Strictly Confidential

Part two of an occasional cryptography series

by Julian Bucknall

Well, here I am about to walk upon England's mountains green, on a well-deserved vacation (even though I say so myself). Compared to the dry brown of Colorado, Yorkshire looks way too green. Lambs are bleating everywhere, the wind is sighing through the trees, the smells of cooking are escaping the kitchen. I'll let this April shower stop first before going hiking, and what better way to wait for the sun to break through than write part two of my cryptography series.

Manifesto

Back in February of this year, I talked about some elementary encryption methods in this column. These algorithms were all of historical importance only; the only ones used these days are ROT13 and XOR, the former being a Caesar cipher and the latter a type of Vigenère cipher. The encryption algorithms I introduced in that article were all what are known these days as *private key ciphers* (or *secret key ciphers*, or *symmetric ciphers*: there's nothing like having a whole bunch of different names to get us all confused!). We suppose that our familiar encryption protagonists, Alice and Bob, want to communicate securely without Eve being able to decipher their messages by eavesdropping. A private key cipher is an encryption method where Alice and Bob have previously agreed on a key and an algorithm using that key and can communicate by encrypting and decrypting messages to their hearts' content, leaving their adversary Eve, the eavesdropper, to use sophisticated cryptanalysis to try and break the encryption. A simple enough scenario, but it has one *big* problem.

If Alice and Bob can meet, they can exchange their secret key without fear that Eve can find it out. But what if Alice and Bob are in different countries, on opposite sides of the Earth? Then Alice would have to send the secret key to Bob by mailing it, by giving it to a (supposedly) trustworthy courier, and so on. If Eve were able to intercept the key between Alice and Bob, she has the opportunity to perform the '(wo)man in the middle' attack. She sends *her* secret key to Bob instead of *Alice's* key, and from then on she's in control. She intercepts encrypted messages from Alice, decrypts them and then encrypts them (or a changed message) with *her* secret key and sends it on to Bob. Messages from Bob can be decrypted and then altered and re-encrypted with Alice's key and sent on. Alice and Bob have no idea what's going on.

Nevertheless, private key ciphers are still extremely important, providing we get round the key-exchange problem. We'll discuss how to do that in a further instalment of this encryption series, but for now we'll continue looking at private key ciphers.

Before going any further, I must draw your attention to an important point. The security of any encryption process should *not* depend on the secrecy of the algorithm, it should depend solely on the secrecy of the key or keys. Secret algorithms are generally insecure: be very wary of any products or software that employ a secret algorithm 'developed in-house by our own analysts'. The field of cryptography has, over the past few years, developed some extremely secure algorithms and has broken some extremely dubious ones. The well-known algorithms (DES, Blowfish, RSA, RC5) are all secure: not because people didn't know how they worked, but because academics and researchers have tried to break them and failed. Their security comes from this intense analysis, not from hiding the internals of the algorithm concerned. Of course, this doesn't mean that someone isn't going to devise some new mathematics that breaks DES tomorrow, but so far the important algorithms remain unbroken (unless you count a brute force attack). A secret algorithm, on the other hand, could be broken quite easily: it hasn't been subjected to peer analysis and so is a complete unknown. It *may* be secure, but equally well it may be insecure and you may never suspect it. And, by the way, just as in programming circles, the author of an algorithm is usually the worst person to try and find the bugs in it.

For Your Pleasure

In this article I would like to continue talking about private key ciphers by introducing possibly the most famous one of them all: the *Data Encryption Standard* (DES).

