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 \( k_1, k_2, k_3, \ldots \). 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 \([0, 2^{32}]\) 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 $x_1, x_2, \ldots$, 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 $x_0$.  

---

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

\[
\begin{align*}
  x_0 &= 311021 \\
  x_1 &= 43952889436 \\
  x_2 &= 1443650955 \\
  x_3 &= 94597186059 \\
  x_4 &= 107971367060 \\
  x_5 &= 97956752491 \\
  x_6 &= 72009624215 \\
  x_7 &= 125065550422 \\
  x_8 &= 61609613347
\end{align*}
\]

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 \( x_0 = x^2 \pmod{n} \). 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

\[
\begin{align*}
  x_0 &= 96734062441 \\
  x_1 &= 49181739960 \\
  x_2 &= 116601795484 \\
  x_3 &= 87163050345 \\
  x_4 &= 65384689558 \\
  x_5 &= 85100903044 \\
  x_6 &= 24926545774 \\
  x_7 &= 98076507435
\end{align*}
\]

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

\[
x_{n+k} \equiv a_{k-1}x_{n+k-1} + a_{k-2}x_{n+k-2} + \cdots + a_0x_n \quad (\text{mod } p)
\]

(3)

where \( a_i \in \mathbb{F}_p \). The polynomial

\[
x^k - a_1x^{k-1} - \cdots - a_{k-2}x - a_0 \quad (4)
\]

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 \( x_1, x_2, \ldots, x_k \in \mathbb{F}_p \) 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 \( \mathbb{F}_{p^k} \).

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

\[
x_{n+5} \equiv x_{n+2} + x_n \quad (\text{mod } 2).
\]

(5)

Its characteristic polynomial \( x^5 + x^3 + 1 \) is an irreducible polynomial over \( \mathbb{F}_2 \) and its root generates the finite field \( \mathbb{F}_{2^5} \). Why is this so? Note that, the multiplicative group of \( \mathbb{F}_{2^5} \) 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

\[
01000010010110011110001101110101. \quad (6)
\]

We can generate this sequence by choosing \((0, 1, 1, 0, 0, 0)\) 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:

\[
\begin{align*}
(\text{plaintext}) & \quad 1111000111010110 \\
(\text{key}) & \quad 0100001001011101 \\
(\text{ciphertext}) & \quad 1011001110001111
\end{align*}
\]

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 \( \mathbb{F}_2 \). 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

\[x_{n+31} \equiv x_{n+3} + x_n.\]  (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 \(\mathbb{F}_2[x]\) 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.](image)

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 \(C_2\)
Stream Ciphers

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

$$x_{n+k} \equiv a_{k-1}x_{n+k-1} + a_{k-2}x_{n+k-2} + \cdots + a_0x_n \pmod{2}$$

the LFSR has $k$ cells $C_1, C_2, \ldots, C_{k-1}, C_k$ and the $i$th 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 $C_2, C_3, \ldots, C_k$ 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 \textit{nth state vector}. Then, $S_n$ describes the contents of the cells $C_1, C_2, \ldots, C_k$ after $n$ clock cycles. Thus, $S_0$ describes the initial state $(x_1, x_2, \ldots, x_k)$ and is called the \textit{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:

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

\[ x_{n+2} = c_0 x_n + c_1 x_{n+1}. \]  

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

\[ 0 \equiv c_0 \cdot 0 + c_1 \cdot 1 \quad (n = 1) \]
\[ 1 \equiv c_0 \cdot 1 + c_1 \cdot 0 \quad (n = 2). \]

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 \( x_{n+2} \equiv x_n \) (mod 2). Since \( x_5 \not\equiv x_3 \) (mod 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 \( c_0 = 1, c_1 = 1, c_2 = 0 \), so the recurrence could be \( x_{n+3} \equiv x_n + x_{n+1} \) (mod 2). However, our guess turns out to be wrong again because for \( n = 3 \) we should have \( x_6 \equiv x_3 + x_4 \) (mod 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:

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

36
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 \(x_1, x_2, \ldots, x_{100}\). For \(k = 2, 3, 4, \ldots\), we form the \(k \times k\) matrix

\[
M_k = \begin{pmatrix}
x_1 & x_2 & \ldots & x_k \\
x_2 & x_3 & \ldots & x_{k+1} \\
\vdots & \vdots & \ddots & \vdots \\
x_k & x_{k+1} & \ldots & x_{2k-1}
\end{pmatrix}
\]

as before and compute its determinant. If, for several consecutive values of \(k\), \(\det(M_k) \equiv 0 \pmod{2}\) 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:

\[
\begin{align*}
100110011001110001100010100011110110011111010101001 \\
01101101011000011011100101011110000000100010010000 \\
\end{align*}
\]

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

\[
x_{16} \equiv x_{17} + x_{20} + x_{24} \equiv 1 + 0 + 1 \equiv 0
\]

