example, describe how the window scale option used in the TCP segment header can overcome this problem.

12.40 The computation of the RTO for a connection using the method in Exercise 12.24 required the RTT to be computed for every data segment. Explain how, with a large window size, this can lead to a poor estimate of the RTT.

Hence, with the aid of the time sequence diagram shown in Figure 12.17, explain how the number of updates can be reduced by using a time-stamp option field. Include in your explanation the use of the *time-stamp value* and *time-stop echo reply* fields and how the receiving TCP overcomes the fact that not all segments are acknowledged.

12.41 Explain the principles behind:
(i) the SACK-permitted option
(ii) how protection against wrapped sequence numbers can be overcome by using the time-stamp option.

12.42 Use the time sequence diagrams associated with the connection established (Figure 12.5) and connection close (Figure 12.12) procedures to follow the state transitions that occur at the client and server sides shown in Figure 12.18(a) and (b) respectively.

Section 12.4

12.43 What are the main differences between UDP and TCP?

12.44 By means of a diagram, show the socket interface associated with UDP in relation to a user AP. Include in your diagram the send and receive buffers associated with the socket and the input and output buffers associated with the UDP entity.

12.45 Show on a diagram a typical sequence of socket primitives that are issued at both the sending and the receiving sides to:
(i) establish a socket connection,
(ii) exchange a single UDP datagram,
(iii) release the socket connection.
Identify the main parameters associated with each primitive.

12.46 In relation to the UDP datagram format shown in Figure 12.20(b), explain how the checksum is corrupted.

State why the maximum size of UDP datagram is often less than the theoretical maximum.

Section 12.5

12.47 Describe the use of the RTP protocol and, by means of a diagram, show its position in relation to the TCP/IP protocol stack.

12.48 In relation to the RTP packet format shown in Figure 12.21(b), explain the meaning and use of the following fields:
(i) CC and CSRC,
(ii) M and payload type,
(iii) sequence number,
(iv) time-stamp,
(v) SSRC.

12.49 Describe the use of the RTCP protocol and, by means of a diagram, show its position in relation to the TCP/IP protocol stack.

12.50 Identify and give a brief explanation of the four main functions performed by RTCP.

13

Application support functions

13.1 Introduction

Having described the various protocols that are used to transfer information across a network/internet we are now in a position to describe a selection of the standard protocols associated with various applications. Before we do this, however, it will be helpful if we first build up an understanding of some of the support functions that are used with many of these application protocols.

For example, if you were asked to write an application program to process a set of fault reports that have been gathered from the various items of computer-based equipment that make up a network – switching exchanges, bridges, gateways, routers, and so on – then you would, of course, want to use a suitable high-level programming language. Each report would then be declared in the form of, say, a (record) structure with the various fields in each record declared as being of suitable types. However, although the data types used may be the same as those used by the programmer who created the software within each item of equipment, the actual representation of each field after compilation may be quite different in each equipment. For example, in one computer an integer type may be represented by a 16-bit value while in another it may be represented by 32 bits.

Even if the two computers both use 16 bits to represent an integer type, the position of the sign bit or the order of the two bytes that make up each integer may still be different. Similarly, the character types used in different computers also differ. For example, EBCDIC may be used in one and ASCII/IA5 in another. The representation of the different data types are thus said to be in an **abstract syntax** form.

The effect of this is that when we pass a block of data – for example, holding a set of records – from one computer to another, we cannot simply transfer the block of data from one computer memory to that of the other since the program in the receiving computer may interpret the data incorrectly such as on the wrong byte boundaries. Consequently, when we transfer data between computers, we must ensure that the syntax of the data is known by the receiving machine and, if this is different from its local syntax, convert the data into this syntax prior to processing.

One approach to this problem is to define a **data dictionary** for the complete (distributed) application which contains an application-wide definition of the representation of all the data types used in the application. If this representation is different from the local representation used by a machine, we must convert all data received into its local syntax prior to processing and convert it back into the standard form if it is to be sent to another machine. The form used in the data dictionary is known as the **concrete** or **transfer** syntax for the application.

This is a common requirement in many distributed applications, especially when computers and other items of equipment from different manufacturers are involved. Hence to meet this requirement, an international standard has been defined for representing information that is to be transferred between (possibly dissimilar) computers and other items of computer/microprocessor based equipment. This is called **abstract syntax notation one** (**ASN.1**) and is defined in **IS 8824**. As we shall see, this comprises both an abstract syntax for defining the data types associated with a distributed application and also a transfer syntax for representing each data value during its transfer over the network.

A second requirement that relates to many distributed applications is network security. Increasingly, people are using networks like the Internet for banking, home shopping, and many other applications that involve the transfer of sensitive information such as credit card details over the network. As the knowledge of computer networking and their protocols has become more widespread, so the threat of intercepting and decoding the data within messages during its transfer across the network has increased. To combat these threats, a number of security techniques have been developed which, when combined together, provide a high level of confidence that any information that is received from the network has come from the stated source and has not been read or changed during its transfer over the network. In this chapter, we shall present an overview of both these topics and, in the following two chapters, some examples of applications that use them.

13.2 ASN.1

ASN.1 is concerned with the representation (syntax) of the data in the messages associated with a distributed/networked application during its transfer between two application processes (APs). The aim is to ensure that the messages exchanged between the two APs have a common meaning – known as **shared semantics** – to both processes. The approach adopted is shown in Figure 13.1.

With any distributed application all APs involved must know the syntax of the messages associated with the application. For example, if the application involves customer accounts, the APs in all systems that process them must be written to interpret each field in an account in the same way. However, as we indicated earlier, the representation of data types associated with a high-level programming language may differ from one computer to another. To ensure that data is interpreted in the same way, before any data is transferred between two processes it must be converted from its local (abstract) syntax into an application-wide transfer syntax. Similarly, before any received data is processed, it must be converted into the local syntax if this is different from the transfer syntax.

Two questions arise from this: firstly, what abstract syntax should be used, and secondly, what representation should be adopted for the transfer syntax. One solution to the first question is to assume that all programming is done in the same high-level programming language and then to declare all data types relating to the application using this language. However, different programmers may prefer to use different languages. Also, the question of the transfer syntax is still unanswered.

As the representation of data in a distributed application is a common requirement, ISO (in cooperation with ITU-T) has defined ASN.1 as a general abstract syntax that is suitable for the definition of data types associated with most distributed applications. An example application that uses ASN.1 is

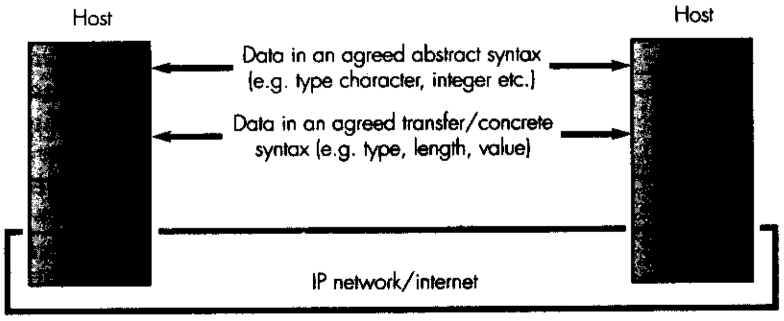

Host
Host
Data in an agreed abstract syntax (e.g. type character, integer etc.)
Data in an agreed transfer/concrete syntax (e.g. type, length, value)
IP network/internet

Figure 13.1 ASN.1: abstract and transfer/concrete syntax relationship.

the simple network management protocol which we describe in Section 14.7 of the next chapter. As the name implies, the data types associated with ASN.1 are abstract types. Hence, in addition to the abstract syntax definition, an associated transfer syntax has also been defined.

As an aid to the use of ASN.1, a number of companies now sell **ASN.1 compilers** for a range of programming languages. There is an ASN.1 compiler for Pascal and another for C. The general approach for using such compilers is shown in Figure 13.2.

Firstly, the data types associated with an application are defined using ASN.1. For example, if two APs are to be written, one in Pascal and the other in C, the ASN.1 type definitions are first processed by each compiler. Their output is the equivalent data type definitions in the appropriate language together with a set of **encoding** and **decoding procedures/functions** for each data type. The data type definitions are linked and used with the corresponding application software, while the encoding and decoding procedures/functions are used as library procedures/functions: each encoding procedure/function is used to encode the related value into its corresponding transfer syntax ready for transferring to the destination AP and each decoding procedure/function is used to decode a received value from its transfer syntax into its local syntax prior to processing.

As we shall see, the output of each encoding procedure/function is in the form of a byte – normally referred to as an octet – string comprising an identifier, the length (in bytes) of the encoded value, and the encoded value. The identifier is then used to determine the type of decoding procedure/function that should be carried out on the value.

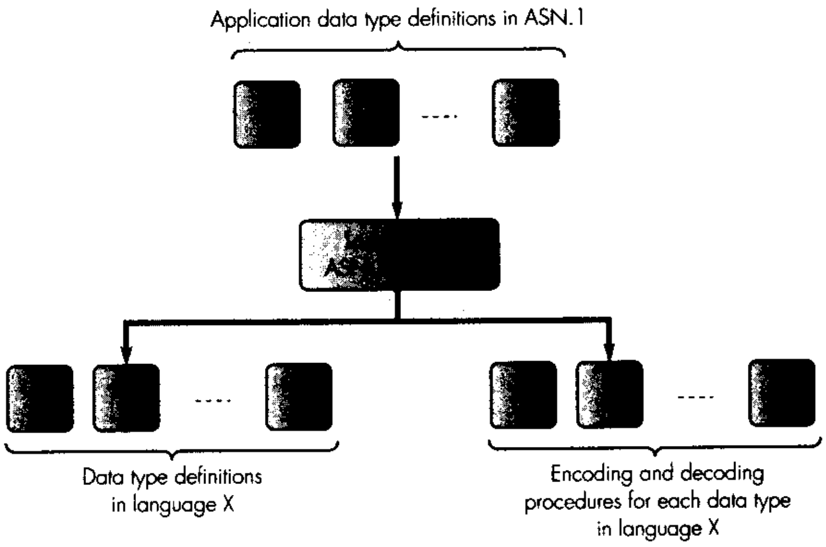

Application data type definitions in ASN.1
AS
Data type definitions in language X
Encoding and decoding procedures for each data type in language X

Figure 13.2 ASN.1 compiler function.

13.2.1 Type definitions

The type definitions used with ASN.1 are defined in IS 8824. They are similar to those used with most high-level programming languages for defining the data types associated with the variables used in a program: as each variable is declared, the data type associated with it is also defined. Then, when a value is assigned to the variable, its syntax is of the defined type.

ASN.1 supports a number of type identifiers, which may be members of the following four classes:

- UNIVERSAL: the generalized types such as integer;
- CONTEXT-SPECIFIC: these are related to the specific context in which they are used;
- APPLICATION: these are common to a complete application;
- PRIVATE: these are user definable but must begin with an upper-case letter.

The data types associated with the UNIVERSAL class may be either primitive (simple) or constructed (structured). A primitive type is either a basic data type that cannot be decomposed – for example, a BOOLEAN or an INTEGER – or, in selected cases, a string of one or more basic data elements all of the same type – for example, a string of one or more bits, octets, or IA5/graphical characters. The keywords used with ASN.1 are always in upper-case letters and the primitive types available include:

UNIVERSAL (primitive): BOOLEAN
INTEGER
BITSTRING
OCTETSTRING
REAL
ENUMERATED
IA5String/DisplayString
NULL
ANY

The names of variables and constants may consist of upper and lower-case letters, digits, and hyphens but must begin with a lower-case letter. Some examples of simple types are shown in Figure 13.3(a).

As with a program listing, we may insert comments at any point in a line; the comments start with a pair of adjacent hyphens and end either with another pair of hyphens or the end of a line. The assignment symbol is ::= and the individual bit assignments associated with a BITSTRING type are given in braces with the bit position in parentheses. A similar procedure is used for the ENUMERATED type to identify the possible values of the vari-

```
(a)          married ::= BOOLEAN -- true or false
     yrsWithCompany ::= INTEGER
       accessRights ::= BITSTRING{read(0), write(1)}
        PDUContents ::= OCTETSTRING
               name ::= IA5String
                 pi ::= REAL -- mantissa, base, exponent
            workDay ::= ENUMERATED{monday(0), tuesday(1) ... friday(4)}

(b)  personnelRecord ::= SEQUENCE{
                              empNumber INTEGER,
                              name IA5String,
                              yrsWithCompany INTEGER
                              married BOOLEAN}

   c.f. personnelRecord =  record
                              empNumber = integer;
                              name = array [1..20] of char;
                              yrsWithCompany = integer;
                              married = boolean
                           end;

(c)  personnelRecord ::= SEQUENCE{
                              empNumber [APPLICATION1] INTEGER,
                              name [1] IA5String,
                              yrsWithCompany [2] INTEGER,
                              married [3] BOOLEAN}

(d)  personnelRecord ::= SEQUENCE{
                              empNumber [APPLICATION1] INTEGER,
                              name [1] IMPLICIT IA5String,
                              yrsWithCompany [2] IMPLICIT INTEGER,
                              married [3] IMPLICIT BOOLEAN}
```

Figure 13.3 Some example ASN.1 type definitions: (a) simple types; (b) constructed type; (c) tagging; (d) implicit typing.

able. INTEGER types are signed whole numbers of, in theory, unlimited magnitude while REAL types are represented in the form {m, B, e} where m = mantissa, B = base, and e = exponent, that is m × B^e.

A NULL type relates to a single variable and is commonly used when a component variable associated with a constructed type has no type assignment. Similarly, the ANY type indicates that the type of the variable is defined elsewhere.