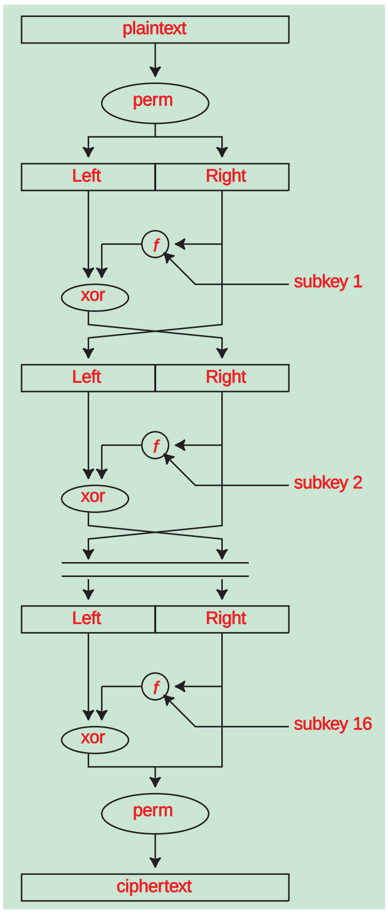
DES has a funny history and illustrates Americans' built-in paranoia about their Government. Back in the 70s, the National Standards Bureau (NSB, nowadays the National Institute of Standards and Technology, NIST) wanted to create a secure encryption algorithm for the Government's secret documents. It published an invitation for academics or companies to put forward a proposal. Now, in those days, cryptography was a new science, in its infancy. It was, to put it mildly, in disarray. No techniques had really been designed for cryptography or cryptanalysis. IBM had been funding an internal project called Lucifer that was an encryption algorithm, but they had no idea whether it was secure or not. Nevertheless, they put it forward. Now comes the paranoia part. The NSB sent it to the National Security Agency (NSA), a secretive government agency responsible for message and radio interceptions around the globe (the nearest equivalent in the UK is GCHQ). They returned the algorithm with some alterations to what are known as the S-boxes, saying that the changes would make the algorithm more secure. The paranoia part? People said that the NSA had managed to install a *trapdoor* into the algorithm (a trapdoor is a deliberate weakness to allow those who know about it to easily decrypt messages). The NSA also reduced the key length from IBM's 112 bits to 56. A lot of research went into Lucifer and the NSA changes, and eventually the algorithm was accepted in 1977 and renamed DES. Even since then, a lot of research has been done to try and prove or disprove the presence of a trapdoor, but none has been found and DES has been re-certified every 5 years from the original acceptance date.

NIST is now in the process of tendering for a new US standard for encryption, the *Advanced Encryption Standard* (AES), the cipher that is supposed to last for the next 30 or 40 years. DES is now too insecure for top-secret use because the speed of computing hardware has grown so fast and so large that brute force attacks are now viable.

DES is a block cipher: one that works on a block of bits at a time. The plaintext message is divided up into 64-bit blocks, with the last block being padded to 64 bits in some way, and each block is encrypted. The key for DES is a 56-bit key, although it is usually expressed as a 64-bit key (8 bytes) of which every eighth bit is ignored.

Since the key length is so short, it is extremely amenable to a brute force attack. To break DES, you would need to try about 2^{55} (3.6 * 10^{16}) keys on average. If you could try five million keys a second (just about possible on a modern Pentium III with a highly optimized implementation), it would take about 230 years to break DES. However, in 1997, the DESCHALL group actually broke a DES-encrypted message in about 4 months by harnessing idle machines connected to the internet. Each machine would try a small subset of the possible keys, and because there were so many machines participating (a peak of 14,000 machines was reached), at one point they were testing 7 billion keys per second. In 1993 a cryptographic researcher called Michael Wiener devised a machine for testing DES keys. He estimated that if you spent about $10,000 and built specialized hardware for decrypting DES, you could break a DES-encrypted message in about 2.5 days, testing at a rate of 1.7 * 10^{11} keys per second (170 billion keys per second). If you wanted to spend one million dollars he estimated that you could break a DES encrypted message in about 35 minutes. Whether anyone has ever built such a machine is unknown.

➤ *Figure 1: DES flowchart.*



Do The Strand

Anyway, let's take a look at how a 'modern' encryption algorithm works and describe the DES cipher.

As I stated before, DES works on blocks of 64 bits, or 8 bytes, at a time. The block is put through the DES mangle (possibly the best way to describe it!) and out pops an encrypted 64-bit block. We continue like this until the entire message has been encrypted. For each block we permute it a little and then split it into two 32-bit halves, known as the left and right halves. Now comes the fun part: we push the two halves through 16 rounds of a peculiar function (known unimaginatively as function *f*) that combines the right half and part of the DES key, spitting out a 32-bit value that's then XORed with the left half. This new value becomes the new right half and the old right half becomes the new left half. After 16 loops of this shuffling, the two halves are joined together and the inverse of the original permutation is performed. The result is the encrypted version of the original 64-bit plaintext. Phew!

