2048XKS - A Software Oriented High Security Block Cipher

Dieter Schmidt*

21. September 2012

Abstract

The block cipher 2048XKS is a derivative of the block ciphers 1024 and 1024XKS, which in turn used the block ciphers MMB, SAFER and Blowfish as building blocks. The block cipher 2048XKS has a block size of 2048 bits and a key length of 4096 bits and 8192 bits, respectively. 2048XKS is a Substition-Permutation-Network (SPN). It is designed for 32 bit microprocessors with a hardware integer multiplier.

1 Introduction

IDEA and its predecessor PES (see [5, 6, 7]) was the first encryption algorithm, that used incompatible group operations to get diffusion and confusion (see [16]). IDEA uses addition modulo 2 (XOR), addition modulo 2^{16} and multiplication modulo $2^{16} + 1$. IDEA has not been broken in the open literature. However, IDEA is about twenty years old and due to its block length of 64 bits and keysize of 128 bits not state of the art.

Interestingly, James Bamford (see [1]) gives a note, what the NSA can achieve. He writes: *Another factor was the growing use of encryption and the NSA's inability, without spending excessive amounts of computer time and human energy, to solve commercial systems more complex than 256 bits.*

2048XKS has a block length of 2048 bits and a keyspace of 4096 bits or 8192 bits. It uses addition modulo 2 (XOR), addition modulo 2^{32}, addition modulo 2^{256}, circular shifts to the left of 32 bit integers, and multiplication modulo $2^{32} - 1$. Note that bitwise rotation (a

*Denkmalstrasse 16, D-57567 Daaden, Germany, dieterschmidt@usa.com

circular shift by a to n positions to the left) can be expressed as $2^n * a \mod 2^{32} - 1$, if $a < 2^{32} - 1$.

2048XKS is a generalised Substition-Permutation-Network (SPN). The s-boxes are based on multiplication modulo $2^{32} - 1$. This was invented by Daemen et al. in block cipher MMB (see [3]). The permutation is a modified Pseudo-Hadamard-Tranformation taken from SAFER (see [8, 9]). The key schedule is a modification from the key schedule of Blowfish (see [14]).

2048XKS has eight primary rounds, followed by a middle transformation, and eight secondary rounds. The diffusion layer of the secondary rounds is the inverse diffusion layer of the primary rounds. The result is that for decryption the same algorithm is used as for encryption. For decryption, one needs only the multiplicative inverse from the multiplication modulo $2^{32} - 1$ and the inverted keys.

Let denote: $\oplus$ addition modulo 2 (XOR), $\boxtimes$ multiplication modulo $2^{32} - 1$, + addition modulo 2^{32}, - subtraction modulo 2^{32}, a $\lll$ n the circular shift to the left of the 32 bit integer a by n positions.

2 The Algorithm

A bitstring of 2048 bit (block size) is partitioned into 64 pieces of 32 bits. Although the algorithm lends itself to 32 bit microprocessors, a 64 bit mecropressors can be used. If so, the key addition modulo 2 (XOR) and the key addition modulo 2^{256} is faster on a 64 bit microprocessor. However, the rest of the algorithm, the s-boxes and the diffusions layer use 32 bit integer. So the increase in speed on a 64 bit microprocessor is limited.

2.1 The Primary Round

The pieces of 32 bit are added modulo 2 (XOR) to the key. Then to 32 bit integers are transformed by the s-boxes.The s-boxes are all different from each other to avoid attacks based on symmetry. However, in each round the order of the s-boxes is the same. Then the 32 bit integers (64 pieces) are added modulo 2^{256} to the key. This addition modulo 2^{256} is done eight times to reach the block size. Finally the 64 pieces are fed through the diffusion layer, which is a derivative of the diffusion layer in SAFER.

2.2 The Middle Transformation

Eight pieces of 32 bit are packed together und added modulo 2^{256} to the key. This is done eight times to reach the blocksize. Then, the 64

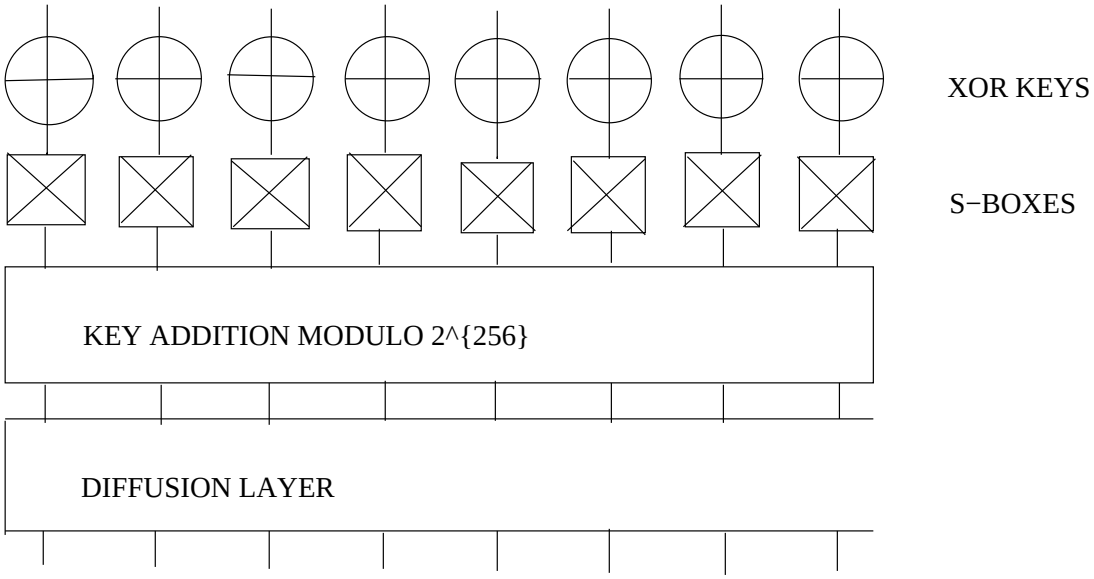

Figure 1: One eighth of the primary round

pieces of 32 bit are fed through the s-boxes. The order of the s-boxes is the same as in the primary rounds. At last, eight addition modulo 2^{256} of the 64 pieces are performed.

