1024 - A High Security Software Oriented Block Cipher*

Dieter Schmidt†

March 26, 2009

Abstract

A cryptographer with weak algorithms gets his feedback almost instantly by the open crypto community. But what about government cryptanalysis ? Given the fact that there is a considerable amount of cryptanalysis behind closed doors, what is to be done to get COMINT deaf ? The NSA, as the most closely examined SIGINT agency, has a workforce of 38,000 [2], among them several thousand cryptologists. The actual wiretapping is done by the Central Security Service with 25,000 women and men. Other industrialised states have also thousands cryptologists at their wage role.

The block cipher 1024 is an attempt to make cryptanalysis more difficult especially with differential (DC) and linear cryptanalysis (LC). The assumption is that the increased security will defeat other cryptanalytical methods not yet known by the open crypto community.

1024 has a block size of 1024 bits und key length of 2048 bits.

1 Introduction

From the beginning of open cryptography in the 70's unease has taken on government cryptographers. They felt that open cryptographic research, especially cryptanalysis, would make their work more difficult. On the other hand open cryptographic works from ATM to the German Health Card and have shown that there is a market for computer security software.

In the 80's Admiral Bobby Inman, Director of the NSA, established a committee [1] that should censor all cryptographic publications. But

*Revised March 26, 2009

†Denkmalstrasse 16, D-57567 Daaden, Germany, dieterschmidt@gmx.com

when scientists did not follow up, he warned that legislative action would be taken. But that has not been the place and open cryptography has been a success story.

It should be clear which stand I support, but there is a place where government censorship is the right thing: Ultra, the decryption of the German Enigma during World War II. Knowledge of that would have the Germans to develope a new cipher machine. That would be a catastrophy for the Allied war effort and would lengthen the war by approx. a year [7].

Nevertheless government cryptography has proven that they are more advanced than open research. Differential cryptanalysis was known by Don Coppersmith in 1974 when he and others designed the Data Encryption Standard. Open research followed up 18 years later [3]. Differential cryptanalysis was a powerful tool and the government asked IBM to be silent for national security reasons (see preface of [3]). Intriguingly differential cryptanalysis is a chosen plaintext attack and you need an insider to mount this attack.

Another point is public key encryption. It is known the a scientist of the UK's Government Communication Head Quarter (GCHQ) knew in 1967 about this. That is approx. ten earlier year then open research.

But stifling open research on cryptology is not the only sin that the NSA has committed. There were the operations Minaret und Shamrock. While Minaret were directed against the leaders of the student revolt in the 60's and took away their telecommunication data without due process, Shamrock was directed against non US governments and the cable operators were asked to hand over telecommunication data.

But don't think here in Germany everything is all right. However, in his verdict on data retention the Constitutional Court in Germany ruled that citizens have the right to privacy (i.e. among others encryption). Given the fact that intelligence agencies in Germany are in the business of wiretapping, one must admit that the right to encryption is also the right to choose the algorithm freely and with maximum security.

Here in Germany in 1998 the minister of interior, a conservative, tried to ban all encryption the government could not read. But the conservatives lost the elections und today encryption is so ubiquitous that even the conservatives don't dare it to outlaw it.

But back to the algorithm. 1024 has a block size of 1024 bits and keylength of 2048 bits. It takes as building blocks the ciphers **MMB** [5] und **SAFER** [8]. While **MMB** has bilding blocks words of 32 bits, the building block of **SAFER** is a byte. So **SAFER** has to be adapted to use 4 bytes. Also **SAFER** [9] has a problem with the diffusion. Thats why the rotation has been used. However **SAFER** has an advantage:

Its diffusion layer was built around the Cooley-Tukey-Fast-Fourier-Transform with complexity $O(n \ln n)$. AES (Rijndael) has a diffusion layer built around MDS-Codes with complexity $O(n^2)$.

The rest of the paper is organized as follows: Section 2 describes the algorithm, Section 3 relates to security with linear cryptanalysis. Section 4 gives implementation details. We conclude in Section 5 and Appendix A gives the reference implementation in C. Appendix B contains a program in C for multipermutation on a byte level [14, 15, 16]. Appendix C gives a program to calculate the maximum bias for linear cryptanalysis.

2 The Algorithm

2.1 The S-Boxes

1024 is a Substitution-Permutation-Network (SPN). It uses as building blocks multiplication modulo $2^{32}-1$ as s-box and a modified diffusion layer from SAFER. Keys are applied before and after the s-boxes. 1024 has 32 s-boxes (multiplication modulo $2^{32}-1$). Let us denote in this subsection addition, subtraction and multiplication modulo 2^n-1 by respectively $+$,- and $\times$, ordinary multiplication by $*$, integer division by $[\div]$,XOR by $\oplus$, rotation by a bits to the left by $\lll a$, rotation by a bits to the right by $\ggg a$ and addition modulo 2^{256} by $\boxplus$. .

Multiplication modulo 2^n-1 as s-box was first used by Daemen et. al. [4, 5, 6]. The studied function is:

$$f^a(x) = \begin{cases} a \times x & \text{if} \quad x < 2^n-1 \\ 2^n-1 & \text{if} \quad x = 2^n-1 \end{cases} \tag{1}$$

The calculation is easy:

$$a*b \mod(2^n-1) = (a*b \mod(2^n) + \lfloor \frac{a*b}{2^n} \rfloor)(1 + \lfloor \frac{1}{2^n} \rfloor) \tag{2}$$

The first righthand term is obtained by taking the least significant bits of the product, the second term by taking the remaining bits and shifting them to the right by n bits and add that to the first term. If a carry (i.e. bit 32 is set) results from that addition the result is incremented by 1. Note that [4] gives a wrong formula. It has been corrected in chapter 11 of Joan Daemens Ph.D. thesis [6]. Note that the last factor of the righthandside of the equation is not distributive.

Multiplication modulo 2^n-1 has interesting properties. A multiplication by 2 modulo 2^n-1 is equivalent by a rotation to left by one. Similarly $2^k \times a = a \lll k$. Further material can be found in [4].

In [5] multiplication factors modulo $2^{32}-1$ are given. In the cipher **MMB** a encryption (in hex. 0x025F1CDB) and decryption (in hex. 0x0DAD4694) factors are introduced. The encryption factor by **MMB** is in 1024 rotated left from 0 to 31. The left most block is assigned encryption factor without rotation. The next block is assigned encryption_factors $\lll$ 1 and so on. The last block (right most) is assigned encryption_factors $\lll$ 31. That is why the number of blocks with 32 bits is 32 (see reference implementation **WIDTH**).

The decryption factor from **MMB** is treated almost the same way. The decryption factor from **MMB** is the decryption factor from 1024 in left most block. The decryption factor is rotated to the right from 0 to 31 and assigned from left most block to right most block. The left most block is assigned the decryption_factor $\ggg$ 0, the right most block is assigned the decryption_factors $\ggg$ 31.

The critical probability of the s-boxes with regard to differential cryptanalysis is 2^{-9}.

2.2 Diffusion Layer

The diffusion layer has as parent the diffusion layer from **SAFER** [8, 9]. However, there are four modifications:

1. 32 blocks instead of eight.
2. Four bytes instead of one byte as primitive unit. See [13].
3. Before the addition primitive units are beeing rotated.
4. One additional layer

Point two is clear. In a modern PC the CPU has a register size of four bytes, sometimes eight bytes. Obviously this will increase the speed.