A constructed type is defined by reference to one or more other types, which may be primitive or constructed. The constructed types used with ASN.1 include the following:

- UNIVERSAL (constructed) SEQUENCE: a fixed (bounded), ordered list of types, some of which may be declared optional, that is, the associated typed value may be omitted by the entity constructing the sequence;

- SEQUENCEOF: a fixed or unbounded, ordered list of elements, all of the same type;
- SET: a fixed, unordered list of types, some of which may be declared optional;
- SETOF: a fixed or unbounded, unordered list of elements, all of the same type;
- CHOICE: a fixed, unordered list of types, selected from a previously specified set of types.

An example of a constructed type is shown in Figure 13.3(b), together with the equivalent type definition in Pascal for comparison purposes.

To allow the individual elements within a structured type to be referenced, ASN.1 supports the concept of **tagging**. This involves assigning a **tag** or **identifier** to each element and is analogous to the index used with the array type found in most high-level languages.

The tag may be declared to be one of the following:

- CONTEXT-SPECIFIC: the tag has meaning only within the scope of the present structured type;
- APPLICATION: the tag has meaning in the context of the complete application (collection of types);
- PRIVATE: the tag has meaning only to the user.

An example of the use of tags in relation to the type definition used in Figure 13.3(b) is given in part (c). In the example, we assume that *empNumber* is used in other type definitions and hence is given a unique application-wide tag. The other three variables need be referenced only within the context of this sequence type.

Another facility supported in ASN.1 is to declare a variable to be of an **implied type**, using the keyword IMPLICIT which is written immediately after the variable name and, if present, the tag number.

Normally, the type of a variable is explicitly defined, but if a variable has been declared to be of an IMPLICIT type, then the type of the variable can be implied by, say, its order in relation to other variables. It is used mainly with tagged types since the type of the variable can then be implied from the tag number. An example is shown in Figure 13.3(d), where the types of the last three variables can be implied – rather than explicitly defined – from their tag number. The benefit of this will become more apparent when we discuss the encoding and decoding rules associated with ASN.1 in the next section.

To illustrate the use of some of the other features and types associated with ASN.1, let us consider the use of ASN.1 for the definitions of the protocol data units (PDUs) associated with a protocol. The PDU type definitions discussed so far are all defined in the form of an ordered bit string with the number of bits required for each field and the order of bits in the string defined unambiguously. This ensures that the fields in each PDU are interpreted in the same way in all systems.

To minimize the length of each PDU, many of the fields have only a few bits associated with them. With this type of definition, it is not easy to use a high-level programming language to implement the protocol since isolating each field in a received octet string – and subsequent encoding – can involve-complex bit manipulations.

To overcome this problem, the PDU definitions of all of the protocols defined by ISO are now defined using ASN.1. By passing each PDU definition through an appropriate ASN.1 compiler, the type definitions of all the fields in each PDU are automatically produced in a suitable high-level language compatible form. The protocol can be written in the selected language using these type definitions. Again, however, since the fields are now in an abstract syntax, the corresponding encoding and decoding procedures produced by the ASN.1 compiler must be used to convert each field into/from its transfer syntax when transferring PDUs between two peer protocol entities.

An example of the use of ASN.1 for the definition of a PDU is shown in Figure 13.4. This relates to an ISO application protocol called **file transfer access and management** (**FTAM**) which is sometimes used instead of FTP.

The complete set of PDUs relating to a particular protocol entity is defined as a **module**. The name of a module is known as the **module definition**. In the example of Figure 13.4, this is *ISO8571-FTAM DEFINITIONS*. It is followed by the assignment symbol (::=); the module body is then defined between the BEGIN and END keywords.

Following BEGIN, the CHOICE type indicates that the PDUs used with FTAM belong to one of three types: *InitializePDU*, *FilePDU*, or *BulkdataPDU*. A further *CHOICE* type indicates that there are six different types of PDU associated with the *InitializePDU* type: *FINITIALIZErequest*, *FINITIALIZE response*, and so on. Note that these are tagged so that they can be distinguished from one another. Also, since the tags are followed by *IMPLICIT*, the type of PDU can be implied from the tag field, that is, no further definition is needed, such as a PDU type. Note that since the *FINITIALIZErequest* PDU is always the first PDU received in relation to FTAM, it is assigned an application-specific tag number of 1. The remaining PDU types then have a context-specific tag; note that the word CONTEXT is not needed as these types will have meaning in the context of FTAM. The definition of each PDU is then given and, in Figure 13.4, the *FINITIALIZErequest* PDU is defined.

The *SEQUENCE* structured type is used in this definition to indicate that the PDU consists of a number of typed data elements, which may be primitive or constructed. Although with the *SEQUENCE* type the list of variable types is in a set order, normally the individual elements are (context-specifically) tagged since, as we shall see in the next section, this can lead to a more efficient encoded version of the PDU. The first element, *protocolId*, is of type *INTEGER* and is set to zero, which indicates it is FTAM (*iso FTAM*).

```
ISO8571-FTAM DEFINITIONS ::=

BEGIN

PDU ::= CHOICE {
                InitializePDU,
                FilePDU,
                BulkdataPDU
                }
InitializePDU ::= CHOICE    {
                      [APPLICATION 1]      IMPLICIT FINITIALIZErequest,
                      [1]                  IMPLICIT FINITIALIZEresponse,
                      [2]                  IMPLICIT FTERMINATErequest,
                      [3]                  IMPLICIT FTERMINATEresponse,
                      [4]                  IMPLICIT FUABORTrequest,
                      [5]                  IMPLICIT FPABORTresponse
                          }

FINITIALIZErequest    ::= SEQUENCE {
                      protocolId [0] INTEGER { isoFTAM (0) },
                      versionNumber [1] IMPLICIT
                                      SEQUENCE { major INTEGER,
                                                 minor INTEGER},
                                           -- initially { major 0, minor 0}
                      serviceType [2] INTEGER { reliable (0),
                                              user correctable (1)}
                      serviceClass [3] INTEGER { transfer (0),
                                               access (1),
                                               management (2)}
                      functionalUnits [4] BITSTRING { read (0),
                                                  write (1),
                                                  fileAccess (2),
                                                  limitedFileManagement (3),
                                                  enhancedFileManagement (4),
                                                  grouping (5),
                                                  recovery (6),
                                                  restartDataTransfer (7) }
                      attributeGroups [5] BITSTRING {storage (0),
                                                  security (1) }
                      rollbackAvailability [6] BOOLEAN DEFAULT FALSE,
                      presentationContextName [7] IMPLICIT ISO646String {"ISO8822"},
                      identifyOfInitiator [8] ISO646String OPTIONAL,
                      currentAccount [9] ISO646String OPTIONAL,
                      filestorePassword [10] OCTETSTRING OPTIONAL,
                      checkpointWindow [11] INTEGER OPTIONAL }

FINITIALIZEresponse ::= SEQUENCE {
                          :
                          :
END
```

Figure 13.4 ASN.1 PDU definition example.

The second element, *versionNumber*, is defined as a *SEQUENCE* of two *INTEGER* types – *major* and *minor*. As before, the use of the word *IMPLICIT* means that the type (*SEQUENCE*) can be implied from the preceding tag field and need not be encoded. A comment field is used to indicate the initial setting of the two variables. The next two elements are both of type *INTEGER*; the possible values of each are shown in the braces.

The next element, *functionalUnits*, is of type *BITSTRING*; the eight bits in the string are set to 1 or 0 depending on whether the particular unit is (1) or is not (0) required. Finally, some of the later elements in the sequence are declared *OPTIONAL*, which means that they may or may not be present in an encoded PDU. Since the individual elements in the PDU have been tagged, the receiver of the PDU can determine if the element is present or not. The keyword *DEFAULT* has a similar meaning except that if the element is not present in a PDU, it is assigned the default value.

Finally, there is a primitive type that has been defined to enable an object definition to be unique within a wider context than its current definition. For example, as we shall expand upon in Section 14.7.1, within the context of network management, the various managed objects associated with a network – a bridge, a router, a protocol, and so on – are each assigned an OBJECT IDENTIFIER that is unique within the context of all the different network types – PSTN/ISDN/Internet/and so on.

Within any one of these networks there is a host of multinational vendors that supply equipment and software to be used within that network. Also, many vendors supply equipment that is used in a number of these networks In order to ensure that the management information produced by a particular piece of equipment or software relates to a specific network type, the various international standards bodies have defined an **object naming tree** so that the set of object identifiers associated with each of these network types are unique within a global context.

13.2.2 Transfer syntax

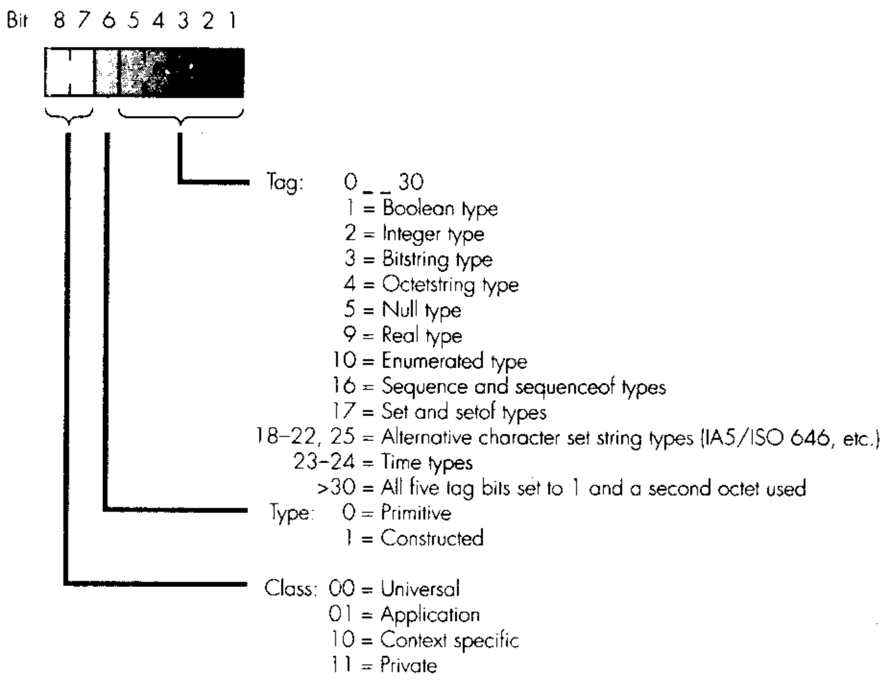
As we indicated earlier, all the ASN.1 data types associated with an application have an abstract syntax. This means that their values may be represented in different ways within the various computers/items of equipment involved in the application. Hence as we saw in Figure 13.1, before each value associated with the various data types used within an application is transferred from one AP to another, it is first encoded into a standard transfer – also called concrete – syntax. Similarly, on receipt of each encoded value, the destination AP first decodes each value into its local (abstract) syntax before it is processed. In this section we describe the principles behind both the encoding and decoding operations.

Encoding

The standard representation for a value of each type is a data element comprising the following three fields:

- *identifier*, which defines the ASN.1 type;
- *length*, which defines the number of octets in the contents field;
- *contents*, which defines the contents (which may be other data elements for a structured type).

Each field comprises one or more bytes/octets. The structure of the identifier octet is shown in Figure 13.5 and example encodings of different typed values are given in Figure 13.6. To help readability, the content of each octet is represented as two hexadecimal digits and the final encoded value (always in the form of a string of octets) is given at the end of each example. If the number



Note: The null type is used to indicate the absence of an element in a sequence.

The two time types are used to specify time in a standardized way as a string of IA5/ISO 646 characters. For example:

```
YY  MM  DD  hh  mm  ss
00  09  30  20  45  58    = current time
```

Figure 13.5 ASN.1 encoding: identifier bit definitions.

of octets in the contents field exceeds 127, the most significant bit of the first length octet is set to 1 and the length is defined in two (or more) octets.

In Figure 13.6(a), the identifier 01 (Hex) indicates that the class is UNIVERSAL (bits 8 and 7 = 00), it is a primitive type (bit 6 = 0), and the tag (bits 1 through 5) is 1, thus indicating it is Universal 1 and hence BOOLEAN. The length is 01 (Hex) indicating that the content is a single octet. TRUE is encoded as FF (Hex) and FALSE as 00 (Hex).

```
(a) BOOLEAN - UNIVERSAL 1

    e.g., Employed ::= BOOLEAN
    -- assume true

    Identifier = 01 (Hex)                          -- Universal 1
    Length     = 01
    Contents   = FF

    i.e., 01 01 FF

    INTEGER - UNIVERSAL 2

    e.g., RetxCount ::= INTEGER
    -- assume = 29 (decimal)

    Identifier = 02                                -- Universal 2
    Length     = 01
    Contents   = 1D                                -- 29 decimal

    i.e., 02 01 1D

    BITSTRING - UNIVERSAL 3

    e.g., FunctionalUnits ::= BITSTRING {read (0), write (1), fileAccess (2)}
    -- assume read only is required

    Identifier = 03
    Length     = 01
    Contents   = 80                                -- read only = 1000 0000

    i.e., 03 01 80

    UTCTime - UNIVERSAL 23

    e.g., UTCTime ::= [UNIVERSAL 23] IMPLICIT ISO646String
    -- assume 2.58 p.m. on 5th November 1999 = 99 11 05 14 58

    Identifier = 17 (Hex)                          -- Universal 23
    Length     = 0A
    Contents   = 38 39 31 31 30 35 31 34 35 38

    i.e., 17 0A 38 39 31 31 30 35 31 34 35 38
```

Figure 13.6 ASN.1 encoding examples: (a) primitive types; (b) constructed type; (c) use of implicit tag.

