Technical Report

UCAM-CL-TR-120
ISSN 1476-2986

Number 120

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Block encryption

D. Wheeler

November 1987

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Block Encryption

David Wheeler
Computer Laboratory
Cambridge University
20 November 1987

Introduction

A fast simple way of encrypting computer data is needed. The UNIX crypt is a good way of doing this although the method is not cryptographically sound for text. The method suggested here is applied to larger blocks than the DES method which uses 64 bit blocks, so that the speed of encyphering is reasonable. The algorithm is designed for software rather than hardware. This forgoes two advantages of the crypt algorithm, namely that each character can be encoded and decoded independently of other characters and that the identical process is used both for encryption and decryption. However this method is better for coding blocks directly.

Method basis.

The method uses table look up as a non linear operation to mix the bits of a byte and an operation such as exclusive or for scrambling the bytes of the block. We describe the method for a block of say 1000 characters each of eight bits. The non linear operation is that of replacing a character by a new character such that the operation is reversible. That is the table gives a permutation of the characters.

If the look up table is perm[0] to perm[255] and the block to be translated is ch[0] to ch[999], then the basic operation is

```
for n=0 to 999 do
    ch[n]=ch[n] XOR perm[ch[n-1]+ch[999-n]]
```

ch[-1] is defined by some rule
and where the argument of perm is modulo 256 which may be done by a mask or byte operation.
XOR is the exclusive or operation.
The statement is in psuedo C.

The operation should be done at least three times. After the first time the translation of the nth character depends on the 0th to n-1th and the 999th to the 1000-nth characters in a weak way. The extra two iterations strengthen the diffusion.

The decoding is done by simply reversing the order of application of the formula. Each step is reversible as long as the block has an even number of characters.

The translation of element 500 depends only on element 499 as the argument of perm is ch[499]+ch[499] in the first iteration. However element 499 at this point depends on elements 0 to 498 and 999 to 500, and the following two iterations will cover up this slight weakness.

The weakness may possibly be exploited by an opponent who can arrange for a large number of blocks to be encrypted, with the central bytes taking all values while the remainder of the blocks were kept the same. If this attack were possible, the system using this encryption is probably too weak for good security.

Complete encoding

To complete the method we need some way of deriving a permutation from a key. This is straightforward but requires some care. The key should be long enough, and preferably the permutation generated should not have any special properties. It should not be possible to derive the key from the permutation. The following method seems to have most of the required properties. The indentation gives the loop structure of the program.

```
Let the key be key[0] to key[15]

 // set initial permutation

 for n= 0 to 255 do
    x=(key[0]+n) XOR key[1]
    x=(key[2]+x) XOR key[3]
    perm[n XOR key[4]]=x

 // randomise the permutation

 perm[256]=perm[0]
 for n=0 to 255 do
    perm[n]=perm[x]
    perm[x]=perm[n+1]
    x=perm[(x+key[n&15])&255]
    key[n&15]=x
```

The last cycle should be repeated once so that it is difficult to derive the initial key from the final permutation.

This is a random number generator which generates a permutation. It is relatively fast and uses all the digits of the key. A single bit change of the key will generate a completely different permutation. If the key is all zero, nevertheless the permutation will seem reasonably random. The cycle is relatively fast so that new permutations can be generated easily. For example, if each block had its own "salt" then the generation time could be ignored for blocks much larger than 256.

Because the reversibility is derived from the exclusive or operation rather than the inverse permutation, the permutation may be replaced by a simple table of random numbers. It seems that the permutation may be stronger as it gives a more uniformly distributed result so that frequency analysis techniques are made more difficult. However the permutation generation loop uses perm even though it has one element repeated and is not quite a true permutation until the last cycle.

Method strengthening

It is very easy to use three separate permutation tables rather than a single one. This only changes the starting overhead and not the coding or decoding rate.

If we permute the characters before we encode, this slightly slows the process, but will probably make attacks using common words harder, although it seems difficult to attack this way now.

If we change the permutation tables from file to file, rather than just the start character ch[-1] then the coding is slower but presumably harder to attack.