The Pseudo-Hadamard-Transform is defined as:

$$b_1 = 2a_1 + a_2 \tag{3}$$
$$b_2 = a_1 + a_2 \tag{4}$$

It can be rewritten:

$$b_2 = a_1 + a_2 \tag{5}$$
$$b_1 = a_1 + b_2 \tag{6}$$

The Pseudo-Hadamard-Transform has one disadvantage. The least significant bit of b_1 is not dependent on a_1. Schneier et. al. [13] were

aware that b_1 is not dependent on the most significant bit of a_1. But there is no word on the least significant bit of a_1 (or at least I did not see it). Because $b_1 = 2a_1 + a_2$ the least significant bit of b_1 is a function of a_2 and not of a_1. Thus the least significant bit of b_1 is incomplete.

In [12] a branch number for invertible linear mappings was introduced. It is defined as

$$B(\theta) = \min_{a \neq 0}(\omega_h(a) + \omega_h(\theta(a))) \tag{7}$$

where ω_h denotes the Hamming weight of a, i.e. the number of nonzero components of a. For example $a = (3x0)F$ has the Hamming weight of 4. θ is the linear mapping. The branch number of the linear mapping θ is at least B. A linear mapping with optimal branch number $B = n + 1$ can be constructed by a maximum distance separable code. I can see no reason why this can not be done on a non linear transform. Bearing that in mind, the branch number of Twofish [13] is two, 2^{31} in left most block and the other blocks 0 as input. The output is 2^{31} on the right most block, 0 else. The same holds for my diffusion layer (a branch number of 2). An input of 2^{31} on the left most block, 0 else, gets an output of the right most block of 2^{31}, the other blocks 0. Obviously this is a poor performance.

That is why the rotation was introduced. To the b_2 a rotated value of a_1 is added. Similarly to the b_1 a rotated value of b_2 is added. The rotation values are pseudo-random and it is the assumption that the branch number is higher. For more details, see the function pht in the reference implementation. The function ipht does the opposite of the function pht, i.e. the rotation is invertible.

On the original diffusion layer of SAFER rotations were introduced. The result is that an odd rotation from the "left" to the "right" and even rotation from the "right" to "left" is a multipermutation [14, 15, 16]. Note the the natural unit of the diffusion layer of SAFER is the byte. My vintage computer of 1997 was able to calculate this, but not 16 bit or 32 bit. A modern computer could calculate 16 bit, but not 32 bit. However, it is conjectured that the multipermutation through rotation and addition is valid for 32 bit. For more details, see appendix B.

2.3 Addition modulo 2^{256}

Addition modulo 2^{256} was introduced to give an upper bound for linear cryptanalysis. If we take [11], we can have an upper bound for linear cryptanalysis without being forced to examine the diffusion layer or

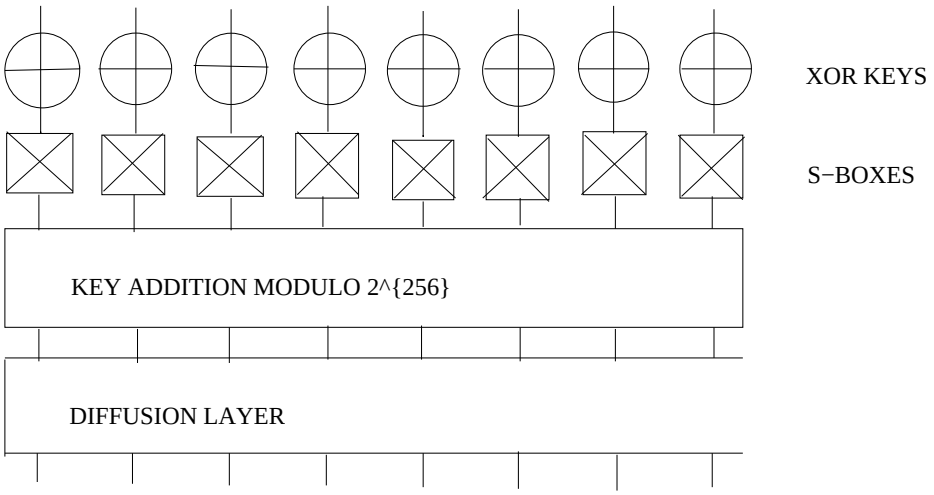

Figure 1: The left quarter of a primary round

the s-boxes. See subsection Key Schedule and section Linear Cryptanalysis for further details.

1024 has a bit length of 1024 bits. This means addition modulo 2^{256} is applied four times, from left to right, sometimes after the s-boxes, sometimes before the s-boxes. Since there are 32 s-boxes of 32 bits the input or output of one addition modulo 2^{256} is eight s-boxes.

One can argue that all the keys should be applied by addition modulo 2^{256}, so one can use less rounds. But the XOR of some keys is there to make the cryptanalysis more difficult by using different groups or to avoid symmetry attacks.

2.4 Primary Round

1024 consists of eight primary rounds, a middle transform and eight secondary rounds. The number of primary rounds und secondary rounds must be equal. A secondary round is the inversion of a primary round, except for the key and the encryption/decryption factors. Figure 1 shows the left quarter of a primary round.

A primary round starts with XORing the first half of the round key. Since 1024 consists of 32 blocks of 32 bits, a 32 bit CPU will do that in 32 steps.

The s-boxes (multiplication modulo $2^{32}-1$) follow. Note that each of the s-boxes has its own multiplication factor. (see subsection s-box). As 1024 consists of 32 blocks of 32 bits, one has 32 multiplications and 32 encryption factors, since the s-boxes should have no symmetry.

Addition modulo 2^{256} of the second half of the round key follows.

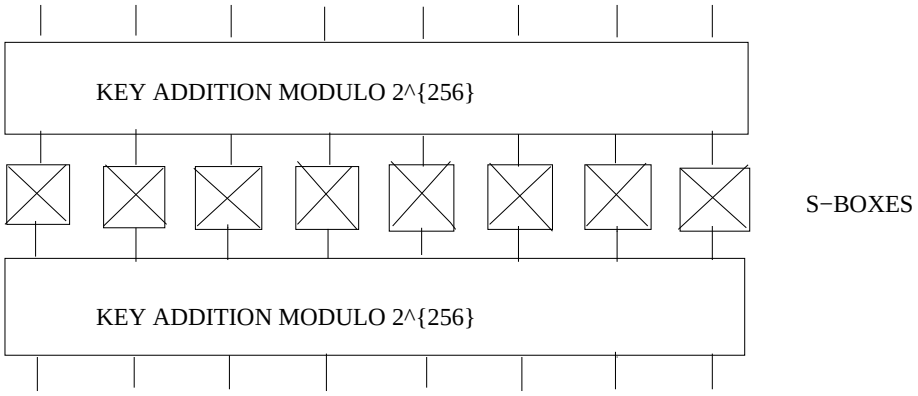

Figure 2: A left quarter of the middle transform

As 1024 has 1024 bits, there are four additions. The left most addition consists of the output of left most eight s-boxes and so on.

Finally the Pseudo-Hadamard-Transform (see function pht in the reference implementation) is done. Note that the least significant four bytes of the addition modulo 2^{256} are least significant unit of the Pseudo-Hadamard-Transform.

