the six bit string obtained in the last step with L_i. $L_{i+1} = R_i$.

Here are some exercises for you.

E1) Complete Example 1 on the previous page by carrying out one more round of encryption using the Toy block cipher.

E2) If the 9 bit key for the Toy block cipher is 011001110 what are the 8-bit keys for the second and fourth rounds of encryption?

We now describe the decryption procedure. Given L_nR_n, we first switch L_n and R_n and obtain R_nL_n. We now use the procedure that we used for encryption, but we use the keys in reverse order, i.e. k_n, k_{n-1}, …, k_1. In the first step, from the input R_nL_n, we get the output $[L_n]\,[R_n \oplus f(L_n, k_n)]$.

From the way the encryption is carried out, we know that

$$L_n = R_{n-1} \text{ and } R_n = L_{n-1} \oplus f(R_{n-1}, k_n).$$

Since $L_n = R_{n-1}$, we have

$$f(R_{n-1}, k_n) \oplus f(L_n, k_n) = f(R_{n-1}, k_n) \oplus f(R_{n-1}, k_n) = 0.$$

Therefore, $R_n \oplus f(L_n, k_n) = L_{n-1} \oplus f(R_{n-1}, k_n) \oplus f(L_n, k_n) = L_{n-1}$. So, under the procedure $[L_n]\,[R_n \oplus f(L_n, k_n)]$ is decrypted as $[R_{n-1}]\,[L_{n-1}]$. Similarly, in the second step $R_{n-1}L_{n-1}$ decrypts to $R_{n-2}L_{n-2}$. Proceeding this way, we get R_0L_0. Switching the left and right halves, we get back the plain text L_0R_0. This is summarised in the box below:

Decryption using the Toy cipher

1) Swap the right and the left halves.
2) Carry out the procedure used for encryption n times using the keys in the reverse order: k_n, k_{n-1}, …, k_1.
3) Swap the right and the left halves.

Let us look at an example to understand the decryption procedure.

Example 2: The text 100100000000 was got by encrypting twice using the key 110101110. Decrypt it using the same key.

Solution:

1) We swap the right and the left halves to get 000000100100.
2) We carry out two rounds of the same process we used for encryption, using the keys in the reverse order.
   First round: The key is $k_2 = 10101110$. Expanding the right half 100100, we get 10101000. XORing with the k_2, we get 00000110. Using the S-boxes, we get 101011. XORing with the left half 000000, we get 101011. Output for the first round is 100100101011.
   Second Round: The key is $k_1 = 11010111$. Expanding 101011, we get 10010111. XORing with the key k_1, we get 01000000. Using the S-boxes, we get 011100. We XOR this with 100100, to get 111000. So, the output of the second round is 101011111000.
3) We swap the left and right halves to get 111000101011. It is worth noting that this is the plain text we started with in Example 1.

* * *

So, for decryption, we use a procedure similar to the encryption procedure. We interchange the left and right halves and use the keys k_i in the reverse order. Since the procedures for encryption and decryption are the same, the sender and receiver use the

same machines. Of course, the receiver has to reverse the left and right inputs. Here is an exercise for you to check your understanding of the decryption process.

E3) Decrypt the text 001011011001 that was encrypted once with the Toy block cipher using the key 010100110.

We conclude this section here. In the next section, we will discuss the DES algorithm.

4.3 THE DATA ENCRYPTION STANDARD

The DES was one of the most popular ciphers for 70s, 80s and early 90s. In a survey of block ciphers [15], written in 1994 , Robshaw remarked "DES is still secure, but it is now at the end of its useful life. It will, however, continue to be used in some other mode, perhaps in some form of triple encryption until an alternative block cipher can be found." It is also one of the most extensively researched ciphers. It has survived many attempts to break it. When it finally become obsolete, it was not because of the cryptanalytic techniques, but because of advances in technology that made it possible to crack the DES by brute force. Because of this, the design principles behind the DES are used in many other modern block ciphers. In this section we will describe the cipher. The discussion in this section is based on Sec. 4.4 and Sec. 4.5 of [21].

In DES, a block of cipher text is 64-bit long and the actual key is 56-bit long. It is padded with 8 more bits to get a 64 bit key by adding parity bits at the 8th, 16th, 24th, …. We choose the parity bits in such a way that each block of 8 bits has an odd number of ones. For example, if there are three 1s and four 0s in the first seven bits, we choose 0 as the eighth bit; if there are, for example, four 1s and three 0s in the first seven bits, we choose 1 as the eighth bit. The parity bits are used for detecting errors.

Suppose $\mathcal{M}$ is a plain text of 64 bits. We encrypt using the DES algorithm as follows:

Initial permutation We apply an initial fixed permutation IP(i.e. IP is independent of $\mathcal{M}$ and the key k) to $\mathcal{M}$ and obtain $\mathcal{M}_0 = \text{IP}(\mathcal{M})$. As we did in the Toy cipher, we divide this into two parts, L_0R_0, each 32 bits long.

Apply round function For $1 \leq i \leq 16$, we do the following:

$$L_i = R_{i-1} \text{ and } R_i = L_{i-1} \oplus f(R_{i-1}, k_i),$$

where k_i is a string of length 64 that we obtain from the key k. We will describe f and the selection of k_i in detail later.

Switch and permute We switch the left and right halves to get $R_{16}L_{16}$, then apply the inverse of the initial permutation to get the cipher text $c = \text{IP}^{-1}(R_{16}L_{16})$.

Note the similarity between this encryption procedure and the encryption procedure in the Toy block cipher.

As we did in the Toy Block Cipher, we decrypt by using the same procedure we use for encryption, but the keys are used in reverse order, k_{16}, k_{15}, …, k_2, k_1. Because of the left-right switch performed in the third step, we need not switch the left-right switch during decryption.

The initial permutation is given in Table 1. According to this table, 58th bit of $\mathcal{M}$ is the first bit of $\mathcal{M}_0$, 50th bit of $\mathcal{M}$ is the second bit of $\mathcal{M}_0$, etc.

Table 1: Initial Permutation

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

We now describe the function $f(R, k_i)$.

Expansion We expand R to E(R) according to Table 2.

Table 2: Expansion permutation

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

This means that the 32nd bit of R is the first bit of E(R), the first bit of R is the second bit of E(R), etc.

XOR with the key We compute $E(R) \oplus k_i$, which has 48 bits, and write it as $B_1B_2 \ldots B_8$ where each B_j is 6 bits long and is the input for S-box S_j.

S-boxes We use the eight S-boxes in Table 3 on the facing page, the first S-box for B_1, the second S-box for B_2, etc. Writing $B_j = b_1b_2 \cdots b_6$, the row of the S-box is specified by b_1b_6 while $b_2b_3b_4b_5$ determines the column. For example, if $B_2 = 101110$, $b_1b_6 = 10$ which is two in base 10. So, we add one to two and choose the third row. Also, $b_2b_3b_4b_5 = 0111$ is seven in base 10. So, we add one to seven and choose the eigth column. The entry in this location in the second S-box is one which is 0001 in binary. In this way, we get eight four-bit outputs $C_1C_2 \cdots C_8$.

Final permutation We permute the bits in the 32-bit number $C_1C_2 \cdots C_8$ according to Table 4. The resulting 32-bit string is $f(R, k_j)$.

Table 4: Final permutation.

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

It remains to explain how to get the keys k_1,k_2, …, k_{16} from the 64 bit key k.

1) We discard the parity bits the permute the remaining bits according to Table 5. Let us write the output as $k_0^1k_0^2$ where k_0^1 and k_0^2 have 28 bits each.

Table 5: Key permutation.

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

2) For $1 \leq i \leq 16$, in the ith round, we get k_i^1 and k_i^2 by shifting the bits in k_{i-1}^1 and k_{i-1}^2 to the left circularly by the number of bits in Table 6. See the appendix at the end of this unit for a discussion of left and right circular shifts.

Table 6: Key shifts

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Shift	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

For example, in the third round, according to the table, we have to shift the bits by two positions. So, the first bit becomes the 27th bit, the second bit becomes the 28th bit and all the bits in positions three to 28th are shifted to the left by two positions. The third bit becomes the first bit, the fourth bit becomes the second bit etc.

Table 3: S-boxes

S-box 1															
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
S-box 2															
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
S-box 3															
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
S-box 4															
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
S-box 5															
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
S-box 6															
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
S-box 7															
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
S-box 8															
13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	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

3) We choose the 48 bits of the key k_i from $k_i^1 k_i^2$ according to Table 7.

Let us now look at an example of encryption by DES. This example is taken from `http://orlingrabbe.com/des.htm`.

Example 3: Let us take the message

$$\mathcal{M} = 0000\ 0001\ 0010\ 0011\ 0100\ 0101\ 0110\ 0111\ 1000\ 1001\ 1010\ 1011\ 1100\ 1101\ 1110\ 1111$$

which is 64 bits long. Let the 64 bit key be

$$00010011\ 00110100\ 01010111\ 01111001\ 10011011\ 10111100\ 11011111\ 11110001$$

Table 7: Key selection

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

After applying the initial permutation to $\mathcal{M}$, we get

$$\mathcal{M}_0 = 1100\ 1100\ 0000\ 0000\ 1100\ 1100\ 1111\ 1111$$
$$1111\ 0000\ 1010\ 1010\ 1111\ 0000\ 1010\ 1010.$$

From this we get

$$L_0 = 11001100000000001100110011111111 \quad (1)$$
$$R_0 = 11110000101010101111000010101010 \quad (2)$$

Using the key permutation in Table 5, we get the following 56-bit key.

$$11110000110011001010101011111010101010110011001111100011111$$

Dividing this into two halves, we get

$$k_0^1 = 1111000\ 0110011\ 0010101\ 0101111$$
$$k_0^2 = 0101010\ 1011001\ 1001111\ 0001111$$

Here are the value of k_0^1, k_0^2, k_1^1, k_1^2, …, k_{16}^1, k_{16}^2. Concatenating the values k_i^1 and k_i^2 in

Table 8: The values of k_i^1 and k_i^2.

i	k_i^1	k_i^2
0	11110000110011001010101111	01010101011001100111100011111
1	11100001100110010101011111	10101010110011001111000111110
2	11000011001100101010111111	01010101100110011110001111101
3	00001100110010101010111111111	01010110011001111000111101101
4	00110011001010101011111111100	01011001100111100011110101101
5	11001100101010101111111110000	01100110011110001111010101101
6	00110010101010111111111000011	10011001111000111101010101101
7	11001010101011111111100001100	01100111100011110101010101110
8	00101010101111111110000110011	10011110001111010101010110011
9	01010101011111111100001100110	00111100011110101010101100111
10	01010101111111110000110011001	11110001111010101010110011001100
11	01010111111111000011001100101	11000111101010101011001100111
12	01011111111000011001100101101	00011110101010101100110011111
13	01111111100001100110010101101	01111010101010110011001111100
14	11111110000110011001010101101	11101010101011001100111110001
15	11111000011001100101010101111	10101010101100110011111000111
16	11110000110011001010101011111	01010101011001100111100011111

Table 8 and using Table 7, we get the 48-bit keys k_1, k_2, …, k_{16} in Table 9. Let us now calculate L_1, R_1 from Eqn. (1) and Eqn. (2). We have
$L_1 = R_0 = 11110000101010101111000010101010$. We have $R_1 = L_0 \oplus f(R_0, k_1)$. Let us first calculate $f(R_0, k_1)$. Expanding R_0 according to Table 2, we get
$E(R_0) = 011110\ 100001\ 010101\ 010101\ 011110\ 100001\ 010101\ 010101$. We then XOR the output with the key k_1:

$E(R_0)$	011110 100001 010101 010101 011110 100001 010101 010101
k_1	000110 110000 001011 101111 111111 000111 000001 110010
	011000 010001 011110 111010 100001 100110 010100 100111

Table 9: Keys for 16 rounds.

k_1	000110 110000 001011 101111 111111 000111 000001 110010
k_2	011110 011010 111011 011001 110110 111100 100111 100101
k_3	010101 011111 110010 001010 010000 101100 111110 011001
k_4	011100 101010 110111 010110 110110 110011 010100 011101
k_5	011111 001110 110000 000111 111010 110101 001110 101000
k_6	011000 111010 010100 111110 010100 000111 101100 101111
k_7	111011 001000 010010 110111 111101 100001 100010 111100
k_8	111101 111000 101000 111010 110000 010011 101111 111011
k_9	111000 001101 101111 101011 111011 011110 011110 000001
k_{10}	101100 011111 001101 000111 101110 100100 011001 001111
k_{11}	001000 010101 111111 010011 110111 101101 001110 000110
k_{12}	011101 010111 000111 110101 100101 000110 011111 101001
k_{13}	100101 111100 010111 010001 111110 101011 101001 000001
k_{14}	010111 110100 001110 110111 111100 101110 011100 111010
k_{15}	101111 111001 000110 001101 001111 010011 111100 001010
k_{16}	110010 110011 110110 001011 000011 100001 011111 110101

We have

$$B_1 = 011000 \qquad B_2 = 010001 \qquad B_3 = 011110 \qquad B_4 = 111010$$
$$B_5 = 100001 \qquad B_6 = 100110 \qquad B_7 = 010100 \qquad B_8 = 100111$$

Using the S-box S_i for B_i, $1 \leq i \leq 8$, we get

$$S_1(B_1)\,S_2(B_2)\,S_3(B_3)\,S_4(B_4)\,S_5(B_5)\,S_6(B_6)\,S_7(B_7)\,S_8(B_8)$$
$$= 0101\ 1100\ 1000\ 0010\ 1011\ 0101\ 1001\ 0111$$

Next, we apply the permutation in Table 4 to
$S_1(B_1)\,S_2(B_2)\,S_3(B_3)\,S_4(B_4)\,S_5(B_5)\,S_6(B_6)\,S_7(B_7)\,S_8(B_8)$. We get

$$f(R_0, k_1) = 0010\ 0011\ 0100\ 1010\ 1010\ 1001\ 1011\ 1011$$

We have $R_1 = L_0 \oplus f(R_0, k_1)$. So,

L_0	1100 1100 0000 0000 1100 1100 1111 1111
$f(R_0, k_1)$	0010 0011 0100 1010 1010 1001 1011 1011
	1110 1111 0100 1010 0110 0101 0100 0100

Proceeding like this, we get

$$L_{16} = 01000011010000100011001000110100$$
$$R_{16} = 00001010010011001101100110010101$$

We then switch the halves apply the permutation IP^{-1} which is the inverse of the permutation IP, to $R_{16}L_{16}$ to get

$$10000101\ 11101000\ 00010011\ 01010100$$
$$00001111\ 00001010\ 10110100\ 00000101$$

* * *

This concludes our discussion of DES.

The design criteria behind DES was not clear because it was not made public. In 1992, IBM published some of the details. See [4]. We give a brief summary below:

1) Each S-box has 6 input and 4 output bits. This was the largest that could be put on one chip in 1974.
2) The relationship between in inputs and outputs of the S-boxes should not be linear. If the relationship is linear, cryptanalysis will be easier.
3) Each row of an S-box contains all numbers from 0 to 15.
4) If two inputs to an S-box differ by 1 bit, the output must differ by at least 2 bits.
5) If two inputs to an S-box differ in their first 2 bits, but have the same last 2 bits, their outputs should not be the same.
6) There are 32 pairs of input having the same XOR. Out of these, no more than 8 pairs should have the same XOR for their outputs.

4.3.1 Modes of Operation

As we saw earlier, DES is a block cipher that encrypts plain texts which are 64-bits long in one go. But, sometimes, it may become necessary to encrypt plain texts of larger or smaller sizes. We can run block ciphers in different modes according to the demands of the application of the ciphers. The following are five common modes:

1) Electronic code cook(ECB).
2) Cipher block chaining(CBC).
3) Cipher feedback(CFB).
4) Output feedback(OFB).
5) Counter(CTR)

All these modes can work with chunks of different sizes. However, in our discussion we will confine ourselves to the case where all the chunks are of 8 bits. Let us discuss these modes one by one.

Electronic code book(ECB) By the definition of a block cipher, we break up the text into blocks of fixed size. This mode of operation is called Electronic code book operation. If $P = P_1P_2\cdots P_k$ is the plain text, then $C = C_1C_2\cdots C_k$ is cipher text where $C_i = E_k(P_i)$.

There is a weakness in this mode. Once we fix a key, the relationship between the plain text and cipher text is fixed. If an adversary is able to decrypt a block of cipher text from the context or by any other means, he/she can decrypt all the occurrences of that cipher text.

Cipher Block Chaining(CBC) In this mode, the encryption of a block depends upon the encryption of the previous blocks and so removes some of the problems in the ECB mode. We use an **initialisation vector(IV)** C_0 in this mode. We encrypt as follows:

$$C_1 = E_k(P_1 \oplus C_0)$$
$$C_i = E_k(P_i \oplus C_{i-1}) \text{ for } i > 1$$

We decrypt as follows:

$$P_i = D_k(C_i) \oplus C_{i-1}$$

Again, we have to change the IVs often to keep the system secure.

Cipher Feedback(CFB): In ECB and CBC, encryption and decryption cannot take place till we receive all the 64 bits of text. In CFB we can encrypt 8 bits of plain text without waiting for all the 64 bits. We break the plain text into 8-bit pieces: $P = P_1P_2\cdots$ where each P_j has eight bits. We encrypt as follows: We choose an initial 64 bit text X_1. Then, for $j = 1,2,3,\ldots$ we encrypt the the P_js:

$$O_j = L_8\left(E_k\left(X_j\right)\right) \tag{3a}$$
$$C_j = P_j \oplus O_j \tag{3b}$$
$$X_{j+1} = R_{56}\left(X_j\right)||C_j \tag{3c}$$

Here $L_8(X)$ denotes the 8 leftmost bits of $X, R_{56}(X)$ denotes the rightmost 56 bits of X, and $X \parallel Y$ denotes the string obtained by writing X followed by Y.

We decrypt as follows:

$$P_j = C_j \oplus L_8\left(E_k\left(X_j\right)\right)$$
$$X_{j+1} = R_{56}\left(X_j\right) \parallel C_j.$$

You would have probably noticed that decryption does not involve calling the decryption function, D_k. This would be an advantage of running a block cipher in a stream mode in a case where the decryption function for the block cipher is slower than the encryption function.