(b) SEQUENCE/SEQUENCEOF – UNIVERSAL 16

```
e.g., File ::= SEQUENCE {userName IA5String, contents OCTETSTRING}
-- assume userName = "FRED" and contents = 0F  27  E4  Hex

Identifier = 30 (Hex)                          -- Constructed, Universal 16
Length     = 0B                                -- Decimal 11
Contents   = Identifier = 16                   -- Universal 22
             Length     = 04
             Contents = 46  52  45  44
             Identifier = 04                   -- Universal 4
             Length     = 03
             Contents = 0F  27  E4

i.e., 30  0B  16  04  46  52  45  44  04  03  0F  27  E4
```

(c) Tagging/IMPLICIT

```
e.g., UserName ::= SET {surname [0] IMPLICIT ISO646String, password [1] ISO646String }
-- assume surname = "BULL" and password = "KING"

Identifier = 31                                -- Constructed, Universal 17
Length     = 0E                                -- Decimal 14
Contents   = Identifier = 80                   -- Context-specific 0 = surname
             Length     = 04
             Contents = 42  55  4C  4C
             Identifier = A1                   -- Context-specific 1 = password
             Length     = 06
             Contents = Identifier = 16        -- Universal 22
                        Length     = 04
                        Contents = 4B 49 4E 47

i.e., 31  0E  80  04  42  55  4C  4C  A1  06  16  04  4B  49  4E  47
```

Figure 13.6 Continued

Integer values are encoded in 2s-complement form with the most significant bit used as the sign bit. Thus, a single octet can be used to represent a value in the range –128 to +127. More octets must be used for larger values. Note, however, that only sufficient octets are used to represent the actual value, irrespective of the number of bits used in the original form, that is, even if the value 29 shown in Figure 13.6(a) is represented as a 16-bit or 32-bit integer locally, only a single octet is used to represent it in its encoded form. Similarly, if the type is BITSTRING, the individual bits are assigned starting at the most significant bit with any unused bits set to zero.

Two examples showing the encoding of constructed types are given in Figure 13.6(b) and (c). With a variable of type SEQUENCE (or SEQUENCEOF), the identifier is 30 (= 0011 0000 binary). This indicates that the class is UNIVERSAL (bits 8 and 7 = 00), it is a constructed type (bit 6 = 1), and the tag equals 16 (bit 5 = 1 and bits 4 through 1 = 0). Similarly, the identifier with a SET (or SETOF) type is 31, indicating it is UNIVERSAL, constructed with tag 17.

Note also that in Figure 13.6(c), the two fields in the type *UserName* have been tagged as context-specific – [0] and [1]. The two identifiers associated with these fields are 80 (= 1000 0000 binary) and A1 (= 1010 0001 binary), respectively. The first indicates that the class is context-specific (bits 8 and 7 = 10), it is a simple type (bit 6 = 0), and the tag is 0. However, the second is context-specific, constructed, and the tag is 1. This is because the first context-specific tag has been declared IMPLICIT, in which case the type field can be implied from the tag. However, with the second, the type field must also be defined so two additional octets are required. Note that in all cases the resulting octet string is transmitted in the order left to right.

In order to illustrate a more complete definition, an example PDU encoding is given in Figure 13.7. The PDU selected is *FINITIALIZErequest*, which we defined earlier in its ASN.1 form in Figure 13.4. The actual values associated with the PDU are defined in Figure 13.7(a) while Figure 13.7(b) shows how the selected values are encoded. Typically, the various fields in the PDU are abstract data types associated with a data structure in a program. However, after encoding, the PDU consists of a precisely defined string of octets which, for readability, are shown in hexadecimal form. The complete octet string is then transferred to the correspondent (peer) FTAM protocol entity where it is decoded back into its (local) abstract form.

Decoding

On receipt of the encoded string, the correspondent AP performs an associated decoding operation. For example, assuming that the received octet string relates to the PDU shown in Figure 13.7, the leading octet in the string is first used to determine the type of PDU received – Application-specific

```
(a)  FINITIALIZErequest = {   protocolId = 0,
                              versionNumber {major = 0, minor = 0}
                              serviceType = 1,
                              serviceClass = 1,
                              functionalUnits {read = 0, write = 1, fileAccess = 2,
                                              limitedFileManagement = 3
                                              enhancedFileManagement = 4,
                                              grouping = 5, recovery = 6,
                                              restartDataTransfer = 7}
                              attributeGroups {storage = 0, security = 1}
                              rollbackAvailability = T,
                              PresentationContextName = "ISO8822"}
```

Figure 13.7 Example PDU encoding: (a) PDU fields and their contents; (b) encoded form.

```
(b) Identifier = 61                                          -- Application-specific 1 = FINITIALIZErequest
    Length     = 31                                          -- decimal 49
    Contents   = Identifier = A0                             -- Context-specific 0 = protocolId
                 Length     = 03
                 Contents   = Identifier = 02                -- Universal 2 - INTEGER
                              Length     = 01
                              Contents   = 00                -- isoFTAM
                 Identifier = A1                             -- Context-specific 1 = versionNumber
                 Length     = 06
                 Contents   = Identifier = 02                -- Universal 2
                              Length     = 01
                              Contents   = 00                -- major
                              Identifier = 02                -- Universal 2
                              Length     = 01
                              Contents   = 00                -- minor
                              Identifier = A2
                              Length     = 03
                              Contents   = Identifier =02
                                           Length     = 01
                                           Contents   = 01              -- serviceType = user correctable
                              Identifier = A3
                              Length     = 03
                              Contents   = Identifier = 02
                                           Length     = 01
                                           Contents   = 01              -- serviceClass = access
                              Identifier = A4                -- Context-specific 4 = functionalUnits
                              Length     = 03
                              Contents   = Identifier = 03   -- Universal 3 = BITSTRING
                                           Length     = 01
                                           Contents   = E0   -- read, write, fileAccess = 1110 000
                              Identifier = A5                -- Context-specific 5 = attributeGroups
                              Length     = 03
                              Contents   = Identifier = 03
                                           Length     = 01
                                           Contents   = 40   -- security 0100 000
                              Identifier = A6                -- Context-specific 6 = rollbackAvailability
                              Length     = 03
                              Contents   = Identifier = 01   -- Universal 1 = BOOLEAN
                                           Length     = 01
                                           Contents   = FF   -- true
                              Identifier = A7                -- Context-specific 7 = PresentationContextName
                              Length     = 07
                              Contents   = 49 53 4F 38 38 32 32     -- "ISO8822"
```

Concrete syntax of the above PDU is thus.

```
61 2F A0 03 02 01 00 A1 06 02 01 00 02 01 00 A2
03 02 01 01 A3 03 02 01 01 A4 03 03 01 E0 A5 03
03 01 40 A6 03 01 01 FF A7 07 49 53 32 38 38 32
32
```

Figure 13.7 Continued

1 = *FINITIALIZErequest.* Clearly, since each PDU has a unique structure, we must have a separate decoding procedure for each PDU type. Hence, on determining the type of PDU received, the corresponding decoding procedure is invoked. Once this has been done, the various fields (data elements) making up the PDU will be in their local (abstract) syntax form and processing of the PDU can start. Thus in the example, the various context-specific tags are used to determine the field within the PDU and the appropriate decoded value – now in its local syntax – is then assigned to this.

13.3 Security

As we indicated in the introduction, increasingly people are using networks such as the Internet for on-line banking, shopping, and many other applications. The generic term used is **electronic commerce** or **e-commerce** and this often involves the transfer of sensitive information such as credit card details over the network. Hence to support this type of networked transaction, a number of security techniques have been developed which, when combined together, provide a high level of confidence that any information relating to the transaction that is received from the network:

- has not been altered in any way – **integrity**;
- has not been intercepted and read by anyone – **privacy/secrecy**;
- has come from an authorized sender – **authentication**;
- has proof that the stated sender initiated the transaction – **nonrepudiation.**

Below we shall describe a number of the techniques that are used to carry out these four functions. As we shall see, secrecy and integrity are achieved by means of **data encryption** while authentication and nonrepudiation require the exchange of a set of (encrypted) messages between the two communicating parties. We shall give some examples of applications that use these techniques later in Chapters 14 and 15.

13.4 Data encryption

As the knowledge of computer networking and protocols has become more widespread, so the threat of intercepting and decoding message data during its transfer across a network has increased. For example, the end systems (stations/hosts) associated with most applications are now attached to a LAN. The application may involve a single LAN or, in an internetworking environment, the Internet. However, with most LANs, transmissions on the shared transmission medium can readily be intercepted by any system if an intruder sets the appropriate MAC chipset into the promiscuous mode and records all transmissions on the medium. Then, with a knowledge of the LAN protocols being used, the intruder can identify and remove the protocol control information at the head of each message, leaving the message contents. The message contents, including passwords and other sensitive information, can then be interpreted.

This is known as **listening** or **eavesdropping** and its effects are all too obvious. In addition and perhaps more sinister, an intruder can use a recorded message sequence to generate a new sequence. This is known as **masquerading** and again the effects are all too apparent. Therefore, encryption should be applied to all data transfers that involve a network. In the context of the TCP/IP reference model, the most appropriate layer to perform such operations is the application layer. This section provides an introduction to the subject of data encryption.

13.4.1 Terminology

Data encryption (or **data encipherment**) involves the sending party – for example, the application protocol entity – in processing all data prior to transmission so that if it is accidentally or deliberately intercepted while it is being transferred it will be incomprehensible to the intercepting party. Of course, the data must be readily interpreted – **deycrypted** or **deciphered** – by the intended recipient. Consequently, most encryption methods involve the use of an **encryption key** which is hopefully known only by the two correspondents. The key features in both the encryption and the decryption processing. Prior to encryption, message data is normally referred to as **plaintext** and after encryption as **ciphertext**. The general scheme is illustrated in Figure 13.8.

When deciding on a particular encryption algorithm we must always assume that a transmitted message can be intercepted and recorded, and that the intruder knows the context in which the messages are being used, that is,

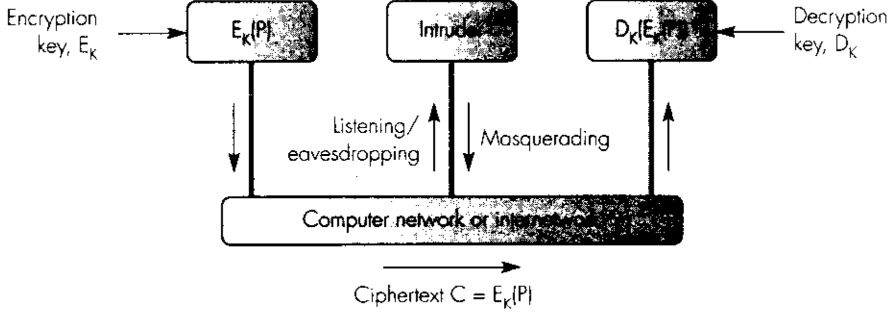


Figure 13.8 Data encryption terminology.

the type of information being exchanged. The aim is to choose an encryption method such that an intruder, even with access to a powerful computer, cannot decipher the recorded ciphertext in a realistic time period. There are two widely used algorithms but before we discuss them, let us consider some of the more fundamental techniques on which they are based.

13.4.2 Basic techniques

The simplest encryption technique involves **substituting** the plaintext alphabet (codeword) with a new alphabet known as the **ciphertext alphabet**. For example, a ciphertext alphabet can be defined which is the plaintext alphabet simply shifted by *n* places where *n* is the key. Hence, if the key is 3, the resulting alphabet is as follows:

Plaintext alphabet:	a b c d e f g . . .
Ciphertext alphabet:	d e f g h i j . . .

The ciphertext is obtained by substituting each character in the plaintext message by the equivalent letter in the ciphertext alphabet.

A more powerful variation is to define a ciphertext alphabet that is a random mix of the plaintext alphabet. For example:

Plaintext alphabet	a b c d e f g . . .
Ciphertext alphabet:	n z q a i y m . . .

The key is determined by the number of letters in the alphabet, for example, 26 if just lower-case alphabetic characters are to be transmitted or 128 if, say, the ASCII alphabet is being used. There are therefore $26! = 4 \times 10^{26}$ possible keys with the first alphabet or many times this with the larger alphabet. Notice that in general, the larger the key the more time it takes to break the code.

Although this may seem to be a powerful technique, there are a number of shortcuts that can be used to break such codes. The intruder is likely to know the context in which the message data is being used and hence the type of data involved. For example, if the messages involve textual information, then the statistical properties of text can be exploited: the frequency of occurrence of individual letters (e, t, o, a, and so on), are all well documented. By performing statistical analyses on the letters in the ciphertext such codes can be broken relatively quickly.

Substitution involves replacing each character with a different character, so the order of the characters in the plaintext is preserved in the ciphertext. An alternative approach is to reorder (**transpose**) the characters in the plaintext. For example, if a key of 4 is used, the complete message can first be divided into a set of 4-character groups. The message is then transmitted starting with all the first characters in each group, then the second, and so on. As an example, assuming a plaintext message of "this is a lovely day", the ciphertext is derived as follows:

1	2	3	4	←	key
t	h	i	s		
–	i	s	–		
a	–	l	o		
v	e	l	y		
–	d	a	y		

Ciphertext = t–av–hi–edisllas–oyy

Clearly, more sophisticated transpositions can be performed but, in general, when used alone transposition ciphers suffer from the same shortcomings as substitution ciphers. Most practical encryption algorithms tend to use a combination of the two techniques and are known as **product ciphers**.

Product ciphers

These use a combination of substitutions and transpositions. Also, instead of substituting/transposing the characters in a message, the order of individual bits in each character (codeword) is substituted/transposed. The three alternative transposition (also known as **permutation**) operations are shown in Figure 13.9(a). Each is normally referred to as a **P-box**.

The first involves transposing each 8-bit input into an 8-bit output by cross-coupling each input line to a different output line as defined by the key. This is known as a **straight permutation**. The second has a larger number of output bits than input bits; they are derived by reordering the input bits and passing selected input bits to more than one output. This is known as an **expanded permutation**.

