Team Members

1- Omar Mohamed Almaghrabi Ali El-gendi.

2- Akram Tarek Fouad Kashef.

3- Islam Hossam El-din Ibrahim Mohamed.

4- Kamal Mohamed Kamal.

Supervisors

1- Dr.Ibrahim El-Henawy.

2- Eng.Mahmoud Mahdy.

Contents

1-Introduction

Security and privacy of data is an overall issue, this fact rules for all kind of data at any time but especially for those which are transmitted in some way.
Communication systems are often seen as possible security leaks for transmitted data even though these systems employ data security techniques.
Voice encryption systems are used to guarantee end-to-end security for speech in real time communication systems such as GSM, VoIP, Telephone, analogue Radio.
Figure 1 illustrates a possible leak in a kind of network as we use it every day.

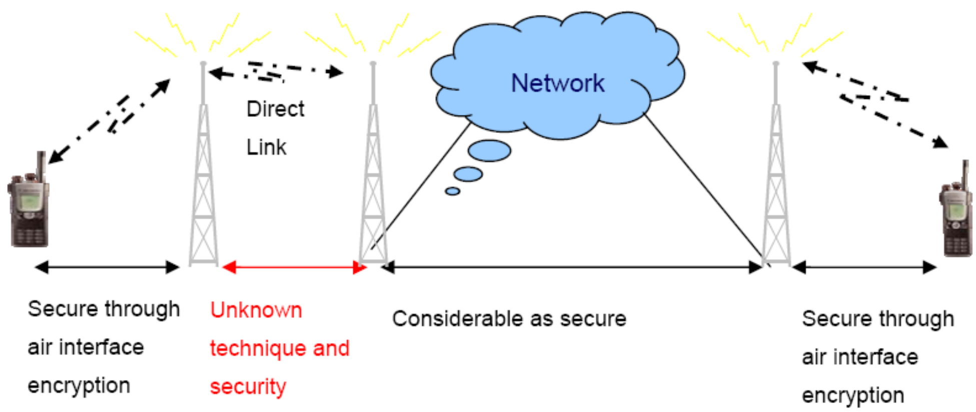

Figure 1: Radio Network with unsecured sectors

Surely there is quite a range of applications available on the market that solves this issue very good. Since this, often even standardised techniques are not meant to be challenged, the approach developed while this thesis can rather be seen as a demo for a lot of signal processing techniques (like channel filtering, up/down-sampling, real-time audio etc.) for academic purposes then as a serious guarantee for end- to end security.

2- Problem

The main problem in the most voice chat application between end – to – end applications is security.

It is possible to a third party system to enter the conversation between two users and hear the voice , in all cases this will lead to a disaster results.

So our main issue in our system is to handle the security threats to make voice conversation more secured.

3- Idea

We have designed and implemented a voice chat application between two end points that enables our users to make his/her voice chatting more secured and efficient.

3.1- System components :

Module 1:
Real time voice chat application over network using UDP sockets.

Module 2:

Implementing Encoder and Decoder to the voice captured from the microphone at the two end-points

Module 3:

This module handles our main issue Security , it encrypts the voice at the first end-point and decrypt it at the second end-point.

4- Implementation

4.1- Voice Chat application

4.1.1- Sockets overview :

A network socket is an endpoint of an inter-process communication flow across a computer network. Today, most communication between computers is based on the Internet Protocol; therefore most network sockets are Internet sockets.

An Internet socket is characterized by a unique combination of the following:

-Local socket address: Local IP address and port number

-Remote socket address: Only for established TCP sockets. As discussed in the client-server section below, this is necessary since a TCP server may serve several clients concurrently. The server creates one socket for each client, and these sockets share the same local socket address.

-Protocol: A transport protocol (e.g., TCP, UDP, raw IP, or others). TCP port 53 and UDP port 53 are consequently different, distinct sockets.

Sockets types :

There are several Internet socket types available:

-Datagram sockets, also known as connectionless sockets, which use User Datagram Protocol (UDP)

-Stream sockets, also known as connection-oriented sockets, which use Transmission Control Protocol (TCP) or Stream Control Transmission Protocol (SCTP).

-Raw sockets (or Raw IP sockets), typically available in routers and other network equipment. Here the transport layer is bypassed, and the packet headers are made accessible to the application.

There are also non-Internet sockets, implemented over other transport protocols, such as Systems Network Architecture (SNA).[2] See also Unix domain sockets (UDS), for internal inter-process communication.

How sockets work?

Within the operating system and the application that created a socket, a socket is referred to by a unique integer number called socket identifier or socket number. The operating system forwards the payload of incoming IP packets to the corresponding application by extracting the socket address information from the IP and transport protocol headers and stripping the headers from the application data.

To manage the connection between application layer network protocols, TCP and UDP use ports and sockets. TCP and UDP operate at the host-to-host layer in the IP communication model and provide host-to-host communication services for the application layer protocol. This means an application layer protocol is on one IP host connecting to an application layer protocol on another IP host.

A socket is like a handle to a file, which is used to open the path to communicate with another machine. It resembles the file IO, as does the serial communication. Using socket programming, we can have communication between two applications. The applications are typically on different computers or in the same computer. For the two applications to talk to each either on the same or different computers, one application is generally a server that keeps listening to the incoming requests and the other application acts as a client and makes the connection to the server application.

The server application can either accept or reject the connection. If the server accepts the connection, a dialog can begin between the client and the server. Once the client is done with whatever it needs to do, it can close the connection with the server. Connections are expensive in the sense that servers allow only finite connections to occur. During the time client has an active connection, it can send the data to the server and/or receive the data.

When the client or server is trying to make a connection, then we have to be careful. When either side (client or server) sends data, the other side is supposed to read the data. But how will the other side know when data has arrived. There are two options - either the application needs to poll for the data at regular intervals or there needs to be some sort of mechanism that would enable the application to get notifications and application can read the data at that time. Windows is an event driven system and the notification system seems an obvious and best choice and it in fact is.

The two applications that need to communicate with each other need to make a connection first. If each machine wants to make a connection they have to identify themselves. Using the IP of the machine, they are identified. IP address is nothing but the eight octal numbers, like 107.108.1.179. First two octal shows the network ID and third and fourth octal shows the host ID.

Using the code

To use sockets in .NET applications, we have to add the following using statements:

```
using System.Net;

using System.Net.Sockets;
```

Now we can create a socket object:

```
Socket sListener;
```

Programming the server

Let's create a click event that will enable the created socket to set its IpEndPoint and the protocol type.