Let us look at one round of encryption using the CFB mode. First, we initialise a 64-bit register with the value of X_1. We then encrypt these 64 bits using E_k. We extract the leftmost 8 bits of $E_k(X_1)$ and XOR it with the 8-bit P_1 to form C_1. We send C_1 to the recipient. We then update the 64-bit register X_1 as follows: We concatenate the rightmost 56 bits of X_1 with 8 bits of C_1 and load the resulting 64 bits into the register. We then encrypt P_2 by the same process, but use X_2 in place of X_1. After encrypting P_2 to C_2, we update the 64-bit register as follows:

A **register** is a memory location in a hardware device. In the context of cryptography, it refers to a device that is used for encryption and decryption.

$$X_3 = R_{56}(X_2) \parallel C_2 = R_{48}(X_1) \parallel C_1 \parallel C_2.$$

After 8th rounds, the initial X_1 has disappeared from the 64-bit register and $X_9 = C_1 \parallel C_2 \parallel \cdots \parallel C_8$. The C_j continue to pass through the register, so for example $X_{20} = C_{12} \parallel C_{13} \parallel \cdots \parallel C_{19}$.

Since CFB mode of transmission can recover form errors in transmission of the cipher text, this mode is useful in practice. Suppose that the transmitter sends the cipher text blocks $C_1, C_2, \ldots, C_k \ldots$, and C_1 is corrupted during transmission, so that the receiver observes $C_1', C_2, \ldots$. On decryption of C_1', the receiver gets a garbled version of P_1 with bit errors in the location that C_1 had bit errors. Since the receiver will get X_2 by concatenating the rightmost 56 bits of X_1 with C_1' and so the value of X_2 obtained also will be wrong. Note that, each time the receiver calculates X_i, the leftmost 8 bits of the register will be pushed out. So,

$$X_2 = \left(v_1, v_2, v_3, v_4, v_5, v_6, v_7, C_1'\right)$$

where v_1, v_2, ... were the values in the shift register before C_1' was received.

$$X_3 = \left(v_2, v_3, v_4, v_5, v_6, v_7, C_1', C_2\right)$$
$$X_4 = \left(v_3, v_4, v_5, v_6, v_7, C_1', C_2, C_3\right)$$
$$\vdots$$
$$X_7 = \left(v_8, C_1', C_2, C_3, \ldots, C_7\right)$$
$$X_8 = \left(C_1', C_2, C_3, \ldots, C_8\right)$$
$$X_9 = \left(C_2, C_3, \ldots, C_8, C_9\right)$$

So, the corrupted cipher is ultimately flushed out of the register and decryption proceeds correctly afterwards.

Output Feedback (OFB): In this mode errors do not propagate like CFB mode where an error in one of the encrypted chunks propagates to the next 8 chunks.

Like CFB, OFB also works on chunks of different sizes. For our discussion, we will focus on the 8-bit version of OFB, where OFB is used to encrypt 8-bit chunks of plain text in a streaming mode. The procedure in OFB is very similar to the procedure in CFB. In, for $j = 1, 2, 3, \ldots$, we encrypt as follows:

$$O_j = L_8\left(E_k\left(X_j\right)\right) \tag{4a}$$
$$X_{j+1} = R_{56}\left(X_j\right) \parallel O_j \tag{4b}$$
$$C_j = P_j \oplus O_j. \tag{4c}$$

Let us compare the methods of encryptions for CFB and OFB modes given by Eqn. (3) and Eqn. (4), respectively. The difference is in computation of X_js. From Eqn. (3b), we see that, in the CFB mode we get X_{j+1} by concatenating the rightmost 56 bits of X_j with $C_j = P_j \oplus E_k\left(X_j\right)$. In OFB mode, from Eqn. (4b), we see that, we get X_j by concatenating the rightmost 56 bits with $O_j = L_8\left(E_k\left(X_j\right)\right)$.

Let us see what is the advantage in OFB when compared with CFB. First, the generation of the O_j output key stream may be performed completely without any plain text. So, the output key stream can be computed in advance thus speeding up the decryption process in those situations where this gain in speed is important. The second advantage is that, when there are errors in some of the bits in the cipher text, this will affect only the corresponding bits in the plain text when we decrypt.

There is a problem with OFB, however, that is common to all stream ciphers that are obtained by XORing pseudo-random numbers with plain text. If an adversary knows a particular plain text P_j and cipher text C_j, he/she can modify the messages. He/She first calculates

$$O_j = C_j \oplus P_j$$

to get out the key stream. He/She may then create any false plain text P'_j he/she wants. Now, to produce a cipher text, he merely has to XOR with output stream he/she calculated:

$$C'_j = P'_j \oplus O_j.$$

Counter (CTR): The counter (CTR) mode builds upon the ideas that were used in the OFB mode. Just like OFB, CTR creates an output key stream that is XORed with chunks of plain text to produce cipher text. However, the CTR mode avoids the problems associated with the OFB mode by delinking the output stream O_j from the previous output streams.

In CTR we start by breaking up the plain text into 8-bit pieces, $P = [P_1, P_2, \ldots]$. We begin with an initial value X_1, which has a length equal to the block length of the cipher, for example, 64 bits of output, and we extract the leftmost 8-bits of the cipher text and XOR it with P_1 to produce 8 bits of cipher text, C_1.

However, for updating the register X_2, we do not use the output of the block cipher, we simply take $X_2 = X_1 + 1$. So, X_2 does not depend on previous output. We create new output stream in CTR by encrypting X_2. Similarly, we proceed by using $X_3 = X_2 + 1$, and so on. We produce the jth cipher text by XORing the left 8 bits from the encryption of the jth register with the corresponding plain text P_j.

In general, the procedure for CTR is

$$\begin{aligned} X_j &= X_{j-1} + 1 \\ O_j &= L_8\left(E_k\left(X_j\right)\right) \\ C_j &= P_j \oplus O_j \end{aligned}$$

for $j = 2, 3, \cdots,$. If we continually add 1 to X_j, it could eventually become too large. This is unlikely to happen, but if it does, we simply wrap around and start back at 0.

We can calculate the registers X_j ahead of time just like OFB and the actual encryption of plain text is simple in that it involves just the XOR operation. As a result, it fares as well as the OFB mode when errors are introduced in the cipher text. The advantage that CTR enjoys over OFB is that we can calculate many output chunks O_J in parallel. We do not have to calculate O_J before calculating O_{J+1}. This makes CTR mode ideal for parallelising.

In the next subsection, we will briefly explain the cryptanalysis techniques used to break the DES. We will also see why the algorithm became obsolete.

4.3.2 Cryptanalysis of DES

DES was the standard cryptographic system from the time it was approved as the standard in the late 90s. It was resistant to differential cryptanalysis, a powerful technique discovered by Shamir and Biham(See [2]). The technique was used successfully against Khafre, REDOC-11, FEAL and LOKI. See [1]. Differential Cryptanalysis can break DES with less than 16 rounds of encryption. The DES was also resistant to linear cryptanalysis, discovered by Matsui(See [9]) in 1994. It turned out that the designers of DES knew about differential and designed DES to be resistant to this attack. However, it seems that they didn't know about linear cryptanalysis.

If this was the situation regarding cryptanalysis techniques, the situation regarding brute force attack was somewhat different. From the time DES was released, the academic community felt that the key length was too small. In fact, a few months after the NBS release of DES, Whitfield Diffie and Martin Hellman published a paper titled "Exhaustive cryptanalysis of the NBS Data Encryption Standard", [5] in which they estimated that a machine, specially built for purpose of attacking the DES, could be built for $20 million. The proposed machine could crack DES in roughly a day. .

By brute force attack, we mean trying all the different 2^{56} possible keys.

Another such device to attack DES was proposed by Michael Wiener in 1993, a researcher at Bell-Northern researcher. This machine used the switching technology used by the telephone industry. In the year 1996, three different approaches were proposed for attacking symmetric ciphers like DES.

1. Distributed computing using a large collection of machines. This was relatively cheap and the cost could distributed amongst many people.
2. Design custom architecture(like Wiener's machine). While this is effective, it is expensive.
3. Using programmable arrays which is somewhere in between the two earlier approaches.

In 1997, RSA Data Security challenged the computing community to find the key and decrypt a message encrypted using the DES. It offered a prize money of $ 10,000 to anybody who succeeds in doing so.

Rock Verser submitted the winning DES key after five months. The noteworthy point in this success is the fact that Rock Verser used distributive computing approach. He used the free time available on the internet. People allowed the use of their computers with the understanding that Verser will share 40% of the prize money with the owner of the computer that found the key. The correct key was found after checking 25% of the possible keys. In 1997, RSA issued another similar challenge. This was cracked much faster by Distributed Computing Technologies, in only 39 days, that too after searching more keys, roughly 85% of the keys.

In 1998, the DES cracker built by the Electronic Frontier Foundation could crack DES in roughly 4.5 days on average. The aim of building the machine was to expose the vulnerability of the DES. The machine cost around $ 200,000 in 1998. See [7].

All these developments signalled the need to find a suitable replacement for DES. One approach was to encrypt using DES more than once. Merkle and Hellman, in [12], showed that encrypting twice with DES is not secure enough. However, encrypting thrice is secure enough and this algorithm is called **Triple DES**. Many other variants of the DES have been proposed. See [19] for a discussion.

Another approach was to develop a new algorithm. This approach lead to the development of the Advanced Encryption standards. We will take this up in the next section.

4.4 THE ADVANCED ENCRYPTION STANDARD

In 1997, the National Institute of Standards and Technology(NIST) called for candidates to replace DES. The following are some of the stipulations made by the NIST:

1. The algorithm should support various key lengths like 128, 192 and 256 bits.
2. The algorithm should work in various kinds of hardware and software.
3. The algorithm should be usable in various modes, ECB, CFB, etc.
4. The algorithm should be widely available on a non-exclusive, royalty-free basis.

The five algorithms that were ultimately shortlisted were

1. MARS from IBM.
2. RC6 from RSA laboratories.
3. Rijndael, proposed by Joan Daeman and Vincent Rijmen.
4. Serpent, proposed by Ross Anderson, Eli Biham, and Lars Knudsen.
5. Twofish, proposed by Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall and Niels Ferguson.

Acceptable pronunciations of Rijndael, according to its designers, are 'Rain doll', 'Reign Dahl' or 'Rhine Dahl'.

Finally, Rijndael was chosen as the Advanced Encryption Standard, the successor of DES. In this section, we will describe this algorithm.

This algorithm, as required by NIST, can be used with 128, 192 and 256 bit keys. Also, it could work in various modes and in various hardware like 8-bit processors used in smart cards and in 32 bit processors in PCs. It can be used in various modes like ECB, CBC, CFB, OFB, and CTR. It is also available worldwide on a non-exclusive, royalty-free basis.

4.4.1 The Basic Algorithm

Rijndael is designed for use with keys of lengths 128, 192, and 256 bits. For simplicity, we'll restrict to 128 bits. First, we give a brief outline of the algorithm, then describe the various components in more detail.

The algorithm consists of 10 rounds when the key has 128 bit, 12 rounds when there are 192 bits, and 14 rounds when the key has 256 bits. Each round has a round key, derived from the original key. There is also a 0th round key, which is the original key. A round starts with an input of 128 bits and produces an output of 128 bits.

There are four basic steps, called **layers**, that are used to form the rounds:

1) The SubBytes Transformation (BS)
2) The ShiftRows Transformation (SR)
3) The MixColumns Transformation (MC)
4) AddRoundKey (ARK)

Putting everything together, we obtain the following:

Rijndael Encryption

1. ARK, using the 0th round key.
2. Nine rounds of BS, SR, MC, ARK, using round keys 1 to 9.
3. A final round: BS, SR, ARK, using the 10th round key.

4.4.2 The Layers

We now describe the steps in more detail. The 128 input bits are grouped into 16 bytes of 8 bits each, call them

$$a_{0,0}, a_{0,1}, a_{0,2}, a_{0,3}, a_{1,0}, a_{1,1}, a_{1,2}, a_{1,3} \ldots, a_{3,3}$$

These are arranged into a 4×4 matrix

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \tag{5}$$

We can regard the entries of the matrix in Eqn. (5) as elements of $\mathbf{F}_{2^8}$. Let us see how. Recall that the finite field $L = \mathbf{F}_{2^8}$ is a vector space of dimension 8 over $\mathbf{F}_2$. So, once we fix a basis for L over $\mathbf{F}_2$, every element of $\mathbf{F}_{2^8}$ can be represented by a 8-tuple of 0s and 1s. Alternately, we can represent an element of $\mathbf{F}_{2^8}$ by a string of length 8 in 0 and 1. For example, we can write the vector $(0,1,1,0,1,0,1,1) \in \mathbf{F}_{2^8}$ as the string 01101011. So, we can represent every element of $\mathbf{F}_{2^8}$ by a byte and conversely, every byte can be regarded as an element of $\mathbf{F}_{2^8}$. If we identify $\mathbf{F}_{2^8}$ with $\mathbf{F}_2[X]/\langle g(X)\rangle$ for some irreducible polynomial $g(X) \in \mathbf{F}_2[X]$ of degree 8, one natural choice for the basis is the set $\left\{\overline{1}, \overline{X}, \overline{X}^2, \ldots, \overline{X}^7\right\}$ where $\overline{X} = X + \langle g(X)\rangle$ and $\overline{1} = 1 + \langle g(X)\rangle$. When there is no possiblity of confusion we will simply write 1 instead of $\overline{1}$.

Let us see how to carry out arithmetic operations with bytes. We identify a byte $a_7a_6\cdots a_1a_0$ with the element

$$a_7\overline{X}^7 + a_6\overline{X}^6 + \cdots + a_1\overline{X} + a_0 \in \mathbf{F}_{2^8} = \mathbf{F}_2[X]/\langle g(X)\rangle.$$

So, we can regard the entries of the 4×4 matrix in Eqn. (5) as elements of L. We can add two such entries by XORing them.

Multiplication is a lot more complicated. We have to find a proper representation of $\mathbf{F}_{2^8}$ for this. In AES, we represent $\mathbf{F}_{2^8}$ as $\mathbf{F}_2[X]/\langle g(X)\rangle$ where

$$g(X) = X^8 + X^4 + X^3 + X + 1.$$

Suppose we want to multiply two bytes, $b = 10110111$ and $b' = 10001100$. Under our representation for $\mathbf{F}_{2^8}$ the byte b corresponds to the element

$$X^7 + X^5 + X^4 + X^2 + X + 1 + \langle g(X)\rangle \in \mathbf{F}_2[X]/\langle g(X)\rangle$$

and b' corresponds to

$$X^7 + X^3 + X^2 + \langle g(X)\rangle \in \mathbf{F}_2[X]/\langle g(X)\rangle.$$

To multiply b and b' we multiply the polynomials

$$X^7 + X^5 + X^4 + X^2 + X^2 + X + 1,\ X^7 + X^3 + X^2 \in \mathbf{F}_2[X]$$

and divide the product by $g(X)$. We leave it to you to verify that

$$\begin{aligned} &\left(X^7 + X^5 + X^4 + X^2 + X^2 + X + 1\right)\left(X^7 + X^3 + X^2\right) \\ &= X^{14} + X^{12} + X^{11} + X^{10} + X^7 + X^6 + X^5 + X^2. \end{aligned}$$

If we divide the polynomial on the RHS in the above equation by $g(X)$, the remainder is

$$X^6 + X^5 + 1 \in \mathbf{F}_2[X]$$

So, $b \cdot b' = 01100001$.

Since $\mathbf{F}_{2^8}$ is a field, every non-zero element is invertible. So, each byte b except the zero byte has a multiplicative inverse; that is, there is a byte b' such that $b \cdot b' = 00000001$. To find the inverse of b, we proceed as follows: Suppose b corresponds to $p(X) + \langle g(X) \rangle$, $p(X) \in \mathbf{F}_2[X]$ of degree less than 8. Since $g(X)$ is irreducible, $(g(X), p(X)) = 1$. So, using extended Euclid's algorithm for finding the gcd of polynomials, we can find $h(X)$, $q(X) \in \mathbf{F}_2[X]$ such that

$$g(X)h(X) + q(X)p(X) = 1.$$

Then, the byte corresponding to $q(X)$ is the inverse of b.

For example, if we take $b = 10001100$ we have

$$\left(X^7 + X^3 + X^2\right)\left(X^7 + X^6 + X^5 + X^4 + X^2 + X + 1\right) + g(X)\left(X^6 + X^5 + X^4 + X^3 + X + 1\right) = 1$$

So, the inverse of $b = 10001100$ is 11110111. Thus, since we can do arithmetic operations on bytes, we can work with matrices whose entries are bytes.

Before you proceed further, try the following exercises to check your understanding.

E4) a) Find the product of the bytes 10011011 and 10001001.

b) Find the inverse of 10001001.

The BytesSub Transformation

This is a non-linear transformation using S-boxes. The purpose is to make the algorithm resistant to differential and linear cryptanalysis attacks.

In this step, each of the bytes in the matrix is changed to another byte by Table 10, called the S-box.

As before, write a byte as 8 bits: $a_7a_6a_5a_4a_3a_2a_1a_0$. We split it into two parts $a_7a_6a_5a_4$ and $a_3a_2a_1a_0$. Note that a four bit number can be represented as a single digit number in hexadecimal system. For example, 1111 is 15 in the decimal system and hence f in the hexadecimal system. Look for the entry in the $a_7a_6a_5a_4$ row and $a_3a_2a_1a_0$ column (the rows and columns are numbered using the hexadecimal digits 0, 1, 2, 3, …, 9, a, b, c, …, f.). This entry, when converted to binary, is the output. For example, suppose the input byte is 10001010. Then, 1000 is 8 in the decimal system and it is represented by 8 in the hexadecimal system also. We have that 1010 is 10 in decimal and it is represented by a in the hexadecimal system. We look in the row labelled 8 and the column labelled a. The entry is 7e which is $7 \cdot 16 + 14 = 126$ in decimal and 1111110 in binary. This is the output of the S-box.