2.5 The Middle Transform

The middle transform is the only part of 1024 that comes with no Pseudo-Hadamard-Transform. Instead the data output from the eighth primary round comes as input for the middle transform. First there is addition modulo 2^{256} of the first half of the round key. Again the least significant four byte output by the eighth primary round is the least significant input to the left most addition modulo 2^{256}. Figure 2 shows one quarter of the middle transform.

After addition modulo 2^{256} is completed, the data (1024 bit) is partitioned in 32 blocks of 32 bits. This data is now input to the s-boxes (multiplication modulo $2^{32} - 1$). The s-boxes are the same as in the primary rounds.

The output by the s-boxes are now input to second addition modulo 2^{256} of the second half the round key. This is the same as in subsection primary round.

2.6 The Secondary Rounds

The input to the first secondary round is the output from the middle transform. At first there is an inverse diffusion layer (see function ipht in the reference implementation). Second there is addition modulo

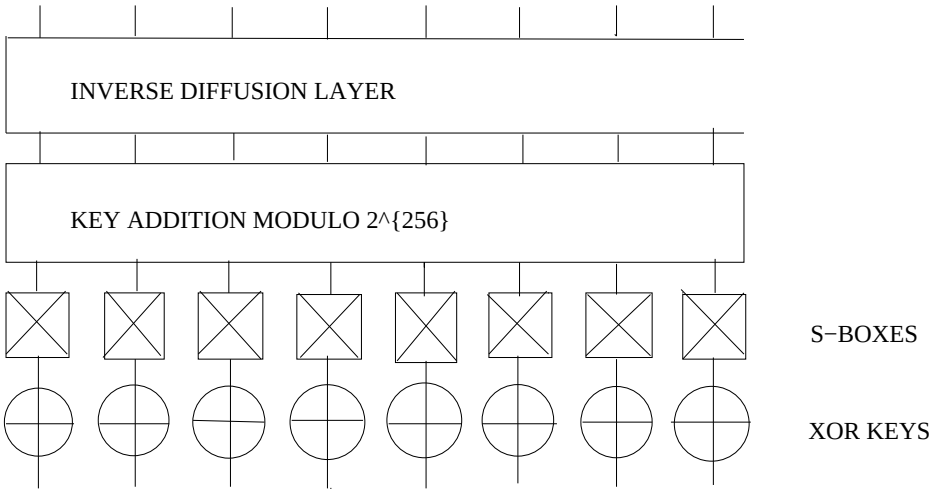

Figure 3: The left quarter of a secondary round

2^{256} with the first half of the round key. Third there are the s-boxes. The s-boxes have the same factors as the s-boxes in the primary round and in the middle transform. Finally there is XOR with the second half of the round key. Figure 3 shows the left quarter of one secondary round.

2.7 Key Schedule

The first round key is the user key (see function key_schedule in the reference implementation). The next round key is the predecessor rotated by 455 bits to the left and so on. Note that a round key and the key to middle transform are applied before and after the s-boxes. For the eight primary rounds one half of the key is applied before the s-boxes (XOR, low bits), the other half of the key is applied after the s-boxes (addition modulo 2^{256}, high bits). For the middle transform one half of the key is applied before the s-boxes (addition modulo 2^{256}, low bits), and one half of the key is applied after the s-boxes (addition modulo 2^{256}, high bits). For the eight secondary rounds one half of the key is applied before the s-boxes (addition modulo 2^{256}, low bits), the other half of the key is applied after the s-boxes (XOR, high bits).

Why is the rotation 455 bits to the left? This is linear cryptanalysis and the so called bias or the so called effectiveness [10, 11]. When we take into account that addition modulo 2^{256} of the keys in the rounds and the middle transform we have nine additions of round keys and middle transform. If we take into account [10, 11] we have as average bias $\epsilon = 2^{-9+128} = 2^{-119}$ (piling up lemma not considered). The

rotation of the round keys was so determined that the maximum bias was as low as possible. This can be done via exhausive search and takes on a modern PC a few secounds. The result is that a rotation to the left of 455 bits is desired one. It has a maximum bias of $\epsilon = 2^{-1018}$ (raw data 1024, piling up lemma 7, bit position 1591). It should be noted that a rotation to the left of 1593 bits has the same bias and same data except the bit position. Note that 455+1593=2048 is the key length. However, a rotation by 1593 bit can not be used by the reference implementation (maximum rotation is 1023). Appendix C gives a program to calculate maximum bias.

For the decryption process the XOR-keys simply swap their position on the primary and secondary rounds. If we want an algorithm to be the same for encryption and decryption we need to have the inverse of addition modulo 2^{256}. The inverse of an integer value, be it signed or unsigned, is to invert the bits of that integer and to add 1. When this is done, the key values swap their position on the primary and secondary rounds and on the middle transform. For details, see the function invert_keys of the reference implementation.

2.8 Decryption

For encryption and decryption the same algorithm is used. For the s-boxes the decryption factors are used. The are the inverse of the encryption keys modulo $2^{32} - 1$. [5] gives one of them. The rest is calculated by rotating this one key to the right from $1 \ldots 31$. See the function decryption_factors of the reference implementation for details.

Also the keys have to be inverted. While XOR is self-invers you will need only to mirror them at the s-boxes of the middle transform. Addition modulo 2^{256} is slightly more difficult: you will need the bit complement and add 1. Having that done you will have to mirror at the s-boxes of the middle transform.

The functions key_schedule and invert_keys of the reference implementation will give you further insight.

3 Linear Cryptanalysis

Linear cryptanalysis is a known plaintext attack and was first published by Matsui [10]. It looks for an effective linear expression for a given cipher, i.e. it looks for plaintext bits P_i, a ciphertext bits C_j and key bits K_k so that

$$P_1 \oplus P_2 \oplus \ldots \oplus P_i \oplus C_1 \oplus C_2 \oplus \ldots \oplus C_j = K_1 \oplus K_2 \oplus \ldots \oplus K_k \quad (8)$$

holds with probability $p \neq 1/2$. The magnitude $\epsilon = |p - 1/2|$ is called the bias or the effectiveness of the equation (8).

Matsui showed that the bias of the linear equation is closely related to the number of plaintext N needed, i.e. roughly speaking $N = 1/(\epsilon^2)$. In the subsection key schedule it is shown that the highest bias is $\epsilon = 2^{-1018}$. Hence the number of plaintext for a successful attack is $N = 2^{2036}$. Since there are only 2^{1024} different plaintext available we can conclude that 1024 is immune from linear cryptanalysis. Please note that the ϵ was only derived using the key addition modulo 2^{2048} making use of [11]. The nonlinearity obtained by the s-boxes and the diffusions layer was not considered, i.e. the limit is an upper bound.

4 Implementation

Modern Central Processing Units (CPU) come with several Arithmetic Logic Units (ALU). The ALUs are the heart of the CPU. All the integer arithmetic ist done here. A Pentium IV has four ALUs and it is clear, that this will increase performance, if the program code allows for parallel executions. Modern computers have several ALUs, and sometimes with several cores. A core is a CPU and a multiple cores on a computer will increase the performance. Again the program code is faster, if it allows parallel executions.

It is estimated that the program code of 1024 in assembler (machine language) on a 32-bit computer , 2.5 Gigahertz clock, one core and one ALU results in a performance of approx. 15 MByte/s. Although the reference implementation in C of 1024 is working, one might ask whether performance can be increased. A C implementation is about 3 to 5 slower then in assembler. One can do the whole implementation in assembler, but I guess the function that have to do with encryption and decryption are enough (function crypt, modmult, pht, ipht).