But before that, a socket needs permission to work, because it will use a closed port number. A window will appear demanding permission to allow sending data.

```
permission = new SocketPermission(NetworkAccess.Accept,
```

```
TransportType.Tcp, "", SocketPermission.AllPorts);
```

As sockets use a network to transmit data, it uses protocols. The most known in this context are UDP which is fast but not reliable, and TCP which is reliable but not fast. Reliability is recommended when sending messages. That's why I use TCP.

```
sListener = new Socket(ipAddr.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
```

The socket needs to have an address. It's of type IpEndPoint. Each socket is identified through the IP address, which is useful to locate the host's machine, and the port number to identify which program is using the socket inside the machine.

```
IPHostEntry ipHost = Dns.GetHostEntry("");

IPAddress ipAddr = ipHost.AddressList[0];

ipEndPoint = new IPEndPoint(ipAddr, 4510);
```

Now we will associate our socket with the IpEndPoint:

```
sListener.Bind(ipEndPoint);
```

So that our socket is ready to use, let's start listening on the chosen port number (4510). You can choose another port number. But the client has to be aware about that. Listening will be handled through this button's event:

Place a socket in a listening state and specify how many client sockets could connect to it:

```
sListener.Listen(10);
```

The server will begin an asynchronous operation to accept an attempt. One of the powerful features of sockets is the use of the asynchronous programming model. Thanks to it, our program can continue running while the socket is performing actions.

```
AsyncCallback aCallback = new AsyncCallback(AcceptCallback);

sListener.BeginAccept(aCallback, sListener);
```

If there is any attempt from the client to connect, the following code will be executed:

```
Socket listener = (Socket)ar.AsyncState;

Socket handler  = listener.EndAccept(ar);
```

To begin to asynchronously receive data, we need an array of type Byte for the received data, the zero-based position in the buffer, and the number of bytes to receive.

```
handler.BeginReceive(buffer, 0, buffer.Length,

     SocketFlags.None, new AsyncCallback(ReceiveCallback), obj);
```

If the client sends any message, the server will try to get it. As sockets send data in binary type, converting them to string type is necessary. It's good to know also that the server, and even the client, don't know anything about the length of the message or the time needed for listening to all of it. That's why we use a special string "<Client Quit>" to put in the end of the message to tell that the text message ends there. To receive data, BeginReceive is called:

```
byte[] buffernew = new byte[1024];

obj[0] = buffernew;

obj[1] = handler;

handler.BeginReceive(buffernew, 0, buffernew.Length,

    SocketFlags.None, new AsyncCallback(ReceiveCallback), obj);
```

After receiving the client's messages, the server may want to reply. But it has to convert the string message in str to bytes data, because sockets manipulate bytes only.

```
byte[] byteData = Encoding.Unicode.GetBytes(str);
```

The server now will send data asynchronously to the connected socket:

```
handler.BeginSend(byteData, 0, byteData.Length, 0, new AsyncCallback(SendCallback), handler);
```

When using the asynchronous programming model to build this sample, there's no risk of application or interface blocking. It will seem like we are using multiple threads.

After writing code, you want to handle the different methods through a user interface. So you may have a UI like this:

Programming the client

The client will try to connect to the server. But it has to know its address.

After creating a SocketPermission for socket access restrictions and creating a socket with a matching IpEndPoint, we have to establish a connection to the remote server host:

```
senderSock.Connect(ipEndPoint);
```

We have to note here that the created IpEndPoint will not be used to identify the client. But it'll be used to identify the server socket.

To send messages, the client adds "<Client Quit>" to mark the end of message and must transform the text message to binary format, as the server did. After that, the socket will send the message by invoking the method Send which will take the binary message as a parameter.

```
byte[] msg = Encoding.Unicode.GetBytes(theMessageToSend + "<Client Quit>");
```

```
int bytesSend = senderSock.Send(msg);
```

For receiving data from the server, it converts the byte array to string and continues to read the data till data isn't available:

```
String theMessageToReceive = Encoding.Unicode.GetString(bytes, 0, bytesRec);
while (senderSock.Available > 0)
{
  bytesRec = senderSock.Receive(bytes);
  theMessageToReceive += Encoding.Unicode.GetString(bytes, 0, bytesRec);
}
```

4.2- Encoder/Decoder

4.2.1- Introduction:

In our application we use G.711

G.711 is an ITU-T standard for audio companding. It is primarily used in telephony.

Its formal name is *Pulse code modulation (PCM) of voice frequencies*. It is required standard in many technologies, for example in H.320 and H.323 specifications. It can also be used for fax communication over IP networks (as defined in T.38specification). G.711, also known as Pulse Code Modulation (PCM), is a very commonly used waveform codec. G.711 is a narrowband audio codec that provides toll-quality audio at 64 kbit/s. G.711 passes audio signals in the range of 300–3400 Hz and sampling them at the rate of 8,000 samples per second, with the tolerance on that rate 50 parts per million (ppm).

4.2.2- Features :

- Sampling frequency 8 kHz.
- 64 kbit/s bitrate (8 kHz sampling frequency x 8 bits per sample).
- Typical algorithmic delay is 0.125 ms, with no look-ahead delay.
- G.711 is a waveform speech coder.
- G.711 Appendix I defines a Packet Loss Concealment (PLC) algorithm to help hide transmission losses in a packetized network.
- G.711 Appendix II defines a Discontinuous Transmission (DTX) algorithm which uses Voice Activity Detection (VAD) and Comfort Noise Generation (CNG) to reduce bandwidth usage during silence periods.
- PSQM testing under ideal conditions yields Mean Opinion Scores of 4.45 for G.711 μ-law, 4.45 for G.711 A-law.
- PSQM testing under network stress yields Mean Opinion Scores of 4.13 for G.711 μ-law, 4.11 for G.711 A-law.

4.2.3 - A-law :

A-law encoding thus takes a 13-bit signed linear audio sample as input and converts it to an 8 bit value as follows:

Linear input code	Compressed code
s0000000wxyz`a	s000wxyz
s0000001wxyz`a	s001wxyz
s000001wxyz`ab	s010wxyz
s00001wxyz`abc	s011wxyz
s0001wxyz`abcd	s100wxyz
s001wxyz`abcde	s101wxyz
s01wxyz`abcdef	s110wxyz
s1wxyz`abcdefg	s111wxyz

Where `s` is the sign bit, and bits after the backtick mark are discarded. So for example, 1'0000'0001'0101 maps to 1000'1010 (according to the first row of the table), and 0'0000'0011'0101 maps to 0001'1010 (according to the second).