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 \( x_{n+1} = x_n \). 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, 0, 1 . . . 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 . . . , it is not clear \textit{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

\[
M_n = \begin{pmatrix}
x_1 & x_2 & \cdots & x_n \\
x_2 & x_3 & \cdots & x_{n+1} \\
\vdots & \vdots & \ddots & \vdots \\
x_0 & x_{n+1} & \cdots & x_{2n-1}
\end{pmatrix}
\]

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

\[
(x_3, x_4, \ldots, x_{n+2}) = (x_2 + x_1, x_3 + x_2, \ldots, x_{n+1} + x_n) = (x_1, x_2, \ldots, x_n) + (x_2, x_3, \ldots, x_{n+1}).
\]

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 \( n \geq 3 \). More generally, if the sequence satisfies the recurrence \((\text{Eqn. (3)})\) and \( i_1, i_2, \ldots, i_r \) are the values of \( k \) for which the coefficient \( a_k \neq 0 \), the \((n+k)\)th row is the linear combination of the rows \( i_1, i_2, \ldots, i_r \).

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 \( \vec{b} = (b_0, \ldots, b_{N-1}) \) such that \( \vec{b}M_N \equiv 0 \).

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 \(\tilde{b}\), we have \(\tilde{b}M^{(0)} = \tilde{b}M_N = 0\). Suppose that we know that \(\tilde{b}M^{(i)} = 0\) for some \(i\). Then

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

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

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

\[
b_0x_n + b_1x_{n+1} + \cdots + b_{N-1}x_{n+N-1} = 0.
\]

Since \(\tilde{b}\) 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 \((\text{mod} \ 2)\), \(b_m x_{n+m} \equiv -x_{n+m}\). 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 \(x_k, x_{k+1}, \ldots, x_{k+m}\), we can determine all the previous terms \(x_i, x_2, \ldots, x_{k-1}\) and all the following terms \(x_{k+m+1}, x_{k+m+2}, \ldots\). 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 \( \chi^2 \) 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 \( \alpha \). Recall that \( \alpha \) 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

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

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 \( k \),

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

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

\[
X_1 = \frac{(n_0 - n_1)^2}{n}
\]

which follows an approximately \( \chi^2 \) distribution with one degree of freedom if \( n \geq 10 \). In our case, \( n_0 = 11 \), \( n_1 = 13 \), \( \chi_s = X_1 = \frac{4}{23} \approx 0.1667 \). The value of \( \chi_\alpha = \chi_{0.05,1} \) which can be found by looking at the row corresponding to \( \nu = 1 \) and the column under \( \alpha = 0.05 \) in the \( \chi^2 \) 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

\[
X_2 = \frac{2}{n-1} (n_{00} + n_{01} + n_{10} + n_{11}) - \frac{2n}{n} (n_0^2 + n_1^2) + 1
\]

It is known that \( X_2 \) follows an approximately \( \chi^2 \) distribution with two degrees of freedom. We have

<table>
<thead>
<tr>
<th></th>
<th>( n_0 )</th>
<th>( n_{00} )</th>
<th>( n_1 )</th>
<th>( n_{01} )</th>
<th>( n_{10} )</th>
<th>( n_{11} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Therefore,

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

The value of \( \chi_\alpha = \chi_{0.05,2} = 5.99146 \) and \( \chi_s < \chi_\alpha \), so this sequence passes this test also.

3. Poker Test

Let \( m \) be a positive integer such that \( \left\lfloor \frac{n}{m} \right\rfloor > 5 \cdot (2^m) \) and \( k = \left\lfloor \frac{n}{m} \right\rfloor \). 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 \( m \), \( 1 \leq i \leq 2^m \), occurs. The poker test checks if all the possible sequences of length \( m \) occurs approximately equal number of times. The statistic used is

\[
X_3 = \frac{2^m}{k} \left( \sum_{i=1}^{2^m} n_i^2 \right) - k
\]

which follows an approximately \( \chi^2 \) 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 \( \frac{24}{m} \times 5 \times (2^m) \) to be satisfied. So, the sequence is too small. So, we take the following longer sequence of length 48 for which we can take \( m = 2 \).

\[
11001001000011111011010100010001011010000101101000
\]

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

\[
11|00|01|00|01|1|1|1|0|1|0|0|0|0|0|0|0|0|0|0|1|0|0|0|0|0
\]

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

<table>
<thead>
<tr>
<th>S.No</th>
<th>Type</th>
<th>No. of occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>01</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>4</td>
</tr>
</tbody>
</table>

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

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

For \( \alpha = 0.05 \) and \( 2^2 - 1 = 3 \) degrees of freedom, the value in the \( \chi^2 \) table is 7.81473. So \( \chi_s = 7.81473 \) 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 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 \( \chi^2 \) 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 \( e_i \) s 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

\[
n - 3 + 3 \geq 5.25 \text{ or } n \geq 160.
\]

So, for this test also, we will use a longer sequence. In our case \( n = 160 \).
From Eqn. (12) we get the following information:

<table>
<thead>
<tr>
<th>i</th>
<th>B_i</th>
<th>G_i</th>
<th>e_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>20</td>
<td>20.25</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>14</td>
<td>10.0625</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

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

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

The value in the \chi^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 \left\lfloor \frac{n}{2} \right\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

\[ X_5 = 2 \left( A(d) - \frac{n-d}{2} \right) / \sqrt{n-d} \]  

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.

\[
\begin{align*}
110010010000111110101 & \quad \text{(First 20 terms of the sequence)} \\
\oplus & 1001000011111011010 \quad \text{(20 terms of the sequence starting from 5th term)} \\
\hline
010110011111001001111 & \quad \text{The last row is 12, so 12 terms of the sequence are not equal to their four shifts, i.e. A(4) = 12. So,}
\end{align*}
\]

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

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
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 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 servers. 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)

1: for \(i \leftarrow 0\) to 255 do
2: \(S[i] \leftarrow i\)
3: end for
4: \(j \leftarrow 0\)
5: for \(i \leftarrow 0\) to 255 do
6: \(j \leftarrow (j + S[i] + \text{key}[i \mod \text{keylength}]) \mod 256\)
7: swap(&S[i],&S[j])
8: 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

1: \(i \leftarrow 0\) \(j \leftarrow 0\)
2: while GeneratingOutput do do
3: \(i \leftarrow (i + 1) \mod 256\)
4: \(j \leftarrow (j + S[i]) \mod 256\)
5: swap(&S[i],&S[j])
6: byte_key \leftarrow S[S[i] + S[j]] \mod 256]
7: result_ciphered \leftarrow byte_key XOR byte_message
8: 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

\[ C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4 \rightarrow C_6 \]

Output

Fig. 5: Answer to exercise 1.

E2) \( x_{n+5} \equiv x_{n+4} + x_{n+2} + x_{n+1} + x_m \pmod{2} \)

E3) Writing \( x_{n+2} = c_0x_1 + c_1x_2 \), we get the matrix equation

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

in \( F_2 \) 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.
<table>
<thead>
<tr>
<th>$\nu$</th>
<th>$\alpha = 0.500$</th>
<th>$\alpha = 0.250$</th>
<th>$\alpha = 0.100$</th>
<th>$\alpha = 0.050$</th>
<th>$\alpha = 0.025$</th>
<th>$\alpha = 0.010$</th>
<th>$\alpha = 0.005$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.45494</td>
<td>1.32330</td>
<td>2.70554</td>
<td>3.84146</td>
<td>5.02389</td>
<td>6.63490</td>
<td>7.87944</td>
</tr>
<tr>
<td>2</td>
<td>1.38629</td>
<td>2.77259</td>
<td>4.60517</td>
<td>5.99146</td>
<td>7.37776</td>
<td>9.21034</td>
<td>10.59663</td>
</tr>
<tr>
<td>15</td>
<td>14.33886</td>
<td>18.24509</td>
<td>22.30713</td>
<td>24.99579</td>
<td>27.48839</td>
<td>30.57791</td>
<td>32.80132</td>
</tr>
<tr>
<td>16</td>
<td>15.33850</td>
<td>19.36886</td>
<td>23.54183</td>
<td>26.29623</td>
<td>28.84535</td>
<td>31.99993</td>
<td>34.26719</td>
</tr>
<tr>
<td>17</td>
<td>16.33818</td>
<td>20.48868</td>
<td>24.76904</td>
<td>27.58711</td>
<td>30.19101</td>
<td>33.40866</td>
<td>35.71847</td>
</tr>
<tr>
<td>18</td>
<td>17.33790</td>
<td>21.60489</td>
<td>25.98942</td>
<td>28.86930</td>
<td>31.52638</td>
<td>34.80531</td>
<td>37.15645</td>
</tr>
<tr>
<td>19</td>
<td>18.33765</td>
<td>22.71781</td>
<td>27.20357</td>
<td>30.14353</td>
<td>32.85233</td>
<td>36.19087</td>
<td>38.58226</td>
</tr>
<tr>
<td>20</td>
<td>19.33743</td>
<td>23.82769</td>
<td>28.41198</td>
<td>31.41043</td>
<td>34.16961</td>
<td>37.56623</td>
<td>39.99685</td>
</tr>
</tbody>
</table>

| Table 3: CHI-SQUARED TABLE |
Table 4: Some percentiles of standard normal distribution. If $X$ is a random variable having a standard normal distribution, then $P(X < x) = a$.

<table>
<thead>
<tr>
<th>$x$</th>
<th>0.841559</th>
<th>0.674490</th>
<th>1.644854</th>
<th>1.959964</th>
<th>2.326348</th>
<th>2.575829</th>
<th>2.807033</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(X &lt; x)$</td>
<td>0.1000</td>
<td>0.0002</td>
<td>0.0005</td>
<td>0.01</td>
<td>0.025</td>
<td>0.05</td>
<td>0.1</td>
</tr>
</tbody>
</table>