4.1 Function modmult

Especially modmult is the most promising candidate for assembler programing. In C one has to declare several unsigned long long (8 byte) to catch the high bits and shift them by 32 to the right and add, and to catch the carry and shift it to the right by 32 bits and increment the result. If an integer hardware multiplier is present (most 32 bit CPU have it) the multiplication is done in 3 clocks. The Intel compatible CPUs have another point of interest. The 64 bit multiplication result is stored in the registers EDX:EAX. So the addition is simple **ADD EAX,EDX**. If in this addition a carry is set, you simply add to

EAX the carry with the command ADC EAX,0x00000000. The whole function modmult in assembler is here:

```
MOV EBX,factor1
MOV EAX,factor2
MUL EBX
ADD EAX,EDX
ADC EAX,0x00000000
MOV modmult,EAX
```

All registers are 32 bit so far. You can see that the assembler program has no conditional jumps and is not susceptible to timing attacks.

4.2 Function crypt and Functions pht/ipht

The programing of the function crypt in the reference implementation is somewhat awkward. If one can see the pertinent figures in the article, she/he finds out, that the programing is "vertical". For example: if you take figure 1 (one quarter of a primary round) and the reference implementation one can see that: first I have done one XOR with the key, then one function modmult, then one block key addition modulo 2^{256}. I should have done it "horizontal", i.e. first all the key XOR, second all the functions modmult and third key addition modulo 2^{256}. Note that addition modulo 2^{256} will only be completed, if all the components are there. Otherwise the the CPU pipeline will be stalled. The addition modulo 2^{256} has two addition commands: first the usual addition (ADD, one time) and then addition with carry (ADC, seven times). In assembler it looks like this:

```
MOV EAX,data1
MOV EBX,key1
ADD EAX,EBX
MOV data1,EAX
MOV EAX,data2
MOV EBX,key2
ADC EAX,EBX
MOV data2,EAX
     .
     .
     .
MOV EAX,data8
MOV EBX,key8
ADC EAX,EBX
```

```
MOV data8,EAI
```

The same is true for the functions pht respectively ipht. First the rotation and the adding are done and then the decimation-by-2-permutation (see Cooley-Tukey-FFT).

If all of the function crypt and the functions pht/ipht are to be programed in assembler, the performance of 1024 will be greatly enhanced. If you are not tired yet, you can unroll the loops and program the functions inline. This is a tedious task, but I guess it is worth it.

5 Conclusion

The block cipher 1024 was introduced. It has a block size of 1024 bits and a key length of 2048 bits. While 1024 is immune from linear cryptanalysis, further work is needed to prove that 1024 immune to other attacks, especially against differential cryptanalysis.

References

[1] Bamford, James: *The Puzzle Palace*, Penguin Books, New York, 1983

[2] Bamford, James: *Body of Secrets*, Doubleday, New York, 2001

[3] Biham, Eli and Adi Shamir: *Differential Cryptanalysis of the Data Encryption Standard*, Springer Verlag, Berlin, 1993

[4] Daemen, Joan; Luc van Linden, Rene Govaerts and Joos Vandewalle: Propagation Properties of Multiplication Modulo $2^e - 1$, *Proceedings of 15th Symposium on Information Theory in the Benelux*, Werkgemeenschaft voor Informatie- en Communicatietheorie, available from `http://www.cosic.esat.kuleuven.be/publication/static/1992.html`, 1992

[5] Daemen, Joan; Rene Govaerts and Joos Vandewalle: Block Cipher based on Modular Arithmetic, *Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography*, W. Wolfowicz (ed.), Fondazione Ugo Bordoni, Rome, Italy available from `http://www.cosic.esat.kuleuven.be/publication/static/1993.html`, 1993

[6] Daemen, Joan: *Cipher and Hash Function Design, Strategies based on linear and differential Cryptanalysis*, Ph.D. thesis, KU Leuven, Belgium, available from

http://homes.esat.kuleuven.be/~cosicart/ps/JD-9500, 1995

[7] Kozaczuk, Wladyslaw: *Geheimoperation Wicher*, Bernard & Graefe Verlag, Koblenz, 1989.

[8] Massey, James: SAFER K-64: A Byte-Oriented Block-Cipher Algorithm, in Anderson Ross (Ed.): *Fast Software Encryption*, Springer Verlag, Berlin, 1994

[9] Massey, James: SAFER K-64: One year later in Preneel, Bart (Ed.): *Fast Software Encryption*, Springer Verlag, Berlin, 1995

[10] Matsui, Mitsuru: Linear Cryptanalysis Method for DES Cipher, in Tor Helleseth (Ed.): *Advances in Cryptology - EUROCRYPT '93*, Springer Verlag, Berlin, 1993

[11] Mukhopadhyay, Debdeep and Dipanwita RoyChowdhury: *Key Mixing in Block Ciphers through Addition modulo* 2^n, available from `http://eprint.iacr.org/2005/383.pdf`

[12] Rijmen, Vincent; Joan Daemen et. al.: The cipher SHARK, in Gollmann, Dieter (Ed.): *Fast Software Encryption*, Springer Verlag, Berlin, 1996

[13] Schneier, Bruce; John Kelsey, Doug Whiting, David Wagner, Chris Hall and Niels Ferguson: *Twofish: A 128-Bit Block Cipher*, 1998, available from `http://www.schneier.com/twofish.html`

[14] Vaudenay, Serge: On the Need for Multipermutation: Cryptanalysis of MD4 and SAFER, in Preneel, Bart (Ed.): *Fast Software Encryption*, Springer Verlag, Berlin, 1995

[15] Vaudenay, Serge and Jacques Stern: CS-Cipher, in Vaudenay, Serge (Ed.): *Fast Software Encryption*, Springer Verlag, Berlin, 1998

[16] Vaudenay, Serge: On the Security of CS-Cipher, in Knudsen, Lars (Ed.): *Fast Software Encryption*, Springer Verlag, Berlin,1999