The output of SubBytes is again a 4×4 matrix of bytes, let's call it

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

The ShiftRows Transformation is a linear mixing step for diffusion of the bits over multiple rounds. It uses arithmetic over the finite field with 2^8 elements. The four rows of the matrix are shifted cyclically to the left by offsets of 0, 1, 2, and 3, to obtain

$$\begin{pmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{pmatrix} = \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,1} & b_{1,2} & b_{1,3} & b_{1,0} \\ b_{2,2} & b_{2,3} & b_{2,0} & b_{2,1} \\ b_{3,3} & b_{3,0} & b_{3,1} & b_{3,2} \end{pmatrix}$$

Table 10: S-Box for the AES

		X														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Y	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

The MixColumns Transformation layer has a purpose similar to ShiftRows. We regard a byte as an element of $\mathbf{F}_{2^8}$ as we explained earlier. Then the output of the ShiftRows step is a 4×4 matrix $\left(c_{i,j}\right)$ with entries in $\mathbf{F}_{2^8}$. We then multiply this by a matrix with entries in $\mathbf{F}_{2^8}$, to produce the output $\left(d_{i,j}\right)$, as follows:

$$\begin{pmatrix} 00000010 & 00000011 & 00000001 & 00000001 \\ 00000001 & 00000010 & 00000011 & 00000001 \\ 00000001 & 00000001 & 00000010 & 00000011 \\ 00000011 & 00000001 & 00000001 & 00000010 \end{pmatrix} \begin{pmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{pmatrix}$$
$$= \begin{pmatrix} d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \end{pmatrix}$$

The RoundKey Addition: In AddRoundKey layer we XOR the round key with the result of the above layer. The round key, derived from the key in a way we'll describe later, consists of 128 bits, which are arranged in a 4×4 matrix $\left(k_{i,j}\right)$ consisting of bytes. This is XORed with the output of the MixColumns step:

$$\begin{pmatrix} d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \end{pmatrix} \oplus \begin{pmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{pmatrix} = \begin{pmatrix} e_{0,0} & e_{0,1} & e_{0,2} & e_{0,3} \\ e_{1,0} & e_{1,1} & e_{1,2} & e_{1,3} \\ e_{2,0} & e_{2,1} & e_{2,2} & e_{2,3} \\ e_{3,0} & e_{3,1} & e_{3,2} & e_{3,3} \end{pmatrix}$$

This is the final output of the round.

The Key Schedule

We arrange the original key, which consists of 128 bits, into a 4×4 matrix of bytes. We then add 40 more columns to this matrix as follows. Let us label the first four columns $W(0), W(1), W(2), W(3)$. We then generate the new columns recursively. Suppose columns up through $W(i-1)$ have been defined. If i is a not a multiple of 4, then

$$W(i) = W(i-4) \oplus W(i-1).$$

If i is a multiple of four then

$$W(i) = W(i-4) \oplus T(W(i-1)).$$

where $T(W(i-1))$ is the transformation of $W(i-1)$ obtained as follows. Let the elements of the column $W(i-1)$ be a, b, c, d. We shift these cyclically to obtain b, c, d, a. Next, we replace each of these bytes with the corresponding element in the S-box from the SubBytes step, to get 4 bytes e, f, g, h. Finally, we compute the round constant.

$$r(i) = 00000010^{(i-4)/4}$$

in GF (2^8) (recall that we are in the case where i is a multiple of 4). Then $T(W(i-1))$ is the column vector

$$(e \oplus r(i), f, g, h).$$

In this way, we generate columns $W(4), \ldots, W(43)$ from the initial four columns.

The **round key** for the ith round consists of the columns

$$W(4i), W(4i+1), W(4i+2), W(4i+3).$$

The Construction of the S-Box

Although the S-box is implemented as a lookup table, it has a simple mathematical description. We start with a byte $a_7a_6a_5a_4a_3a_2a_1a_0$, where each a_i is a binary bit. We compute its inverse as we described earlier. If the byte is 00000000, it has no inverse, so we use 00000000 in place of its inverse. The resulting byte $b_7b_6b_5b_4b_3b_2b_1b_0$ represents an eight-dimensional column vector, with the rightmost bit b_0 in the top position. We multiply this column vector by a matrix and add the column vector $(1,1,0,0,0,1,1,0)$ to obtain a vector $(c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7)$ as follows:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{pmatrix}$$

The byte $c_7c_6c_5c_4c_4c_3c_2c_1c_0$ is the entry in the S-box. For example, suppose we start with the byte 11001011. Its inverse in $\mathbf{F}_{2^8}$ is 00000100. We have

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

This yields the byte 00011111. The first four bits 1100 represent 12 in binary and the last 4 bits 1011 represent 11 in binary. We add 1 to each of these numbers (since the first row and column are numbers 0) and look in the 13th row and 12th column of the S-box. The entry is 31, which in binary is 00011111.

Some of the considerations in the design of the S-box were the following. To make choice of an element from the S-box non-linear, the designers of the AES chose the map $x \mapsto x^{-1}$. Since this map was very simple, it was possibily vulnerable to certain attacks, so it was combined with multiplication by the matrix and adding the vector, as described previously. The matrix was chosen mostly because of its simple form (note how the rows are shifts of each other). The vector was chosen so that no input ever equals its S-box output or the complement of its S-box output (complementation means changing each 1 to 0 and each 0 to 1).

4.4.3 Decryption

Each of the steps SubBytes, ShiftRows, MixColumns, and AddRoundKey is invertible:

1. The inverse of SubBytes is another lookup table, called **InvSubBytes**.
2. The inverse of ShiftRows is obtained by shifting the rows to the right instead of to the left, yielding **InvSubBytes**.
3. The inverse of MixColumns exists because the 4×4 matrix used in MixColumns is invertible. The transformation **InvMixColumns** is given by multiplication by the matrix

$$\begin{pmatrix} 00001110 & 00001011 & 00001101 & 00001001 \\ 00001001 & 00001110 & 00001011 & 00001101 \\ 00001101 & 00001001 & 00001110 & 00001011 \\ 00001011 & 00001101 & 00001001 & 000011101 \end{pmatrix}$$

4. AddRoundKey is its own inverse.

The Rijndael encryption consists of the steps

ARK
BS, SR, MC, ARK
...
BS, SR, MC, ARK
BS, SR, ARK.

Recall that MC is missing in the last round.

To decrypt, we need to run through the inverses of these steps in the reverse order. This yields the following preliminary version of decryption:

ARK, ISR, IBS
ARK, IMC, ISR, IBS
...
ARK, IMC, ISR, IBS
ARK.

We conclude our discussion of AES here.

4.5 SUMMARY

In this unit, we discussed the following:

1) We defined what are block ciphers and stream ciphers;
2) We disussed the general design principles behind block ciphers as enunciated by Claude Shannon;
3) We saw how to encrypt and decrypt using a toy block cipher;
4) We saw how DES encrypts and decrypts messages;
5) We saw various modes of operation of block ciphers;
6) We saw why DES had to be replaced; and
7) We discussed how the AES encrypts and decrypts messages;

4.6 SOLUTIONS/ANSWERS

E1) **Step 1** Split the input into two halves 101011 and 100100.

Step 2 Expand the right half to 10101000.

Step 3 XOR with the key 10101110 to get 00000110. Divide into two parts and use the S-boxes:

$0000 \rightarrow$ First element in the first row of S_1 : 101
$0110 \rightarrow$ Seventh element in the first row of S_2 : 011

Concatenate to get $f(R_1, k_2) = 101011$.

Step 4 To get R_2, XOR $f(R_1, k_2)$ with L_1:

$$R_2 = f(R_1, k_2) \oplus L_1 = 101011 \oplus 101011 = 000000.$$

So, R_2 is 000000 and $L_2 = R_1 = 100100$

E2) The key k_2 for the second round is 11001110. The key for the fourth round is 00111001.

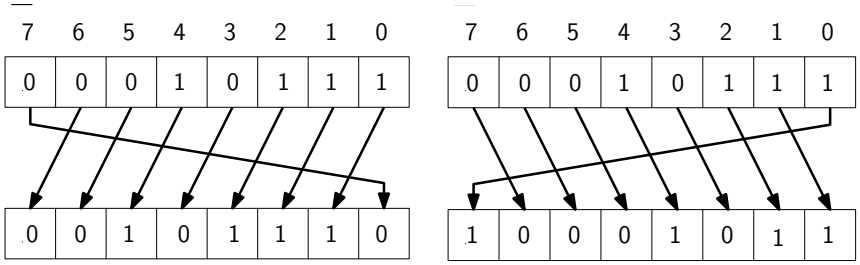
E3) The plain text is 000110001011

E4) a) 10011010.

b) 10011101

APPENDIX: LEFT AND RIGHT CIRCULAR SHIFTS

In this appendix, we will discuss the **right and left circular shifts**. The operator rotates the bits to the left or to the right. The symbol for the left shift operator is $\lll$ and the symbol for the right shift operator is $\ggg$. We write x $\lll$ n(resp. x $\ggg$ n) to denote the shift of the bits by n positions to the left(resp. n positions to the right). For example, x $\lll$ 1 shifts the bits to the left by one place and puts the left most bit at the right most position. Fig. 2(a) shows what happens to the bits in this case. Similarly, Fig. 2(b) shows what happens when we rotate the bits by one place to the right. Note that, x $\lll$ n(resp. x $\ggg$ n) has the same effect as applying x $\lll$ 1(resp. x $\ggg$ 1) n times.



(a) Left circular shift. **(b) Right circular shift.**

Fig. 2: Right and left circular shifts.

```
/* Assumes that the number of bits in an unsigned int is 32 */
/*Left circular shift*/
unsigned int_rotl(unsigned int value, int shift) {
    if ((shift &= 31) == 0)
      return value;
    return (value << shift) | (value >> (32 - shift));
}
/*Right circular shift*/
unsigned int_rotr(unsigned int value, int shift) {
    if ((shift &= 31) == 0)
      return value;
    return (value >> shift) | (value << (32 - shift));
}
```

Listing 4.1: C function for left and right circular shifts. Source: Wikipaedia

UNIT 5 STREAM CIPHERS

Structure **Page No.**

5.1 Introduction 29
Objectives
5.2 Linear Recurrences and Linear Feedback Shift Register 30
5.3 Statistical Tests for Pseudo-Random Number Generators 39
5.4 Stream Ciphers 43
5.5 Summary 45
5.6 Solutions/Answers 46

5.1 INTRODUCTION

> *Block ciphers operate with a fixed width transformation on large blocks of plaintext data; stream ciphers operate with a time-varying transformation on individual plaintext digits.*
>
> — *Rueppel*

In the previous Unit, we discussed block ciphers. They work on blocks, i.e. they encrypt block by block. In this section, we will discuss **stream ciphers**. In stream ciphers, we encrypt text character by character or bit by bit. We can regard the one-time pad, invented in 1917, as the first example of a stream cipher. However, as we will see in this Unit, one-time pad is difficult to use. The ideas behind the design of the one-time pad were used between the two world wars to develop rotor machines for encryption. These machines were used in the Second World War by both the sides. The most famous of the rotor machines was the ENIGMA used by the Germans. We can consider all these machines as examples of stream ciphers.

In the case of block ciphers, the DES dominated the scene and the cryptanalysis was focused mainly on this. However, in the case of stream ciphers a wide variety of ciphers are available. This is because the basic components of stream ciphers are very simple. Researchers have analysed these basic components of stream ciphers extensively. So, highly developed techniques are available for both design and analysis of stream ciphers.

In Sec. 5.2 of this Unit, we begin with a brief description of the one-time pad. We then discuss pseudo-random number generators and the Linear Feedback Shift Register, a simple but fast method for generating pseudo-random numbers. In Sec. 5.3, we discuss some statistical tests for pseudo-random number generators. In Sec. 5.4, we discuss stream ciphers.

Objectives

After studying this unit, you should be able to

- explain what is a pseudo-random number sequence;
- explain what is a cryptographically secure pseudo-random bit generator;
- explain how to generate pseudo-random numbers using LFSRs;
- explain the statistical testing of randomness; and
- explain how the RC4 cipher works.

5.2 LINEAR RECURRENCES AND LINEAR FEEDBACK SHIFT REGISTER

In 1917, Gilbert Vernam, who was working in AT& T, invented an encryption system based on teletype technology. He patented it in 1919. This was an electrical device which combined each character of the plain text with a character on a character tape. Joseph Mauborgne, who was a captain in the US army realised that if the characters are chosen at random, the encryption system was very hard to break. However, the random string of characters should be used only once; otherwise, it is possible to break the system.

In his famous paper, [20], Shannon proved that this system offers perfect secrecy. You are probably wondering why this system is not used. This is because the key has to be of the same length as the message, so it is too long when the message is long. It is very difficult to generate long truly random sequences of numbers. True random number sequences are generated using natural phenomena like decays of radioactive particles or thermal noise from a semiconductor resistor or tossing a coin. Such random numbers are highly unpredictable. Even if we know many numbers in the sequence, the probability that we can predict the next number correctly is very low. However, such random numbers are difficult to generate in large quantities that we need for the one-time pad. Also, securely storing and transmitting large number of bits are quite important.

One way around this problem is to use pseudo-random bit generators. This is the idea behind stream ciphers. Stream ciphers are somewhat similar to the one time pad. Recall that, in stream ciphers also, we use a **key stream** $k_1k_2k_3\ldots$. But, instead of true random numbers, we use **pseudo-random numbers** as the key stream in stream ciphers. They share many of the properties of true random numbers, but they are generated using some mathematical function. We call such a mathematical function a **pseudo-random number generator(PRNG)**. A pseudo-random number generator generates a large sequence of numbers based on the initial input called the **seed**. Once the seed is known, we can generate the entire sequence. Typically, in a stream cipher, the seed used to generate the pseudo-random number sequence is the encryption key. Apart from stream ciphers, PRNGs are also useful for generating keys for algorithms like the AES, DES, etc and for padding messages.

We now define the terminology that we will use in the rest of this section.

Definition 2: A **random bit generator** is a device or algorithm which outputs a sequence of statistically independent and unbiased binary digits.

Note that, we can use a random bit generator to generate random numbers that are uniformly distributed. For example, we can generate a number in $\left[0, 2^{32}\right]$ by generating 32 bits and converting it to an integer. We can discard all the integers that are greater than 2^{32}.

Definition 3: A **pseudo-random bit generator(PRBG)** is a deterministic algorithm which, given a truly random binary sequence of length k, outputs a binary sequence of length l much larger than k which "appears" to be random. The input to the PRBG is called the **seed**, while the output of the PRBG is called a **pseudo-random bit sequence**.

Given the bits generated using a PRBG, we would like that it should be computationally infeasible to find the seed by searching all the 2^k possible choices for it. So, our k has to be sufficiently large. We would like that the bits generated by a PRBG be indistinguishable from a random number sequence and no adversary with limited computational resources should be able to predict the bits of the PRBG. We formally state these requirements in the form of the next two definitions.

Definition 4: We say that a pseudo-random bit generator **passes** all polynomial-time statistical tests if no polynomial-time algorithm can correctly distinguish between an output sequence of the generator and a truly random sequence of the same length with probability significantly greater than $\frac{1}{2}$.

Definition 5: We say that a pseudo-random bit generator **passes** the next-bit test if there is no polynomial-time algorithm which, on input of the first l bits of an output sequence s, can predict the $(l+1)$st bit of s with probability significantly greater than $\frac{1}{2}$.

We can prove that a pseudo-random bit generator that passes the next-bit test passes all polynomial-time statistical tests.

Definition 6: A PRBG that passes the next-bit test (possibly under some plausible but unproved mathematical assumption such as the intractability of factoring integers) is called a **cryptographically secure pseudo-random bit generator (CSPRBG)**.

One of the popular methods for generating pseudo-random numbers is the **linear congruential generator**, invented by D. H. Lehmer in 1949. This produces a sequence of numbers $x_1, x_2, \cdots$, where

$$x_n = ax_{n-1} + b \pmod{m}, \; n \geq 1. \tag{1}$$

The sequence $\{x_n\}$ starts repeating after m terms, if not earlier. A random sequence $x = x_1, x_2, \ldots$, is **periodic** if $x_{n+r} = x_n$ for all $n \geq 1$. The smallest r for which $x_{n+r} = x_n$ for all n is called the period of the sequence. The output of this generator depends on the **initial seed** x_0, and the numbers a, b, and m. If we choose a, b and m correctly, we can generate pseudo-random numbers using this congruence with maximum possible period. The following theorem tells us how to choose a, b and m.

Theorem 1: If $b \neq 0$, the linear congruence generator in Eqn. (1) generates a sequence of length m if and only if

1) b and m are relatively prime.
2) $a-1$ is divisible by all the prime factors of m.
3) $a-1$ is a multiple of 4 if m is a multiple of 4.

You can use the output from linear congruential generators for experimental purposes. For example, if you want to carry out tests on the running time of a sorting algorithm you can use the linear congruential generator to generate the test data. However, they are not very useful for cryptographic purposes. This is because, if we know a few numbers of the sequence, we can predict with high probability the numbers in the sequence even if we do not know a, b and m. In fact, it can be proved that any polynomial congruential generator is cryptographically insecure. For a thorough discussion of linear congruential generators, we refer you to [8].

In cryptographic applications, we need a cryptographically secure random bit generator. One set of candidates for CSPRBGs are the **one-way functions**. These are functions $f(x)$ with the property that we can easily compute $f(x)$ for a given x, but, for given y, it is computationally infeasible to find a x such that $f(x) = y$. We use such functions to generate pseudo-random number sequences as follows: We choose a random seed s and define $x_j = f(s+j)$ for $j = 1, 2, 3, \ldots$. This is a proven and practical method for pseudo-random number generation. Two examples of CSPRBCs are

1) RSA pseudo-random bit generator.
2) BBS(Blum-Blum-Shub) pseudo-random generator.

Let us now discuss the RSA pseudo-random bit generator. In this, we choose two large primes p and q and take their product n. Let $\phi = \phi(n) = (p-1)(q-1)$. We choose a random e such that $0 < e < \phi$ and $(e, \phi) = 1$. We choose a random seed x_0,

$1 \leq x_0 \leq n-1$ and for $i = 1, 2, \ldots, l$, we let $x_i \equiv \mod x_{i-1}^e \pmod n$ and let b_i be the least significant bit of x_i. Then, we get l random bits b_1, b_2, …, b_l.

Let us now look at an example to understand the RSA pseudo-random generator.

Example 4: Let $p = 311009$, $q = 411001$. Then $n = 127825010009$ and $\phi(n) = 127824288000$. Let us choose $x_0 = 311021$ and $e = 1931$. Then, the next 8 values are

$$x_0 = 311021$$
$$x_1 = 43952889436$$
$$x_2 = 1443650955$$
$$x_3 = 94597186059$$
$$x_4 = 107971367060$$
$$x_5 = 97956752491$$
$$x_6 = 72009624215$$
$$x_7 = 125065550422$$
$$x_8 = 61609613347$$

Now, we find b_i as follows:

$$b_i = \begin{cases} 0 & \text{if } x_i \text{ is even.} \\ 1 & \text{if } x_i \text{ is odd.} \end{cases} . \tag{2}$$

Accordingly, we get $b_1 = 0$, $b_2 = 1$, $b_3 = 1$, $b_4 = 0$, $b_5 = 1$, $b_6 = 1$, $b_7 = 0$ and $b_8 = 1$.