Figure 1 shows a stylized rendition of this transformation. We start out with the 64-bit plaintext at the top, push it through the initial permutation, and then split it into two. Then we have 16 similar operations. For each operation we select a different part (subkey) of the DES key (here shown as subkey 1, subkey 2, etc), feed it into the function *f* with the right half and XOR the result with the left half. The figure then shows the switching of left and right halves.

It looks complicated enough without going into what happens inside function *f*.

The amazing thing is that to decrypt you *follow exactly the same process*, with a single change: the subkeys are fed into the process in reverse order.

Before we delve into function *f*, we can start getting our feet wet by writing some code. This will help us solidify what we've learned so far. Note that the code I'm going to be writing is not going to be the most efficient DES implementation ever written: my objective is not to dazzle you with my brilliant code writing skills but to illustrate the algorithm. You are welcome to take my code and optimize it or rewrite it in assembler.

The first thing we have to do is to play around with the key. We have to select the 56 bits we are going to use out of the 64 bits the key is usually expressed as. This is, how can I put it, not as simple as you might expect. The DES specification provides us with a table that tells us which bits go where. Table 1 shows the bit permutation we have to do. For example, bit 0 of the DES key we use in the encryption is equal to bit 56 of the input key, bit 1 is equal to bit 48, etcetera, for all 56 bits. To make things easier for us we'll create a `Boolean` array to hold the 56 bits of the key: it'll help in a moment when we create the subkeys. Listing 1 has the routine that generates the 56-element `Boolean` array. (A note to the interested reader: the bits in each byte are numbered from the most significant, bit 0, to the least significant, bit 7. This is the opposite way that many people number bits where bit 0 is the *least* significant.)

Now we need to know how to create the 16 subkeys we'll be feeding into the mysterious function *f*. For each subkey, we select 48 bits from the derived 56-bit key. The process, as with everything to do with DES, is complex. We split the key into two 28-bit halves. Each half is then rotated left by one or two bits, depending on the round, with the bits falling off the left end being fed into the right end (it's a circular left shift, in other words). The number of steps to rotate the key at each round is given by the special table shown in Table 2. We then select the 48 required bits for the subkey (yet *another* special table defines this, Table 3). At this point we might as well gather the 48 bits together into an array of 6 bytes: it'll make it easier to feed into the function *f*. For decryption, we do the same process, except the subkeys are stored in reverse order. Listing 2 shows this set of transformations that generates the 16 subkeys; notice that we pass in a `Boolean` that defines whether we are going to encrypt or decrypt.

At this point we have initialized the DES state and are ready to begin the actual encryption.

We divide up the plaintext into 64-bit, or 8-byte, chunks. For each chunk we perform the same process. First, we permute the bits in the chunk. Believe it or not, this is also defined by a special table

➤ *Table 1: Selecting 56 bits for a DES key from the initial 64.*

56	48	40	32	24	16	8	0
57	49	41	33	25	17	9	1
58	50	42	34	26	18	10	2
59	51	43	35	62	54	46	38
30	22	14	6	61	53	45	37
29	21	13	5	60	52	44	36
28	20	12	4	27	19	11	3

➤ *Table 2: Number of bits to shift the DES key per round.*

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

➤ *Listing 1: Extracting the 56 bits from the DES key.*

```
procedure ConvertDesKey(const aDesKey64 : TaaDesKey64;
  var aKey56 : TDesKeyArray);
var
  i : integer;
  ByteNum : integer;
  BitNum  : integer;
begin
  for i := 0 to 55 do begin   {for each outbit bit...}
    {work out which byte in the input key and which bit within
     that byte that the output bit will be found}
    ByteNum := DesKeyBitSelection[i] div 8;
    BitNum := DesKeyBitSelection[i] mod 8;
    {set the output boolean equal to this bit}
    aKey56[i] := (aDesKey64[ByteNum] and BitMask[BitNum]) <> 0;
  end;
end;
```