This can be seen as a floating point number with 4 bits of mantissa and 3 bits of exponent.

In addition, the standard specifies that all resulting even bits are inverted before the octet is transmitted. This is to provide plenty of 0/1 transitions to facilitate the clock recovery process in the PCM receivers. Thus, a silent A-law encoded PCM channel has the 8 bit samples coded 0x55 instead of 0x00 in the octets (or 0xD5 if the sign bit happens to be set).

Note that the ITU define bit 1 to have the value 128 and bit 8 to have the value 1.

The more widely accepted convention has bit 7 = 128 and bit 0 = 1.

Note that when data is sent over E0 (G.703), MSB (signbit) is sent first and LSB is sent last.

ITU-T STL defines the algorithm as follows:

4.2.4 - A-law encoder :

```
public class ALawEncoder
    {
        public const int MAX = 0x7fff;

        private static byte[] pcmToALawMap;

        static ALawEncoder()
        {
            pcmToALawMap = new byte[65536];
            for (int i = short.MinValue; i <= short.MaxValue; i++)
                pcmToALawMap[(i & 0xffff)] = encode(i);
        }


        private static byte encode(int pcm)
        {
                        int sign = (pcm & 0x8000) >> 8;

            if (sign != 0)
                pcm = -pcm;
                        if (pcm > MAX) pcm = MAX;


            int exponent = 7;
                        for (int expMask = 0x4000; (pcm & expMask) == 0 && exponent>0;
exponent--, expMask >>= 1) { }


            int mantissa = (pcm >> ((exponent == 0) ? 4 : (exponent + 3))) & 0x0f;

                        byte alaw = (byte)(sign | exponent << 4 | mantissa);

                        return (byte)(alaw^0xD5);
        }


        public static byte ALawEncode(int pcm)
        {
            return pcmToALawMap[pcm & 0xffff];
```

```
        }

        public static byte ALawEncode(short pcm)
        {
            return pcmToALawMap[pcm & 0xffff];
        }

        public static byte[] ALawEncode(int[] data)
        {
            int size = data.Length;
            byte[] encoded = new byte[size];
            for (int i = 0; i < size; i++)
                encoded[i] = ALawEncode(data[i]);
            return encoded;
        }

        public static byte[] ALawEncode(short[] data)
        {
            int size = data.Length;
            byte[] encoded = new byte[size];
            for (int i = 0; i < size; i++)
                encoded[i] = ALawEncode(data[i]);
            return encoded;
        }

        public static byte[] ALawEncode(byte[] data)
        {
            int size = data.Length / 2;
            byte[] encoded = new byte[size];
            for (int i = 0; i < size; i++)
                encoded[i] = ALawEncode((data[2 * i + 1] << 8) | data[2 * i]);
            return encoded;
        }

        public static void ALawEncode(byte[] data, byte[] target)
        {
            int size = data.Length / 2;
            for (int i = 0; i < size; i++)
                target[i] = ALawEncode((data[2 * i + 1] << 8) | data[2 * i]);
        }
    }
}
```

4.2.5 - A-law decoder :

```
public static class ALawDecoder
    {

        private static short[] aLawToPcmMap;

        static ALawDecoder()
        {
            aLawToPcmMap = new short[256];
            for (byte i = 0; i < byte.MaxValue; i++)
                aLawToPcmMap[i] = decode(i);
        }
```

```
    }

    private static short decode(byte alaw)
    {
                alaw ^= 0xD5;

                int sign = alaw & 0x80;
                int exponent = (alaw & 0x70) >> 4;
                int data = alaw & 0x0f;

                data <<= 4;
                data += 8;


        if (exponent != 0)
            data += 0x100;

        if (exponent > 1)
            data <<= (exponent - 1);

        return (short)(sign == 0 ? data : -data);
    }


    public static short ALawDecode(byte alaw)
    {
        return aLawToPcmMap[alaw];
    }


    public static short[] ALawDecode(byte[] data)
    {
        int size = data.Length;
        short[] decoded = new short[size];
        for (int i = 0; i < size; i++)
            decoded[i] = aLawToPcmMap[data[i]];
        return decoded;
    }


    public static void ALawDecode(byte[] data, out short[] decoded)
    {
        int size = data.Length;
        decoded = new short[size];
        for (int i = 0; i < size; i++)
            decoded[i] = aLawToPcmMap[data[i]];
    }


    public static void ALawDecode(byte[] data, out byte[] decoded)
    {
        int size = data.Length;
        decoded = new byte[size * 2];
        for (int i = 0; i < size; i++)
        {
                    decoded[2 * i] = (byte)(aLawToPcmMap[data[i]] & 0xff);
                    decoded[2 * i + 1] = (byte)(aLawToPcmMap[data[i]] >> 8);
        }
    }
}
```

4.2.6 - MU-law :

μ-law encoding takes a 14-bit signed linear audio sample as input, increases the magnitude by 32 (binary 100000), and converts it to an 8 bit value as follows:

Linear input code	Compressed code
s00000001wxyz`a	s000wxyz
s0000001wxyz`ab	s001wxyz
s000001wxyz`abc	s010wxyz
s00001wxyz`abcd	s011wxyz
s0001wxyz`abcde	s100wxyz
s001wxyz`abcdef	s101wxyz
s01wxyz`abcdefg	s110wxyz
s1wxyz`abcdefgh	s111wxyz

Where s is the sign bit, and bits after the backtick mark ` are discarded.

In addition, the standard specifies that all result bits are inverted before the octet is transmitted. Thus, a silent μ-law encoded PCM channel has the 8 bit samples coded 0xFF instead of 0x00 in the octets.

Adding 32 is necessary so that all values fall into a compression group. It is added back at the receiver to the inverted 8bit values.

This means µ-law does not encode all 14-bit values; inputs must be within ±8159.