* * *

We now explain the **Blum-Blum-Shub (BBS) pseudo-random bit generator**, also known as the quadratic residue generator. In this scheme, we generate two large primes p and q that are both congruent to 3 mod 4. We set $n = pq$ and choose a random integer x that is relatively prime to n. To initialise the BBS generator, we set the initial seed to $x_0 \equiv x^2 \pmod n$. The BBS generator produces a sequence of random bits $b_1, b_2, \cdots$ by

1. $x_j \equiv x_{j-1}^2 \pmod n$
2. b_j is the least significant bit of x_j.

Let us look at an example to understand the BBS generator.

Example 5: Let $p = 311009$, $q = 411001$ and $x = 311021$. Then $n = 127825010009$. Then, the next 8 values are

$$x_0 = 96734062441$$
$$x_1 = 49181739960$$
$$x_2 = 116601795484$$
$$x_3 = 87163050345$$
$$x_4 = 65384689558$$
$$x_5 = 85100903044$$
$$x_6 = 24926545774$$
$$x_7 = 98076507435$$

Using Eqn. (2), the values of b_0, b_1, b_2, b_3, b_4, b_5, b_6 and b_7 are, respectively, 1, 0, 0, 1, 0, 0, 0, 1.

* * *

The security of the BBS generator depends on the fact that it is difficult to factorise n if we choose p and q carefully. See [8], page 36 and Section 3.5F for more details.

While the BBS generator is secure, it is not very fast. In many situations involving encryption, there is a trade-off between speed and security. If one wants a very high level of security, speed is often sacrificed, and vice-versa. For example, in cable television, the picture is encrypted to make sure that only the legitimate subscribers can access the signal. Since images involve huge amount of data, we need a fast way of encryption. Breaking the encryption is very costly when compared to the subscription, so it is not viable to attack the system. In such situations, we can generate pseudo-random number sequences very fast using certain linear recurrences.

Let p be a prime. Consider the recurrence

$$x_{n+k} \equiv a_{k-1}x_{n+k-1} + a_{k-2}x_{n+k-2} + \cdots + a_0 x_n \pmod{p} \tag{3}$$

where $a_i \in \mathbf{F}_p$. The polynomial

$$x^k - a_{k_1}x^{k-1} - \cdots - a_{k-2}x - a_0 \tag{4}$$

is called the **characteristic polynomial** of the recurrence in Eqn. (3). Then, we know from the theory of finite fields that we can choose $a_0, a_1, \ldots, a_{k-1}$ and the first k values $x_1, x_2, \ldots, x_k \in \mathbf{F}_p$ in such a way that the sequence $\{x_n\}$ has period $p^k - 1$, i.e. the sequence repeats itself only after $p^k - 1$ values. We can choose $x_1, x_2, \ldots, x_k$ arbitrarily as long as they are not all zero. We have to choose $a_0, a_1, \ldots, a_{k-1}$ in such a way that the characteristic polynomial $x^k - a_{k-1}x^{k-1} - \cdots - a_{k-2}x - a_0$ is a **primitive polynomial**. Recall that it means that a root of the polynomial generates the multiplicative group of the finite field $\mathbf{F}_{p^k}$.

Let us call the first k elements of the sequence as initial vectors and write the initial values in the vector form as $(x_1, x_2, \ldots, x_k)$. Let us now look at an example, let us take $p = 2$ and $k = 5$. Let the linear recurrence relation be

$$x_{n+5} \equiv x_{n+2} + x_n \pmod{2}. \tag{5}$$

Its characteristic polynomial $x^5 + x^3 + 1$ is an irreducible polynomial over $\mathbf{F}_2$ and its root generates the finite field $\mathbf{F}_{2^5}$. Why is this so? Note that, the multiplicative group of $\mathbf{F}_{2^5}$ has 31 elements, i.e. it is of prime order. So, any non-zero element will generate the multiplicative group. In particular, any root of the polynomial $x^5 + x^3 + 1$ will generate the multiplicative group.

Consider the sequence

$$0100001001011001111100011011101 01. \tag{6}$$

We can generate this sequence by choosing $(0,1,0,0,0)$ as the initial vector and generate the terms $x_5, x_6, \ldots$ using the recurrence relation Eqn. (5).

We use the resulting sequence of 0s and 1s as the key for encryption by XORing with the plain text. For example, if we want to encrypt the plain text 1011001110001111 which of length 16. We choose the first 16 terms of the sequence in Eqn. (6), and XOR it with the plain text to get the following:

(plaintext)	1111000111010110
(key) +	0100001001011001
(ciphertext)	1011001110001111

We decrypt by adding the key sequence to the ciphertext in exactly the same way.

Note that the sequence in Eqn. (6) is periodic of period 31. It starts repeating from the 32nd term which is the same as the first term. The 33rd term is the same as the second term and so on. We got this by specifying the initial vector $(0,1,0,0,0)$ and the coefficients $a_0 = 1$, $a_1 = 0$, $a_2 = 1$, $a_3 = 0$ and $a_4 = 0$. So we could produce 31 bits using 5 bits. The polynomial $x^{31} + x^3 + 1$ is an irreducible polynomial over $\mathbf{F}_2$. Further

$2^{31} - 1 = 2147483647$ is a prime. Using the argument we used before, it follows that $x^{31} + x^3 + 1$ is a primitive polynomial. So, the recurrence

$$x_{n+31} \equiv x_{n+3} + x_n. \tag{7}$$

and any non-zero initial vector will produce a sequence that has period $2^{31} - 1 = 2147483647$. Therefore, 62 bits produce more than two billion bits of key. Thus, we can generate a key with large period using very little information. Compared to a one-time pad, where the full two billion bits must be sent in advance, this is a great advantage.

Remark 1: Note that, $2^5 - 1$, $2^{31} - 1$ are all examples of **Mersenne primes**, i.e. primes of the form $2^p - 1$ where p is a prime. So, if $2^p - 1$ is a Mersenne prime, any irreducible polynomial of degree p over $\mathbf{F}_2[x]$ will be a primitive polynomial.

The method of generating pseudo-random number sequence using linear recurrences, when we implement in hardware using what is known as an **linear feedback shift register** LFSR, is very fast. Linear Feedback Shift Registers(LFSRs) are hardware devices used for encryption. LFSRs generate pseudo-random numbers very fast and encrypt text by XORing with the plaintext with the pseudo-random numbers they generate. However, we will also see why this method of encryption is weak. The main components of a feedback shift register(see Fig. 1) are:

1) n-stage shift register with 2-state storage units.
2) Initial state.
3) Feedback function.

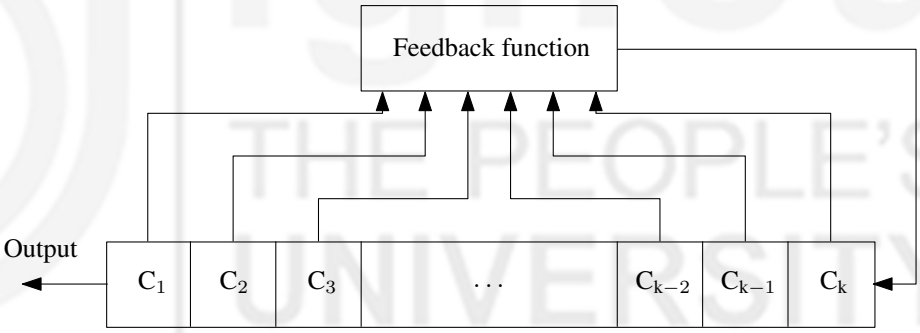


Fig. 1: Schematic diagram of a feedback shift register.

In this register, some cells are designated as taps and their values are passed on to the feedback function in each clock cycle. In each clock cycle, the content of each cell is shifted to the neighbouring cell and the contents of the first cell from the left is given as the output and shifted out of the register. The value of the feedback function is then fed into the first cell from the right which became empty because its contents were shifted to the cell next to it. When feedback function is linear, the feedback shift register is called a linear feedback shift register or LFSR in short. In this case we XOR the contents of the cells designated as taps and feed it into the last cell.

The schematic diagram of the LFSR that generates a pseudo-random bit sequence using the congruence $x_{m+3} \equiv x_{m+1} + x_m \pmod 2$ is shown in Fig. 2.

Suppose we think of LFSR as having three cells labelled C_1, C_2 and C_3 and each of them holds a bit. In this LFSR, we designate the cells C_1 and C_2 as taps. In each clock cycle, the bits in the 'boxes' in the LFSR in Fig. 2 change as follows:

New value of C_1 = Old value of C_2

New value of C_2 = Old value of C_3

New value of C_3 = Old value of $C_1 \oplus$ Old value of C_2

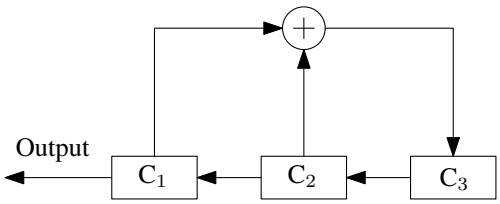


Fig. 2: A linear feedback shift register satisfying $x_{m+3} \equiv x_{m+1} + x_m \pmod 2$

For example, suppose we choose $x_1 = 1$, $x_2 = 0$ and $x_3 = 1$ as the initial vector. So, to start with, the cell labelled C_1 contains the value 1, the cell labelled C_2 contains the value 0 and the cell labelled C_3 contains the value 1. At the end of the first clock cycle, the following happens:

1) The value of C_1, which is 1, is given as output.

2) The value of C_2, which is 0, is shifted to C_1.

3) The values of C_1 and C_2 are XORed and the answer one is stored in C_3.

This process is repeated again and again.

More generally, if the recurrence is

$$x_{n+k} \equiv a_{k-1}x_{n+k-1} + a_{k-2}x_{n+k-2} + \cdots + a_0 x_n \pmod 2$$

the LFSR has k cells C_1, C_2, $\cdots$, C_{k-1}, C_k and the ith cell C_i is designated as a tap if $a_{i-1} = 1$. At the end of each clock cycle, the value in C_1 is given as output, the contents of each of the cells C_2, C_3, …, C_k are shifted to the adjacent cells on the left and the values in the tapped cells are XORed together(before they are shifted to the left) and put in the cell C_k.

Let us call

$$S_n = (x_{n+1}, x_{n+2}, \ldots, x_{n+k})$$

the nth **state vector**. Then, S_n describes the contents of the cells C_1, C_2, …, C_k after n clock cycles. Thus, S_0 describes the initial state $(x_1, x_2, \ldots, x_k)$ and is called the **initial state vector**. If we use a LFSR for generating the sequence given by the relation in Eqn. (5), the initial state vector will be $(0,1,0,0,0)$.

To test your understanding of LFSRS try the following exercise now.

E1) Draw the LFSR circuit for the recurrence relation in Eqn. (5).

E2) Give the recurrence relation corresponding to the LFSR given below:

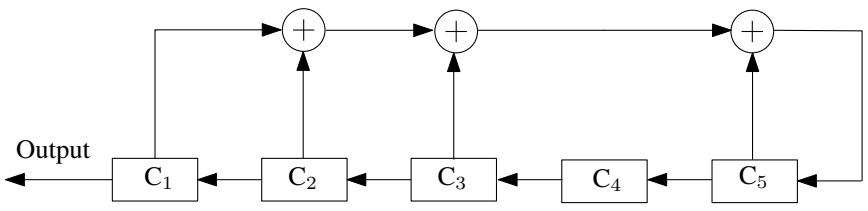


Fig. 3: Figure for exercise 2.

However, encryption using LFSRs is not strong. We can easily break it using a known plain text attack. Suppose we know only a few consecutive bits of plain text, along with the corresponding bits of ciphertext. If we subtract(or add) the plaintext and the ciphertext $\pmod 2$ we obtain a few bits of the key. From this few bits of the key we

Note that addition and subtraction are the same $\pmod 2$.

can find out the recurrence and hence the remaining bits of the key also. Let us now see how we can do this through an example.

Example 6: Suppose we know the initial segment 010110010001 of the sequence 010110010001111 …, which has period 15, and suppose we know it is generated by a linear recurrence. Let us see how we can determine the coefficients of the recurrence.

Since we do not know the length, let us try out recurrences of all possible lengths. We can start with a recurrence of length two since a recurrence of length one yields a constant sequence. Suppose the recurrence is

$$x_{n+2} = c_0 x_n + c_1 x_{n+1}. \tag{8}$$

We set $n = 1$ and $n = 2$ and use the known values $x_1 = 0, x_2 = 1, x_3 = 0, x_4 = 1$. We obtain the equations

$$\begin{aligned} 0 &\equiv c_0 \cdot 0 + c_1 \cdot 1 && (n = 1) \\ 1 &\equiv c_0 \cdot 1 + c_1 \cdot 0 && (n = 2). \end{aligned}$$

Let us write the equations above in matrix form.

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} \equiv \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

Solving the equations, we get $c_0 = 1, c_1 = 0$, so the recurrence could be $x_{n+2} \equiv x_n \pmod 2$. Since $x_5 \not\equiv x_3 \pmod 2$, our guess is not correct. Therefore, let us try length 3. The resulting matrix equation is

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} \equiv \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}.$$

The solution to this equation is $c_0 = 1$, $c_1 = 1$ and $c_2 = 0$, so the recurrence could be $x_{n+3} \equiv x_n + x_{n+1} \pmod 2$. However, our guess turns out to be wrong again because for $n = 3$ we should have $x_6 \equiv x_3 + x_4 \pmod 2$, but this is not true.

Now let us consider length 4. The matrix equation is

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} \equiv \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

The solution is $c_0 = 1, c_1 = 0, c_2 = 0, c_3 = 1$. Our guess for the recurrence is

$$x_{n+4} \equiv x_n + x_{n+3}.$$

Finally, we have hit upon the correct recurrence because we can generate the remaining elements of the key using this recurrence.

* * *

Here is an exercise for you to check your understanding of Example 6.

E3) Given the initial sequence 110010111001, find the recurrence that generates it.

Here is what we do in general: Suppose we know 2m bits. Then, we can test for a recurrence of length m. We set up the following matrix equation:

$$\begin{pmatrix} x_1 & x_2 & \cdots & x_m \\ x_2 & x_3 & \cdots & x_{m+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_m & x_{m+1} & \cdots & x_{2m-1} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix} \equiv \begin{pmatrix} x_{m+1} \\ x_{m+2} \\ \vdots \\ x_{2m} \end{pmatrix}.$$

We will see in Proposition 1 that the matrix is invertible mod 2 if and only if there is no linear recurrence of length less than m that is satisfied by $x_1, x_2, \ldots, x_{2m-1}$.

Let us now summarise our strategy for finding the coefficients of the recurrence that generated a binary sequence once we know a few of the terms. Suppose we know the first 100 bits of the key, say x_1, x_2, …, x_{100}. For $k = 2, 3, 4, \ldots$, we form the $k \times k$ matrix

$$\mathcal{M}_k = \begin{pmatrix} x_1 & x_2 & \ldots & x_k \\ x_2 & x_3 & \ldots & x_{k+1} \\ \vdots & \vdots & \vdots & \vdots \\ x_k & x_{k+1} & \ldots & x_{2k-1} \end{pmatrix}$$

as before and compute its determinant. If, for several consecutive values of k, $\det(\mathcal{M}_k) \equiv 0 \pmod 2$ is zero, we stop. The last k to yield a nonzero (i.e., 1 mod 2) determinant is probably the length of the recurrence. We then solve the matrix equation to get the coefficients $c_0, \ldots, c_{k-1}$. We then check whether the sequence that this recurrence generates matches the sequence of known bits of the key. If not, we try larger values of k.

What do we if we don't get the first 100 bits, but rather some other 100 consecutive bits of the key? We can still apply the same procedure, using these bits as the starting point. In fact, once we find the recurrence, we can also work backwards to find the bits preceding the starting point. Let us look at an example to understand this.

Example 7: Suppose we have the following sequence of 100 bits:

$$1001100100111000110001010001111011001111101010 1001$$

$$0110110101100001101110010101111000000010001001 0000.$$

The first 20 determinants, starting with $m = 1$, are

$$1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.$$

After the eighth term there a string of zeroes in the sequence above, we guess is that $m = 8$ gives the last non-zero determinant. Solving the matrix equation for the coefficients we get

$$\{c_0, c_1, \ldots, c_7\} = \{1, 1, 0, 0, 1, 0, 0, 0\},$$

so we the recurrence is possibly

$$x_{n+8} \equiv x_n + x_{n+1} + x_{n+4}.$$

This recurrence generates all 100 terms of the original sequence, so we have the correct answer, at least based on the knowledge that we have.

Suppose that the 100 bits were in the middle of some sequence, and we want to know the preceding bits. For example, suppose the sequence starts with $x_{17} = 1, x_{18} = 0, x_{19} = 0, \ldots$. Since $-1 \equiv 1 \pmod 2$, we can write the recurrence as

$$x_n \equiv x_{n+4} + x_{n+4} + x_{n+8}$$

Letting $n = 16$ we get

$$\begin{aligned} x_{16} &\equiv x_{17} + x_{20} + x_{24} \\ &\equiv 1 + 0 + 1 \equiv 0 \end{aligned}$$

Continuing in this way, we successively determine $x_{15}, x_{14}, \ldots, x_1$.

***

Remark 2: Suppose a sequence satisfies relation of length three such as $x_{n+3} \equiv x_{n+2}$. It would clearly then also satisfy shorter relations such as $x_{n+1} = x_n$ (at least for $n \geq 2$). However, there are less obvious ways in which a sequence could satisfy a recurrence of length less than expected.

For example, the sequence $1,0,1,1,0,1,1,0,1,1,0,1\ldots$ satisfies the relation

$$x_{n+2} \equiv x_{n+1} + x_n \pmod 2$$

From this relation, we have

$$x_{n+4} \equiv x_{n+3} + x_{n+2} \pmod 2$$

Adding these two relations, we get

$$x_{n+4} + x_{n+2} \equiv x_{n+1} + x_n + x_{n+3} + x_{n+2}$$

or

$$x_{n+4} \equiv x_{n+2} + x_{n+1} + x_n$$

If we are just given the relation $x_{n+4} \equiv x_{n+2} + x_{n+1} + x_n$ for the sequence $1,0,1,1,0,1,1,0,1,1,0,1\ldots$, it is not clear *a priori* that the sequence doesn't satisfy a smaller relation. The next result gives a necessary condition for checking whether the recurrence that we have found for a sequence is of the smallest order possible.

Proposition 1: Let $x_1, x_2, x_3, \ldots$ be a sequence of bits produced by a linear recurrence mod 2. For each $n \geq 1$, let

$$M_n = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ x_2 & x_3 & \cdots & x_{n+1} \\ \vdots & \vdots & \cdots & \vdots \\ x_n & x_{n+1} & \cdots & x_{2n-1} \end{pmatrix}.$$

Let N be the length of the shortest recurrence that generates the sequence $x_1, x_2, x_3, \ldots$. Then $\det(M_N) \equiv 1 \pmod 2$ and $\det(M_n) \equiv 0 \pmod 2$ for all $n \geq N+1$.