➤ *Listing 2: Generating the 16 subkeys.*

```
procedure CalcSubKeys(const aKey56 : TDesKeyArray;
  var aSubKeys : TSubKeyArray; aForEncrypt : boolean);
var
  i         : integer;
  LeftInx   : integer;
  ToInx     : integer;
  Round     : integer;
  Temp56    : TDesKeyArray;
  SubKey    : TSubKey;
  ToByte    : integer;
  Accum     : byte;
  CurBit    : byte;
  TotalRotation : integer;
begin
  {calculate the subkeys for all 16 rounds...}
  for Round := 1 to 16 do begin
    {calculate the total rotation required for this round}
    TotalRotation := 0;
    for i := 1 to Round do
      inc(TotalRotation, DesSubKeyShifts[i]);
    {left rotate the two halves of the 56-bit key by the
     required shift for this round}
    for LeftInx := 0 to 27 do begin
      {calculate the next destination index}
      ToInx := LeftInx - TotalRotation;
      if (ToInx < 0) then {we have wrapped}
        inc(ToInx, 28);
      {move the bit from the original key to our temp key,
       first for the left half and then for the right half}
      Temp56[ToInx] := aKey56[LeftInx];
      Temp56[ToInx+28] := aKey56[LeftInx+28];
    end;
    {now calculate this round's subkey by selecting the
      correct bits}
    ToByte := 0;
    Accum := 0;
    CurBit := $80;
    for i := 0 to 47 do begin
      if Temp56[DesSubKeyPerm[i]] then
        Accum := Accum or CurBit;
      CurBit := CurBit shr 1;
      if (CurBit = 0) then begin
        SubKey[ToByte] := Accum;
        inc(ToByte);
        Accum := 0;
        CurBit := $80;
      end;
    end;
    {save the subkey
     note: for decryption we save subkeys in reverse order}
    if aForEncrypt then
      aSubKeys[Round] := SubKey
    else
      aSubKeys[17-Round] := SubKey;
  end;
end;
```

(Table 4)! Now we get into the fun stuff. We split the plaintext chunk into two halves and start doing the rounds. The right half and the correct subkey are fed into the mysterious function *f*, which returns a 32-bit value. This is XORed with the left half and becomes the new right half. The new left half is set equal to the old right half. We do this operation 16 times.

Finally, we apply the initial permutation, but in reverse, to the bits of the result. No new table this time, it's embodied in Table 4. Listing 3 shows the code to perform all this data mangling. Notice that it uses a separate routine to perform the permutations: Listing 4 has the details of this procedure. It's now time for you and I to get a quick cuppa before we go on to... drum roll, fanfare... function *f*.

13	16	10	23	0	4	2	27	14	5	20	9	22	18	11	3
25	7	15	6	26	19	12	1	40	51	30	36	46	54	29	39
50	44	32	47	43	48	38	55	33	52	45	41	49	35	28	31

➤ *Table 3: Selecting the 48 bits for a subkey.*

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
56	48	40	32	24	16	8	0	58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4	62	54	46	38	30	22	14	6

➤ *Table 4: Initial permutation of the input data.*

Re-Make Re-Model

Welcome back. Function *f*, then. First, I shall describe how it works in principle, and then we'll take a look at how to implement it more efficiently.

As a reminder, function *f* takes a 32-bit block of data (the right half of a 64-bit data block), and a 48-bit subkey, and then, switching the blender to the maximum speed setting, mixes them to produce a 32-bit encrypted block. The blending is performed by the use of a series of bizarre transformations called E-boxes, S-boxes, and P-boxes (as far as I know they're called boxes because they're represented by rectangles in the flow charts for the algorithm).

The first step is the E-box. This is an expansion permutation of the input 32-bit block to produce 48 bits. The E-box has two main purposes. The first is to expand the data block by duplicating bits so that it's the same size as the subkey, ready for an inevitable XOR operation. The second is more important, perhaps. DES is designed so that the value of each bit of ciphertext depends on every bit of plaintext and key; a thorough mixing, in other words. By duplicating certain bits, output bits get to be made up of more input bits more quickly.

The E-box is, surprise, surprise, defined by a table, Table 5. If you look carefully you can see that we are duplicating bits 4, 5, 8, 9, 12, 13, 16, 17, etc, to create the 16 missing bits. We then XOR this expanded data block with the subkey for this round.