4.2.7 - MU-law Encoder :

```
public class MuLawEncoder
  {
      public const int BIAS = 0x84;
      public const int MAX = 32635; //


      public bool ZeroTrap
      {
          get { return (pcmToMuLawMap[33000] != 0); }
          set
          {
              byte val = (byte)(value ? 2 : 0);
              for (int i = 32768; i <= 33924; i++)
                  pcmToMuLawMap[i] = val;
          }
      }


      private static byte[] pcmToMuLawMap;

      static MuLawEncoder()
      {
          pcmToMuLawMap = new byte[65536];
          for (int i = short.MinValue; i <= short.MaxValue; i++)
              pcmToMuLawMap[(i & 0xffff)] = encode(i);
      }


      private static byte encode(int pcm)
      {
                 int sign = (pcm & 0x8000) >> 8;
                 if (sign != 0)
              pcm = -pcm;
                 if (pcm > MAX) pcm = MAX;
                 pcm += BIAS;
          int exponent = 7;
          for (int expMask = 0x4000; (pcm & expMask) == 0; exponent--, expMask >>= 1) { }


          int mantissa = (pcm >> (exponent + 3)) & 0x0f;

                      byte mulaw = (byte)(sign | exponent << 4 | mantissa);

                      return (byte)~mulaw;
      }


      public static byte MuLawEncode(int pcm)
      {
          return pcmToMuLawMap[pcm & 0xffff];
      }
```

```
        public static byte MuLawEncode(short pcm)
        {
            return pcmToMuLawMap[pcm & 0xffff];
        }


        public static byte[] MuLawEncode(int[] data)
        {
            int size = data.Length;
            byte[] encoded = new byte[size];
            for (int i = 0; i < size; i++)
                encoded[i] = MuLawEncode(data[i]);
            return encoded;
        }


        public static byte[] MuLawEncode(short[] data)
        {
            int size = data.Length;
            byte[] encoded = new byte[size];
            for (int i = 0; i < size; i++)
                encoded[i] = MuLawEncode(data[i]);
            return encoded;
        }


        public static byte[] MuLawEncode(byte[] data)
        {
            int size = data.Length / 2;
            byte[] encoded = new byte[size];
            for (int i = 0; i < size; i++)
                encoded[i] = MuLawEncode((data[2 * i + 1] << 8) | data[2 * i]);
            return encoded;
        }



        public static void MuLawEncode(byte[] data, byte[] target)
        {
            int size = data.Length / 2;
            for (int i = 0; i < size; i++)
                target[i] = MuLawEncode((data[2 * i + 1] << 8) | data[2 * i]);
        }
    }
}
```

4.2.8 - MU-law Decoder :

```
public static class MuLawDecoder
    {

        private static short[] muLawToPcmMap;

        static MuLawDecoder()
        {
            muLawToPcmMap = new short[256];
            for (byte i = 0; i < byte.MaxValue; i++)
                muLawToPcmMap[i] = decode(i);
        }


        private static short decode(byte mulaw)
        {
                   mulaw = (byte)~mulaw;
            int sign = mulaw & 0x80;
            int exponent = (mulaw & 0x70) >> 4;
            int data = mulaw & 0x0f;
            data |= 0x10;
            data <<= 1;
            data += 1;

            data <<= exponent + 2;
                       data -= MuLawEncoder.BIAS;
                       return (short)(sign == 0 ? data : -data);
        }


        public static short MuLawDecode(byte mulaw)
        {
            return muLawToPcmMap[mulaw];
        }


        public static short[] MuLawDecode(byte[] data)
        {
            int size = data.Length;
            short[] decoded = new short[size];
            for (int i = 0; i < size; i++)
                decoded[i] = muLawToPcmMap[data[i]];
            return decoded;
        }
        public static void MuLawDecode(byte[] data, out short[] decoded)
        {
            int size = data.Length;
            decoded = new short[size];
            for (int i = 0; i < size; i++)
                decoded[i] = muLawToPcmMap[data[i]];
        }


        public static void MuLawDecode(byte[] data, out byte[] decoded)
        {
            int size = data.Length;
            decoded = new byte[size * 2];
            for (int i = 0; i < size; i++)
            {
```

```
                        decoded[2 * i] = (byte)(muLawToPcmMap[data[i]] & 0xff);
                        decoded[2 * i + 1] = (byte)(muLawToPcmMap[data[i]] >> 8);
                }
            }
        }
    }
```

4.2.9 - Comparison a-law to mu-law:

The µ-law algorithm provides a slightly larger dynamic range than the A-law at the cost of worse proportional distortion for small signals. By convention, A-law is used for an international connection if at least one country uses it.

5- Voice Encryption/Decryption

5.1 - Introduction

Developed in 1974 by IBM in cooperation with the National Securities Agency (NSA), DES has been the worldwide encryption standard for more than 20 years. For these 20 years it has held up against cryptanalysis remarkably well and is still secure against all but possibly the most powerful of adversaries [4]. Because of its prevalence throughout the encryption market, DES is an excellent interoperability standard between different encryption equipment.

The predominant weakness of DES is its 56-bit key which, more than sufficient for the time period in which it was developed, has become insufficient to protect against brute-force attack by modern computers [5] (see table 1). As a result of the need for a greater encryption strength, DES evolved into triple-DES. Triple-DES encrypts using three 56-bit keys, for an encryption strength equivalent to a 168-bit key. This implementation, however, requires three times as many rounds for encryption and decryption and highlights a second weakness of DES – speed. DES was developed for implementation on hardware, and DES implementations in software are often less efficient than other standards which have been developed with software performance in mind.

As is highlighted in the Results section of this application report, however, the C6000 core is able to achieve very impressive data rates for DES and triple-DES in software. Data rates on the C6201 (200 MHz) are measured as high as 53 Mbits per second for DES and 22 Mbits per second for triple-DES. Data rates on the C6211 (150 MHz) device are measured as high as 39 Mbits per second for DES and 18 Mbits per second for triple-DES. On a 300 MHz device such as the C6203, we can extrapolate data rates of 80 Mbits per second for DES and 33 Mbits per second for triple-DES.

This performance is typically more than sufficient for multichannel applications involving 8 Kbit per second vocoders and/or 56 Kbytes per second modems. Even DES encryption at the full G.lite ADSL rate of 1.5 Mbytes per second requires less than 50 MHz performance on the C6000 core and could be added to an application with a simple upgrade from C6201 to the C6202 or from the C6202 to the C6203. Due to the C6000 platform's strong performance in multi-channel communications applications, DES may be implemented as part of a one-chip re-programmable multi-channel solution. This provides great savings in board space, cost, power consumption and design time. Furthermore, because the solution is re-programmable, it may be easily evolved to match the rapidly changing encryption market.