The third has fewer output bits than inputs; it is formed by transposing only selected input bits. This is known as a **compressed** or **choice permutation**.

To perform a straight substitution of 8 bits requires a new set (and hence key) of 2^8 (=256) 8-bit bytes to be defined. This means the key for a single substitution is 2048 bits. To reduce this, a substitution is formed by encapsulating a P-box between a decoder and a corresponding encoder, as shown in Figure 13.9(b). The resulting unit is known as an **S-box**. The example performs a 2-bit substitution operation using the key associated with the P-box. An 8-bit substitution will require four such units.

Product ciphers are formed from multiple combinations of these two basic units, as shown in Figure 13.10. In general, the larger the number of stages the more powerful the cipher. A practical example of product ciphers is the **data encryption standard (DES)** defined by the US National Bureau of Standards. This is now widely used. Consequently, various integrated circuits are available to perform the encryption operation in hardware thereby speeding up the encryption and decryption operations.

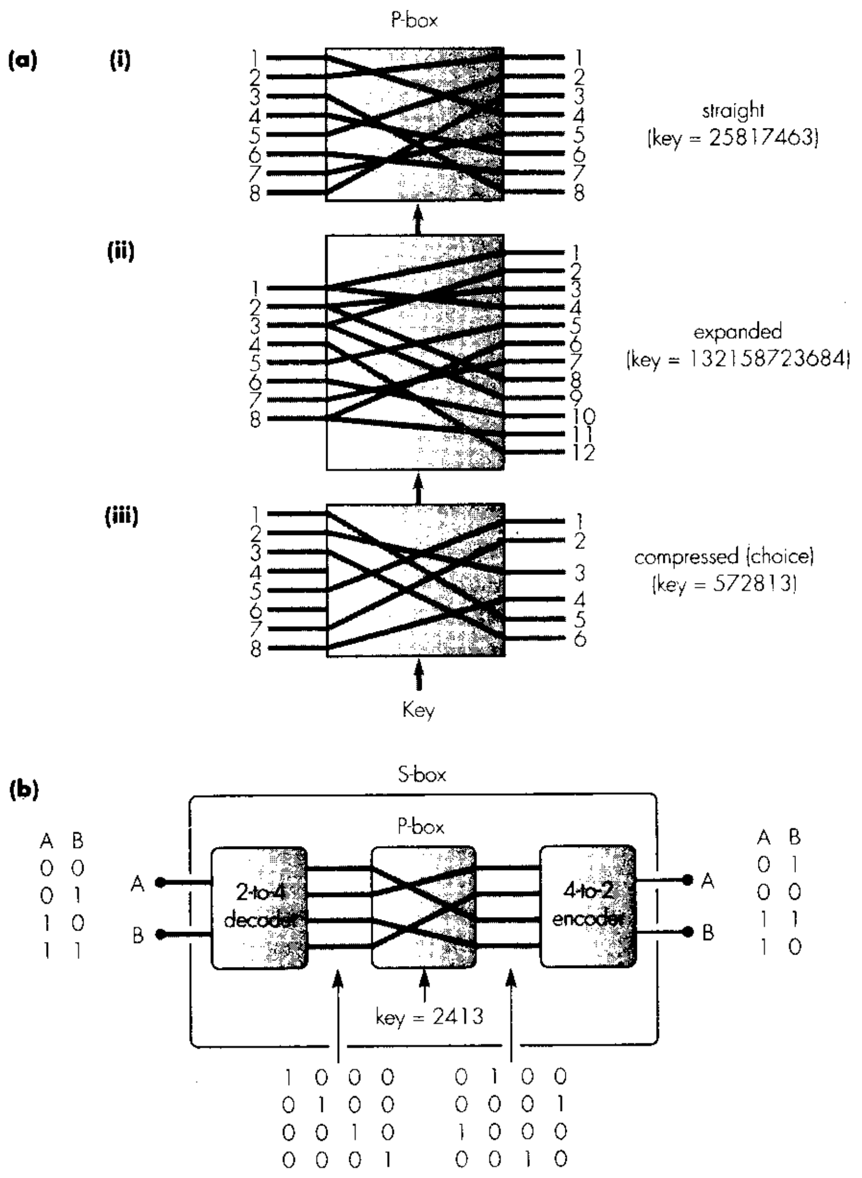


Figure 13.9 Product cipher components: (a) P-box examples; (b) S-box example.

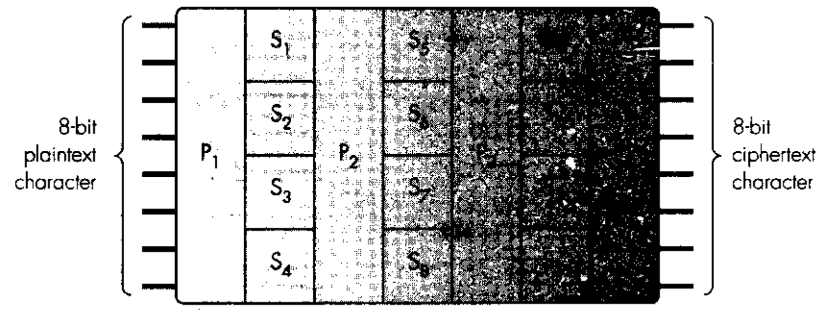


Figure 13.10 Example of a product cipher.

13.4.3 The data encryption standard

The DES algorithm is a **block cipher**, which means that it works on fixed-sized blocks of data. Thus, a complete message is first split (segmented) into blocks of plaintext, each comprising 64 bits. A (hopefully) unique 56-bit key is used to encrypt each block of plaintext into a 64-bit block of ciphertext, which is subsequently transmitted through the network. The receiver uses the same key to perform the inverse (decrpytion) operation on each 64-bit data block it receives, thereby reassembling the blocks into complete messages.

The larger the number of bits used for the key, the more likely it is that the key will be unique. Also, the larger the key, the more difficult it is for someone to decipher it. The use of a 56-bit key in the DES means that there are in the order of 1017 possible keys from which to choose. Consequently, DES is regarded as providing sufficient security for most commercial applications.

A diagram of the DES algorithm is shown in Figure 13.11(a). The 56-bit key selected by the two correspondents is first used to derive 16 different subkeys, each of 48 bits, which are used in the subsequent substitution operations. The algorithm comprises 19 distinct steps. The first step is a simple transposition of the 64-bit block of plaintext using a fixed transposition rule. The resulting 64 bits of transposed text then go through 16 identical iterations of substitution processing, except that at each iteration a different subkey is used in the substitution operation. The most significant 32 bits of the 64-bit output of the last iteration are then exchanged with the least significant 32 bits. Finally, the inverse of the transposition that was performed in step 1 is carried out to produce the 64-bit block of ciphertext to be transmitted. The DES algorithm is designed so that the received block is deciphered by the receiver using the same steps as for encryption, but in the reverse order.

The 16 subkeys used at each substitution step are produced as follows. Firstly, a fixed transposition is performed on the 56-bit key. The resulting transposed key is then split into two separate 28-bit halves. Next, these two

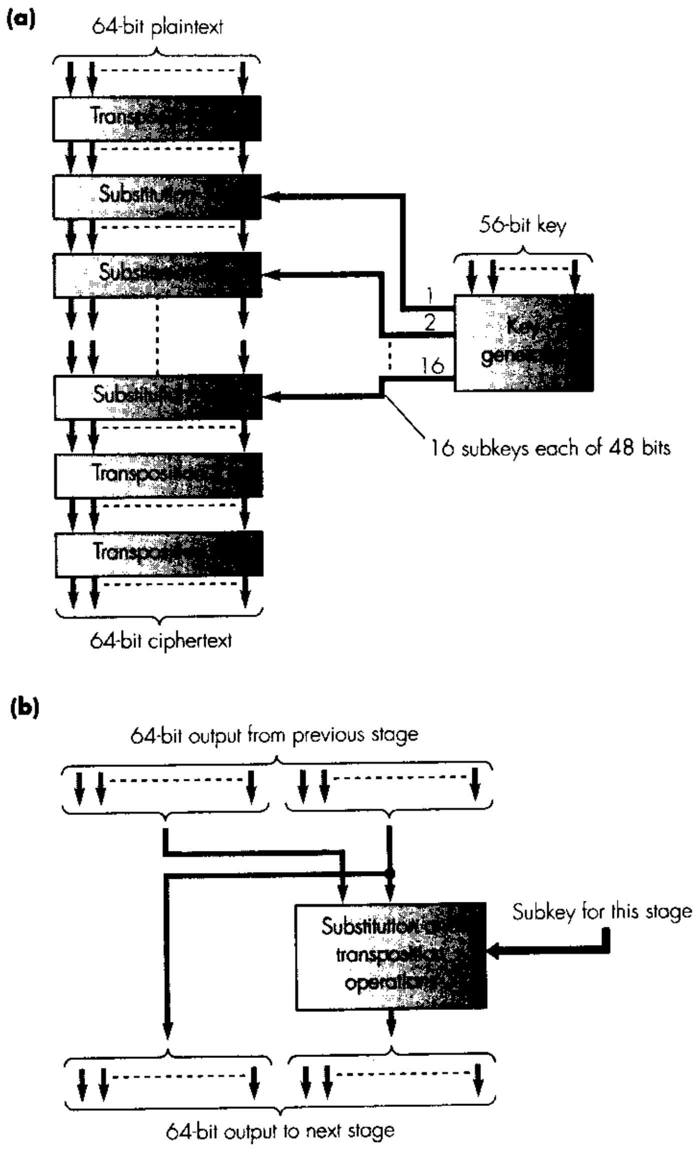


Figure 13.11 DES algorithm principles: (a) overall schematic; (b) substitution schematic; (c) substitution operation.

halves are rotated left independently and the combined 56 bits are then transposed once again using a compression operation to produce a subkey of 48 bits. The other subkeys are produced in a similar way except that the number of rotations performed is determined by the number of the subkey.

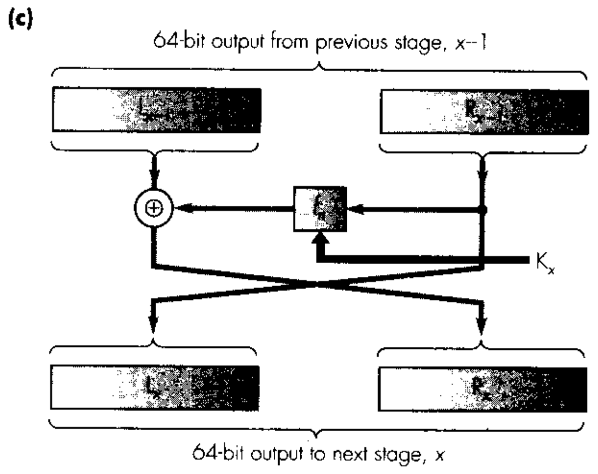

(c)
64-bit output from previous stage, x–1
K_x
64-bit output to next stage, x

Figure 13.11 Continued.

The processing performed at each of the 16 intermediate substitution steps in the encryption process is relatively complex as it is this that ensures the effectiveness of the DES algorithm. This processing is outlined in Figure 13.11(b). The 64-bit output from the previous iteration is first split into two 32-bit halves. The left 32-bit output is simply the right 32-bit input. However, the right 32-bit output is a function of both the left and right inputs and the subkey for this stage. The principle is shown in Figure 13.11(c).

As we can deduce from the figure, in the forward (encryption) direction:

$$L_x = R_{x-1}$$

and

$$R_x = L_{x-1} \oplus f_n\,(R_{x-1}, K_x)$$

where f_n is a bitwise function called the **Feistel cipher**. First, since the subkey for the stage, K_x, is 48 bits, R_{x-1} is expanded into a 48-bit value using a P-box – similar to that shown in part (ii) of Figure 13.9(a) – with a fixed key. This is then exclusive-ORed with K_x and the 48-bit output is then converted back again into a 32-bit value. This is done by first dividing the 48-bit value into eight 6-bit groups and then passing each group through an S-box – similar to that shown in Figure 13.9(b) – each with a different key. In this case, however, the internal P-box performs a compression operation by transposing the 64-bit output from the 6-to-64 decoder into 16 bits. The resulting 16 bits are then passed to a 16-to-4 bit encoder to produce a 4-bit value. The 4-bit output from each of the eight S-boxes is then combined to form a 32-bit value which is passed through a second (straight) P-box to produce the output of the function block (f_n).

As we indicated earlier, the DES algorithm is designed so that the received block is deciphered by the receiver using the same steps as for encryption, but in the reverse order. As we can deduce from Figure 13.11(a), since transposition 1 is the inverse of transposition 3 and transposition 2 is a simple swap operation, by passing the received input through the stack in the reverse direction the output at the top will be the original plaintext. This is only true of course, if passing the 64-bit block through each substitution operation in the reverse direction also produces the inverse of that produced in the forward direction.

To illustrate that the Feisel cipher has this property, as we can deduce from Figure 13.11(c), in the reverse (decryption) direction:

$$R_{x-1} = L_x$$

and

$$L_{x-1} = R_x \oplus f_n(L_x, K_x)$$

Hence, since $L_x = R_{x-1}$, the output of the Feisel cipher – and hence each substitution operation – is invertible.

For example, if we work with two 4-bit groups and assume f_n is a simple exclusive OR operation with a key, K_x, of 1011, then if:

$$L_{x-1} = 1001 \quad \text{and} \quad R_{x-1} = 0110$$

in the forward (encryption) direction:

$$L_x = 0110 \quad \text{and} \quad R_x = 0100$$

And in the reverse (decryption) direction:

$$R_{x-1} = 0110 \quad \text{and} \quad L_{x-1} = 1001$$

which, as we can see, are the same as the two original inputs.