Well, that wasn't too bad. Now comes the real fun part (evil laugh). We now perform a substitution operation using the eight S-boxes (S standing for *substitution*). Split the 48-bit result from the XOR operation into eight 6-bit blocks. Each 6-bit block will use its own S-box. Table 6, no groaning at

➤ *Listing 3: The high-level DES implementation.*

```
procedure TaaDesEngine.ProcessBlock(const aSrc : TaaDesBlock;
  var aDest: TaaDesBlock);
var
  Round : integer;
  Temp  : longint;
  Block : packed record
    Left  : longint;
    Right : longint;
  end;
begin
  {perform the initial permutation of the source block}
  CalcPermutation(aSrc, Block, DesStartPerm, sizeof(DesStartPerm));
  for Round := 1 to 16 do begin
    Temp := Block.Right;
    Block.Right := Block.Left xor F(Block.Right, PSubKeyArray(FState)^[Round]);
    Block.Left := Temp;
  end;
  {this will have done one too many swaps of the right and
   left halves, so swap them back}
  Temp := Block.Right;
  Block.Right := Block.Left;
  Block.Left := Temp;
  {perform the final permutation of the source block}
  CalcPermutation(Block, aDest, DesFinalPerm, sizeof(DesFinalPerm));
end;
```

➤ *Listing 4: Permuting a set of bits according to a mapping.*

```
procedure CalcPermutation(const aSource; var aDest;
  const aMapping; aMapCount : integer);
var
  Src   : TByteArray absolute aSource;
  Dest  : TByteArray absolute aDest;
  Map   : TByteArray absolute aMapping;
  i     : integer;
  FromByte : integer;
  FromBit  : integer;
  ToByte   : integer;
  Accum    : byte;
  CurBit   : byte;
begin
  {using mapping, transfer bits from source to destination}
  ToByte := 0;
  Accum := 0;
  CurBit := $80;
  for i := 0 to pred(aMapCount) do begin
    FromByte := Map[i] div 8;
    FromBit := Map[i] mod 8;
    if ((Src[FromByte] and BitMask[FromBit]) <> 0) then
      Accum := Accum or CurBit;
    CurBit := CurBit shr 1;
    if (CurBit = 0) then begin
      Dest[ToByte] := Accum;
      inc(ToByte);
      Accum := 0;
      CurBit := $80;
    end;
  end;
end;
```

the back, please, defines the S-box for the first 6-bit block. The way it is used goes like this: take the first and last bits from the 6-bit block. Combine them to form a 2-digit binary number and this number defines a row in the S-box table. So for the input block 101010, the row number is binary 10 (or 2 decimal), so we should look at the third row (the first row being row 0, of course). Now we take the middle four bits as a binary number and then look at that element along the row we've selected. For 101010, the middle four bits form binary 0101 (or 5 decimal), so we look at the sixth element of the second row (elements are counted from zero as well). We get the value 2. We convert this into a 4-digit binary number, 0010, and output that as the substitution value for the original 6 bits.

We repeat that for the other seven 6-bit blocks. As I said, each of these has its own S-box: it's OK, I'm not going to show all the tables, and eventually we'll end up with eight 4-bit blocks which we join together to make, fanfare, a 32-bit block. It is through the S-boxes that DES gets its power and inscrutability. Up until now the transformations we've been describing have all been simple stuff indeed and are just there for mixing's sake. With the S-boxes we arrive at some decidedly bizarre and hard-to-analyze operations. Essentially, DES is unbreakable because of the S-boxes.

31	0	1	2	3	4	3	4	5	6	7	8
7	8	9	10	11	12	11	12	13	14	15	16
15	16	17	18	19	20	19	20	21	22	23	24
23	24	25	26	27	28	27	28	29	30	31	0

➤ *Table 5: The E-box expansion permutation.*

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

➤ *Table 6: The first S-box.*

15	6	19	20	28	11	27	16	0	14	22	25	4	17	30	9
1	7	23	13	31	26	2	8	18	12	29	5	21	10	3	24

➤ *Table 7: P-box permutation.*

Unfortunately, we're not quite finished yet. There is one last permutation to do, the P-box, which shuffles all the bits in the result from the S-boxes. Table 7, the last one I promise, shows this final permutation. The result that comes out of the P-box is the result of function *f*.