A Reference Implementation

```
#include<stdio.h>

#define NUM_ROUNDS 8
#define INT_LENGTH 32
#define ROL(x,a) (((x)<<(a))|((x)>>(INT_LENGTH-(a))))
#define ROR(x,a) (((x)<<(INT_LENGTH-(a))|((x)>>(a))))
#define WIDTH 32
```

```
#define ROTROUND 455

void encryption_factors(unsigned long e_factors[WIDTH]){
  unsigned long i;

  e_factors[0]=0x025F1CD8;
  for(i=1;i<WIDTH;i++){
    e_factors[i]=ROL(e_factors[0],i);
  }
}

void decryption_factors(unsigned long d_factors[WIDTH]){
  unsigned long i;

  d_factors[0]=229459604;
  for(i=1;i<WIDTH;i++){
    d_factors[i]=ROR(d_factors[0],i);
  }
}

unsigned long modmult(unsigned long factor1,unsigned long factor2){
  unsigned long long f1,f2,ergebnis,k;

  f1=(unsigned long long) factor1;
  f2=(unsigned long long) factor2;
  ergebnis=f1*f2;
  k=(ergebnis>>INT_LENGTH);
  ergebnis&=0xFFFFFFFF;
  ergebnis+=k;
  ergebnis+=(ergebnis>>INT_LENGTH) & 1;
  return(ergebnis & 0xFFFFFFFF);
}

void invert_keys(unsigned long keys[4*NUM_ROUNDS+2][WIDTH]){
  unsigned long i,j,help;
  unsigned long long h1,h2,carry1,carry2;

  for(i=0;i<NUM_ROUNDS;i++){
    for(j=0;j<WIDTH;j++){
      help=keys[2*i][j];
      keys[2*i][j]=keys[4*NUM_ROUNDS-2*i+1][j];
      keys[4*NUM_ROUNDS-2*i+1][j]=help;
    }
```

```
  }
  for(i=0;i<(NUM_ROUNDS+1);i++){
    carry1=1;
    carry2=1;
    for(j=0;j<WIDTH;j++){
      h2=(unsigned long long) keys[4*NUM_ROUNDS-2*i][j];
      h1=(unsigned long long) keys[2*i+1][j];
      h1^=0xFFFFFFFF;
      h2^=0xFFFFFFFF;
      h1+=carry1;
      h2+=carry2;
      carry2=(h2>>INT_LENGTH) & 1;
      carry1=(h1>>INT_LENGTH) & 1;
      if((j & 7)==7){
        carry1=1;
        carry2=1;
      }
      keys[4*NUM_ROUNDS-2*i][j]=h1 & 0xFFFFFFFF;
      keys[2*i+1][j]=h2 & 0xFFFFFFFF;
    }
  }
}

void key_schedule(unsigned long user_key[2][WIDTH],\
  unsigned long key[4*NUM_ROUNDS+2][WIDTH]){
  unsigned long i,j;

  for(i=0;i<2;i++){
    for(j=0;j<WIDTH;j++){
      key[i][j]=user_key[i][j];
    }
  }
  for(i=0;i<(2*NUM_ROUNDS);i++){
    for(j=0;j<WIDTH;j++) {
      key[2*i+3][j]=(key[2*i+(j+((WIDTH*INT_LENGTH-ROTBOUND)\
      /INT_LENGTH))/WIDTH]\
      [(j+((WIDTH*INT_LENGTH-ROTBOUND)/INT_LENGTH))\
      %WIDTH]<<(ROTBOUND%INT_LENGTH))|\
      (key[2*i+(j+(WIDTH*INT_LENGTH-ROTBOUND)/INT_LENGTH+1)/WIDTH]\
      [(j+(WIDTH*INT_LENGTH-ROTBOUND)/INT_LENGTH+1)%WIDTH]>>\
      (INT_LENGTH-ROTBOUND%INT_LENGTH));

      key[2*i+2][j]=(key[2*i+1-(j+((WIDTH*INT_LENGTH-ROTBOUND)\
```

```
      /INT_LENGTH))/WIDTH]\
      [(j+((WIDTH*INT_LENGTH-ROTBOUND)/INT_LENGTH))\
      %WIDTH]<<(ROTBOUND%INT_LENGTH))\
      |(key[2*i+1-(j+(WIDTH*INT_LENGTH-ROTBOUND)/INT_LENGTH+1)/WIDTH]\
      [(j+(WIDTH*INT_LENGTH-ROTBOUND)/INT_LENGTH+1)%WIDTH]>>\
      (INT_LENGTH-ROTBOUND%INT_LENGTH));
    }
  }
}

void pht(unsigned long a[WIDTH]){
  unsigned long i,b[WIDTH];

  a[1]+=ROL(a[0],1);
  a[0]+=ROL(a[1],2);
  a[3]+=ROL(a[2],7);
  a[2]+=ROL(a[3],16);
  a[5]+=ROL(a[4],13),
  a[4]+=ROL(a[5],30);
  a[7]+=ROL(a[6],19);
  a[6]+=ROL(a[7],12);
  a[9]+=ROL(a[8],25);
  a[8]+=ROL(a[9],26);
  a[11]+=ROL(a[10],31);
  a[10]+=ROL(a[11],8);
  a[13]+=ROL(a[12],5);
  a[12]+=ROL(a[13],22);
  a[15]+=ROL(a[14],11);
  a[14]+=ROL(a[15],4);
  a[17]+=ROL(a[16],17);
  a[16]+=ROL(a[17],18),
  a[19]+=ROL(a[18],23);
  a[18]+=a[19];
  a[21]+=ROL(a[20],29);
  a[20]+=ROL(a[21],14);
  a[23]+=ROL(a[22],3);
  a[22]+=ROL(a[23],28);
  a[25]+=ROL(a[24],9);
  a[24]+=ROL(a[25],10);
  a[27]+=ROL(a[26],15);
  a[26]+=ROL(a[27],24);
  a[29]+=ROL(a[28],21);
  a[28]+=ROL(a[29],6);
```

```
a[31]+=ROL(a[30],27);
a[30]+=ROL(a[31],20);
for(i=0;i<(WIDTH/2);i++){
  b[i]=a[2*i];
  b[i+(WIDTH/2)]=a[2*i+1];
}

b[1]+=ROL(b[0],1);
b[0]+=ROL(b[1],2);
b[3]+=ROL(b[2],11);
b[2]+=ROL(b[3],20);
b[5]+=ROL(b[4],21);
b[4]+=ROL(b[5],6);
b[7]+=ROL(b[6],31);
b[6]+=ROL(b[7],24);
b[9]+=ROL(b[8],9);
b[8]+=ROL(b[9],10);
b[11]+=ROL(b[10],19);
b[10]+=ROL(b[11],28);
b[13]+=ROL(b[12],29);
b[12]+=ROL(b[13],14);
b[15]+=ROL(b[14],7);
b[14]+=b[15];
b[17]+=ROL(b[16],17);
b[16]+=ROL(b[17],18);
b[19]+=ROL(b[18],27);
b[18]+=ROL(b[19],4);
b[21]+=ROL(b[20],5);
b[20]+=ROL(b[21],22);
b[23]+=ROL(b[22],15);
b[22]+=ROL(b[23],8);
b[25]+=ROL(b[24],25);
b[24]+=ROL(b[25],26);
b[27]+=ROL(b[26],3);
b[26]+=ROL(b[27],12);
b[29]+=ROL(b[28],13);
b[28]+=ROL(b[29],30);
b[31]+=ROL(b[30],23);
b[30]+=ROL(b[31],16);
for(i=0;i<(WIDTH/2);i++){
  a[i]=b[2*i];
  a[i+(WIDTH/2)]=b[2*i+1];
}
```

```
a[1]+=ROL(a[0],1);
a[0]+=ROL(a[1],2);
a[3]+=ROL(a[2],15);
a[2]+=ROL(a[3],24);
a[5]+=ROL(a[4],29);
a[4]+=ROL(a[5],14);
a[7]+=ROL(a[6],11);
a[6]+=ROL(a[7],4);
a[9]+=ROL(a[8],25);
a[8]+=ROL(a[9],26);
a[11]+=ROL(a[10],7);
a[10]+=ROL(a[11],16);
a[13]+=ROL(a[12],21);
a[12]+=ROL(a[13],6);
a[15]+=ROL(a[14],3);
a[14]+=ROL(a[15],28);
a[17]+=ROL(a[16],17);
a[16]+=ROL(a[17],18);
a[19]+=ROL(a[18],31);
a[18]+=ROL(a[19],8);
a[21]+=ROL(a[20],13);
a[20]+=ROL(a[21],30);
a[23]+=ROL(a[22],27);
a[22]+=ROL(a[23],20);