Proof: If there is a recurrence of length N and if $n > N$, then one row of the matrix M_n is congruent mod 2 to a linear combination of other rows.

For example, suppose the recurrence is $x_{n+2} = x_{n+1} + x_n$. Writing the first three rows of the matrix in vector form, they are $(x_1, x_2, \ldots, x_n)$, $(x_2, x_3, \ldots, x_{n+1})$ and $(x_3, x_4, \ldots, x_{n+2})$. But, using the recurrence $x_{n+2} = x_{n+1} + x_n$, we have

$$\begin{aligned}(x_3, x_4, \ldots, x_{n+2}) &= (x_2 + x_1, x_3 + x_2, \ldots, x_{n+1} + x_n) \\ &= (x_1, x_2, \ldots, x_n) + (x_2, x_3, \ldots, x_{n+1}).\end{aligned}$$

So, we see that the third row is the sum of the first and second rows. Therefore, $\det(M_n) \equiv 0 \pmod 2$ for all $n > 3$. More generally, if the sequence satisfies the recurrence Eqn. (3) and i_1, i_2, …, i_r are the values of k for which the coefficient $a_k \neq 0$, the $(n+k)^{\text{th}}$ row is the linear combination of the rows i_1, i_2, …, i_r.

We'll prove the other part by contradiction. We'll suppose that $\det(M_n) \equiv 0 \pmod 2$ and arrive at a contradiction by producing a recurrence of length less than N. Suppose $\det(M_N) \equiv 0 \pmod 2$. Then, since this means that the matrix M_N is singular, there is a non-zero row vector $\overline{b} = (b_0, \ldots, b_{N-1})$ such that $\overline{b} M_N \equiv 0$.

Let the recurrence of length N be

$$x_{N+n} \equiv c_0 x_n + \cdots + c_{N-1} x_{n+N-1}.$$

For each $i \geq 0$, let

$$M^{(i)} = \begin{pmatrix} x_{i+1} & x_{i+2} & \cdots & x_{i+N} \\ x_{i+2} & x_{i+3} & \cdots & x_{i+N+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{i+N} & x_{i+N+1} & \cdots & x_{i+2N-1} \end{pmatrix}$$

Then $M^{(0)} = M_N$. From our recurrence relation, it follows that

$$M^{(i)} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{pmatrix} \equiv \begin{pmatrix} x_{i+N+1} \\ x_{i+N+2} \\ \vdots \\ x_{i+2N} \end{pmatrix},$$

which is the last column of $M^{(i+1)}$.

By our choice of $\bar{b}$, we have $\bar{b}M^{(0)} = \bar{b}M_N = 0$. Suppose that we know that $\bar{b}M^{(i)} = 0$ for some i. Then

$$\bar{b} \begin{pmatrix} x_{i+N+1} \\ x_{i+N+2} \\ \vdots \\ x_{i+2N} \end{pmatrix} \equiv \bar{b}M^{(i)} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix} \equiv 0.$$

So, $\bar{b}$ annihilates the last column of $M^{(i+1)}$. Since the remaining columns of $M^{(i+1)}$ are columns of $M^{(i)}$, we find that $\bar{b}M^{(i+1)} \equiv 0$. We can now apply induction to obtain $\bar{b}M^{(i)} \equiv 0$ for all $i \geq 0$.

Let $n \geq 1$. The first column of $M^{(n-1)}$ yields

$$b_0x_n + b_1x_{n+1} + \cdots + b_{N-1}x_{n+N-1} \equiv 0.$$

Since $\bar{b}$ is not the zero vector, $b_j \neq 0$ for at least one j. Let us choose m be the largest j such that $b_j \neq 0$, which means that $b_m = 1$. Since we are working $\pmod 2$, $b_mx_{n+m} \equiv -x_{n+m}$. Therefore, we can rearrange the relation to obtain

$$x_{n+m} \equiv b_0x_n + b_1x_{n+1} + \cdots + b_{m-1}x_{n+m-1}.$$

This is a recurrence of length m. Since $m \leq N - 1 < N$, and we assumed N to be of the shortest possible length, we have a contradiction. ■

Remark 3: Suppose the length of the recurrence is m. As we saw in our earlier discussion, if we know any m consecutive terms of the sequence, say x_k, x_{k+1}, …, x_{k+m}, we can determine all the previous terms x_1, x_2, …, x_{k-1} and all the following terms x_{k+m+1}, x_{k+m+2}, …. Clearly, if we have m consecutive 0s, then all values that follow are also 0. Also, all previous values are 0. Therefore, we exclude this case from consideration. There are $2^m - 1$ strings of 0s and 1s of length m in which at least one term is non-zero. Therefore, as soon as there are more than $2^m - 1$ terms, some string of length m must occur twice, so the sequence repeats. The period of the sequence is at most $2^m - 1$.

So far, we have discussed some methods for generating pseudo-random number sequences. We have to test these sequences to see how far they imitate the properties of true random number sequences. In the next section, we will discuss some statistical tests for testing pseudo-random numbers.

5.3 STATISTICAL TESTS FOR PSEUDO-RANDOM NUMBER GENERATORS

Earlier, we said that generating true random numbers is very tedious and we use pseudo-random numbers generated using some mathematical functions. When we use such methods for generating random numbers, we have to make sure that the numbers we get are 'sufficiently random'. We will discuss some empirical tests for testing

whether a given sequence of bits is pseudo-random. We assume that you are familiar with Testing of Hypothesis. You can refer to Block two of the IGNOU course AST-01 for a discussion. In particular, Unit 7 of Block two of AST-01 discusses χ^2 test that we will use in this section. See `http://www.egyankosh.ac.in/bitstream/123456789/14910/1/Unit-7.pdf`(You may have to register for downloading the material.) The discussion in this section is based on Section 5.4 of [11]. Our discussion only provides an introduction. A standard reference for the topic is [8]. See also [13] for a discussion of tests for cryptographic random number generators and the software developed by NIST for carrying out statistical tests.

In the statistical tests for randomness we usually formulate the null hypothesis H_0 that a given sequence of numbers is random and test this using an appropriate statistic for a particular level of significance α. Recall that α is the probability that we commit a type I error; in this case this is probability of wrongly rejecting a data that is random as non-random. For cryptographic applications, we usually choose $\alpha = 0.01$. The statistic used for the tests varies from test to test. If we accept the null hypothesis of a test for our chosen level of significance, we say the sequence **passes the test**. The fact that a sequence passes the test doesn't necessary mean that it is random. Tests help only in identifying the weaknesses in a sequence and a sequence may not be random even if it passes the tests. The tests also tells us to what extent a pseudo-random number sequence is similar to a true random number sequence.

We will discuss the following tests:

1. Frequency test.
2. Serial test.
3. Poker test.
4. Runs test.
5. Auto correlation test.

These tests are based on what are known as Golomb's randomness postulates. Before we discuss the postulates, we have to introduce some terminology. Given a random sequence s, a **run** is a subsequence consisting of consecutive 0s or consecutive 1s that satisfies the following condition: The sequence should not be preceded or followed by the same symbol, i.e. if the subsequence consists of zeros(resp. ones), the bit preceding and following the subsequence must be ones(resp. zeros). We call a run of 0s a **gap** and run of 1s a **block**.

Let $x_1, x_2, \ldots$, be a sequence of period r. Then, the **autocorrelation function** of the sequence is defined by

$$C(t) = \frac{1}{r}\sum_{i=1}^{r}(2x_i - 1)(2x_{i+t} - 1) \text{ for } 0 \leq t \leq r-1.$$

The function $C(t)$ measures the similarity between the sequence x and its shift by t positions. For a random sequence x of period r, we expect $|rC(t)|$ to be small for $0 < t < r$.

For a sequence $x_1x_2\ldots$ of period r, the following are the Golomb's postulates:

1) The number of 1s in $x_1x_2\ldots x_r$ differ from the number of 0s by at most 1.

2) In $x_1x_2\ldots x_r$, at least half the runs have length 1, at least one-fourth have length 2, at least one-eighth have length 3, etc., as long as the number of runs so indicated exceeds 1. Moreover, for each of these lengths, there are (almost) equally many gaps and blocks.

3) The autocorrelation function C(t) is two-valued. That is, if the sequence is of period k,

$$rC(t) = \sum_{i=1}^{r}(2x_i - 1)(2x_{i+t} - 1) = \begin{cases} r; & \text{if } t = 0; \\ k; & \text{if } 1 \le t \le r-1 \end{cases}$$

We will illustrate all the tests, except the poker test and the runs test, using the sequence 110010010000111111011010 of length 24. In practice, we choose longer sequences of length much larger than 10000. Here, we have chosen a small sequence because we merely want to explain the test procedure. We will apply all the tests with $\alpha = 0.05$.

1. Frequency Test

In this test, we check that the number of 0s and 1s are approximately equal. Let n_0 and n_1 denote the number of 0s and 1s, respectively. The statistic for this test is

$$X_1 = \frac{(n_0 - n_1)^2}{n} \tag{9}$$

which follows an approximately χ^2 distribution with one degree of freedom if $n \ge 10$. In our case, $n_0 = 11$, $n_1 = 13$, $\chi_s = X_1 = \frac{4}{24} \approx 0.1667$. The value of $\chi_\alpha = \chi^2_{0.05,1}$ which can be found by looking at the row corresponding to $\nu = 1$ and the column under $\alpha = 0.05$ in the χ^2 table in Table 3 on page 47. We see that this is 3.84146, so $\chi_\alpha = 3.841146$. Since $\chi_s < \chi_\alpha$, our sequence passes this test.

2. Serial Test

In a random sequence, we would expect the pairs 00, 01, 11 and 10 occur approximately equal number of times. The serial test checks if this is the case in the sequence we are testing for randomness. As before, let n_0 and n_1 denote the number of 0s and 1s that occur and n_{00}, n_{01}, n_{10} and n_{11} denote the number of occurrences of 00, 01, 10 and 11, respectively. Note that $n_{00} + n_{01} + n_{10} + n_{11} = n - 1$ because the sequences are allowed to overlap. The statistic for this test is

$$X_2 = \frac{4}{n-1}\left(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2\right) - \frac{2}{n}\left(n_0^2 + n_1^2\right) + 1 \tag{10}$$

It is known that X_2 follows an approximately χ^2 distribution with two degrees of freedom. We have

$$n_0 = 11 \qquad n_{00} = 5 \qquad n_{01} = 5$$
$$n_1 = 13 \qquad n_{10} = 6 \qquad n_{11} = 7$$

Therefore,

$$\chi_s = X_2 = \frac{4}{23}(25 + 25 + 36 + 49) - \frac{2}{24}(121 + 169) + 1 \approx 0.311594$$

The value of $\chi_\alpha = \chi^2_{0.05,2} = 5.99146$ and $\chi_s < \chi_\alpha$, so this sequence passes this test also.

3. Poker Test

Let m be a positive integer such that $\lfloor \frac{n}{m} \rfloor > 5 \cdot (2^m)$ and $k = \lfloor \frac{n}{m} \rfloor$. Divide the sequence into k non-overlapping parts of length m each. Let n_i be the number of times the i^{th} type of sequence of length m, $1 \le i \le 2^m$, occurs. The poker test checks if all the possible sequences of length m occurs approximately equal number of times. The statistic used is

$$X_3 = \frac{2^m}{k}\left(\sum_{i=1}^{2^m} n_i^2\right) - k \tag{11}$$

which follows an approximately χ^2 distribution with $2^m - 1$ degrees of freedom. Note that, the poker test is a generalisation of frequency test. If $m = 1$, we get the frequency test.

If we are to apply the poker test to the sequence used in the earlier two tests, we will have to choose m = 1 for the condition $\lfloor \frac{24}{m} \rfloor > 5 \cdot (2^m)$ to be satisfied. So, the sequence is too small. So, we take the following longer sequence of length 48 for which we can take m = 2.

110010010000111111011010101000100010000101101000

Splitting into blocks of size two, we get the following:

11|00|10|01|00|00|11|11|11|01|10|10|10|10|00|10|00|10|00|01|01|10|10|00

There are four possible types of sequences, 00, 01, 10 and 11. Table 1 gives the number of sequences of various types.

S.No	Type	No. of occurrences
1	00	7
2	01	4
3	10	9
4	11	4

Table 1: Number of occurrences of different types of sequences.

Let us now apply the test with m = 2, k = 24 and n = 48. We have

$$\chi_s = X_3 = \frac{4}{24}\left(7^2 + 4^2 + 9^2 + 4^2\right) - 24 = \frac{162}{6} - 24 = 3.$$

For $\alpha = 0.05$ and $2^2 - 1 = 3$ degrees of freedom, the value in the χ^2 table is 7.81473. So $\chi_\alpha = 7.81473$ and $\chi_s < \chi_\alpha$, so our sequence passes the poker test.

4. Runs Test

The expected number of gaps or blocks of length i in a random sequence of length e_i is $(n - i + 3)/2^{i+2}$. In this test, we want to test whether the number of runs in s is as expected.

For this we proceed as follows: Suppose that k is the largest integer i for which $e_i \geq 5$. Let B_i, G_i be the number of blocks and gaps, respectively of length i in the random sequence s for each i, $1 \leq i \leq k$. The statistic for this test is

$$X_4 = \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i}$$

The statistic X_4 follows approximately χ^2 distribution with $2k - 2$ degrees of freedom.

Note that, for a fixed n, $(n - i + 3)/2^{i+2}$ **decreases** with i. Since we can apply the test only if there are e_is which are greater than five, we have to choose n large enough. For example, if we want e_3 to be greater than five, we should have

$$n - 3 + 3 \geq 5.2^5 \text{ or } n \geq 160.$$

So, for this test also, we will use a longer sequence. In our case n = 160.

0 | 1 | 0000 | 1 | 0 | 11 | 0000 | 1 | 00 | 11 | 00 | 1 | 0 | 1 | 0
| 1 | 0 | 1 | 00 | 111 | 00 | 111 | 0 | 11 | 00 | 1 | 00 | 1 | 0 | 1 | 0 |
111 | 0 | 1 | 0000 | 1 | 0 | 11 | 0 | 1 | 0 | 111 | 00000 | 1111 |
000 | 1 | 0 | 111 | 00000 | 111 | 0 | 111 | 00
0 | 1 | 00 | 1 | 00 | 1 | 00 | 1 | 0 | 111 | 0 | 11 | 00000 | 11 |
0 | 1 | 00 | 11 | 00 | 11 | 000 | 1 | 0 | 1 | 00 | 1111 | 0 | 11 |
0 | 1 | 00 | 1 | 000 | 11 | (12)

From Eqn. (12), we get the following information:

i	B_i	G_i	e_i
1	23	20	20.25
2	10	14	10.0625
3	8	3	5

Table 2: Values of B_i, G_i and e_i

In our case $e_4 \approx 2.5 < 5$, so we stop with e_3. We have

$$\chi_s = X_4 = \frac{(24-20.25)^2}{20.25} + \frac{(21-20.25)^2}{20.25} + \frac{(14-10.0625)^2}{10.0625} + \frac{(10-10.0625)^2}{10.0625} + \frac{(8-5)^2}{5} + \frac{(5-3)^2}{5} = 5.1133$$

The value in the χ^2 table for four degrees of freedom for $\alpha = 0.05$ is 9.48773, i.e. $\chi_\alpha = 9.48773$. So, $\chi_s < \chi_\alpha$ and the sequence passes the runs test.

5. Autocorrelation Test

In this test, we check the correlation between the sequence s and its (non-cyclic) shifted versions. We fix an integer d, $1 \leq d \leq \lfloor \frac{n}{2} \rfloor$. Then, the number of bits in s not equal to their d-shifts is $A(d) = \sum_{i=0}^{n-d-1} s_i \oplus s_{i+d}$, where $\oplus$ denotes the XOR operator. The statistic used for the test is

$$X_5 = 2\left(A(d) - \frac{n-d}{2}\right)/\sqrt{n-d} \tag{13}$$

which approximately follows an $N(0,1)$ distribution if $n-d \geq 10$. We use a two-sided test since we expect the value of $A(d)$ to be neither small nor big.

For our sequence, $n = 24$, so we have $d \leq 12$. Let us apply the test with $d = 4$. So, $n-d-1 = 19$.

$$\begin{array}{rl} & 11001001000011111101 \quad \text{(First 20 terms of the sequence)} \\ \oplus & 10010000111111011010 \quad \text{(20 terms of the sequence starting from 5th term)} \\ \hline & 01011001111100100111 \end{array}$$

The number of 1s in the last row is 12, so 12 terms of the sequence are not equal to their four shifts, i.e. $A(4) = 12$. So,

$$X_5 = \frac{2\left(12 - \frac{20}{2}\right)}{\sqrt{20}} = \frac{4}{\sqrt{20}} = 0.8944$$

From the tables of normal distribution, $P(X > a) = 0.025$ if $a = 1.96$. So, $P(X > 1.96) + P(X < -1.96) = 0.05$. So, our sequence passes the autocorrelation test.

The universal Maurer test that was published in [10] can detect a larger class of defects in random bit sequences including the defects detected by the five tests mentioned above. However, we will not discuss the Maurer test in our course. You can refer to Chapter 5, Section 4.5 of [11] for a discussion of this test.

We conclude this section here. In the next section, we discuss stream ciphers.

5.4 STREAM CIPHERS

In this section, we will discuss symmetric key stream ciphers. Note that, LFSR based ciphers and the Vigenère cipher are also examples of stream ciphers. Notice also that, a

block cipher, when used in CTR mode is like a stream cipher with the initial vector X_1 acting like a seed for pseudo-random number generation. While LFSR based stream ciphers are suitable for hardware implementation, they are not amenable to software implementation. This lead to several proposals for stream ciphers particularly designed for software implementation.

In this section we discuss the popular symmetric stream cipher, RC4, pronounced as ARC4. We begin with an overview of stream cipher structure, and then examine RC4.