Both Ends Burning

So, to recap, for every 64-bit block we push into the DES algorithm, we permute the bits, we split that into two and perform 16 rounds of function *f* with the right half and an XOR with the left. For each call to function *f* we expand the input 32-bits to 48, XOR it with the correct subkey for that round, split the answer into eight 6-bit blocks, each of which are fed through an S-box, combined to a 32-bit answer that is then shuffled. The miracle is not that we can encrypt a message with this little lot, but that we can decrypt it afterwards!

I mentioned at the beginning of the discussion about function *f* that we would try and make it a little more efficient when we implement it. Take the S-boxes, for example. Although the algorithm is pretty clear about what to do, if we actually did it in that way, we'd end up with the world's slowest DES implementation. What we do instead is to rearrange the S-box tables so that they become arrays indexed by 6-bit values (a 32-element array). For our example, 101010, we would set element 42 to 2.

There are other things we could do as well, but we'll leave all that to the commercial products and just make the simple S-box change I described above. (If you own

➤ *Listing 5: The magic internal function f.*

```
function F(const aRight: longint; const aSubKey: TSubKey):
  longint;
var
  i        : integer;
  BigRight : T48Bits;
  Accum    : byte;
  Temp32   : T32Bits;
begin
  {start off with the expansion permutation: the E-Box}
  CalcPermutation(aRight, BigRight, DesEBoxPerm,
    sizeof(DesEBoxPerm));
  {XOR the subkey into the expanded data}
  for i := 0 to 5 do
    BigRight[i] := BigRight[i] xor aSubKey[i];
  {now wade into the S-Boxes}
  {..first}
  Accum := (BigRight[0] and $FC) shr 2;
  Temp32[0] := DesSBox1[Accum] shl 4;
  {..second}
  Accum := ((BigRight[0] and $03) shl 4) or
           ((BigRight[1] and $F0) shr 4);
  Temp32[0] := Temp32[0] or DesSBox2[Accum];
  {..third}
  Accum := ((BigRight[1] and $0F) shl 2) or
           ((BigRight[2] and $C0) shr 6);
  Temp32[1] := DesSBox3[Accum] shl 4;
  {..fourth}
  Accum := (BigRight[2] and $3F);
  Temp32[1] := Temp32[1] or DesSBox4[Accum];
  {..fifth}
  Accum := (BigRight[3] and $FC) shr 2;
  Temp32[2] := DesSBox5[Accum] shl 4;
  {..sixth}
  Accum := ((BigRight[3] and $03) shl 4) or
           ((BigRight[4] and $F0) shr 4);
  Temp32[2] := Temp32[2] or DesSBox6[Accum];
  {..seventh}
  Accum := ((BigRight[4] and $0F) shl 2) or
           ((BigRight[5] and $C0) shr 6);
  Temp32[3] := DesSBox7[Accum] shl 4;
  {..eighth}
  Accum := (BigRight[5] and $3F);
  Temp32[3] := Temp32[3] or DesSBox8[Accum];
  {end up with the final permutation: the P-Box}
  CalcPermutation(Temp32, Result, DesPBoxPerm,
    sizeof(DesPBoxPerm));
end;
```

Advanced Cryptography by Bruce Schneier, you can find the source to DES that does perform all this messing around. Although I'm somewhat proficient in C, I find that reading source that follows some algorithm using a completely different method with no comments is a fast way to get frustrated.) Listing 5 shows the final function *f* that performs all the fiddling around.

The Thrill Of It All

Having seen the DES algorithm, let's discuss some of its pros and cons.

The first problem is the one I've glossed over repeatedly and that some of you may have noticed. DES only works on messages that are a whole number of 64-bit blocks in size. The plaintext must be an integral multiple of 8 bytes in size. Only one in eight possible plaintexts is so obliging on average, the other seven are inconsiderate and have a final block that is too small. What can we do? We could certainly pad the last block with zero bytes (or $FF bytes, or anything in between) to make it exactly 8 bytes in length, but that leaves a problem for the decryption part: how does it know how big the original plaintext was? If we're not careful, we'll end up with a decrypted plaintext that's longer than the original. For a text message, that's not too bad, but for a zipfile or a database table, it's a disaster.