Triple DES

Although DES is still widely used, the use of a 56-bit key means that, with increasingly powerful computers, the time taken to exhaustively try each of the possible keys is reducing steadily. To counter this, a variant called **triple DES** has been developed. The principle of the scheme is shown in Figure 13.12.

As we can see, the scheme involves the use of two keys and three executions of the DES algorithm. Key K_1 is used with the first (DES) block, K_2 with the second block, and K_1 again with the third block. The use of two keys gives an effective key length of 112 bits and, because of this, the scheme is now widely used in many financial applications.

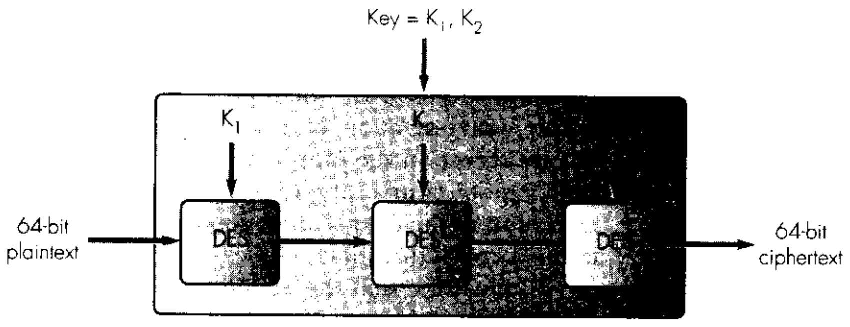


Figure 13.12 Triple DES schematic.

Chaining

The basic mode of working of DES is known as **electronic code book (ECB)** since each block of ciphertext is independent of any other block. Thus each 64-bit block of ciphertext has a unique matching block of plaintext, which is analogous to entries in a code book. The ECB mode of working is shown in Figure 13.13(a).

As we can deduce from Figure 13.11, the ECB mode of operation of DES has good secrecy properties and gives good protection against errors or changes that may occur in a single block of enciphered text. It does not, however, protect against errors arising in a stream of blocks. Since each block is treated separately in the ECB mode, the insertion of a correctly enciphered block into a transmitted stream of blocks is not detected by the receiver; it simply deciphers the inserted block and treats it as a valid block. Consequently, the stream of enciphered blocks may be intercepted and altered by someone who knows the key without the recipient being aware that any modifications have occurred. Also, if the order of the blocks is changed in some way then this will not be detected. The ECB mode, therefore, has poor integrity properties. Hence to obtain integrity as well as secrecy, an alternative mode of operation of DES based on a technique called **chaining** is often used. It is called the **chain block cipher (CBC)** mode and is shown in Figure 13.13(b).

As we can see, although the chaining mode uses the same block encryption method as previously described, each 64-bit block of plaintext is first exclusive-ORed with the enciphered output of the previous block before it is enciphered. The first 64-bit block of plaintext is exclusive-ORed with a 64-bit random number called the **initial vector**, which is sent prior to the cipher text. Then, after the first block has been encoded/decoded using this, subsequent blocks are encoded/decoded in the chained sequence shown in the figure. Thus, since the output of each block is a function both of the block contents and the output of the previous block, any alternations to the trans-

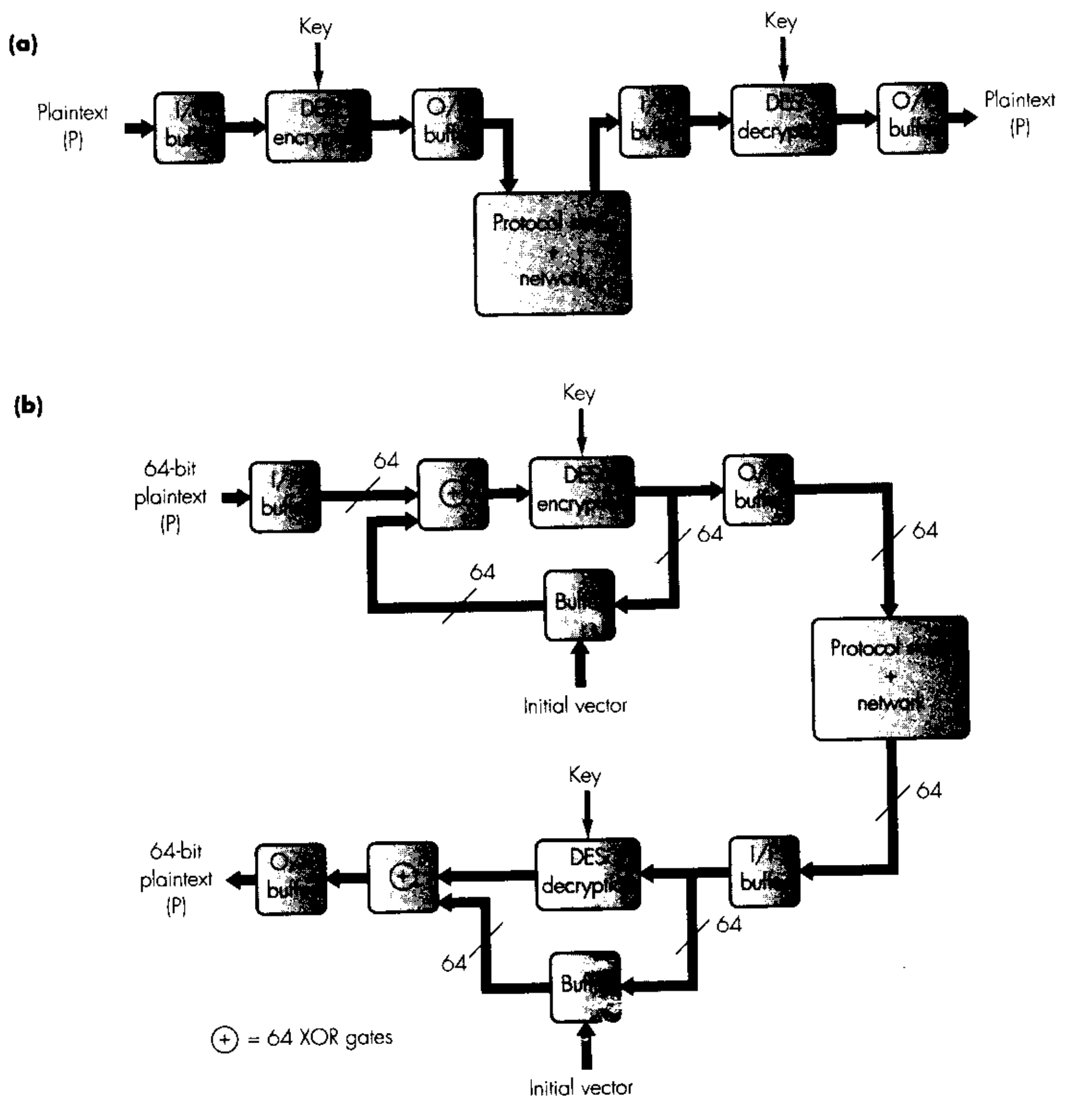


Figure 13.13 DES operational modes: (a) electronic code book (ECB); (b) chain block cipher (CBC); (c) cipher feedback mode (CFM).

(c)

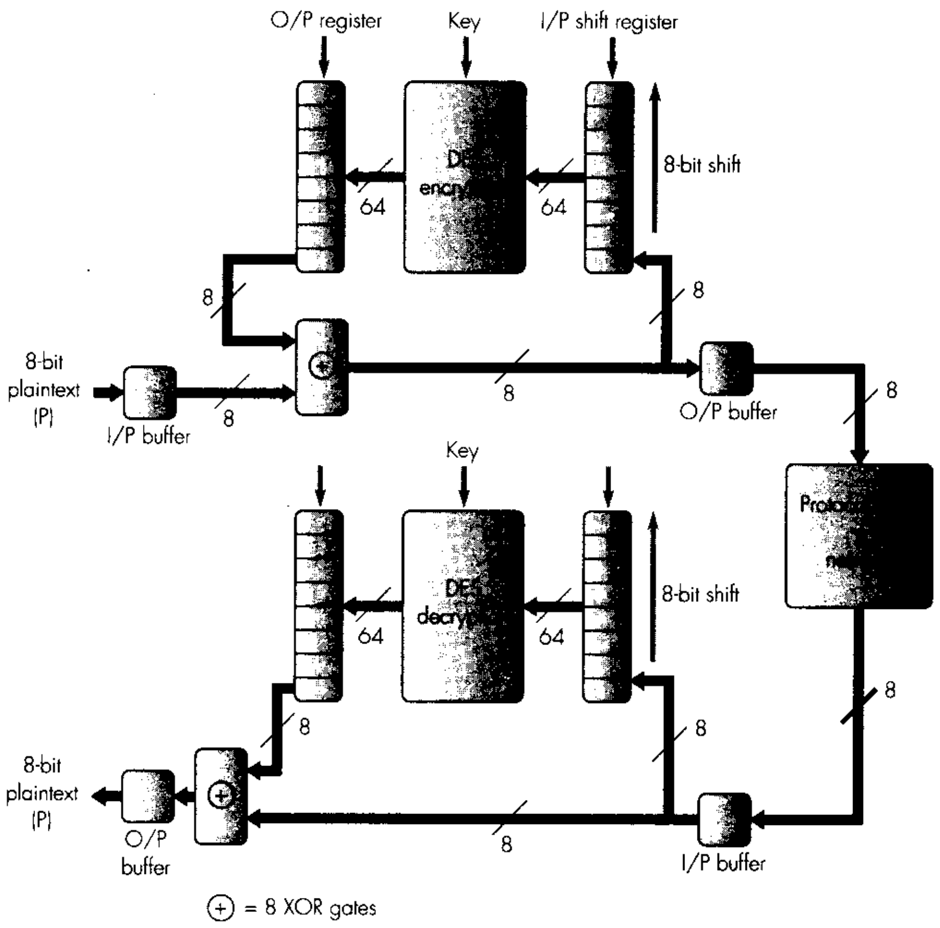


Figure 13.13 Continued.

mitted sequence can be detected by the receiver so giving a high level of integrity. Also, identical blocks of plaintext yield different blocks of ciphertext which makes the breaking of the code much more difficult. For these reasons, this is the mode of operation normally used for digital communication applications.

Since the basic CBC mode operates with 64-bit blocks, all messages must be multiples of 64 bits. Otherwise padding bits must be added. However, as we have seen in earlier chapters, the contents of all messages consist of strings of octets, so the basic unit of all messages is 8 bits rather than 64. An alternative mode of DES known as the **cipher feedback mode (CFM)** has also been defined which operates on 8-bit boundaries. A schematic of the scheme is shown in Figure 13.13(c).

With this mode, a new DES encryption operation is performed after every 8 bits of input rather than 64 with the CBC mode. A new 8-bit output is also produced which is the least significant 8 bits of the DES output, exclusive-ORed with the 8 input bits. Then, after each 8-bit output has been loaded into the output buffer, the 64-bit contents of the input shift register are shifted by 8 places. The 8 most significant bits are thus lost and the new 8-bit input is loaded into the least significant 8 bits of the input shift register. The DES operation is performed on these new 64 bits and the resulting 64-bit output is loaded into the output register. The least significant 8 bits of the latter are then exclusive-ORed with the 8 input bits and the process repeats.

CFM is particularly useful when the encryption operation is being performed at the interface with the serial transmission line. This mode of operation is used with the DES integrated circuits; each new 8-bit output is loaded directly into the serial interface circuit.

13.4.4 IDEA

The **international data encryption algorithm** (**IDEA**) is another block cipher method that is similar in principle to DES since it also operates on 64-bit blocks of plaintext. It can also be used, therefore, in the various chaining modes we described in the last section. To obtain added resilience, however, it uses a 128-bit key and more sophisticated processing during each phase of the encryption operation. Also, it has been designed so that it can be implemented equally well in both hardware and software and, in particular, with 16-bit microcomputers. A schematic diagram of the encryption operation is shown in Figure 13.14(a).

As we can see, each 64-bit block of plaintext passes through a series of eight bit-manipulation iterations followed by a final transposition. At each of the eight iterations, each of the 64 output bits is a function of all 64 input bits. The various processing operations that are carried out to achieve this are shown in Figure 13.14(b).

The 128-bit key is first used to generate 52 subkeys each of 16 bits. As we can see in the figure, six subkeys are used at each iteration and the remaining four subkeys are used in the final transposition stage. Decryption uses the same algorithm but with a modified set of keys.

Each 64-bit input is first divided into four 16-bit words each of which goes through a series of multiplication, addition, and exclusive-OR operations. All the lines shown in the figure are 16 bits and all the multiplication operations involve first the 32-bit product of the two 16-bit inputs being computed and then dividing this by $2^{16} + 1$. The output is then the 16-bit remainder. In the case of the addition operations, the two 16-bit inputs are added together and any carry that is generated is ignored.