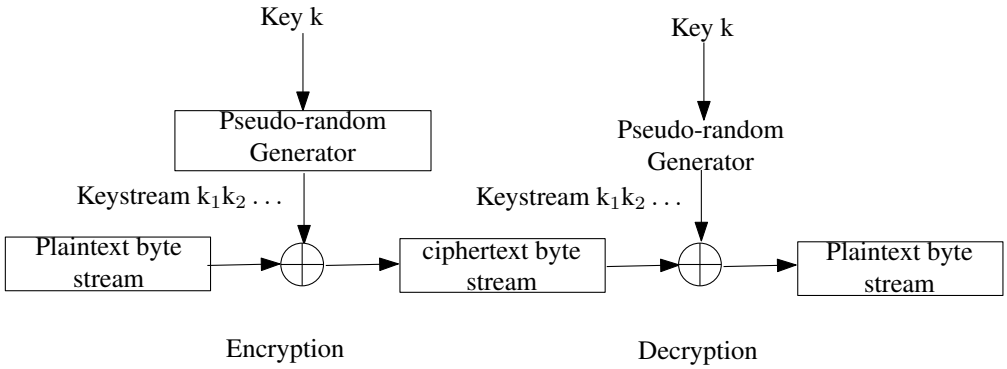


Fig. 4: Schematic diagram of a stream cipher.

A typical stream cipher encrypts the plain text one byte at a time, although a stream cipher may be designed to operate on one bit at a time or on units larger than a byte at a time. Fig. 4 is a schematic diagram of stream cipher structure. In this structure a key is input to a pseudo-random bit generator that produces a stream of 8-bit numbers that are apparently random. The output of the generator, the **keystream**, is combined one byte at a time with plain text stream using the bitwise exclusive-OR (XOR) operation.

Stream ciphers with properly designed CSPRBGs are as secure as block ciphers. Stream ciphers are generally faster than block ciphers and used wherever speed is important. Also, compared to block ciphers, they can be implemented in software with fewer lines of code. In block ciphers, we can reuse the keys. In stream ciphers we cannot reuse the keys. If two different messages are encrypted with the same key, by XORing the ciphertexts, we can get the XOR of the plaintexts and this can be used in cryptanalysis.

For applications that require encryption/decryption of a stream of data, such as over a data communications channels like mobile phones or a browser/Web link, a stream cipher might be the better alternative. For applications that deal with blocks of data, such as file transfer, e-mail, and database, block ciphers may be more appropriate. However, either type of cipher can be used in virtually any application.

The RC4 Algorithm

RC4 is a stream cipher designed in 1987 by Ron Rivest for RSA Security. It is a variable key-size stream cipher with byte-oriented operations, suitable for bulk encryption. Analysis shows that the period of the cipher is overwhelmingly likely to be greater than 10^{100}. Cf. [16]. RC4 is probably the most widely used stream cipher. It is used in the SSL/TLS (Secure Sockets Layer/ Transport Layer Security) Standards that have been defined for communication between Web browsers and severs. It is also used in the WEP (Wired Equivalent Privacy) protocol that is part of the IEEE 802.11 wireless LAN standard. RC4 was kept as a trade secret by RSA Security. In September 1994, the RC4 algorithm was anonymously posted on the Internet on the Cypher punks anonymous remailers list. Our discussion on RC4 is based on the Wikipedia article [24].

To generate the keystream, the cipher makes use of a secret internal state which consists of two parts:

1. A permutation of all 256 possible bytes (denoted "S" below).
2. Two 8-bit index-pointers (denoted "i" and "j").

The permutation is initialised with a variable length key, typically between 40 and 256 bits, using the key-scheduling algorithm (KSA). Once this has been completed, the stream of bits is generated using the pseudo-random generation algorithm (PRGA).

The Key-scheduling Algorithm (KSA)

The key-scheduling algorithm is used to initialise the permutation in the array "S". "keylength" is defined as the number of bytes in the key and can be in the range $1 \leq \text{keylength} \leq 256$, typically between 5 and 16, corresponding to a key length of 40—128 bits. First, the array "S" is initialised to the identity permutation. S is then processed for 256 iterations in a similar way to the main PRGA, but also mixes in bytes of the key at the same time.

Algorithm 1 The key-scheduling algorithm(KSA)

```
for i ← 0 to 255 do
    S[i] ← i
end for
j ← 0
for i ← 0 to 255 do
    j ← (j + S[i] + key[i mod keylength]) mod 256
    swap(&S[i],&S[j])
end for
```

The Pseudo-random Generation Algorithm (PRGA)

The lookup stage of RC4. The output byte is selected by looking up the values of S(i) and S(j), adding them together modulo 256, and then looking up the sum in S; S(S(i) + S(j)) is used as a byte of the key stream, K.

For as many iterations as are needed, the PRGA modifies the state and outputs a byte of the keystream. In each iteration, the PRGA increments i, adds the value of S pointed to by i to j, exchanges the values of S[i] and S[j], and then outputs the value of S at the location S[i] + S[j] (modulo 256). Each value of S is swapped at least once every 256 iterations.

Algorithm 2 The stream generator

```
i ← 0 j ← 0
while GeneratingOutput do do
    i← (i + 1) mod 256
    j← (j + S[i]) mod 256
    swap(&S[i],&S[j])
    byte_key ← S[(S[i] + S[j]) mod 256]
    result_ciphered ← byte_key XOR byte_message
end while
```

5.5 SUMMARY

In this Unit, we have discussed

1. what is a pseudo-random number sequence;
2. what is a cryptographically secure pseudo-random bit generator;

3. the method of generating pseudo-random numbers using LFSRs;
4. the statistical testing of randomness; and
5. how the RC4 cipher works.

5.6 SOLUTIONS/ANSWERS

E1) See Fig. 5.

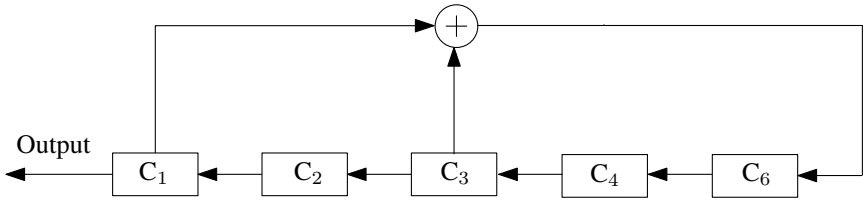


Fig. 5: Answser to exercise 1.

E2) $x_{m+5} \equiv x_{m+4} + x_{m+2} + x_{m+1} + x_m \pmod 2$

E3) Writing $x_{n+2} = c_0 x_1 + c_1 x_2$, we get the matrix equation

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

in $\mathbf{F}_2$ for which the solution is $(1,1,0)$. This leads to the recurrence $x_{n+2} \equiv x_n + x_{n+1} \pmod 2$. For $n = 3$, $x_5 \neq x_4 + x_3$. So, this is not the correct recurrence.

In the next step, we get the matrix equation

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

You can check that $x_{n+3} \equiv x_n + x_{n+1} \pmod 2$ is the correct recurrence.

ν	α =0.500	α =0.250	α =0.100	α =0.050	α =0.025	α =0.010	α =0.005
1	0.45494	1.32330	2.70554	3.84146	5.02389	6.63490	7.87944
2	1.38629	2.77259	4.60517	5.99146	7.37776	9.21034	10.59663
3	2.36597	4.10834	6.25139	7.81473	9.34840	11.34487	12.83816
4	3.35669	5.38527	7.77944	9.48773	11.14329	13.27670	14.86026
5	4.35146	6.62568	9.23636	11.07050	12.83250	15.08627	16.74960
6	5.34812	7.84080	10.64464	12.59159	14.44938	16.81189	18.54758
7	6.34581	9.03715	12.01704	14.06714	16.01276	18.47531	20.27774
8	7.34412	10.21885	13.36157	15.50731	17.53455	20.09024	21.95495
9	8.34283	11.38875	14.68366	16.91898	19.02277	21.66599	23.58935
10	9.34182	12.54886	15.98718	18.30704	20.48318	23.20925	25.18818
11	10.34100	13.70069	17.27501	19.67514	21.92005	24.72497	26.75685
12	11.34032	14.84540	18.54935	21.02607	23.33666	26.21697	28.29952
13	12.33976	15.98391	19.81193	22.36203	24.73560	27.68825	29.81947
14	13.33927	17.11693	21.06414	23.68479	26.11895	29.14124	31.31935
15	14.33886	18.24509	22.30713	24.99579	27.48839	30.57791	32.80132
16	15.33850	19.36886	23.54183	26.29623	28.84535	31.99993	34.26719
17	16.33818	20.48868	24.76904	27.58711	30.19101	33.40866	35.71847
18	17.33790	21.60489	25.98942	28.86930	31.52638	34.80531	37.15645
19	18.33765	22.71781	27.20357	30.14353	32.85233	36.19087	38.58226
20	19.33743	23.82769	28.41198	31.41043	34.16961	37.56623	39.99685

Table 3: CHI-SQUARED TABLE

a	0.1	0.25	0.05	0.025	0.01	0.005	0.0025	0.0002	0.0001
x	1.281552	0.674490	1.644854	1.959964	2.326348	2.575829	2.807033	3.540084	3.719016

Table 4: Some percentiles of standard normal distribution. If X is a random variable having a standard normal distribution, then $P(X > x) = a$

UNIT 6 HASH FUNCTIONS

Structure	Page No.
6.1 Introduction	49
Objectives	
6.2 Design of Hash Functions	49
6.3 Examples of Hash Functions	54
MD5	
SHA-II	
6.4 Birthday Attacks	59
6.5 Summary	59
6.6 Solutions/Answers	59

6.1 INTRODUCTION

Hash functions have many applications in Computer Science. Loosely speaking, hash functions are functions that take a variable length input and give an output of fixed length, i.e. it compresses the input of arbitrary size to a fixed size. From the point of view of applications in cryptography, we need hash functions satisfying some extra conditions. Such hash functions, called **one-way hash functions** or **cryptographic hash functions**, are the main topic of discussion in this Unit. In Sec. 6.2 we discuss the considerations that go into the design of cryptographic hash functions. In Sec. 6.3, we will discuss two examples of hash functions, namely, MD5 and SHA-II. In the Sec. 6.4, we will discuss some attacks on hash functions.

Objectives

After studying this unit, you should be able to

- define a cryptographic hash function;
- define a compression function;
- explain how to construct compression functions from block ciphers using Davies-Meyer, Matyas-Meyer-Oseas and Miyaguchi-Preneel methods;
- explain the Merkle-Damgård method for constructing hash functions from compression functions;
- explain the working of MD5 and SHA-II algorithms; and
- explain the birthday attack on hash functions.

6.2 DESIGN OF HASH FUNCTIONS

Let us begin our discussion of hash functions with an example of application of hash functions. Many useful softwares are available for free download from the internet. Sometimes, the file may not download properly for the following reason: All the data is transferred over the internet using the TCP/IP protocol. In this protocol, the data is divided into packets and sent over the network. All the packets are put together again at the destination. In this process, some packets may get lost. If this happens, you would have a damaged version of the file. How can you check if the software has downloaded correctly.

In some sites, you would have noticed that a number having 32 hexadecimal digits, called MD5 sum, is also given.(See Fig. 1 on the next page.) We will see later in this

Unit what this MD5 sum is. For the moment, we will discuss only the reason for giving the MD5 sum. There are softwares available on the internet to find the MD5 sums of files. You can find the MD5 sum for the file you have downloaded using any of these software and compare it with the MD5 sum given in the website. Even if the file you downloaded is slightly different from the file available on the internet, the MD5 sums will not match. So, you can check if your software has downloaded correctly by comparing the hash values. The MD5 sum is actually the **hash value** of the file

Fig. 1: Software and its MD5 sum.

calculated using a **hash function**.

The term hash functions comes from Computer Science where it denotes a function, not necessarily one-way, that compresses an input string of arbitrary length to a string of fixed length. There, the hash functions are used in the data structure called dictionary used for sorting. Depending on the application to which it is put, one-way hash function has many names in cryptographic literature like hashcode, hash total, checksum, Message Integrity Code(MIC), finger print, MDC, etc.

Let us now formally define a cryptographic hash function. Before we do that let us set up some notation. Note that there is a bijection between $\{0,1\}^n$, $n \geq 1$, and the set of all binary strings of length n. Similarly, there is bijection between the set of all possible finite binary strings and the set $\cup_{n=1}^{\infty} \{0,1\}^n$. Let us denote the set of all possible finite binary strings by $\{1,0\}^*$.

Definition 7: A function $h\colon \{0,1\}^* \longrightarrow \{0,1\}^n$, $n \geq 1$, is called a **a cryptographic hash function** if h has the following properties:

1) We should be able to calculate $h(\mathcal{M})$ easily from a given a message $\mathcal{M}$.
2) For a given y in the image of h, it should computationally infeasible to find an input $\mathcal{M}'$ with $h(\mathcal{M}') = y$. (One way)
3) It is computationally infeasible to find two inputs $\mathcal{M}_1$ and $\mathcal{M}_2$, $\mathcal{M}_1 \neq \mathcal{M}_2$, with $h(\mathcal{M}_1) = h(\mathcal{M}_2)$.(Collision resistance.)
4) Given $\mathcal{M}$ and $h(\mathcal{M})$, it is difficult to find $\mathcal{M}'$ such that $h(\mathcal{M}) = h(\mathcal{M})$.(Second pre-image resistant)

For reasons that will be clear later, we call the input to a hash function a **message** and the output the **message digest**. We call a function that satisfies condition 2) and 4) a **one way function**. We call a function that satisfies condition 3) a **collision resistant** hash function.

Note that, in condition 2), if $y = h(\mathcal{M})$, we are not trying to find $\mathcal{M}$. Rather, we are trying to find an $\mathcal{M}'$ with image y. Instead of condition 3), we often settle for the following weaker condition:

Definition 8: We call a function h **weakly collision free** if, given $\mathcal{M}$, it is computationally infeasible, to find another $\mathcal{M}' \neq \mathcal{M}$ such that $h(\mathcal{M}) = h(\mathcal{M}')$.

Recall that, the pigeon hole principle says that if the number of pigeons is greater than the number of pigeon holes, some pigeon holes must have more than one pigeon. In the case of hash functions, by mapping a large set to a small set, we are trying to put a large number of pigeons in a small number of pigeon holes. The set of possible messages is much larger than the set of possible message digests. So, there should be many instances of messages $\mathcal{M}_1$ and $\mathcal{M}_2$ with $h(\mathcal{M}_1) = h(\mathcal{M}_2)$. The requirement 3) only says that it should be computationally infeasible to such pairs $\mathcal{M}_1$ and $\mathcal{M}_2$.

In the example we discussed at the beginning of the unit we saw how we can detect if a file has changed using MD5 sum. We call the MD5 sum a **Modification Detection Code(MDC)** or **Manipulation Detection Code** because it helps us to check if a file has been modified. However, if a malicious attacker gains access to the server where the software is made available he/she can do the following:

1) Replace the software file with another file containing a harmful software.
2) Calculate the MD5 sum for this modified file and replace the hash sum on the website with the hash sum of the malicious program.

The attacker is able to do this because any one can calculate the MD5 sum. To prevent this, we use another kind of hash functions called **keyed hash-functions**. In this case a secret key is required to calculate the hash. Such keyed hash functions are also called **Message Authentication Codes(MACs)** for the following reason: Suppose Alice and Bob share a secret key k. Also, suppose that Bob wants to send a message to Alice and Alice wants to be sure that the message was actually sent by Bob. Then, if $\mathcal{M}$ is the message, Bob can calculate the hash $h_k(\mathcal{M})$ and send $(\mathcal{M}, h_k(\mathcal{M}))$ to Alice. Here, $h_k(\mathcal{M})$ is a hash function that uses the secret key k for calculating the hash. Since Alice also has the key k, she can calculate $h_k(\mathcal{M})$ and compare it with the value sent by Bob and convince herself that the message was actually sent by Bob. If Eve modifies the message to $\mathcal{M}'$, she will not be able to calculate the hash $h_k(\mathcal{M}')$ since she doesn't know the secret key k. So, she will not be able to replace the hash value Bob has sent with $h_k(\mathcal{M}')$. Thus, she cannot modify the message without Alice's knowledge. Let us now define keyed hash functions formally:

Definition 9: By a family of **keyed hash functions** we mean a set of hash functions $\{h_k\}_{k\in K}$ parametrised by a finite set $\mathcal{K}$ called the keyspace. In other words, for each $k \in \mathcal{K}$, we have a hash function $h_k : \{0,1\}^* \to \{0,1\}^n$.

Definition 10: A **HMAC(Hashed Message Authentication Code)** takes a message $\mathcal{M}$of arbitrary length and a secret key k as input and gives an output of fixed length. It is defined as follows:

$$\text{HMAC}_k(\mathcal{M}) = h\left(k' \oplus \text{opad}, h\left(k' \oplus \text{ipad}, \mathcal{M}\right)\right)$$

where:

1) h is a usual hash function that acts on blocks of size m.
2) k' is got by padding k with sufficient number of zeros on the right so that k' has size m.
3) We form ipad by repeating the byte 0x36 as many times as necessary so that the length is m. Similarly, we form opad by repeating the byte 0x5c.

An **NMAC(Nested Message Authentication Code)** is a generalised version of HMAC which uses two secret keys k_1 and k_2 and is defined as follows:

$$\text{NMAC}_{K_1,k_2}(\mathcal{M}) = h(k_2, h(k_1, \mathcal{M}))$$

Here h is a usual hash function.

Note that, in the definition of the NMAC, if we take $k_1 = k' \oplus \text{ipad}$ and $k_2 = k' \oplus \text{opad}$, we get a HMAC.

A method for constructing hash functions that is used in the construction of many hash functions is the **Merkle-Damgård** method. The basic idea behind the method is to use a **one-way compression function** repeatedly.

Definition 11: A **collision resistant function** f is a map

$$f:\ \{0,1\}^{n+m} \longrightarrow \{0,1\}^{n},\ m,n \in \mathbf{N},\ m > n,$$

which is collision resistant, i.e. it satisfies condition 3) in the definition of a cryptographic hash function.

The nice thing about the Merkle-Damgård method is that we can prove that the hash function constructed using this method satisfies conditions 1), 2) and 3) if the one-way compression function satisfies them. The Merkle-Damgård method is as follows:

Let f be a compression function. Suppose, we want to find the hash of a message $\mathcal{M}$. Let us call the number of bits in the message, the **length** of the message. We first pad it with zeroes so that the number of bits in the message is a multiple of m. However, there is a problem in this. Suppose for simplicity that $m = 64$ and the message $\mathcal{M}$ is the string "Hashexample". If we assume that we use 8 bits to store a character, then this string is of length 88 bits. Then, dividing into blocks of length 64 each and padding it we will get two blocks "Hashexam" and "ple$\underbrace{000\ldots00}_{40\text{ zeroes}}$". (Note here, that we consider the zeroes added as bits and not characters and so each zero is of length 1 bit.) However, if we take the string "Hashexample00",the phrase "Hashexample" followed by two zero bits, we will again get the same two blocks and so the strings "Hashexample" and "Hashexample00" will have the same hash value. To avoid this we insert a one bit at the beginning of the zeroes. For example, using this method, "Hashexample" will yield the blocks "Hashexam" and "ple1$\underbrace{00\ldots00}_{39\text{ zeroes}}$" while "Hashexample00" will yield the strings "Hashexam" and "ple001$\underbrace{000\ldots00}_{37\text{ zeroes}}$" which are different.