One thing we could do is to add the length of the original plaintext to the plaintext in some fashion and encrypt it along with the plaintext. There are several choices here. The first might be to attach the length (as a long integer) as a suffix to the plaintext and then start encrypting. This might be an attack point, though: in practice we can deduce the value of the `longint` that's been added at the start by looking at the length of the ciphertext. This gives us a good start for the attack. A better way might be to append the `longint` length to the plaintext (in practice we would make the length the last four bytes of the last 8-byte block, so we know where to find it). This could lead to a similar attack, though.

A better answer is to recognize that we don't need to store the *whole* length of the plaintext, we only need to store the length of the last, short, block. This is going to be a value from 0 to 7. If the plaintext length is exactly divisible by eight, we add a complete new block to the plaintext and set the last byte of it to zero; for all other lengths we can pad out the last block to the full eight bytes, making the last byte equal to the number of valid bytes in the final block. The padding bytes can be any value we like: zero, $FF, or whatever, but, since we throw them away anyway, set them to bytes output from a random number generator. To be really paranoid, since the value of the last byte is contained within a mere 3 bits, set the other 5 bits to some random value too.

A much better alternative to this scheme is to use a method called *ciphertext stealing*. In this method we use the length of the plaintext itself to tell us what to do; we don't need to do any padding and the ciphertext turns out to always be the same length as the plaintext. If the length of the plaintext is a multiple of eight, then we simply encrypt all the 64-bit blocks and we're done. Otherwise, the plaintext has an extra 1 to 7 bytes that we need to encrypt. We encrypt the last-but-one block as normal. Steal enough bytes from this encrypted block to pad out the final plaintext block to 8 bytes. This leaves from 1 to 7 encrypted bytes; we'll use them in a moment. Encrypt the padded final block. Now we switch the ciphertext blocks, outputting the final 8-byte block first, followed by the remaining 1 to 7 encrypted bytes. The ciphertext ends up the same length as the plaintext. On decryption, we will decrypt the final full ciphertext block. This gives us the final partial plaintext, plus the borrowed encrypted bytes. We append these bytes to the final partial ciphertext and decrypt them. This will give us the final full plaintext block, which comes before the partial plaintext.

That's a bit complicated, so let's take an example. Suppose the final nine bytes of the plaintext were 'abcedfghi'. We encrypt the first eight to produce 'ABCDEFGH'. Borrow the last seven bytes of this, saving the 'A', and add to the final 'i' to make 'iBCDEFGH' and encrypt to make 'STUVWXYZ'. We output this as nine bytes: 'STUVWXYZA'. On decryption, we reverse the process: decrypt 'STUVWXYZ' to make 'iBCEDFGH'. Save the 'i' and return the borrowed encrypted bytes to the final byte of ciphertext to make 'ABCDEFGH'. Decrypt to make 'abcdefgh', append the saved 'i' and output.

When used in practice, DES has several well-defined *modes*. The one I've discussed is known as *electronic codebook* mode (ECB). With ECB we simply encrypt and decrypt each and every 64-bit block in the same manner. If we have two plaintext blocks that are the same in the original message, they will produce the same ciphertext. (We can view the algorithm as a huge codebook with every possible variation of 8 bytes having its own code.) This is a point of attack for the cryptanalyst. On the plus side, it also allows for parallel implementations: since each block is processed independently of all the others, several blocks can be encrypted at the same time on a multiprocessor machine. Another positive spin is that if there is a transmission error and a few bits get garbled, it doesn't affect the entire decryption, only a mere 8 bytes.

For not very much more coding effort, we can use *cipher block chaining* mode (CBC). With CBC we XOR each plaintext block with the previous ciphertext block before encrypting it with the standard DES. This way equal plaintext blocks do *not* produce the same ciphertext blocks. The CBC process produces a cascading or avalanche effect with the encryption of a plaintext block depending on every previous ciphertext block. CBC mode helps protect against certain attacks, and is deemed to be a more secure implementation.