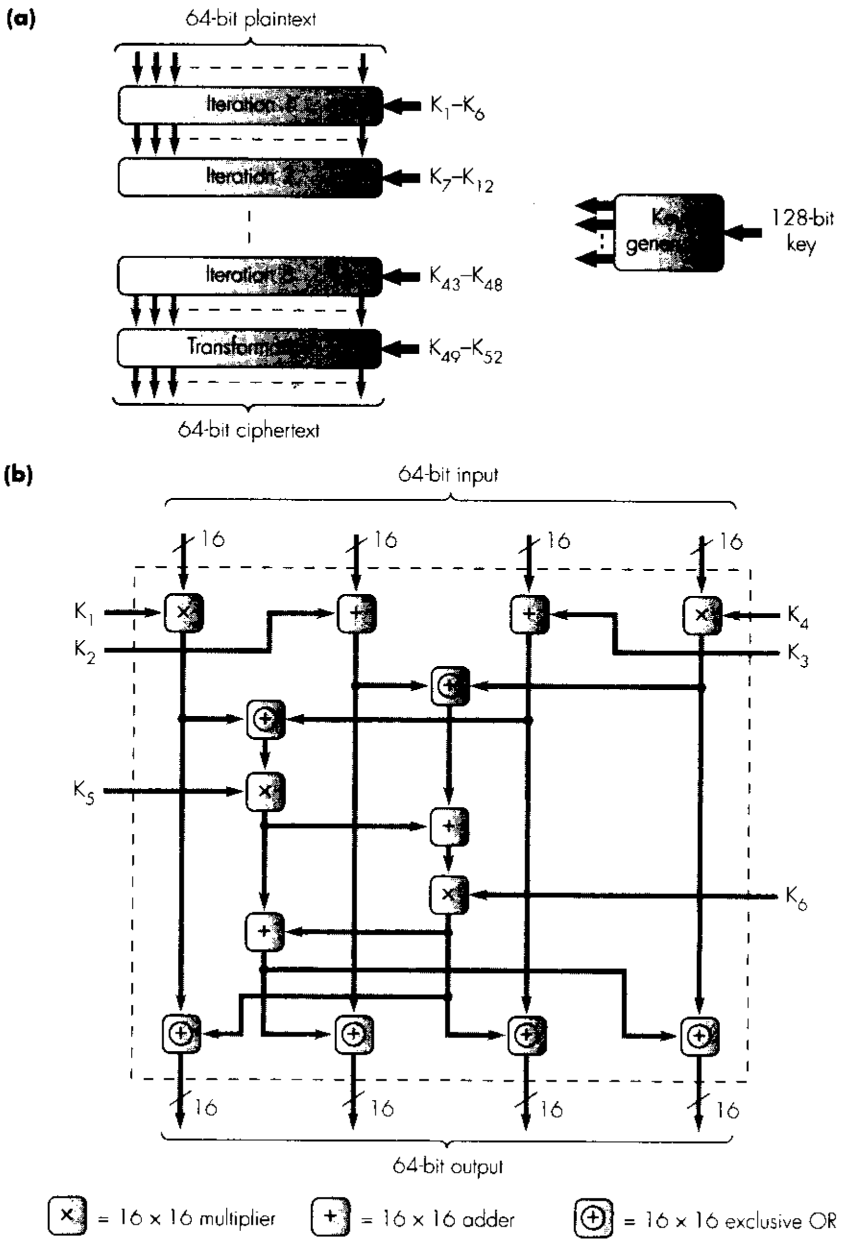


Figure 13.14 IDEA: (a) encryption schematic; (b) single iteration detail.

13.4.5 The RSA algorithm

Both DES and IDEA rely, of course, on the same key being used for both encryption and decryption. An obvious disadvantage is that some form of key notification must be used before any encrypted data is transferred between two correspondents. This is perfectly acceptable as long as the key does not change very often, but in fact it is common practice to change the key on a daily, if not more frequent, basis. Clearly, the new key cannot reliably be sent via the network, so an alternative means, such as a courier, must be used. The distribution of keys is a major problem with private key encryption systems. An alternative method, based on a public rather than a private key, is sometimes used to overcome this problem. The best known public key method is the **RSA algorithm**, named after its three inventors: Rivest, Shamir, and Adelman.

The fundamental difference between a private key system and a public key system is that the latter uses a different key to decrypt the ciphertext from the key that was used to encrypt it. A public key system uses a pair of keys: one for the sender and the other for the recipient.

Although this may not seem to help, the inventors of the RSA algorithm used number theory to develop a method of generating a pair of numbers – the keys – in such a way that a message encrypted using the first number of the pair can be decrypted only by the second number. Furthermore, the second number cannot be derived from the first. This second property means that the first number of the pair can be made available to anyone who wishes to send an encrypted message to the holder of the second number since only that person can decrypt the resulting ciphertext message. The first number of the pair is known as the **public key** and the second the private or **secret key**. The principle of the method is shown in Figure 13.15.

As indicated, the derivation of the two keys is based on number theory and is therefore outside the scope of this book. However, the basic algorithm used to compute the two keys is simple and is summarized here together with a much simplified example.

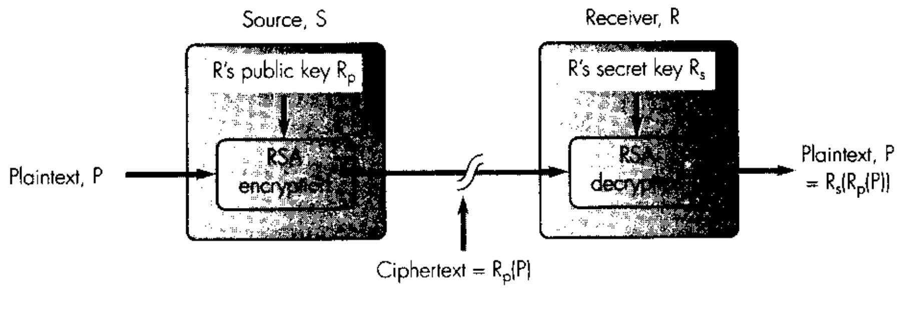


Figure 13.15 RSA schematic.

To create the public key K_p:	*Example:*
■ select two large positive prime numbers P and Q	$P = 7$, $Q = 17$
■ compute $X = (P - 1) \times (Q - 1)$	$X = 96$
■ choose an integer E which is prime relative to X, i.e., not a prime factor of X or a multiple of it, and which satisfies the condition indicated below for the computation of K_s	$E = 5$
■ compute $N = P \times Q$	$N = 119$
■ K_p is then N concatenated with E	$K_p = 119, 5$
To create the secret key K_s:	
■ compute D such that MOD $(D \times E, X) = 1$	$D \times 5/96 = 1$, $D = 77$
■ K_s is then N concatenated with D	$K_s = 119, 77$
To compute the ciphertext C of plaintext P:	
■ treat P as a numerical value	$P = 19$
■ $C =$ MOD (P^E, N)	$C =$ MOD $(19^5, 119)$ $C = 66$
To compute the plaintext P of ciphertext C:	
■ $P =$ MOD (C^D, N)	$P =$ MOD $(66^{77}, 119)$ $P = 19$

The choice of E and D in this example is best seen by considering the factors of 96. These are 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48. The list of numbers which are prime relative to 96 are thus 5, 7, 9, 10, 11, and so on. If we try the first of these, $E = 5$, then there is also a number $D = 77$ which satisfies the condition MOD $(D \times E, X) = 1$ and hence these are chosen.

We can deduce from this example that the crucial numbers associated with the algorithm are the two prime numbers P and Q, which must always be kept secret. The aim is to choose a sufficiently large N so that it is impossible to factorize it in a realistic time. Some example (computer) factorizing times are:

$N = 100$ digits $\approx$ 1 week

$N = 150$ digits $\approx$ 1000 years

$N > 200$ digits $\approx$ 1 million years

The RSA algorithm requires considerable computation time to compute the exponentiation for both the encryption and decryption operations. However, there is a simple way of avoiding the exponentiation operation by performing instead the following algorithm which uses only repeated multiplication and division operations:

```
C : = 1
begin for i = 1 to E do
    C : = MOD (C × P, N)
end
```

Decryption is performed in the same way by replacing E with D and P with C in the above expression; this yields the plaintext P. For example, to compute C = MOD (19^5, 119):

Step 1: C = MOD (1 × 19, 119) = 19
2: C = MOD (19 × 19, 119) = 4
3: C = MOD (4 × 19, 119) = 76
4: C = MOD (76 × 19, 119) = 16
5: C = MOD (16 × 19, 119) = 66

Note also that the value of *N* determines the maximum message that can be encoded. In the example this is 119 and is numerically equivalent to a single ASCII-encoded character. Therefore, a message comprising a string of ASCII characters would have to be encoded one character at a time.

Although a public key system offers an alternative to a private key system to overcome the threat of eavesdropping, if the public key is readily available it can be used by a masquerader to send a forged message. The question then arises as to how the recipient of a correctly ciphered message can be sure that it was sent by a legitimate source. As we indicated earlier, this relates to authentication and nonrepudiation and there are a number of solutions to this problem.

13.5 Nonrepudiation

Public key systems like RSA are particularly useful for nonrepudiation; that is, proving that a person sent an electronic document. With a paper document, normally a person adds his or her signature at the end of the document – sometimes with the name and signature of a witness – and, should it be necessary, this is then used to verify that the person whose signature is on the document sent it.

One solution is to exploit the dual property of public key systems, namely that not only is a receiver able to decipher all messages it receives (which have been encrypted with its own public key) using its own private key, but any receiver can also decipher a message encrypted with the sender's private key, using the sender's public key.

Figure 13.16(a) shows how this property may be exploited to achieve nonrepudiation. Encryption and decryption operations are performed at two levels. The inner level of encryption and decryption is as already described. However, at

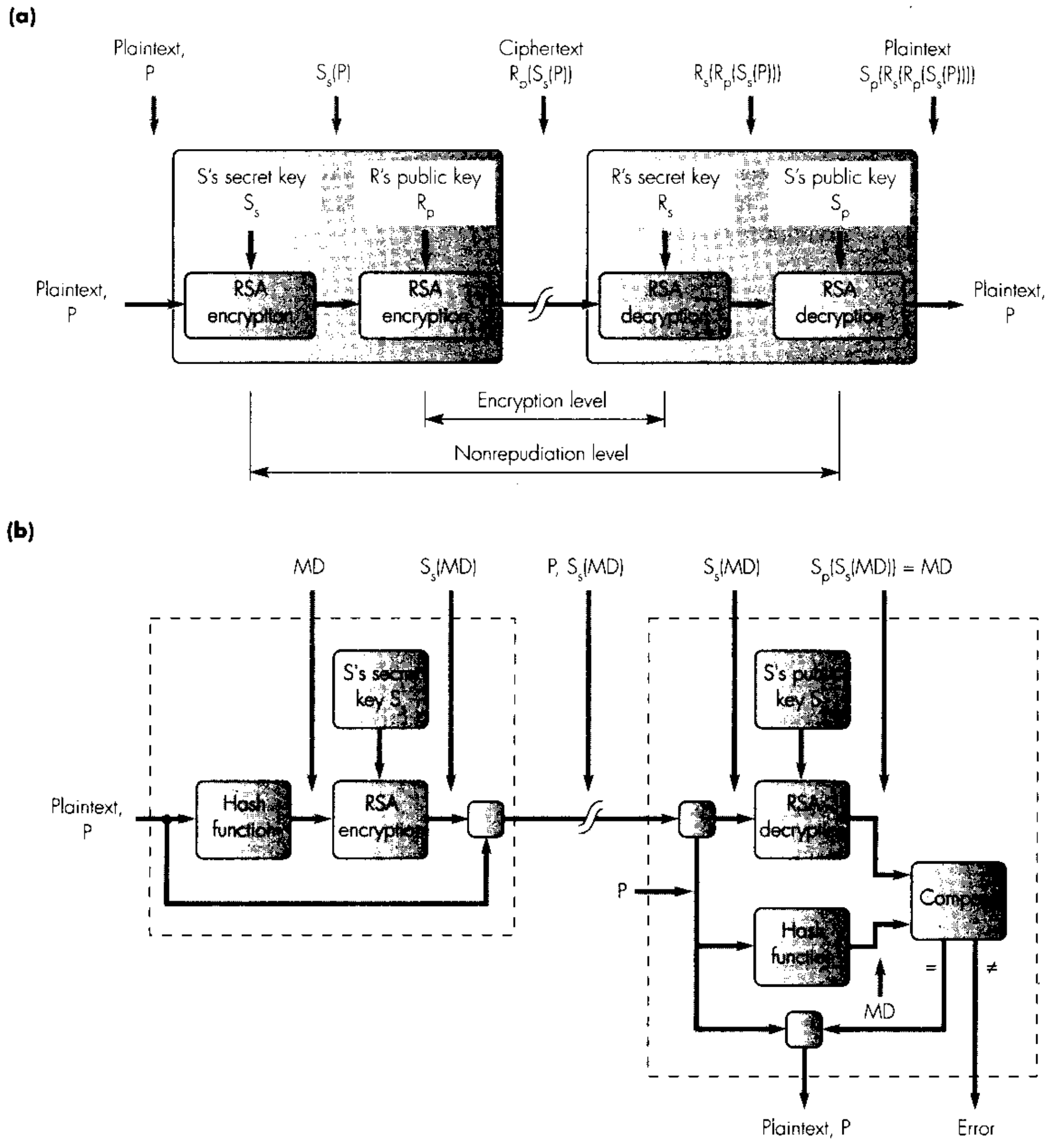


Figure 13.16 Nonrepudiation using RSA: (a) on complete message; (b) on message digest.

the outer level, the sender uses its own private key to encrypt the original (plaintext) message. If the receiver can decrypt this message using that sender's public key, this is proof that the sender did in fact initiate the sending of the message. The scheme is said therefore to produce a **digital signature**.

Although this is an elegant solution, it has a number of limitations. Firstly, the processing overheads associated with the RSA algorithm are high. As we saw with the earlier (much simplified) example, even with a small message (value), the numbers involved can be very large. Therefore a complete message must be divided into a number of smaller units, the size of which is a function of the computer being used. Hence, even though integrated circuits are available to help with these computations, the total message throughput with RSA is still relatively low. Secondly, the method requires two levels of encryption even though it may not be necessary to encrypt the actual message, that is, although only nonrepudiation is required, the actual message contents must still be encrypted.

One solution is to compute a much shorter version of the message based on the message contents, similar in principle to the computation of a CRC. The shorter version is called the **message digest** (**MD**) and the computation function that is used to compute it the hash function. The principle of the scheme is shown in Figure 13.16(b).