It is certain that increasing the number of iterations from three to say six would increase the protection against some form of attack, but it is not clear that it is necessary. The usual attacks are

```
use chosen text
use known text
use guessed strings
use the frequency characteristics, single and multiple
use unexpected coincidences
try all keys
use algebaic properties of the keys.
```

The most obvious attack is to use given text in an endeavour to find the perm value. The centre characters are the weakest and with three iterations one may find combinations of the middle five or so characters which leave the rest of the block unchanged. The relations thus discovered may then lead to decoding. This attack can be circumvented merely by increasing the number of iterations. For each iteration the search space is increased by a factor of 2**16.

Suggested Algorithm Set

Use key[0 to 15], perm[0 to 256], ch[0 to N-1], and x all of type "character" or "byte".

```
define generate(key,perm) as

 //  generate initial permutation from about 2**36 possibilities

for n=0 to 255 do
   x=(n+key[0]) XOR key[1]
   x=(x+key[2]) XOR key[3]
   perm[n XOR key[4]]=x

 // randomise the start permutation

for t=0 to 1 do
   perm[256]=perm[0]
   for n=0 to 255 do
       perm[n]=perm[x]
       perm[x]=perm[n+1]
       x=perm[ (x+key[n&15])&255 ]
       key[n&15]=x
```

It is thought to be difficult to find the given key from the permutation and the permutation from the final key.

The given key is changed, so a copy may be needed.

```
define encode(ch,N,perm) as

if N is odd or less than 4 then error

 for t=0 to 2 do
    ch[0]  = ch[0] XOR perm[ (ch[N-1]+ch[N-1])&255 ]
    for n=1 to N-1 do
        ch[n]  = ch[n] XOR perm[ (ch[n-1]+ch[N-1-n])&255 ]


define decode(ch,N,perm) as


if N is odd or less than 4 then error

 for t=0 to 2 do
    for n=N-1 to 1 by -1 do
        ch[n]  = ch[n] XOR perm[ (ch[n-1]+ch[N-1-n])&255 ]
    ch[0]  = ch[0] XOR perm[ (ch[N-1]+ch[N-1])&255 ]
```

Notes

Some of the masking operations may not be needed if the type "byte" is used. Some of the effort in generating the permutation may be redundant but for a large block the percentage of time will be small. The second iteration is both a random number generator and using the numbers generated x, forces 256 random swaps. The random numbers use the current permutation table in their generation process. By repeating the process once, it becomes very much more difficult to go from the final permutation to the initial key, unless one also has the changed key. This is probably a safe, fairly fast way of generating the permutation from the key. The generation is only a small part of the overhead if large blocks are being used so that the excess work is not very significant.

Protocols should arrange that permutations are not used too often or for too long. A "salt" could well be part of the key so that the same permutation is never used for more than one file. In such a case the overhead would be maximised but probably would still be small enough.

The choice of byte rather than word operations, will slow the process slightly, but will fit a wider range of processors more directly.

The author would like to be informed of significant weaknesses of the basic encryption method.

A stronger but more complicated encode can be defined as

```
M=n/2+1
for n=M to 0 by -1 do
   ch[n] = ch[n] XOR perm[ ch[n+1]+ch[n-M+N-1] ]
for n=M-1 to N-1 do
   ch[n] = ch[n] XOR perm[ ch[n-1]+ch[n-M+1] ]
```

This avoids the centre weakness, and increases the change diffusion (propagation of single and multiple character changes). The two overlapping passes from the middle avoid the problem of n-1 pointing outside the block when n=0. However the speed of encrypting and decrypting is no slower. Is the possible extra security worth the increase in complication? There is no easy answer so the simpler is the current solution.

Performance

For a block of reasonable size, the performance is determined by the code derived from a single line of code and includes an XOR operation, byte addition and a table look up. This is done about three times, so that the work per byte is small and comparable with file movement or very simple processing. Thus the time penalty is likely to be admissable in applications such as filing and message handling. It is obvious however that in many cases it will be system considerations which dominate the acceptability of encrytion and not performance.

Results of M Burrows

Using a MicroVAX II, the time per character per three passes is 12.5 microseconds. The inner loop was unrolled 8 times to get this performance. The VAX8800 is five times as fast.

Acknowledgements

Thanks are due to R.M.Needham, J.H.Davenport, D.W.Davies for useful suggestions, T.M.Lomas for pointing out the inner cycle reduces to a mere two instructions on the VAX and to M.Burrows for the timing runs..