In practice, CBC mode is the most widely accepted mode to use with DES, ECB being a less secure mode.

Triptych

Next, it is important to realize that the security of DES relies on the keys. The algorithm itself has been subjected to many, many attacks and tests and has passed them all for standard usage. (There are a couple of attacks that enable you to find the key providing that you are allowed to encrypt about 250 known plaintexts and get the ciphertexts thus produced. In practice, Alice and Bob are not likely to allow Eve the opportunity to encrypt 250 separate messages with their private key!) There are, however, a handful of keys that you must not use. Four of them are known as *weak keys*: if you encrypt plaintext twice with one of these keys, you get the original plaintext. These keys are fairly obvious to spot: internally they correspond to 56-bit keys with all bits set, all bits clear, the first 28 bits set and the last clear, or vice versa. Twelve are known as *semi-weak keys*. These are used in pairs: you encrypt plaintext with the first key in the pair, and then encrypting the ciphertext so produced with the second key produces the initial plaintext.

However, it's not usually worth worrying about these weak and semi-weak keys in practice. There is a universe of 2^{56} possible keys for DES (72 quadrillion), and so the possibility of choosing a 'bad' key is small indeed. Generating a DES key with some kind of random number generator is pretty safe; better might be to use a message digest algorithm on a password.

Because DES is based on a key containing a mere 56 bits (approaching the point at which brute force attacks are viable) there have been proposals to make DES more secure through increasing the key length. One plan is to provide a key that is directly split into the sixteen 48-bit subkeys. In other words, have a key that is 768 bits in length, the first 48 bits of which is used in the first round, the next 48 bits in the second round, and so on. Although this sounds great in theory to counter the brute force attack (if we used every PC in the world, say 100 million PCs, and assumed they could all test 100 million keys per second, it would take longer to find a key than is left before the Universe is estimated to reach its final demise), it does not make DES any less vulnerable to sophisticated mathematical attacks such as differential cryptanalysis.

The most popular DES extension is called triple-DES. With triple-DES we effectively encrypt plaintext three times. There are several different ways of doing this. The first one is the most obvious and is known as DES-EEE3. We select three separate 56-bit keys, encrypt the plaintext with the first key, encrypt the resulting ciphertext with the second, and then encrypt the ciphertext from this second encryption with the third. To decrypt, we decrypt with the third key, then the second and the first.

The second triple-DES method is a little more peculiar and is known as DES-EDE3. Again we have three DES keys. We encrypt the plaintext with the first key, *decrypt* the resulting ciphertext with the second and finally encrypt the mish-mashed result of that with the third key. To decrypt, we decrypt with the third, encrypt with the second and decrypt with the first.

There are two further common triple-DES implementations called DES-EEE2 and DES-EDE2. These are implemented exactly as their similarly named brethren, except they use just two keys. The first and third encryptions are both done with the first key, the second key being used merely for the second operation.

As you have probably guessed, triple-DES is three times as slow as standard DES.

Well, if you made it this far, congratulations on getting through such a long, intense article. DES is not easy to understand (if it was, it probably would not be so secure). It's an important encryption algorithm, to be sure, but admittedly wasn't designed for modern 32-bit processors, as can be seen from the clunky code. Since DES became a standard, there have been better private-key algorithms properly designed for 32-bit processors, and part of the new AES standard is to make sure that the encryption algorithm for the next 30 to 40 years can be implemented efficiently on 32-bit and 64-bit processors.

On the disk I've supplied a DES engine class that can encrypt and decrypt streams (some of the listings show methods of this class). It can use either ECB or CBC modes. I leave it as an easy exercise for the reader to implement a triple-DES engine class using this class.

As for me, I'm off to hike Swaledale with my wife before dinner. See you next time.

References:

Advanced Cryptography by Bruce Schneier.

RSA Laboratories' Frequently Asked Questions About Today's Cryptography by RSA Data Security, Inc.

A Brute-force Search of DES Keyspace, available at: www.usenix.org/publications/login/1998-5/curtin.html

Julian Bucknall likes going to the cinema, not necessarily for the music. He can be reached at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.

© Julian M Bucknall, 2000