5.2 - DES Algorithm:

The Data EncryptionStandard (DES) has been developed as a cryptographicstandard for generalusebythepublic. DESwasdesignedwiththefollowingobjectivesinmind[NIS77,Pfl89]:

1. Highlevelofsecurity
2. Completely specifiedandeasytounderstand
3. Cryptographicsecurity donotdepend onalgorithm secrecy
4. Adaptabletodiverseapplications
5. Economical hardware implementation
6. Efficient(e.g. highdata rates)
7. Canbevalidated
8. Exportable

5.2.1 – Data Encryption Algorithm

- Substitution-permutationalgorithm:
  - 64-bitinputandoutputblocks
  - 56-bitkey(with anadditional8parity bits)
  - informationdata iscycled16timesthrough asetofsubstitutionandpermutationtrans- formations: highlynon-linear input-outputrelationship
- Veryhighthroughputrates achievable (upto100Mbits/s)
- Availabilityofeconomicalhardware toimplementDES

- Lowtomedium security applications(e.g. securespeechcommunications)

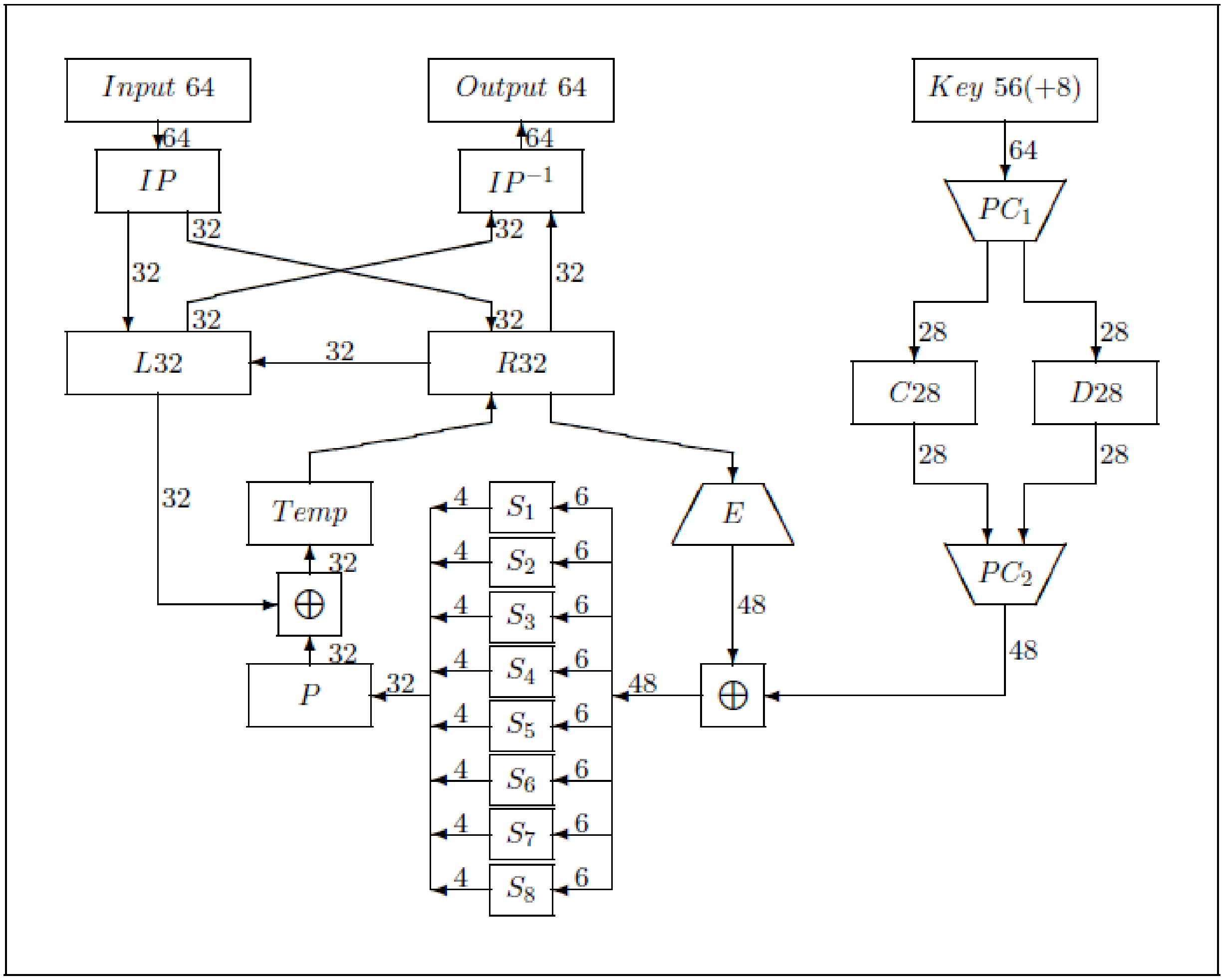

Figure 1: DES encryption/decryption algorithm.