a[25]+=ROL(a[24],9);
a[24]+=ROL(a[25],10),
a[27]+=ROL(a[26],23);
a[26]+=a[27];
a[29]+=ROL(a[28],5);
a[28]+=ROL(a[29],22);
a[31]+=ROL(a[30],19);
a[30]+=ROL(a[31],12);
for(i=0;i<(WIDTH/2);i++){
  b[i]=a[2*i];
  b[i+(WIDTH/2)]=a[2*i+1];
}
b[1]+=ROL(b[0],1);
b[0]+=ROL(b[1],2);
b[3]+=ROL(b[2],19);
b[2]+=ROL(b[3],12);
b[5]+=ROL(b[4],5);
b[4]+=ROL(b[5],22);
b[7]+=ROL(b[6],23);
```

```
b[6]+=b[7];
b[9]+=ROL(b[8],9);
b[8]+=ROL(b[9],10);
b[11]+=ROL(b[10],27);
b[10]+=ROL(b[11],20);
b[13]+=ROL(b[12],13);
b[12]+=ROL(b[13],30);
b[15]+=ROL(b[14],31);
b[14]+=ROL(b[15],8);
b[17]+=ROL(b[16],17);
b[16]+=ROL(b[17],18);
b[19]+=ROL(b[18],3);
b[18]+=ROL(b[19],28);
b[21]+=ROL(b[20],21);
b[20]+=ROL(b[21],6);
b[23]+=ROL(b[22],7);
b[22]+=ROL(b[23],16);
b[25]+=ROL(b[24],25);
b[24]+=ROL(b[25],26);
b[27]+=ROL(b[26],11);
b[26]+=ROL(b[27],4);
b[29]+=ROL(b[28],29);
b[28]+=ROL(b[29],14);
b[31]+=ROL(b[30],15);
b[30]+=ROL(b[31],24);
for(i=0;i<(WIDTH/2);i++){
  a[i]=b[2*i];
  a[i+(WIDTH/2)]=b[2*i+1];
}
a[1]+=ROL(a[0],1);
a[0]+=ROL(a[1],2);
a[3]+=ROL(a[2],23);
a[2]+=ROL(a[3],28);
a[5]+=ROL(a[4],13);
a[4]+=ROL(a[5],22);
a[7]+=ROL(a[6],3);
a[6]+=ROL(a[7],16);
a[9]+=ROL(a[8],25);
a[8]+=ROL(a[9],10);
a[11]+=ROL(a[10],15);
a[10]+=ROL(a[11],4);
a[13]+=ROL(a[12],5);
a[12]+=ROL(a[13],30);
```

```
a[15]+=ROL(a[14],27);
a[14]+=ROL(a[15],24);
a[17]+=ROL(a[16],17);
a[16]+=ROL(a[17],18);
a[19]+=ROL(a[18],7);
a[18]+=ROL(a[19],12);
a[21]+=ROL(a[20],29);
a[20]+=ROL(a[21],6);
a[23]+=ROL(a[22],19);
a[22]+=a[23];
a[25]+=ROL(a[24],9);
a[24]+=ROL(a[25],26),
a[27]+=ROL(a[26],31);
a[26]+=ROL(a[27],20);
a[29]+=ROL(a[28],21);
a[28]+=ROL(a[29],14);
a[31]+=ROL(a[30],11);
a[30]+=ROL(a[31],8);
for(i=0;i<(WIDTH/2);i++){
  b[i]=a[2*i];
  b[i+(WIDTH/2)]=a[2*i+1];
}
b[1]+=ROL(b[0],1);
b[0]+=ROL(b[1],2);
b[3]+=ROL(b[2],27);
b[2]+=ROL(b[3],8);
b[5]+=ROL(b[4],21);
b[4]+=ROL(b[5],14);
b[7]+=ROL(b[6],15);
b[6]+=ROL(b[7],20);
b[9]+=ROL(b[8],9);
b[8]+=ROL(b[9],26);
b[11]+=ROL(b[10],3);
b[10]+=b[11];
b[13]+=ROL(b[12],29);
b[12]+=ROL(b[13],6);
b[15]+=ROL(b[14],23);
b[14]+=ROL(b[15],12);
b[17]+=ROL(b[16],17);
b[16]+=ROL(b[17],18);
b[19]+=ROL(b[18],11);
b[18]+=ROL(b[19],24);
b[21]+=ROL(b[20],5);
```

```
  b[20]+=ROL(b[21],30);
  b[23]+=ROL(b[22],31);
  b[22]+=ROL(b[23],4);
  b[25]+=ROL(b[24],25);
  b[24]+=ROL(b[25],10);
  b[27]+=ROL(b[26],19);
  b[26]+=ROL(b[27],16);
  b[29]+=ROL(b[28],13);
  b[28]+=ROL(b[29],22);
  b[31]+=ROL(b[30],7);
  b[30]+=ROL(b[31],28);
  for(i=0;i<WIDTH;i++) a[i]=b[i];
}

void ipht(unsigned long a[WIDTH]){
  unsigned long i,b[WIDTH];

  a[0]-=ROL(a[1],2);
  a[1]-=ROL(a[0],1);
  a[2]-=ROL(a[3],8);
  a[3]-=ROL(a[2],27);
  a[4]-=ROL(a[5],14);
  a[5]-=ROL(a[4],21);
  a[6]-=ROL(a[7],20);
  a[7]-=ROL(a[6],15);
  a[8]-=ROL(a[9],26),
  a[9]-=ROL(a[8],9);
  a[10]-=a[11];
  a[11]-=ROL(a[10],3);
  a[12]-=ROL(a[13],6);
  a[13]-=ROL(a[12],29);
  a[14]-=ROL(a[15],12);
  a[15]-=ROL(a[14],23);
  a[16]-=ROL(a[17],18);
  a[17]-=ROL(a[16],17);
  a[18]-=ROL(a[19],24);
  a[19]-=ROL(a[18],11);
  a[20]-=ROL(a[21],30);
  a[21]-=ROL(a[20],5);
  a[22]-=ROL(a[23],4);
  a[23]-=ROL(a[22],31);
  a[24]-=ROL(a[25],10);
  a[25]-=ROL(a[24],25);
```

```
a[26]-=ROL(a[27],16);
a[27]-=ROL(a[26],19);
a[28]-=ROL(a[29],22);
a[29]-=ROL(a[28],13);
a[30]-=ROL(a[31],28);
a[31]-=ROL(a[30],7);
for(i=0;i<(WIDTH/2);i++){
  b[2*i]=a[i];
  b[2*i+1]=a[i+(WIDTH/2)];
}
b[0]-=ROL(b[1],2);
b[1]-=ROL(b[0],1);
b[2]-=ROL(b[3],28);
b[3]-=ROL(b[2],23);
b[4]-=ROL(b[5],22);
b[5]-=ROL(b[4],13);
b[6]-=ROL(b[7],16);
b[7]-=ROL(b[6],3);
b[8]-=ROL(b[9],10);
b[9]-=ROL(b[8],25);
b[10]-=ROL(b[11],4);
b[11]-=ROL(b[10],15);
b[12]-=ROL(b[13],30);
b[13]-=ROL(b[12],5);
b[14]-=ROL(b[15],24);
b[15]-=ROL(b[14],27);
b[16]-=ROL(b[17],18);
b[17]-=ROL(b[16],17);
b[18]-=ROL(b[19],12);
b[19]-=ROL(b[18],7);
b[20]-=ROL(b[21],6);
b[21]-=ROL(b[20],29);
b[22]-=b[23];
b[23]-=ROL(b[22],19);
b[24]-=ROL(b[25],26);
b[25]-=ROL(b[24],9);
b[26]-=ROL(b[27],20);
b[27]-=ROL(b[26],31);
b[28]-=ROL(b[29],14);
b[29]-=ROL(b[28],21);
b[30]-=ROL(b[31],8);
b[31]-=ROL(b[30],11);
for(i=0;i<(WIDTH/2);i++){
```

```
  a[2*i]=b[i];
  a[2*i+1]=b[i+(WIDTH/2)];
}
a[0]-=ROL(a[1],2);
a[1]-=ROL(a[0],1);
a[2]-=ROL(a[3],12);
a[3]-=ROL(a[2],19);
a[4]-=ROL(a[5],22);
a[5]-=ROL(a[4],5);
a[6]-=a[7];
a[7]-=ROL(a[6],23);
a[8]-=ROL(a[9],10),
a[9]-=ROL(a[8],9);
a[10]-=ROL(a[11],20);
a[11]-=ROL(a[10],27);
a[12]-=ROL(a[13],30);
a[13]-=ROL(a[12],13);
a[14]-=ROL(a[15],8);
a[15]-=ROL(a[14],31);
a[16]-=ROL(a[17],18);
a[17]-=ROL(a[16],17);
a[18]-=ROL(a[19],28);
a[19]-=ROL(a[18],3);
a[20]-=ROL(a[21],6);
a[21]-=ROL(a[20],21);
a[22]-=ROL(a[23],16);
a[23]-=ROL(a[22],7);
a[24]-=ROL(a[25],26);
a[25]-=ROL(a[24],25);
a[26]-=ROL(a[27],4);
a[27]-=ROL(a[26],11);
a[28]-=ROL(a[29],14);
a[29]-=ROL(a[28],29);
a[30]-=ROL(a[31],24);
a[31]-=ROL(a[30],15);
for(i=0;i<(WIDTH/2);i++){
  b[2*i]=a[i];
  b[2*i+1]=a[i+(WIDTH/2)];
}
b[0]-=ROL(b[1],2);
b[1]-=ROL(b[0],1);
b[2]-=ROL(b[3],24);
b[3]-=ROL(b[2],15);
```