The MD is first computed using the chosen hash function. This is then encrypted using the sender's private key. The encrypted MD is then sent together with the plaintext message. At the receiver, the encrypted MD is decrypted using the sender's public key. The MD of the received plaintext message is also computed and, if this is the same as the decrypted MD, this is taken as proof that no one has tampered with the message and the sender whose public key was used to decrypt the MD did in fact send the message.

There are two widely used schemes that use this approach. One is called **MD5**, which was designed by Rivest, and the other the **secure hash algorithm** (**SHA**) which is a US government scheme. Both schemes operate on 512-bit blocks of plaintext. In the case of MD5 the computed MD is 128 bits long and for SHA, 160 bits long. As we shall see in the following chapters, MD5 is widely used with Internet applications and, because of its origin, SHA is used in government applications.

13.6 Authentication

In general, authentication is required when a client wishes to access some information or service from a networked server. Before the client is allowed access to the server, he or she must first prove to the server that they are a registered user. Once authenticated the user is then allowed access. The authentication process can be carried out using either a public key or a private key scheme. We shall give an example of each approach.

13.6.1 Using a public key system

The general principle of a scheme that is based on a public key system is shown in Figure 13.17. The scheme assumes that all potential users know the public key of the server, Sp. The client first creates a message containing the

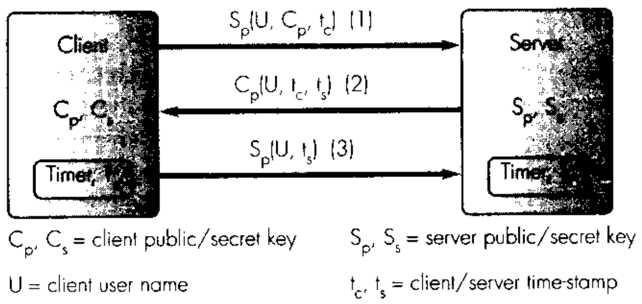


Figure 13.17 User authentication using a public key scheme.

client's user name, U, the client's public key, C_p, and a time-stamp, t_c. The latter indicates when the message was created and a record of this is kept by the client. The message is then encrypted using S_p and sent to the server (1).

The server first decrypts the message using its own secret key, S_s, and then proceeds to validate that there is such a registered user from the user name, U. Assuming this is the case, it then proceeds to create a response message comprising the client's user name, U, and time-stamp, t_c, plus a second time-stamp indicating when the response was created by the server, t_s. The server keeps a record of this and encrypts the message using the client's public key, C_p. It then sends the encrypted message to the client (2).

On receipt of the response, the client first decrypts the message using its own secret key, C_s, and, on determining that the t_c within it is the same as it sent, assumes that it has been authenticated by the server. It then proceeds to acknowledge this by creating a second message containing the client's user name, U, and the server's time-stamp, t_s. This again is encrypted using the server's public key, S_p, and sent to the server (3). The server decrypts this using its own secret key, S_s, and, on determining the t_s within it is the same as it sent, prepares to accept service requests from the client.

Note that in both cases, if the time the message was received exceeds the time-stamp value in the related response message by more than a defined time interval, then the message is discarded and access remains blocked. Also, should the transaction require the subsequent messages to be encrypted, then the key to be used would be returned by the server in message (2).

13.6.2 Using a private key system

An example of a method that is based on a private key system is **Kerberos**. This is widely used in many practical systems and, as we shall see, the method requires a trusted third party to act as a **key distribution server**.

The basic security control mechanism employed in Kerberos is a set of encrypted **tickets** – also known as control or permission tokens – which are used to control access to the various servers that make up the system. These include a range of application servers – file servers, electronic mail servers, and so on – and the system server that issues the tickets is known as the **ticket granting server**. All messages that are exchanged between a user and the ticket granting server and between a user and an application server, are encrypted using private keys which form part of the corresponding ticket. In addition, each message/ticket has a **nonce** associated with it. This comprises two date-and-time values the first of which specifies when the nonce was first generated. A nonce is used both to verify the origin of a message and to limit the validity of a ticket to a defined lifetime. This is determined by the second date-and-time value in the nonce. This feature means an eavesdropper has only a limited time to decrypt an intercepted ticket.

The key distribution server is a networked system with which all users and application servers must be registered. It comprises two servers: an **authentication server** and a ticket granting server. The authentication server provides, firstly, management services to allow all users, together with their related passwords, to be registered. It also provides the names and secret keys of all Kerberos servers, including the ticket granting server and all application servers. This information is retained in an **authentication database**. It then provides additional runtime services to enable a user to be authenticated as a registered user of the system before being allowed to access any of the Kerberos servers. A schematic diagram summarizing the various interactions between a user and the two types of server is given in Figure 13.18(a).

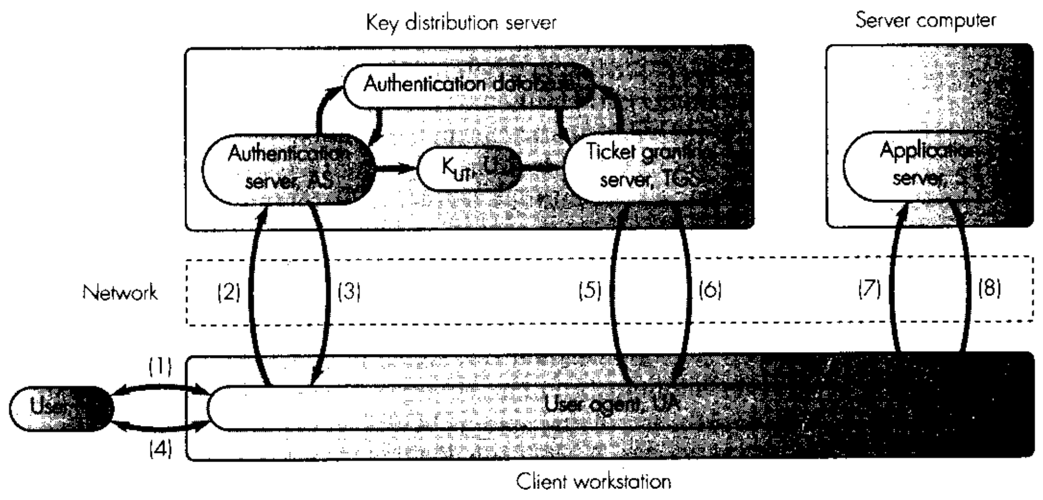
Running within each client workstation is a process called the **user agent** and it is through this that all interactions between a user and the various Kerberos servers take place. Before a user (agent) can access an application server, it must first obtain from the ticket granting server an **authentication ticket** and a **session key**; the first verifies that the user has been authenticated as a registered user and the second is used to encrypt all the subsequent dialog units that are exchanged in this session between the user agent and the application server. Note that in practice more than one application server can be involved in a single session. Also, both keys have a limited lifetime associated with them to guard against a user, whose registration has expired for example, from reusing a ticket.

At the start of a session, the user is prompted by the user agent (UA) for his/her user name (1). Before the UA can communicate with the ticket granting server (TGS), the user must first be authenticated as a registered user and a (permission) ticket obtained to access the TGS. Both these functions are performed by the authentication server (AS). On receipt of the user name, the UA creates a message containing the names of the user and the TGS and a nonce. The UA keeps a record of the nonce used and sends the message to the AS (2).

All subsequent messages associated with the session are encrypted using various keys. These are defined in Figure 13.18(b) together with the components that make up the two tickets; the first granting permission for the UA to communicate with the TGS and the second for the UA to communicate with the application server. The contents of the messages exchanged during a successful session are listed in Figure 13.18(c).

On receipt of the initial UA request message, the AS first validates the user is registered and, if positive, proceeds to create a response message. The

(a)



(b)

K_U =	The private key of the user – the user password
K_T =	The private key of the TGS
K_S =	The private key of the application server
K_{UT} =	A session key to encrypt UA ↔ TGS dialog units
K_{US} =	A session key to encrypt UA → S dialog units

TGS ticket, $T_{UT} = K_T\{U, T, t_1, t_2, K_{UT}\}$
Application server ticket, $T_{US} = K_S\{U, S, t_1, t_2, K_{US}\}$
t_1, t_2 = start, end of ticket lifetime

(c)

	Direction	Message
(1)	U ↔ UA	User name, U
(2)	UA → AS	$\{U, T; n_1\}$
(3)	AS → UA	$K_U\{K_{UT}, n_1\}; T_{UT}$
(4)	U ↔ UA	User password, K_U
(5)	UA → TGS	$K_{UT}\{U, t\}; T_{UT}, S, n_2$
(6)	TGS → UA	$K_{UT}\{K_{US}, n_2\}; T_{US}$
(7)	UA → S	$K_{US}\{U, t\}; T_{US}, n_3$
(8)	S → UA	$K_{US}\{n_3\}$

$K_{UT}/K_{US}\{U, t\}$ are both authenticators and t is a time-stamp

Figure 13.18 User authentication using Kerberos: (a) terminology and message exchange; (b) key and ticket definitions; (c) message contents.

latter comprises two parts. The first part consists of a newly generated session key, K_{UT} – to be used to encrypt the subsequent UA/TGS dialog units – together with the nonce, n_1, contained in the initial UA request message. A record is kept of K_{UT} and this part of the response message is then encrypted using the user's password, K_U – obtained from the authentication database – as a key. The second part comprises the permission ticket for the UA to access the TGS, T_{UT} – encrypted using the private key of the TGS, K_T. The two-part message is then returned to the UA (3).

At this point the UA prompts the user to enter his/her password, K_U (4). The latter is then used to obtain, firstly, the nonce, n_1 – which verifies the message relates to its earlier request – and secondly, the key K_{UT}. Clearly, a user impersonating as the registered user would not be able to decrypt this message and hence would be foiled at this stage. The UA then proceeds to use the retrieved key, K_{UT}, to create what is referred to as an **authenticator**. This is a token which verifies the user has been authenticated and comprises the user name, U, and a time-stamp, t, both encrypted using K_{UT}. To this is added the encrypted permission ticket, T_{UT}, the name of the required application server, S, and a second nonce, n_2. The complete message is then sent to the TGS (5).

The authenticator is decrypted by the TGS using the retained key K_{UT} and, since this was granted for the same user, U, it is accepted by the TGS as proof that the user has permission to be granted a session key to communicate with an application server. In response, the TGS generates a new session key, K_{US}, to be used by UA to encrypt the dialog units exchanged with the named server, S. Note that if multiple servers were to be accessed during the session, then multiple keys are issued at this stage, one for use with each server. The TGS then creates a message, the first part comprising K_{US} and the nonce n_2 – encrypted using K_{UT} – and the second comprising an encrypted permission ticket for the UA to access B, T_{US}. The complete message is then sent to the UA (6).

On receipt of this, the UA uses K_{UT} to decrypt the first part of the message to obtain K_{US} and the nonce n_2. The latter is used to confirm the message relates to its own earlier request message to the TGS, and K_{US} is used to create an authenticator, which verifies the user has been granted permission to access the named server. The authenticator is then combined with the permission ticket granted by the TGS, T_{US}, and a third nonce n_3. The resulting message is then sent by the UA to server S (7).

As Figure 13.18(b) shows, the permission ticket, T_{US}, is encrypted using the private key of S, K_S. Hence, on receipt of the message, S proceeds to use its own private key to decrypt T_{US} and obtain the name of the user, U, and the allocated session key, K_{US}. It uses the latter to decrypt the authenticator and confirm that U has been authenticated as a registered user and granted permission to access S. The server responds by returning the nonce, n_3, encrypted using K_{US} (8). This concludes the authentication procedure and the exchange of data messages between UA and S can now commence. If required, the data messages are encrypted using K_{US}.

13.7 Public key certification authorities

In the examples in previous sections that used the RSA public key scheme it was assumed that the public key was obtained in some way; for example by sending it in an email message prior to the transfer or making it available in a Web page. However, sending a public key in this way without any supporting proof of identify has potential drawbacks. The example shown in Figure 13.19 illustrates one of these.

In this example it is assumed that the person sending the message wants to discredit the person whose name is on the message. To do this, he or she sends a message that will achieve this in such a way that the recipient thinks the message has been sent by the person whose name is on it. To avoid this type of misuse of public key systems, in most applications the public key is obtained from a recognized **certification authority** (**CA**).

When a person registers with a recognized CA, after careful checks, an electronic **certificate** is created by the CA. This contains, in addition to the public key of the owner of the certificate, other information about the owner. The IETF have defined the complete list of contents of a certificate in **RFC 1422** and this is now used widely by a number of CAs. The fields present include:

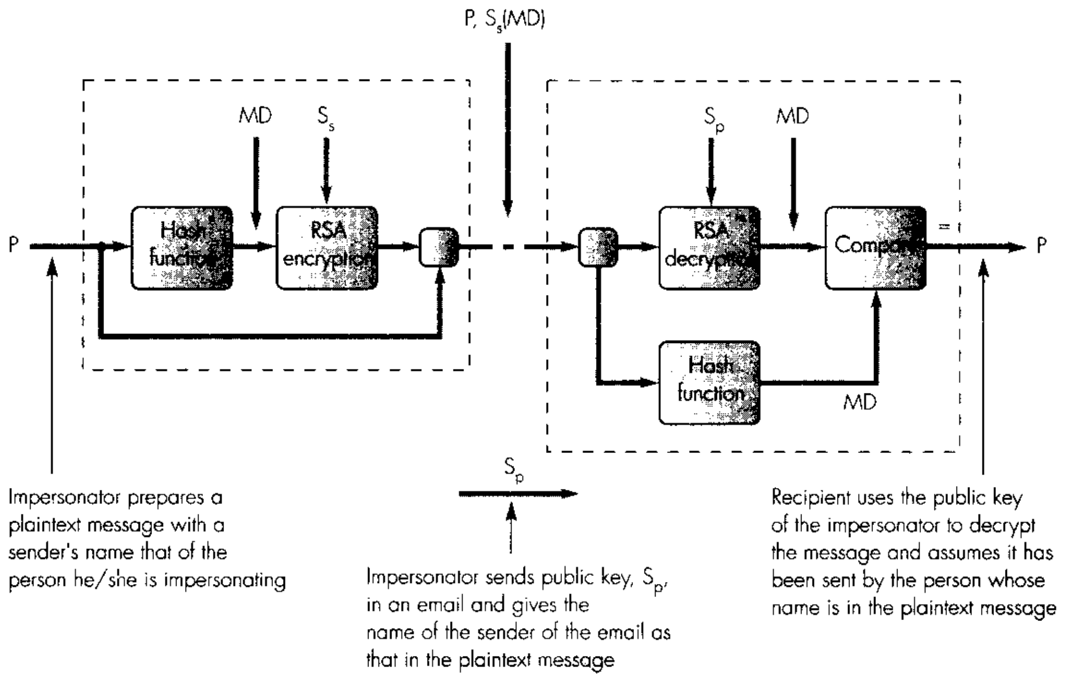

P, S_s(MD)
MD
S_s
P
Hash function
RSA encryption
S_p
MD
RSA decryption
Compare
=
P
Hash function
MD
S_p
Impersonator prepares a plaintext message with a sender's name that of the person he/she is impersonating
Impersonator sends public key, S_p, in an email and gives the name of the sender of the email as that in the plaintext message
Recipient uses the public key of the impersonator to decrypt the message and assumes it has been sent by the person whose name is in the plaintext message

Figure 13.19 A possible threat when using a public key system.