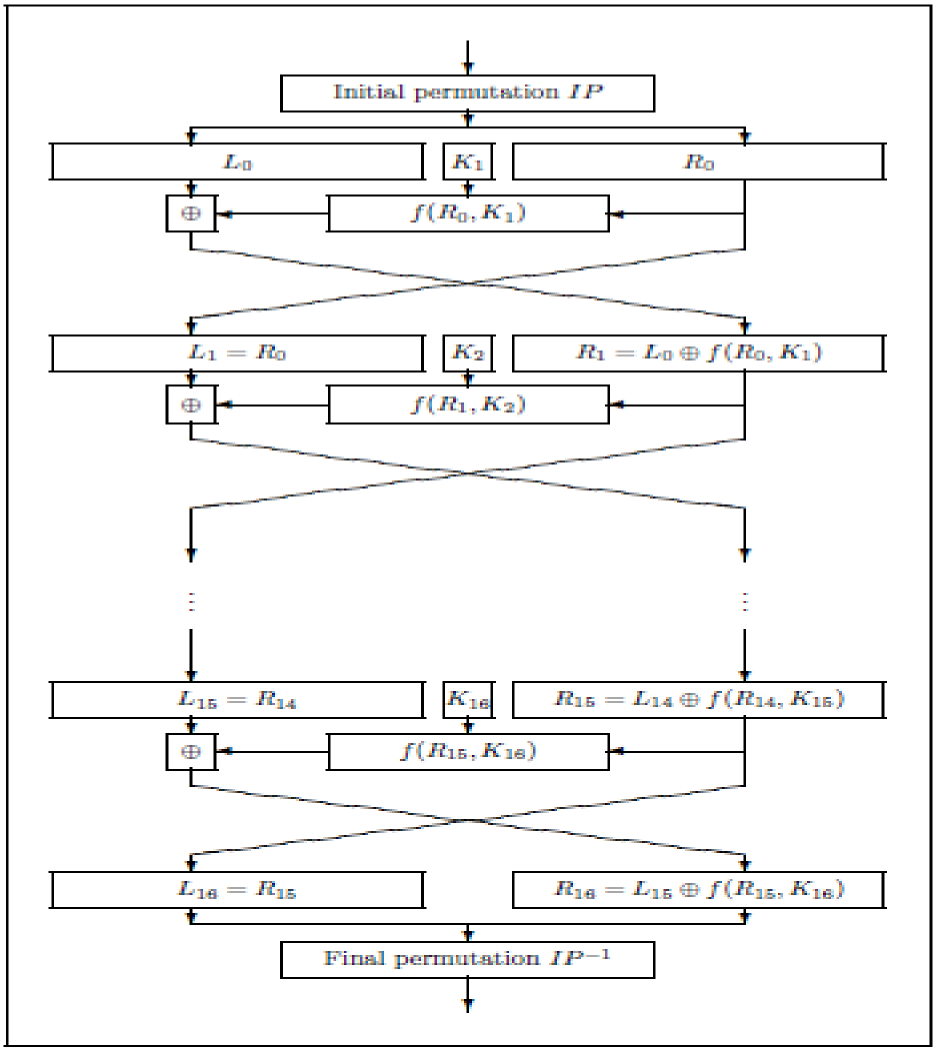

Figure 2: DES sequence of substitution and permutation transformations.

Table 1: Initial IP and inverse initial IP^{-1} permutation tables.

Initial permutation IP

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

Final permutation IP^{-1}

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

Table 2: Expansion permutation E and permutation P tables.

Expansion permutation E

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Permutation P

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

5.2.2 - DES Weakness

5.2.2.1 – Key Spacesize

InDES,thekeyconsistsina56-bitvectorprovidingakeyspaceKof2^{56}=7.2058×10^{16}elements. Inanexhaustive searchknown-plaintextattack,thecryptanalystwillobtain thesolutionafter2^{55}, or3.6029×10^{16} trials, onaverage.

In1977,DiffieandHellman[DH77]haveshownthataspecialpurposemultipleparallelprocessor consistingof10^6intergratedcircuits,eachonetryingakeyevery1µs,coulddetermine thekeyused inabout10hours onaverage inaknown-plaintextattack. The costofsuchamultiple processor machinewouldhavebeenaround $50,000,000in1977[Pfl89].Ifsuchamachinewasused365days ayear, 24hours aday, amortizing the priceoverthe number ofkeysolutions obtained, then the pricepersolution wouldhavebeenabout$20,000persolution.

DiffieandHellmanarguedthatifthekeylengthwasincreasedfrom56to64bits,itwouldmake theDESalgorithm secureevenfor“intelligence agenciesbudgets...” [Sim92],whiledecreasing the keylengthfrom56toonly48wouldmakeDES“vulnerabletoattackbyalmostanyreasonable sized organization”[Sim92].Thekeylength isthus averycritical parametertothesecurity ofDES.

5.2.2.2 – Complement property

Another possibleweaknessofDESliesinthecomplementpropertyoftheDESalgorithm. LetM bea64-bitplaintextmessagetobeencryptedintoa64-bitciphertextCusingthe56-bitkeyK:

$$C=DES_K(M)$$

ThecomplementpropertyofDES[Pfl89]indicates that thebit-by-bitmodulo-2complementof theciphertextC,i.e.$\bar{C}$,canbeobtained fromtheplaintextM andkeyK as:

$$\bar{C} = DES_{\bar{K}}(\bar{M})$$
$$\bar{C} = \overline{DES_K(M)}$$

Sincecomplementingtheciphertextvector$\bar{C}$takesmuchlesstimethan actually performingthe DESencryptiontransformation,theexhaustive keysearchattackcanbereduced almost byhalf.

5.2.2.3 – DES weak keys

TheDESalgorithm generates fromthe 56-bitkeyK aset,orsequence,of16distinct 48-bitsub-keyswhicharethen usedineachround ofsubstitutionand permutationtransformationofDES. However,ifthe leftand rightregisters C_i and D_i ofthe sub-key schedule calculation branch are filledwith“0”or“1”,thesub-keyswillbeidentical:

$$k_1=k_2=...=k_{16}$$

The encryptionand decryptionprocessesbeingthe sameexcept forthe order ofsub-keys, when suchweakkeysareemployed, enciphering aplaintextmessagesM twicewillresult intheoriginal plaintextmessage[DP84]:

$$DES_K[DES_K(M)]=M$$

TheweakkeysoftheDESarelistedhereafter:

K_1	=	01	01	01	01	01	01	01	01
K_2	=	FE	FE	FE	FE	FE	FE	FE	FE
K_3	=	1F	1F	1F	1F	0E	0E	0E	0E
K_4	=	E0	E0	E0	E0	F1	F1	F1	F1

5.2.2.4 – DES semi – weak key pairs

Another propertyobserved inthe DESalgorithm isthe existence ofsemi-weakpairs ofkeysfor which the patternofalternatingzeroes and ones inthe twosub-key registers C_i and D_i. This results inthe firstkey,sayK_1, producing the sub-keysequence: $k_1, k_2, \ldots, k_{16}$,whilethe second keyofthepair, K_2,generates theinversesub-keysequence: $k_{16}, k_{15}, \ldots, k_1$. Thus theencryption ofmessageMbykeyK_1 followedbyasecondencryptionwithkeyK_2 willgivetheoriginalmessage M:

$$DES_{K_2}[DES_{K_1}(M)]=M$$

Thesemi-weakkeysoftheDESare[DP84]:

$K_{1,1}$	=	01	FE	01	FE	01	FE	01	FE
$K_{1,2}$	=	FE	01	FE	01	FE	01	FE	01
$K_{2,1}$	=	1F	E0	1F	E0	0E	F1	0E	F1
$K_{2,2}$	=	E0	1F	E0	1F	F1	0E	F1	0E
$K_{3,1}$	=	01	E0	01	E0	01	F1	01	F1
$K_{3,2}$	=	E0	01	E0	01	F1	01	F1	01
$K_{4,1}$	=	1F	FE	1F	FE	0E	FE	0E	FE
$K_{4,2}$	=	FE	1F	FE	1F	FE	0E	FE	0E
$K_{5,1}$	=	01	1F	01	1F	01	0E	01	0E
$K_{5,2}$	=	1F	01	1F	01	0E	01	0E	01
$K_{6,1}$	=	E0	FE	E0	FE	F1	FE	F1	FE
$K_{6,2}$	=	FE	E0	FE	E0	FE	F1	FE	F1

5.3 – Differential and linear cryptoanalysis

Traditional cryptanalysisofblock ciphers such as the Data EncryptionStandard rely on such known plaintextmethods asdoingexhaustive search overthe wholekeyspace. While this type ofbrute forcecryptanalyticattackmay seem practical on conventional singleDES encryption, it becomesimpracticalto perform ondouble DES and triple DES enciphering implementations. Moresophisticatedcryptanalysismethods have been proposed inthe recentyears to reduce the computationalcomplexity ofabrute forceattack.Twosuchmethods aredifferentialcryptanalysis andlinearcryptanalysiscryptanalysis.Differential cryptanalysisisbrieflydescribed insection4.1 andlinearcryptanalysiscryptanalysisinsection4.2.

5.3.1 – Differential cryptoanalysis

Differentialcryptanalysishasbeenproposedsince1990 tobreakblockcipherssuchasDESandits predecessor LUCIFER.Whilesuccessfulforbreaking LUCIFER,differential cryptanalysisisstill, atleastforthetimebeing,of“academic” interestforbreaking the16-roundfull-fledgedDES.The reasonwhyDESisresistantagainst differentialcryptanalysisisthatwhiledifferentialcryptanalysis hasbeenknowntothegeneralpublicforless than tenyears,itstechniques wereknowntotheDES developersintheseventies. Neverthless differential cryptanalysis,aslinearcryptanalysis,isoneof themostpromisingcryptanalysismethods.

Differential cryptanalysisinvolvesthe analysis ofthe distributionofthe difference(modulo-2 bit perbit) between twoplaintextsX_1 and X_2 and the twociphertextsY_1 and Y_2 resulting from their encryption.HeretheplaintextsX_1 andX_2 areinfactthe32-bitcontents oftherightregister prior the extension permutationE(X)inaDESround. The twociphertextsY_1 and Y_2 are the 32-bitoutputfromthestandardpermutationP(C)after thesubstitutionboxes.

Figure 4showsasingleround ofDESencryption.LetΔXrepresentthedifferenceofthetwo known(and chosen)plaintextsX_1 andX_2:

$$\Delta X = X_1 \oplus X_2$$

where$X_1 \oplus X_2$ represents theaddition modulo-2bitbybitofthe2plaintextvectors. Inachosen plaintextattack, the twoplaintextsX_1 and X_2 are chosen such as to givea desired plaintext differenceΔX.

Since$\Delta X = X_1 \oplus X_2$ and$A=E(X)$issimplyanexpansion permutationoftheplaintextbits A,then thedifferenceΔAisalsoknown:

$$\begin{aligned} \Delta A &= A_1 \oplus A_2 \\ \Delta A &= E(X_1) \oplus E(X_2) \\ \Delta A &= E(\Delta X) \end{aligned}$$

AteachDESround,theunknown48-bitsubkey$K_i$isaddedtothe48-bitvectorAattheoutput oftheexpansion permutationbox:

$$B_1 = A_1 \oplus K_i \quad \text{and}$$
$$B_2 = A_2 \oplus K_i$$

Since the 48-bit subkey K_i issecret, the two48-bit vectors B_1 and B_2 are also unknown. However,their differenceΔBisknown!

$$\begin{aligned} \Delta B &= B_1 \oplus B_2 \\ \Delta B &= (A_1 \oplus K_i) \oplus (A_2 \oplus K_i) \\ \Delta B &= A_1 \oplus A_2 \\ \Delta B &= \Delta A \\ \Delta B &= E(\Delta X) \end{aligned}$$

So,bychosingtheplaintextsX_1 andX_2 andtherefore theirdifferenceΔX,onefindstheinputs tothe8substitutionboxesevenifthesubkeysareunknown.

Nowworkingbackward fromknownciphertextsY_1 andY_2 obtained fromtheencryption of the aboveplaintextsX_1 andX_2,wecanalsodetermine their differenceΔY[1]:

$$\Delta Y = Y_1 \oplus Y_2$$

BothY_1andY_2vectorsarepermutedversionsofthe32-bitoutputsC_1andC_2ofthesubstitution boxes:

$$Y_1 = P(C_1) \quad \text{and}$$
$$Y_2 = P(C_2)$$

or,expressingthesubstitutionboxesoutputsC_1 andC_2 asafunction oftheciphertextsY_1 andY_2:

$$C_1 = P^{-1}(Y_1) \quad \text{and}$$
$$C_2 = P^{-1}(Y_2)$$

Finally, thedifferenceattheoutputofthesubstitutionboxes$\Delta C$is:

$$\begin{aligned} \Delta C &= C_1 \oplus C_2 \\ \Delta C &= (P^{-1}(Y_1)) \oplus (P^{-1}(Y_2)) \\ \Delta C &= P^{-1}(\Delta Y) \end{aligned}$$

Differential cryptanalysiscompares the distributionofthe differenceΔXforaplaintextpair X_1 andX_2 withthedistributionoftheciphertextdifferenceΔYforthecorresponding ciphertext pairY_1 andY_2. Inachosenplaintext-ciphertextattack,theplaintextischosensuchastoprovide thedesireddifferenceΔX.ItexploitthefactthattheplaintextdifferencesΔXandtheciphertext differencesΔYarenotequallylikely.Somedifferencesinplaintextpairshaveahigherprobability ofcausingdifferenceinciphertextpairsthan others.

[1]Asweknow the actualciphertextsare obtainedbyadding Y to the previous (and known) contentsofthe left register.