```
b[4]-=ROL(b[5],14);
b[5]-=ROL(b[4],29);
b[6]-=ROL(b[7],4);
b[7]-=ROL(b[6],11);
b[8]-=ROL(b[9],26);
b[9]-=ROL(b[8],25);
b[10]-=ROL(b[11],16);
b[11]-=ROL(b[10],7);
b[12]-=ROL(b[13],6);
b[13]-=ROL(b[12],21);
b[14]-=ROL(b[15],28);
b[15]-=ROL(b[14],3);
b[16]-=ROL(b[17],18);
b[17]-=ROL(b[16],17);
b[18]-=ROL(b[19],8);
b[19]-=ROL(b[18],31);
b[20]-=ROL(b[21],30);
b[21]-=ROL(b[20],13);
b[22]-=ROL(b[23],20);
b[23]-=ROL(b[22],27);
b[24]-=ROL(b[25],10);
b[25]-=ROL(b[24],9);
b[26]-=b[27];
b[27]-=ROL(b[26],23);
b[28]-=ROL(b[29],22);
b[29]-=ROL(b[28],5);
b[30]-=ROL(b[31],12);
b[31]-=ROL(b[30],19);
for(i=0;i<(WIDTH/2);i++){
  a[2*i]=b[i];
  a[2*i+1]=b[i+(WIDTH/2)];
}
a[0]-=ROL(a[1],2);
a[1]-=ROL(a[0],1);
a[2]-=ROL(a[3],20);
a[3]-=ROL(a[2],11);
a[4]-=ROL(a[5],6);
a[5]-=ROL(a[4],21);
a[6]-=ROL(a[7],24);
a[7]-=ROL(a[6],31);
a[8]-=ROL(a[9],10),
a[9]-=ROL(a[8],9);
a[10]-=ROL(a[11],28);
```

```
a[11]-=ROL(a[10],19);
a[12]-=ROL(a[13],14);
a[13]-=ROL(a[12],29);
a[14]-=a[15];
a[15]-=ROL(a[14],7);
a[16]-=ROL(a[17],18);
a[17]-=ROL(a[16],17);
a[18]-=ROL(a[19],4);
a[19]-=ROL(a[18],27);
a[20]-=ROL(a[21],22);
a[21]-=ROL(a[20],5);
a[22]-=ROL(a[23],8);
a[23]-=ROL(a[22],15);
a[24]-=ROL(a[25],26);
a[25]-=ROL(a[24],25);
a[26]-=ROL(a[27],12);
a[27]-=ROL(a[26],3);
a[28]-=ROL(a[29],30);
a[29]-=ROL(a[28],13);
a[30]-=ROL(a[31],16);
a[31]-=ROL(a[30],23);
for(i=0;i<(WIDTH/2);i++){
  b[2*i]=a[i];
  b[2*i+1]=a[i+(WIDTH/2)];
}
b[0]-=ROL(b[1],2);
b[1]-=ROL(b[0],1);
b[2]-=ROL(b[3],16);
b[3]-=ROL(b[2],7);
b[4]-=ROL(b[5],30);
b[5]-=ROL(b[4],13);
b[6]-=ROL(b[7],12);
b[7]-=ROL(b[6],19);
b[8]-=ROL(b[9],26);
b[9]-=ROL(b[8],25);
b[10]-=ROL(b[11],8),
b[11]-=ROL(b[10],31);
b[12]-=ROL(b[13],22);
b[13]-=ROL(b[12],5);
b[14]-=ROL(b[15],4);
b[15]-=ROL(b[14],11);
b[16]-=ROL(b[17],18);
b[17]-=ROL(b[16],17);
```

```
  b[18]-=b[19];
  b[19]-=ROL(b[18],23);
  b[20]-=ROL(b[21],14);
  b[21]-=ROL(b[20],29);
  b[22]-=ROL(b[23],28);
  b[23]-=ROL(b[22],3);
  b[24]-=ROL(b[25],10);
  b[25]-=ROL(b[24],9);
  b[26]-=ROL(b[27],24);
  b[27]-=ROL(b[26],15);
  b[28]-=ROL(b[29],6);
  b[29]-=ROL(b[28],21);
  b[30]-=ROL(b[31],20);
  b[31]-=ROL(b[30],27);
  for(i=0;i<WIDTH;i++) a[i]=b[i];
}
void crypt(unsigned long key[4+NUM_ROUNDS+2][WIDTH],\
unsigned long factors[WIDTH],\
unsigned long data[][WIDTH],unsigned long long size){

  unsigned  long i,j;
  unsigned long long m,n,o,carry1,carry2;

  for(m=0;m<size;m++){
    for(i=0;i<NUM_ROUNDS;i++){
      carry1=0;
      for(j=0;j<WIDTH;j++){
        data[m][j]^=key[2*i][j];
        data[m][j]=modmult(data[m][j],factors[j]);
        n=(unsigned long long) key[2*i+1][j];
        o=(unsigned long long) data[m][j];
        n+=o;
        n+=carry1;
        carry1=(n>>INT_LENGTH) & 1;
        data[m][j]=n & 0xFFFFFFFF;
        if((j & 7)==7) carry1=0;
      }
      pht(&data[m][0]);
    }
    carry1=0;
    carry2=0;
    for(j=0;j<WIDTH;j++){
      n=(unsigned long long)data[m][j];
```

```
      o=(unsigned long long)key[2*NUM_ROUNDS][j];
      n+=o;
      n+=carry1;
      carry1=(n>>INT_LENGTH) & 1;
      data[m][j]=n & 0xFFFFFFFF;
      data[m][j]=modmult(data[m][j],factors[j]);
      n=(unsigned long long) data[m][j];
      o=(unsigned long long) key[2*NUM_ROUNDS+1][j];
      n+=o;
      n+=carry2;
      carry2=(n>>INT_LENGTH) & 1;
      data[m][j]=n & 0xFFFFFFFF;
      if((j & 7)==7){
        carry1=0;
        carry2=0;
      }
    }
    for(i=0;i<NUM_ROUNDS;i++){
      carry1=0;
      ipht(&data[m][0]);
      for(j=0;j<WIDTH;j++){
        n=(unsigned long long) data[m][j];
        o=(unsigned long long) key[2*NUM_ROUNDS+2+2*i][j];
        n+=o;
        n+=carry1;
        carry1=(n>>INT_LENGTH) & 1;
        data[m][j]=n & 0xFFFFFFFF;
        data[m][j]=modmult(data[m][j],factors[j]);
        data[m][j]^=key[2*NUM_ROUNDS+3+2*i][j];
        if((j & 7)==7) carry1=0;
      }
    }
  }
}



void encrypt(unsigned long userkey[2][WIDTH],unsigned long long size,\
  unsigned long data[][WIDTH]){

  unsigned long key[4 *NUM_ROUNDS+2][WIDTH];
  unsigned long factors[WIDTH];
```

```
  key_schedule(userkey,key);
  encryption_factors(factors);
  crypt(key,factors,data,size);
}
void decrypt(unsigned long userkey[2][WIDTH],unsigned long long size,\
  unsigned long data[][WIDTH]){

  unsigned long key[4*NUM_ROUNDS+2][WIDTH];
  unsigned long factors[WIDTH];

  key_schedule(userkey,key);
  invert_keys(key);
  decryption_factors(factors);
  crypt(key,factors,data,size);
}

int main(){
  unsigned long i,j;
  unsigned long userkey[2][WIDTH], data[1][WIDTH];

  for(i=0;i<WIDTH;i++) data[0][i]=i;
  for(i=0;i<WIDTH;i++){
    for(j=0;j<2;j++) userkey[j][i]=32*j+i;
  }
  encrypt(userkey,1ULL,data);
  for(i=0;i<WIDTH;i++) printf("%lx\n",data[0][i]);
  scanf("%ld",&j);
  for(i=0;i<WIDTH;i++){
    for(j=0;j<2;j++) userkey[j][i]=32*j+i;
  }
  decrypt(userkey,1ULL,data);
  for(i=0;i<WIDTH;i++) printf("%lx\n",data[0][i]);
  scanf("%ld",&j);
return(0);
}
```