We further pad the message by adding one more block which we construct as follows: Suppose the length of the message $\mathcal{M}$ before padding is ℓ, in binary. Here, we assume that $\ell < 2^m$. We add as many zeroes to the left of ℓ as necessary to get a block of size m. For example, suppose we want to apply this to the string "Hashexample". Then, after padding it to get "Hashexam" and "ple$\underbrace{000\ldots00}_{40\text{ zeroes}}$", we add another block "$\underbrace{00\ldots00}_{57\text{ zeroes}}$1011000". Here the length of the string "Hashexample" is 88 and the binary representation of 88 is 1011000. We add 57 zeroes to the left of the bitstring 1011000 so that the total length is 64. This technique is called **length padding** or **Merkle-Damgård** strengthening.

After padding the message so that its length is a multiple of m, we divide the message into blocks $\mathcal{M}_1$, $\mathcal{M}_2$, …, $\mathcal{M}_t$ where each block is of size m. We then use the one-way compression function f to create the hash in t steps. Let 0^n denote a string consisting of n zeroes. We set

$$\mathcal{H}_0 = f(0^n||\mathcal{M}_1), \mathcal{H}_i = f\left(\mathcal{H}_{i-1}||\mathcal{M}_i\right) \text{ for } 1 \leq i \leq t$$

Here 0^n is an example of **initial vector**. In the hash algorithms, we use some other initial vectors also instead of 0^n. Note that each $\mathcal{H}_i$ is m bits long. The last output $\mathcal{H}_t$ is the hash of the message $\mathcal{M}$. (See Fig. 2 on the facing page.) The values $\mathcal{H}_0$, $\mathcal{H}_1$, …, $\mathcal{H}_{t-1}$ are called **chaining variables** because they are chained to the next stage of application of the compression function, as shown in Fig. 2 on the next page.

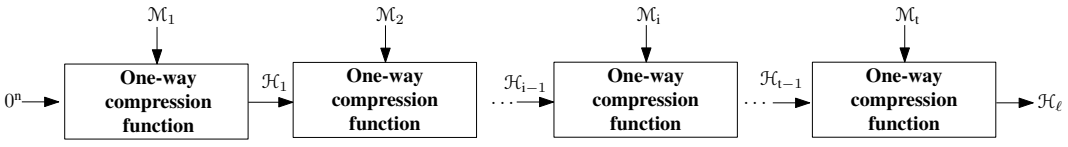


Fig. 2: The Merkle-Damgård method.

Before we move on to the next topic of discussion, here is an exercise for you.

E1) Assuming a block size of 64 and that we use 8 bits to store a character, what will be the string you will get by applying the Merkle-Damgård strengthening to the string "tohashornottohash"?

Let us now address the next issue, namely the construction of one-way compression function. One method is to use a block cipher to construct a one-way compression function. In this Unit, we will discuss the methods for this due to Davies-Meyer, Matyas-Meyer-Oseas, Miyaguchi-Preneel. For a discussion of other methods, you can see the Wikipaedia article [23] or [11].

In our discussion, let $E_k(\mathcal{M})$ denote ciphertext we get by encrypting a message $\mathcal{M}$ with a block cipher E and key k.

Davies-Meyer method: Here, we assume that the block cipher takes as input a string of size n and outputs a string of size n and the size of the key is m. In this method, the one way compression function is defined as follows: Let X be a string of length $n+m$. Let X_L denote first n bits on the left of X and let X_R denote the remaining m bits so that $X = X_L||X_R$. Then, the compression function is

$$f(X) = f(X_L||X_R) = E_{X_R}(X_L) \oplus X_L.$$

So, we have

$$\mathcal{H}_i = E_{\mathcal{M}_i}\left(\mathcal{H}_{i-1}\right) \oplus \mathcal{H}_{i-1}.$$

See Fig. 3.

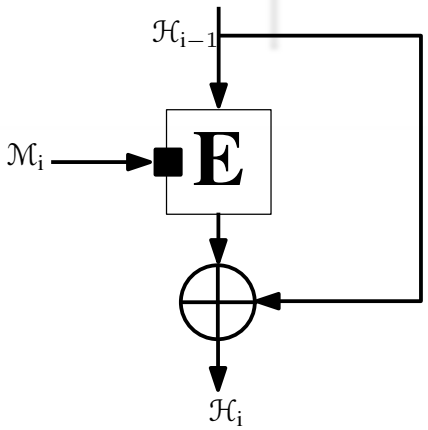


Fig. 3: Davies-Meyer Method.

Matyas-Meyer-Oseas method: In this method, we use a block cipher E which takes an input of size m, uses a key of size n and produces an output of size m. Note the interchange of the roles of the message and the hash in this method. If the block cipher has different block and key sizes, the length of the previous hash may not be of the correct size for a key. Also, the key for the block cipher may have some special requirements. So, the previous hash $\mathcal{H}_{i-1}$ has to be processed with another function g

to produce a key from the hash $\mathcal{H}_{i-1}$. In this case we define the compression function as follows:

$$f(X) = f(X_L||X_R) = E_{g(X_L)}(X_R) \oplus X_R.$$

So,

$$\mathcal{H}_i = E_{g(\mathcal{H}_{i-1})}(\mathcal{M}_i) \oplus \mathcal{M}_i.$$

See Fig. 4.

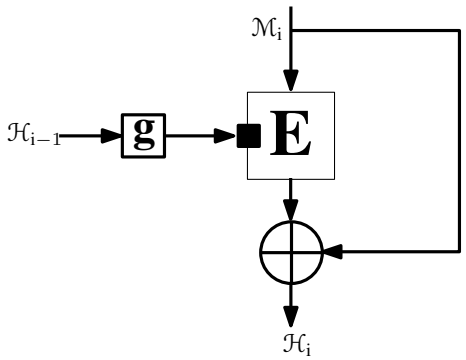


Fig. 4: Matyas-Meyer-Oseas method.

Miyaguchi-Preneel method: This was proposed independently by Miyaguchi and Preneel. It is an extended version of Matyas-Meyer-Oseas method. In this method, in addition to XORing with $\mathcal{M}_i$, the previous hash value is also XORed with the output of the block cipher. In this case we define the compression function as follows:

$$f(X) = f(X_L||X_R) = E_{g(X_L)}(X_R) \oplus X_R \oplus X_L.$$

So,

$$\mathcal{H}_i = E_{g(\mathcal{H}_{i-1})}(\mathcal{M}_i) \oplus \mathcal{M}_i \oplus \mathcal{H}_{i-1}.$$

See Fig. 5.

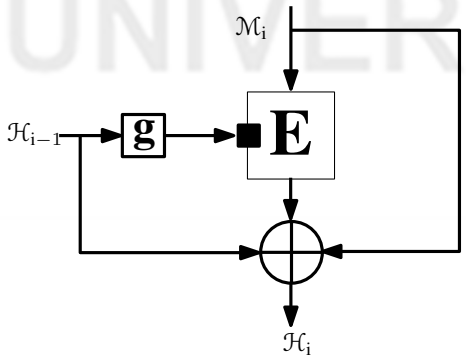


Fig. 5: Miyaguchi-Preneel method.

We close this section here. In the next section, we will discuss some popular hash functions.

6.3 EXAMPLES OF HASH FUNCTIONS

In the previous section, we saw some methods of designing hash functions. In this section, we are going to see some hash functions based on these design principles. We start our discussion with MD5 hash function. This was invented by Ronald Rivest in the year 1992 to replace MD4, a hash function he invented earlier, after cryptographers found some flaws in it.

6.3.1 MD5

MD5 takes as input messages of arbitrary size and outputs a hash of size 128 bits. We describe this process now. The block size of the function is 512 bits, i.e. the compression function acts on 512 bits of the message at a time.

Step 1: Padding. We pad the message with 1 followed by as many zeroes as necessary so that the message length is $\equiv 448 \pmod{512}$, i.e. 64 less than a multiple of 512. We do this even if the length of the message is already a multiple of 512. Then, we add the 64 bit representation of the length of the message. For example, suppose the length of the message is 234. Then, $234 = 11101010$ has eight binary digits. We add 56 zeros to the left to get a 64 bit representation of 234 and append this to the message. If the length of the message is greater than 2^{64} we reduce the length $\bmod 2^{64}$. At the end of this step, let the message be $\mathcal{M}_1\mathcal{M}_2\cdots\mathcal{M}_\ell$ where 512ℓ is the total length of the message after padding and each $\mathcal{M}_i$ is 512 bits long.

Step 2: Initialise the Buffer. The intermediate results of the process of creating the hash are stored in a 128 bit buffer. We initialise these registers to the following values written in hexadecimal:

$$\begin{aligned}A &= 67452301\\ B &= EFCDAB89\\ C &= 98BADCFE\\ D &= 10325476\end{aligned}$$

These values are stored in **little-endian** format. What is this format? Note that we can divide 32 bits into four bytes of length 8 bits each. In the little-endian format, the lower bytes are stored at the lower addresses and the higher order bytes at higher addresses. For example, the four bytes in 67452381 will be stored as follows:

Word A : 81 23 45 67

We store the least significant byte, 01 in the left most byte of the 32 bit word which has the smallest address. We store the bytes of higher significance at higher addresses. In the other format, which is called **big-endian**, we store the most significant byte at the lowest address and bytes of progressively higher significance are stored in progressively higher addresses.

Step 3: Process the message in 512 bit blocks. This involves the use of a compression function that processes the 512 bit input in four steps or rounds. In each of these rounds, it uses four different functions F, G, H and I which take three 32-bit words X, Y and Z and output one 32 bit word:

$$\begin{aligned}F(X,Y,Z) &= (X\wedge Y)\vee(\neg X\wedge Z)\\ G(X,Y,Z) &= (X\wedge Z)\vee(Y\wedge\neg Z)\\ H(X,Y,Z) &= X\oplus Y\oplus Z\\ I(X,Y,Z) &= Y\oplus(X\vee\neg Z)\end{aligned}$$

Let us see what these functions are. Here, $\wedge$, $\vee$ and $\oplus$ are the familiar functions from logic. Let us quickly recall what they are. Suppose $x, y \in \{0,1\}$. Recall that:

1) We have $x \vee y = 0$ if and only if both x and y are zero. For all other values of x and y, $x \vee y$ is 1.
2) We have $x \wedge y = 1$ if and only if both x and y are one. For all other values of x and y, $x \wedge y$ is 0.
3) We have $x \oplus y = 1$ if $x \neq y$ and 0 if $x = y$.
4) We have $\neg x = 0$ if $x = 1$ and $\neg x = 1$ if $x = 0$.

Here, X, Y and Z are 32 bit words. Suppose $X = x_1x_2\cdots x_{32}$ and $Y = y_1y_2\cdots y_{32}$ where x_i and y_i are 0 or 1. What do we mean by $(X \vee Y)$, for example? It has the following obvious meaning:

$$X = (x_1x_2\cdots x_{32}) \vee (y_1y_2\cdots y_{32}) = a_1a_2\cdots a_{32} \qquad \text{where } a_i = x_i \vee y_i, 1 \leq i \leq 32$$

Similarly, we apply all the other operations bit by bit.

Addition Modulo 2^{32}: Suppose X and Y are 32 bit representations of two integers x and y, where $0 \leq x < 2^{32}$ and $0 \leq y < 2^{32}$. Then, we define $X + Y$ as follows: Let $z = x + y \pmod{32}$, $0 \leq z \leq 2^{32}$. Let Z be the 32 bit word corresponding to z. Then $Z = X + Y$. For example, suppose $x = 2^{32} - 3$ and $y = 5$. Then,

$$X = \underbrace{111\cdots1}_{30\ 1s}01, Y = \underbrace{000\cdots0}_{29\ 0s}101.$$

Then $x + y = 2^{32} + 2 \equiv 2 \pmod{2^{32}}$. Therefore, $z = 2$ and

$$Z = \underbrace{000\cdots0}_{29\ 0s}10.$$

Another function that we are going to use is the **right and left circular shifts**. We have already discussed this in the the appendix to unit 4.

Step 4 This step uses a table T. Let T[i] be the i-th element of the table. Then, $T[i] = \lfloor |2^{32} \sin i| \rfloor$, i.e. the absolute value of the integer part of $2^{32} \sin i$, where i is in radians. Note that $\mathcal{M}_i$ is 512 bits long. Let $\mathcal{M}_i^{(j)}$ denote the chunk of bits from $32j + 1$th bit to $32(j + 1)$th bit, $0 \leq j \leq 15$. Now, we do as in Algorithm 3 on the next page.

Step 5: A, B, C, D is the 128 bit message digest produced. (Note that the length of each of A, B, C and D is 32 bits.)

We conclude our discussion with some remarks on the current status of MD5. In 1996, a flaw was found by Dobbertin. See [6]. Although the flaw was not considered very serious, cryptographers started recommending other hash functions like SHA-1. However, in [22], Wang, Feng, Lai and Yu found many examples of collisions in many popular hash functions including MD5. See also [3]. In 2007, Marc Stevens, Arjen Lenstra and Benne de Weger MD5 showed how to create a pair of files with the same hash sum. In 2008, Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, Benne de Weger used the technique to fake certificate validity. Also, according to US-CERT of the U. S. Department of Homeland Security MD5 "should be considered cryptographically broken and unsuitable for further use," and most U.S. government applications will be required to move to the SHA-2 family of hash functions after 2010. As of now, MD5 seems useful only for the application we mentioned in the introduction, namely checking files for accidental damages. In 2005, security flaws were found in 2005. Because of these reasons, we discuss SHA-2 in the next subsection.

6.3.2 SHA-2

The SHA-2 is actually a family of hash functions consisting of SHA-224, SHA-256, SHA-384, SHA-512. NSA designed these functions and NIST published them in 2001. In 2005, some attacks were identified in SHA-1, indicating that a stronger hash function is needed. Although SHA-2 is similar to SHA-1, these attacks don't seem feasible for SHA-2 at present. See [17] and [18] for some recent work on SHA-2. A competition organised by NIST is underway to develop SHA-3 and this will end in 2012 with the selection of a winning hash function. Our discussion will follow [14] although our notation is different from [14] in some ways. We will restrict ourselves to SHA-256 only. SHA-224 is very similar to SHA-256 and we will mention how SHA-224 differs from SHA-256.

Algorithm 3 MD5 Algorithm

procedure MD5($\mathcal{M}$) ▷ $\mathcal{M}$ is the message after padding.
  for i $\leftarrow 1, \ell$ **do** ▷ ℓ is the number of blocks in $\mathcal{M}$ of length 512
    for j $\leftarrow 0, 15$ **do** ▷ Copy $\mathcal{M}_i$ into X.
      $X[j] \leftarrow \mathcal{M}_i^{(j)}$
    end for ▷ End of the loop over j.
    ▷ Save A as AA, B as BB, C as CC and D as DD.
    AA $\leftarrow$ A
    BB $\leftarrow$ B
    CC $\leftarrow$ C
    DD $\leftarrow$ D
    ▷ Round 1
    ▷ Let [abcd k s i] denote the operation $a = b + ((a + F(a,b,c,d) + X[k] + T[i]) \lll s)$. Do the following 16 operations.
    [ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3] [BCDA 3 22 4]
    [ABCD 4 7 5] [DABC 5 12 6] [CDAB 6 17 7] [BCDA 7 22 8]
    [ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
    [ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]
    ▷ Round 2
    ▷ Let [abcd k s i] denote the operation $a = b + ((a + G(b,c,d) + X[k] + T[i]) \lll s)$. Do the following 16 operations.
    [ABCD 1 5 17] [DABC 6 9 18] [CDAB 11 14 19] [BCDA 0 20 20]
    [ABCD 5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA 4 20 24]
    [ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28]
    [ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]
    ▷ Round 3.
    ▷ Let [abcd k s i] denote the operation $a = b + ((a + H(b,c,d) + X[k] + T[i]) \lll s)$. Do the following 16 operations.
    [ABCD 5 4 33] [DABC 8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
    [ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40]
    [ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44]
    [ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]
    ▷ Round 4.
    ▷ Let [abcd k s t] denote the operation $a = b + ((a + I(b,c,d) + X[k] + T[i]) \lll s)$. Do the following 16 operations.
    [ABCD 0 6 49] [DABC 7 10 50] [CDAB 14 15 51] [BCDA 5 21 52]
    [ABCD 12 6 53] [DABC 3 10 54] [CDAB 10 15 55] [BCDA 1 21 56]
    [ABCD 8 6 57] [DABC 15 10 58] [CDAB 6 15 59] [BCDA 13 21 60]
    [ABCD 4 6 61] [DABC 11 10 62] [CDAB 2 15 63] [BCDA 9 21 64]
    ▷ Then perform the following additions. (That is, increment each of the four registers by the value it had before this block was started.)
    A = A + AA
    B = B + BB
    C = C + CC
    D = D + DD
  end for ▷ End of loop over i.
end procedure

Table 1: The 64 constants used in SHA-256.