- issuer name: the identify (called the **distinguished name**) of the CA in a syntax defined in RFC 1779;
- serial number: the unique identifier of the certificate;
- subject name: the identity of the owner of the certificate, for example, an email address, a URL of a Web server, or an IP address of, say, a router;
- public key: the owner's public key and the key algorithm, for example RSA;
- validity period: start and end date of the validity of the certificate;
- signature: the algorithm used by the CA to encrypt the certificate contents, for example RSA.

Once a certificate has been created, the public key it contains can only be accessed through the CA. Typically, the CA is on a list of recognized CAs. The list is located at a well-known Web site and, for each CA on the list, its location and its public key are given. Then, when a public key is required from the CA, the subject's name is submitted and, in response, the CA returns the related certificate. The recipient proceeds to decrypt the contents of the certificate using the public key of the CA. It then reads the public key from the certificate and, before using it, validates that the name of the person on the certificate is that whose key is required.

Summary

In this chapter we have discussed two topics that are used widely in a range of distributed/networked applications. The first is concerned with ensuring that the shared information relating to a distributed application has the same meaning in all the computers/items of equipment that process the information. To achieve this, an international standard called ASN.1 has been defined. As we show in Figure 13.20, this comprises a standard abstract syntax that is used to define the data types associated with the shared information and also a set of encoding and decoding procedures that are used to convert the value associated with each data type into and from a standard transfer syntax.

The second topic relates to network security. This is concerned with four interrelated issues: secrecy, integrity, authentication, and nonrepudiation. We described the principle of operation of both the data encryption standard (DES) and the RSA algorithm. The first is useful for providing both secrecy and integrity and the second authentication and nonrepudiation. We also described the Kerberos system which is a widely used system for authentication.

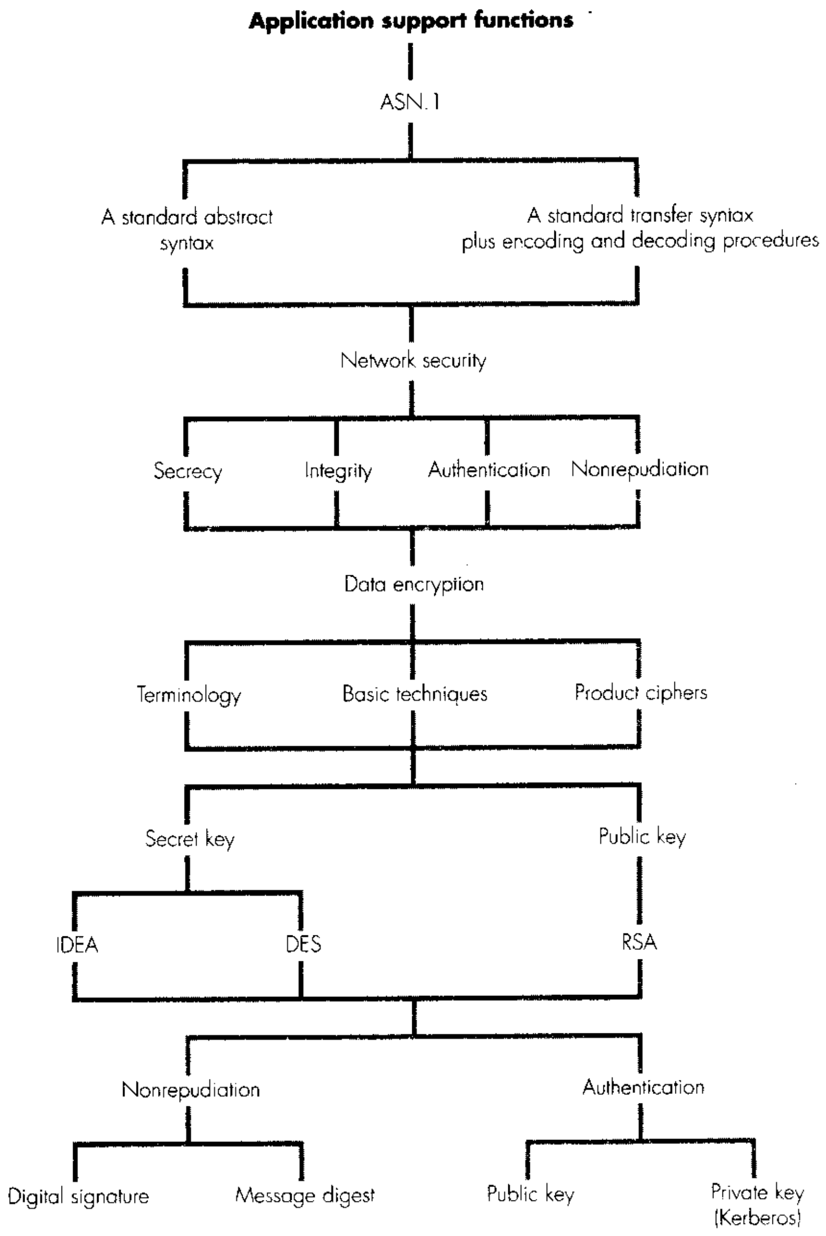


Figure 13.20 Summary of topics discussed in Chapter 13.

Exercises

Section 13.1

13.1 Explain the meaning of the following terms relating to a distributed application involving multiple different computers:
(i) data dictionary,
(ii) abstract syntax,
(iii) transfer/concrete syntax,
(iv) shared semantics.

Section 13.2

13.2 Explain the role of ASN.1 in relation to a distributed application and, with the aid of a diagram, describe the functions performed by an ASN.1 compiler.

13.3 Give an example variable name and ASN.1 type definition for each of the following primitive data types:
(i) boolean,
(ii) integer,
(iii) bitstring,
(iv) character string.

13.4 Give an example variable name and ASN.1 type definition for a sequence constructed type which includes the set of variables you listed in Exercise 13.3.

13.5 Explain the meaning of the terms "tag", "context-specific", and "application-specific". Hence modify the sequence type definition you used in Exercise 13.4 to include three context-specific tags and an application-specific tag.

13.6 Explain the meaning and use of the terms "implicit" and "explicit" in relation to a tag. Hence modify the sequence type definition you used in Exercise 13.5 to include a number of implicit type definitions.

13.7 Outline the role that ASN.1 can play in relation to the definition of the messages/PDUs relating to a protocol.
State an advantage and a disadvantage of using ASN.1 as an alternative to defining a message/PDU in the form of a number of fixed-length fields.

13.8 With the basic encoding rules associated with ASN.1, the transfer syntax used to transfer the value of a variable consists of an identifier, length, and contents field. Explain the use of each of these fields and, in the case of the identifier, the use of the class, type, and tag subfields.

13.9 Use example value assignments for each of the variables you defined in Exercise 13.3 to illustrate how each data type is encoded.

13.10 Assuming the same value assignments you used in Exercise 13.9, encode the two sequence-type variables you defined in Exercises 13.5 and 13.6.
Hence identify the benefits of using an implicit tag.

13.11 As an example, use the second of the encoded octet/byte strings you derived in Exercise 13.10 to explain how the decoding procedure in a correspondent application process determines the value to be assigned to each variable.

Section 13.3

13.12 Explain the meaning of the following terms relating to a secure transaction:
(i) integrity,
(ii) privacy/secrecy,
(iii) authentication,
(iv) nonrepudiation.

Section 13.4

13.13 With the aid of a diagram, explain the meaning of the following terms relating to data encryption:
(i) plaintext and ciphertext,
(ii) encryption key and decryption key,
(iii) eavesdropping and masquerading.

13.14 The following encrypted phrase has been produced using a simple substitution for the ciphertext alphabet. Assuming the phrase relates to communications, derive the ciphertext alphabet and hence the plaintext of the phrase:

ciphertext = frpsxwhu qhwzrunlgj

13.15 The following encrypted phrase has been produced using a simple transposition for the ciphertext alphabet. Derive the key that has been used and hence the plaintext of the phrase:

ciphertext = dcniaoiotmcnnamas ut

13.16 By means of a diagram, show the difference between a straight, expanded, and compressed or choice permutation/transposition. Use for example purposes a P-box with 8 input bits. State the key used in each case.

13.17 Assuming an S-box with 8 input bits, derive the size of the key that is required to perform a straight substitution operation.

With the aid of a diagram, show how the size of the key can be reduced by encapsulating a P-box between a binary decoder and a corresponding encoder. Use for example purposes an S-box with 2-input bits and 2-output bits.

Define a key for the P-box and hence list the four outputs for the four possible combinations of the two input bits.

13.18 A product cipher uses a combination of transpositions and substitutions. Design an encryption unit based on a product cipher that operates on 8 input bits. The unit is to be composed of a straight P-box followed by a block of 4 S-boxes of the type used in Exercise 13.17 and a second straight P-box. Define a suitable key for each stage. Hence for a selected 8-bit input, derive the 8-bit encrypted output from the unit.

13.19 With the aid of a diagram, outline the structure of the product cipher used with the DES algorithm. Show the steps that are carried out for each substitution operation within the product cipher assuming the bitwise function used in each substitution is the Feistel cipher.

13.20 State the three transposition operations that are carried out in the DES product cipher algorithm. Hence use a simple example to show that the output of the set of three transpositions is reversible.

Use a simple example to show how each substitution operation is reversible also. What are the implications of this?

13.21 With the aid of a diagram, describe the operation of the triple DES scheme. Hence explain why it is now being used in place of DES in some applications.

13.22 With the aid of schematic logic diagrams, describe the following operational modes of DES: electronic code book (ECB), chain block cipher (CBC), cipher feedback mode (CFM).

Quantify the number of encryption operations that are used to encrypt a 2048-byte file using
(i) CBC and
(ii) CFM.
Hence identify the main use of CFM.

13.23 With the aid of a schematic diagram, describe the operation of the IDEA scheme. Include in your description the size of the key used and the number and size of the subkeys associated with each encryption stage.

13.24 Each encryption stage in the IDEA scheme involves multiple addition and multiplication operations on pairs of 16-bit operands. Explain how the 16-bit product and 16-bit sum are derived.

13.25 With the aid of an example, explain the principle of operation of the RSA algorithm including how the public and private keys are derived. Use for example purposes the two prime numbers 3 and 11. State the maximum numeric value that can be encrypted with your choice of keys.

13.26 By means of an example, show how the exponentiation operations associated with the encryption and decryption stages of the RSA algorithm can be avoided. Hence assuming only messages composed of the 26 uppercase characters are to be sent, encrypt the string of characters AFKP using the public and private keys you derived in Exercise 13.25. Remember the limitation imposed by the choice of prime numbers.

Section 13.5

13.27 With the aid of a diagram, show how nonrepudiation can be obtained using the RSA algorithm:

(i) on the complete message,
(ii) on a digest of the message.

Clearly identify on your diagram the keys used and the encrypted/decrypted values at each stage.

Section 13.6

13.28 With the aid of a diagram, explain how user authentication can be carried out using a public key scheme. Include in your diagram the contents of each message and the key used to encrypt the message.

13.29 Explain the meaning and use of the following terms in relation to the authentication procedure associated with Kerberos:

(i) tickets,
(ii) ticket granting server,
(iii) nonce,
(iv) authentication server,
(v) authentication database,
(vi) authenticator.

13.30 With the aid of a diagram, identify a possible security threat that can occur with a public key system when the public key is made readily available. Describe how a certification authority can be used to overcome this threat. Include in your description a list of the fields that are present in the certificate and their use.

Internet applications

14.1 Introduction

As we showed in Figure 12.1 and explained in the accompanying text, in the TCP/IP protocol suite, given the IP address and port number of a destination application protocol/process (AP), the services provided by TCP or UDP enable two (or more) peer APs to communicate with each other in a transparent way. That is, it does not matter whether the correspondent AP(s) is(are) running in the same computer, another computer on the same network, or another computer attached to a network on the other side of the world. Also, since neither TCP nor UDP examines the content of the information being transferred, this can be a control message (PDU) associated with the application protocol, a file of characters from a selected character set, or a string of bytes output by a particular audio or video codec. Hence application protocols are concerned only with, firstly, ensuring the PDUs associated with the protocol are in the defined format and are exchanged in the specified sequence and secondly, the information/data being transferred is in an agreed transfer syntax so that it has the same meaning to each of the applications.

In this chapter, we discuss both the role and operation of a selection of the application protocols associated with the Internet. These are the simple