B Multipermutation

```
#include<stdio.h>
#define CHARROTL(x,a) (((x)<<(a))|((x)>>(8-(a))))


int main(){
```

```
unsigned int a[65536];
unsigned int i,j,k,l,m,n,o,p;
unsigned char b,c;
unsigned char ok[3];

  for(i=0;i<7;(i=i+2)){
    for(j=1;j<8;(j=j+2)){
      for(m=0;m<65536;m++) a[m]=0;
      ok[0]='O';
      ok[1]='K';
      ok[2]='\0';
      for(k=0;k<256;k++){
        for(l=0;l<256;l++){
          b=k;
          c=l;
          c+=CHARROTL(b,j);
          b+=CHARROTL(c,i);
          n=b;
          o=c;
          o|=(n<<8);
          a[o]++;
        }
      }
      for(m=0;m<65536;m++){
        if ((a[m]==0)||(a[m]>1)){
          ok[0]='N';
          ok[1]='O';
          ok[2]='\0';
          break;
        }
      }
      printf("%2d %2d %s  ",j,i,ok);
    }
  }
  scanf("%d",&p);
  return(0);
}
```

C Maximum bias for linear cryptanalysis

```
#include<stdio.h>


main(){
  unsigned long  keybites[2048], vari[2048];
  unsigned long i,j,k,l,m,mm,n,nn,auswahl[2048],ausvari[2048];
  unsigned long zahl[2048],p[2048],pos[2048];

  for(m=1;m<2048;m++){

    for(i=0;i<2048;i++){
      keybites[i]=0;
      vari[i]=0;
    }
    for(j=0;j<8;j++){
      for(i=1024;i<2048;i++){
        keybites[i]+=((i & 0xFF)+1);
        vari[i]+=1;
      }
      for(k=0;k<m;k++){
        auswahl[k]=keybites[2048-m+k];
        ausvari[k]=vari[2048-m+k];
      }
      for(k=2047;k>=m;k--){
        keybites[k]=keybites[k-m];
        vari[k]=vari[k-m];
      }
      for(k=0;k<m;k++){
        keybites[k]=auswahl[k];
        vari[k]=ausvari[k];
      }
    }
    for(j=0;j<2048;j++){
      keybites[j]+=((j & 0xFF)+1);
      vari[i]+=1;
    }
    for(k=0;k<m;k++){
      auswahl[k]=keybites[2048-m+k];
      ausvari[k]=vari[2048-m+k];
    }
```

```
  for(k=2047;k>=m;k--){
    keybites[k]=keybites[k-m];
    vari[k]=vari[k-m];
  }
  for(k=0;k<m;k++){
    keybites[k]=auswahl[k];
    vari[k]=ausvari[k];
  }
  for(j=0;j<8;j++){
    for(i=0;i<1024;i++){
      keybites[i]+=((i & 0xFF)+1);
      vari[i]+=1;
    }
    for(k=0;k<m;k++){
      auswahl[k]=keybites[2048-m+k];
      ausvari[k]=vari[2048-m+k];
    }
    for(k=2047;k>=m;k--){
      keybites[k]=keybites[k-m];
      vari[k]=vari[k-m];
    }
    for(k=0;k<m;k++){
      keybites[k]=auswahl[k];
      vari[k]=ausvari[k];
    }
  }
  l=1000000;r=1000;
  for(j=0;j<2048;j++){
    if(keybites[j]<l){
      l=keybites[j];
      r=vari[j];
      mm=j;
    }
  }
  zahl[m]=l;
  p[m]=r;
  pos[m]=mm;
  printf("%8lx\n",m);
}
l=0;
r=0;
mm=0;
for(i=0;i<2048;i++){
```

```
    if((zahl[i]-p[i])>(l-n)){
      l=zahl[i];
      n=p[i];
      nn=pos [i];
      mm=i;
    }
  }
  printf("%8ld\n",l);
  printf("%8ld\n",n);
  printf("%8ld\n",nn);
  printf("%ld",mm);
  scanf("%ld",&nnn);
}
```