$K_0^{\{256\}} = 428a2f98$	$K_1^{\{256\}} = 71374491$	$K_2^{\{256\}} = b5c0fbcf$	$K_3^{\{256\}} = e9b5dba5$
$K_4^{\{256\}} = 3956c25b$	$K_5^{\{256\}} = 59f111f1$	$K_6^{\{256\}} = 923f82a4$	$K_7^{\{256\}} = ab1c5ed5$
$K_8^{\{256\}} = d807aa98$	$K_9^{\{256\}} = 12835b01$	$K_{10}^{\{256\}} = 243185be$	$K_{11}^{\{256\}} = 550c7dc3$
$K_{12}^{\{256\}} = 72be5d74$	$K_{13}^{\{256\}} = 80deb1fe$	$K_{14}^{\{256\}} = 9bdc06a7$	$K_{15}^{\{256\}} = c19bf174$
$K_{16}^{\{256\}} = e49b69c1$	$K_{17}^{\{256\}} = efbe4786$	$K_{18}^{\{256\}} = 0fc19dc6$	$K_{19}^{\{256\}} = 240ca1cc$
$K_{20}^{\{256\}} = 2de92c6f$	$K_{21}^{\{256\}} = 4a7484aa$	$K_{22}^{\{256\}} = 5cb0a9dc$	$K_{23}^{\{256\}} = 76f988da$
$K_{24}^{\{256\}} = 983e5152$	$K_{25}^{\{256\}} = a831c66d$	$K_{26}^{\{256\}} = b00327c8$	$K_{27}^{\{256\}} = bf597fc7$
$K_{28}^{\{256\}} = c6e00bf3$	$K_{29}^{\{256\}} = d5a79147$	$K_{30}^{\{256\}} = 06ca6351$	$K_{31}^{\{256\}} = 14292967$
$K_{32}^{\{256\}} = 27b70a85$	$K_{33}^{\{256\}} = 2e1b2138$	$K_{34}^{\{256\}} = 4d2c6dfc$	$K_{35}^{\{256\}} = 53380d13$
$K_{36}^{\{256\}} = 650a7354$	$K_{37}^{\{256\}} = 766a0abb$	$K_{38}^{\{256\}} = 81c2c92e$	$K_{39}^{\{256\}} = 92722c85$
$K_{40}^{\{256\}} = a2bfe8a1$	$K_{41}^{\{256\}} = a81a664b$	$K_{42}^{\{256\}} = c24b8b70$	$K_{43}^{\{256\}} = c76c51a3$
$K_{44}^{\{256\}} = d192e819$	$K_{45}^{\{256\}} = d6990624$	$K_{46}^{\{256\}} = f40e3585$	$K_{47}^{\{256\}} = 106aa070$
$K_{48}^{\{256\}} = 19a4c116$	$K_{49}^{\{256\}} = 1e376c08$	$K_{50}^{\{256\}} = 2748774c$	$K_{51}^{\{256\}} = 34b0bcb5$
$K_{52}^{\{256\}} = 391c0cb3$	$K_{53}^{\{256\}} = 4ed8aa4a$	$K_{54}^{\{256\}} = 5b9cca4f$	$K_{55}^{\{256\}} = 682e6ff3$
$K_{56}^{\{256\}} = 748f82ee$	$K_{57}^{\{256\}} = 78a5636f$	$K_{58}^{\{256\}} = 84c87814$	$K_{59}^{\{256\}} = 8cc70208$
$K_{60}^{\{256\}} = 90befffa$	$K_{61}^{\{256\}} = a4506ceb$	$K_{62}^{\{256\}} = bef9a3f7$	$K_{63}^{\{256\}} = c67178f2$

We can use SHA-256 to hash a message $\mathcal{M}$ of length N bits where $0 \leq N < 2^{64}$. The algorithm uses the following:

1) A message schedule of 64 words of length 32 bits, W_0, W_1, …, W_{63}.

2) Eight working variables **a**, **b**, **c**, **d**, **e**, **f**, **g** and **h**.

3) A hash value of eight 32 bit words. During the i^{th} round of the processing(there are ℓ rounds in all, where ℓ is the number of blocks of length 512 in the message after padding) $\mathcal{H}_0^{(i)}$, $\mathcal{H}_1^{(i)}$, …, $\mathcal{H}_7^{(i)}$ hold the intermediate hash values. The final hash values $\mathcal{H}_\ell^{(0)}$, $\mathcal{H}_\ell^{(1)}$, …, $\mathcal{H}_\ell^{(7)}$ are concatenated to give the output of the algorithm.

4) The 64 constants in Table 1 which are the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four prime numbers.

Step 1: Preprocessing. We first pad the message exactly the way we did in the case of MD5 algorithm. We then divide the resulting message into ℓ blocks $\mathcal{M}_1$, $\mathcal{M}_2$, …, $\mathcal{M}_\ell$ of 512 bits each.

Step 2: Initialise the Hash values. We set the initial hash values $\mathcal{H}_0^{(0)}$, $\mathcal{H}_1^{(0)}$, …, $\mathcal{H}_7^{(0)}$.

$$\mathcal{H}_0^{(0)} = 6a09e667$$
$$\mathcal{H}_1^{(0)} = bb67ae85$$
$$\mathcal{H}_2^{(0)} = 3c6ef372$$
$$\mathcal{H}_3^{(0)} = a54ff53a$$
$$\mathcal{H}_4^{(0)} = 510e527f$$
$$\mathcal{H}_5^{(0)} = 9b05688c$$
$$\mathcal{H}_6^{(0)} = 1f83d9ab$$
$$\mathcal{H}_7^{(0)} = 5be0cd19$$

Step 3: Hash computation. Before we describe this step, we have to discuss some new notations that we need in our description.

Let us now define the functions used in SHA-256. They are:

$$\mathrm{Ch}(X,Y,Z) = (X \wedge Y) \oplus (\neg X \wedge X) \tag{1}$$

$$\mathrm{Maj}(X,Y,Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z) \tag{2}$$

$$\sum_0^{\{256\}}(X) = (X \ggg 2) \oplus (X \ggg 13) \oplus (X \ggg 22) \tag{3}$$

$$\sum_1^{\{256\}}(X) = (X \ggg 6) \oplus (X \ggg 11) \oplus (X \ggg 25) \tag{4}$$

$$\sigma_0^{\{256\}}(X) = (X \ggg 7) \oplus (X \ggg 18) \oplus (X \gg 3) \tag{5}$$

$$\sigma_1^{\{256\}}(X) = (X \ggg 17) \oplus (X \ggg 19) \oplus (X \gg 10) \tag{6}$$

Here $\gg$ is the right shift in the C language. This is discussed in page 139 of Block 2, MMT-001.

Now, we process the message as in Algorithm 4 on the next page.

Apart from SHA-2, another algorithm that is still usable is RIPMED-160. This was developed by Dobbertin, Bosselaers and Preneel in the framework of the European Union Project RIPE(Race Integrity Primitives Evaluation). This is based on the MD4 algorithm due to Rivest. It gives an output of length 160 bits. We will not discuss this algorithm in our course.

So far we have discussed the design of hash functions. In the next section, we will discuss the analysis of hash functions.

6.4 BIRTHDAY ATTACKS

In this section, we will discuss one of the techniques for cryptanalysis of hash functions, namely the Birthday Attack. This attack is based on the Birthday Problem in probability theory. So, let us first discuss the Birthday Problem.

Suppose there are n people in a room. What is the probability that two of them have the same birthday? We will ignore leap years and assume that the year has 365 days. Let us assign the n persons n numbers and call them $p_1, p_2, p_3, \ldots, p_n$. We can assume that $n \leq 365$. If $n > 365$, pigeon principle tells us that at least two of the persons should have the same birthday, so the probability is one in this case.

Suppose there are only 2 persons. Then, birthday of the first person falls in a particular day of the year. The probability that the birthday of the second person p_2 also falls on the same day as p_1 is $\frac{1}{365}$. So, the probability that p_2 doesn't have the same birthday as p_1 is $\left(1 - \frac{1}{365}\right)$. If there are three persons, the probability that p_3 doesn't have the same birthday as p_1 and p_2 is $\left(1 - \frac{2}{365}\right)$. So, the probability that all the three have different birthdays is

$$\left(1 - \frac{1}{365}\right)\left(1 - \frac{2}{365}\right)$$

A simple inductive argument tells us that the probability that the n persons have different birthdays is

$$\prod_{i=1}^{n-1}\left(1 - \frac{i}{365}\right) = \left(1 - \frac{1}{365}\right)\left(1 - \frac{2}{365}\right)\cdots\left(1 - \frac{n-1}{365}\right) \tag{7}$$

Therefore, the probability of at least two having the same birthday is

$$p(n) = 1 - \prod_{i=1}^{n-1}\left(1 - \frac{i}{365}\right) \tag{8}$$

Here is the table of probabilities:

Algorithm 4 SHA-256 Algorithm

procedure SHA-256($\mathcal{M}$) ▷ $\mathcal{M}$ is the message after padding.
  for i $\leftarrow 1, \ell$ **do**
    for j $\leftarrow$ 1, 15 **do** ▷ Prepare the message schedule.
      $W_i \leftarrow \mathcal{M}_j^{(i)}$
    end for ▷ End of the loop over j.
    for k $\leftarrow$ 16, 63 **do**
      $W_k \leftarrow \sigma_1^{\{256\}}(W_{k-2}) + W_{k-7} + \sigma_0^{\{256\}}(W_{k-15}) + W_{k-16}$
    end for ▷ End of the loop over k.
  ▷ Initialise the eight working variables **a**, **b**, **c**, **d**, **e**, **f**, **g** and **h** with the (i-1)st hash value.
    $\mathbf{a} \leftarrow \mathcal{H}_0^{(i-1)}$
    $\mathbf{b} \leftarrow \mathcal{H}_1^{(i-1)}$
    $\mathbf{c} \leftarrow \mathcal{H}_2^{(i-1)}$
    $\mathbf{d} \leftarrow \mathcal{H}_3^{(i-1)}$
    $\mathbf{e} \leftarrow \mathcal{H}_4^{(i-1)}$
    $\mathbf{f} \leftarrow \mathcal{H}_5^{(i-1)}$
    $\mathbf{g} \leftarrow \mathcal{H}_6^{(i-1)}$
    $\mathbf{h} \leftarrow \mathcal{H}_7^{(i-1)}$
    for t $\leftarrow$ 0,63 **do**
      $T_1 \leftarrow \mathbf{h} + \sum_1^{\{256\}}(\mathbf{e}) + Ch(\mathbf{e},\mathbf{f},\mathbf{g}) + K_t^{\{256\}} + W_t$
      $T_2 \leftarrow \sum_0^{\{256\}}(\mathbf{a}) + Maj(\mathbf{a},\mathbf{b},\mathbf{c})$
      $\mathbf{h} \leftarrow \mathbf{g}$
      $\mathbf{g} \leftarrow \mathbf{f}$
      $\mathbf{f} \leftarrow \mathbf{e}$
      $\mathbf{e} \leftarrow \mathbf{d} + T_1$
      $\mathbf{d} \leftarrow \mathbf{c}$
      $\mathbf{c} \leftarrow \mathbf{b}$
      $\mathbf{b} \leftarrow \mathbf{a}$
      $\mathbf{a} \leftarrow T_1 + T_2$
    end for ▷ End of the loop over t.
  ▷ Compute the ith intermediate hash value $\mathcal{H}^{(i)}$.
    $\mathcal{H}_0^{(i)} \leftarrow \mathbf{a} + \mathcal{H}_0^{(i-1)}$
    $\mathcal{H}_1^{(i)} \leftarrow \mathbf{b} + \mathcal{H}_1^{(i-1)}$
    $\mathcal{H}_2^{(i)} \leftarrow \mathbf{c} + \mathcal{H}_2^{(i-1)}$
    $\mathcal{H}_3^{(i)} \leftarrow \mathbf{d} + \mathcal{H}_3^{(i-1)}$
    $\mathcal{H}_4^{(i)} \leftarrow \mathbf{e} + \mathcal{H}_4^{(i-1)}$
    $\mathcal{H}_5^{(i)} \leftarrow \mathbf{f} + \mathcal{H}_5^{(i-1)}$
    $\mathcal{H}_6^{(i)} \leftarrow \mathbf{g} + \mathcal{H}_6^{(i-1)}$
    $\mathcal{H}_7^{(i)} \leftarrow \mathbf{h} + \mathcal{H}_7^{(i-1)}$
  end for ▷ End of the loop over i.
end procedure

Table 2: Probability of at least two among n persons having the same birthday.

n	p(n)
10	0.1388
15	0.2816
20	0.4422
21	0.4743
22	0.5059
23	0.5371
24	0.5675
25	0.5971
30	0.7297
35	0.8317
40	0.9029
45	0.9481
50	0.9744
52	0.9811
54	0.9862
56	0.9901
57	0.9916

From Table 2, we see that, if there are 23 people in a room, the probability is slightly more than 50% that two of them have the same birthday. If there are 30, the probability is around 70%. If there are 57 people, the probability is nearly 99%! This might seem surprising; this phenomenon is called the **Birthday Paradox**.

Note that $1+x \approx e^x$. So, $1-x \approx e^{-x}$. Using this approximation in Eqn. (8) on page 59, we get

$$p(n) = 1 - \prod_{i=1}^{n-1}\left(1 - \frac{i}{365}\right) \approx 1 - \prod_{i=1}^{n} e^{\frac{i}{365}} = 1 - e^{\frac{n(n-1)}{2\times 365}} \approx 1 - e^{n^2/730} \tag{9}$$

Let us consider a more general situation. Suppose there are N objects and r persons. Each person selects an object with replacement so that more than one person can select the same object. What is the probability that two people select the same object?

We can use the same argument to get the probability as

$$1 - \prod_{i=1}^{r-1}\left(1 - \frac{i}{N}\right) \approx 1 - e^{r^2/N}. \tag{10}$$

More generally, if there are N objects and two groups of sizes r_1 and r_2, the probability that two persons from different groups pick the same object is approximately

$$1 - e^{-\frac{r_1 r_2}{N}} \tag{11}$$

If $r_1 = r_2 = \sqrt{N}$, then the probability is approximately $1 - e^{-1} \approx 0.6321 > \frac{1}{2}$. If we take $r_1 = r_2 = 2\sqrt{N}$, then the probability is $1 - e^{-2} \approx 0.8647$ which is much higher.

These ideas can be used to find collisions for hash functions. Suppose a hash function h produces an output that is n-bits long. Then, the number of possible values of the hash function is 2^n. We make a list of values of $h(x)$ of length $\sqrt{N} = 2^{n/2}$. If any of the two values among $2^{n/2}$ match, we have found a collision. Here, we are in the situation of $N = 2^n$ objects and $r = 2^{n/2}$ persons. As we saw earlier, the probability that two hash values match is ≈ 0.6321. The probability of finding a match becomes much higher if we take $r = 10\sqrt{N}$.

If the output of the hash function is 60 bits, for example, the above attack has a high chance of finding a collision. We need to make a list of size approximately

$2^{n/2} = 2^{30} \approx 10^9$ and to store them. This is possible on most computers. However, if the hash function outputs 128-bit values, the list should have a size of $2^{64} \approx 10^{20}$ which is large in terms of time and memory needed at present.

6.5 SUMMARY

In this Unit we have discussed the following:

1. What is a cryptographic hash function;
2. What is a compression function;
3. How to construct compression functions from block ciphers using Davies-Meyer, Matyas-Meyer-Oseas and Miyaguchi-Preneel methods;
4. The Merkle-Damgård method for constructing hash functions from compression functions;
5. The working of MD5 and SHA-2 algorithms; and
6. The birthday attack on hash functions.

6.6 SOLUTIONS/ANSWERS

E1) The length of the string is 136. We split up the string into three strings "tohashor", "nottohas" and "h". We add a one to the end of "h1" to get "h1". This is of length nine, so we add 55 zeros to get a block of size 64. The length of the string, 136, in binary is 10001000. The length of 136, written in binary, is 8. So, we add 56 zeros to the left of this string to get $\underbrace{00\ldots00}_{56\text{ zeros}}10001000$ So, we get the three blocks "tohashor", "nottohas", "h1$\underbrace{00\ldots00}_{55\text{ zeros}}$" and "$\underbrace{00\ldots00}_{56\text{ zeros}}10001000$".

[1] E. Biham and A. Shamir, *Differential Cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer*, Proc. CRYPTO 91 (J. Feigenbaum, ed.), Springer, 1992, Lecture Notes in Computer Science No. 576, pp. 156–171.

[2] _____, *A Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.

[3] J. Black, M. Cochran, and T. Highland, *A Study of the MD5 Attacks: Insights and Improvements*, 2006, `http://www.cs.colorado.edu/~jrblack/papers/md5e-full.pdf`.

[4] Don Coppersmith, *The data encryption standard (DES) and its strength against attacks*, Tech. Report RC 18613(81421), IBM T.J. Watson Research Center, December 1992.

[5] W. Diffie and M. E. Hellman, *Exhaustive cryptanalysis of the NBS data encryption standard*, Computer **10** (1977), 74–84.

[6] Hans Dobbertin, *The Status of MD5 After a Recent Attack.*, CryptoBytes **2** (1996), no. 2, 1–6.

[7] Electronic Frontier Foundation, *Cracking des: Secrets of encryption research, wiretap politics and chip design*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[8] D. E. Knuth, *Seminumerical algorithms*, 3rd ed., The Art of Computer Programming, vol. 2, Addison Wesley, Reading MA, 1997.

[9] M. Matsui, *Linear cryptanalysis method for DES cipher*, Advances in Cryptology — Eurocrypt '93 (Berlin) (T. Helleseth, ed.), Lecture Notes in Computer Science, vol. 765, Springer-Verlag, 1994, pp. 386–397.

[10] Ueli M. Maurer, *A universal statistical test for random bit generators*, Proc. CRYPTO 90 (A. J. Menezes and S. A. Vanstone, eds.), Springer-Verlag, 1991, Lecture Notes in Computer Science No. 537, pp. 409–420.

[11] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.

[12] R. C. Merkle and M. E. Hellman, *On the security of multiple encryption*, Communications of the ACM **24** (1981), 465–467.

[13] National Bureau of Standards, *Random and pseudo random number generators for cryptographic applications*, Tech. Report Special Publication 800-22, Revision 1, National Bureau of Standards, 2008.

[14] National Bureau of Standard(SHS), *Secure hash standard*, Tech. Report FIPS Publication 180-3, National Bureau of Standards, October 2009.

[15] M. J. B. Robshaw, *Block ciphers*, Tech. Report TR - 601, RSA Laboratories, July 1994.

[16] M.J.B. Robshaw, *Security of RC4*, Tech. Report TR-401, RSA Laboratories, revised July 1995.

[17] Somitra Kumar Sanadhya and Palash Sarkar, *New collision attacks against up to 24-step sha-2*, INDOCRYPT '08: Proceedings of the 9th International Conference on Cryptology in India (Berlin, Heidelberg), Springer-Verlag, 2008, pp. 91–103.

[18] Yu Sasaki, Lei Wang, and Kazumaro Aoki, *Preimage attacks on 41-step sha-256 and 46-step sha-512*, Cryptology ePrint Archive, Report 2009/479, 2009, http://eprint.iacr.org/.

[19] Bruce Schneier, *Applied cryptography (second edition)*, John Wiley & Sons, 1996.

[20] C. E. Shannon, *Communication theory of secrecy systems*, Bell Sys. Tech. J. **28** (1949), 657–715.

[21] Wade Trappe and Lawrence C. Washington, *Introduction to cryptography with coding theory*, second ed., Pearson, 2006.

[22] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu, *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*, Cryptology ePrint Archive, Report 2004/199, 2004, http://eprint.iacr.org/.

[23] Wikipedia, *One-way compression function — Wikipedia, The Free Encyclopedia*, 2010, http://en.wikipedia.org/w/index.php?title=One-way_compression_function&oldid=359260715.

[24] ———, *Rc4 — Wikipedia, The Free Encyclopedia*, 2010, http://en.wikipedia.org/w/index.php?title=RC4&oldid=385372983.