For each ofthe 8DES substitutionboxes, wecan constructa table ofjointplaintextand ciphertextdifferences(seeTable9below)whereeachrowrepresentsagivenplaintextdifferenceΔX andeachcolumnrepresentagivenciphertextdifferenceΔY. The entry $p_{i,j}$ inTable 9represents the number ofoccurences thatagivenplaintextdifferenceΔX_ihasproduced agivenciphertext differenceΔY_j.

Table 9:Plaintextandciphertextdifferencesrelative frequencies.

	$\Delta Y_1 \cdots$		$\Delta Y_j \cdots$	
ΔX_1	$p_{1,1}$	$\cdots$	$p_{1,j}$	$\cdots$
$\vdots$	$\vdots$	$\ddots$	$\ddots$	$\ddots$
ΔX_i	$p_{i,1}$	$\cdots$	$p_{i,j}$	$\cdots$
$\vdots$	$\vdots$	$\ddots$	$\ddots$	$\ddots$

Biham and Shamir [BS93]havedemonstratedthatafull-fledged16-round DEScryptanalysis requires2^{47}chosenplaintext-ciphertext pairsor2^{55}knownplaintext-ciphertext pairswith2^{37}DES operations,thus makingthistypeattackonDESnotpractical yet.

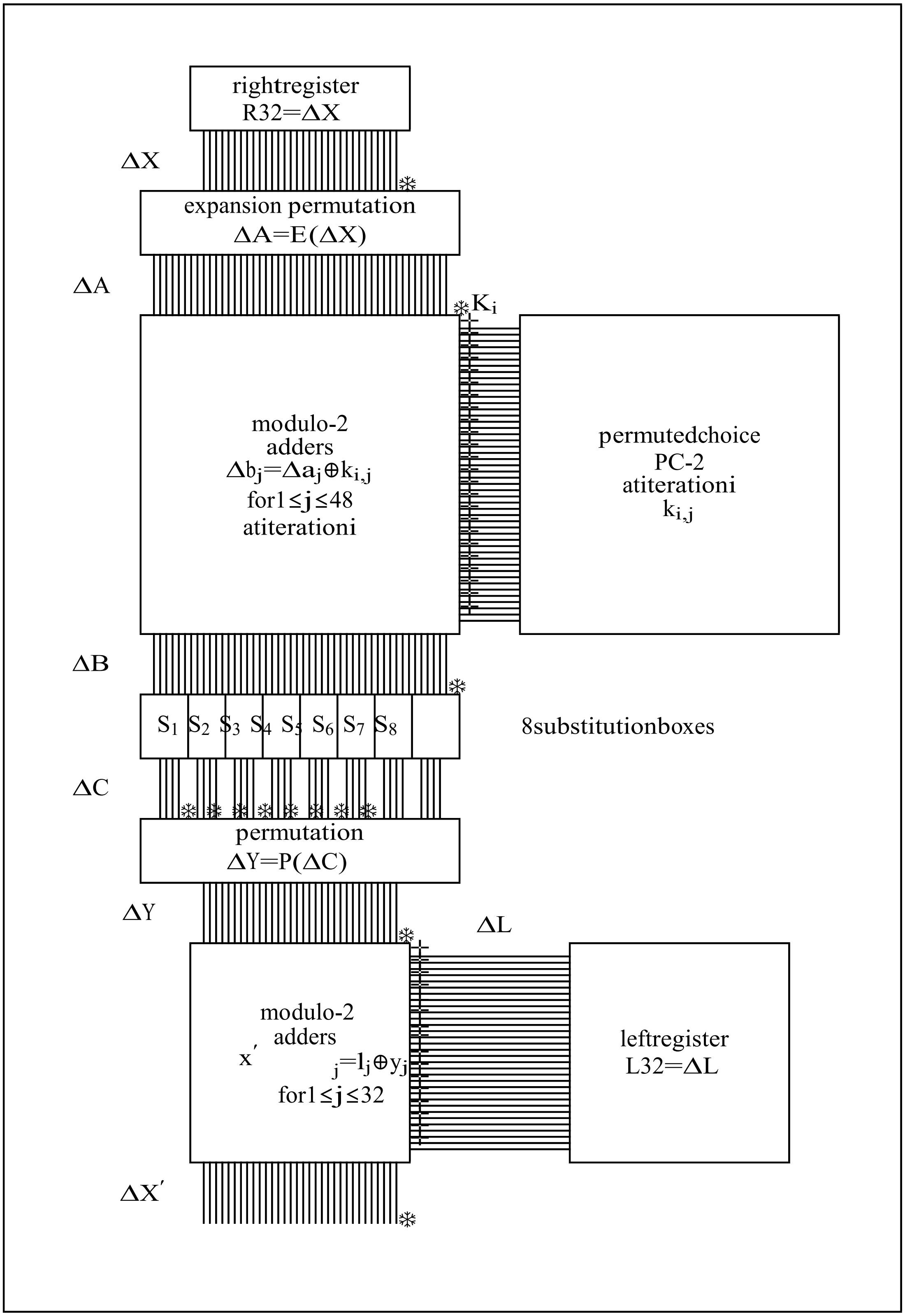

Figure 4:Differential cryptanalysisofasingleDESencryptionround.

6 – Results

1- Voice chat application using sockets.
2- Encoder and Decoder for the voice.
3- Encrypt and Decrypt between two end points.

7 – Future works

1- Implement a plug-in as an intermediate software between all voice chat application eg. Skype , Yahoo , etc..
2- Enhance our encryption and decryption algorithm to make a high level security layer.
3- Convert the voice chat application to work over VOIP.

8 - References

URLs

[1] http://www.nsa.gov/publications/publi00019.cfm#N4
[2] http://en.wikipedia.org/wiki/Voice_frequency
[3] http://www.nsa.gov/public/publi00007.cfm
[4] http://en.wikipedia.org/wiki/SIGSALY
[5] http://en.wikipedia.org/wiki/Scrambler
[6] http://seussbeta.tripod.com/crypt.html
[7] http://en.wikipedia.org/wiki/Vocoder
[8] http://ccrma.stanford.edu/courses/422/projects/WaveFormat
[9] http://www.staudio.de/kb/english/drivers/
[10] http://msdn2.microsoft.com/en-us/library/ms678518(VS.85).aspx
[11] http://clam.iua.upf.edu/
[12] http://www.openal.org/
[13] http://www.portaudio.com/
[14] http://en.wikipedia.org/wiki/Duplex_%28telecommunications%29
[15] http://en.wikipedia.org/wiki/Real_time