2.3 The Secondary Round

At first, the 64 pieces run through the inverse diffusion layer of the primary round. Then eight pieces are packed together to form an addition modulo 2^{256}. This is done eight times. The next stage is the transformation through the s-boxes. The order of the s-boxes are same as in the primary round. Then addition modulo 2 (XOR) is performed.

2.4 The S-Boxes

Multiplication modulo $2^n - 1$ as s-box was first used by Daemen et. al. [2, 3, 4]. 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:

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

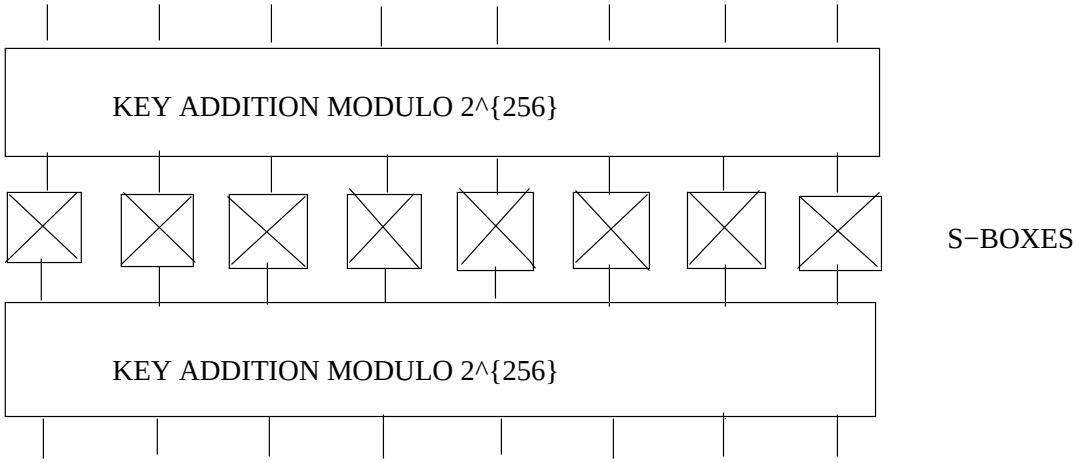

Figure 2: One eighth of the middle transformation

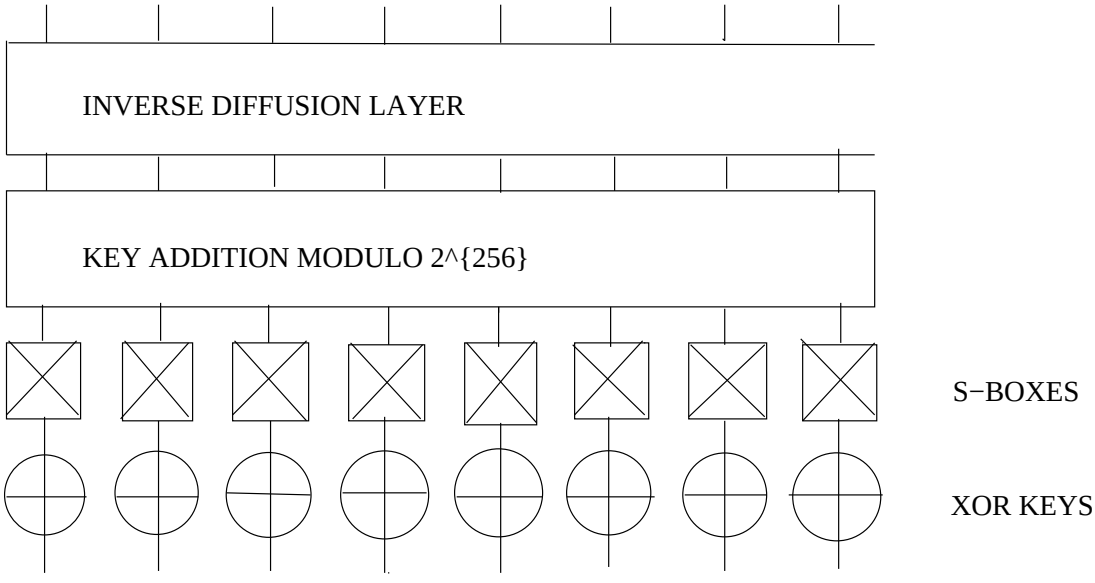

Figure 3: One eighth of the secondary round

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 [3] gives a wrong formula. It has been corrected in chapter 11 of Joan Daemens Ph.D. thesis [4]. 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 [3].

In 2048XKS the s-boxes, which contain the even rotation numbers, are same as in 1024. To fill the left half of the 64 s-boxes, we use the the factor 0x25F1CDB. There is an increase by two for the rotational values for each step ranging from 0 (left most s-box) to 30. The rotational values for the s-boxes with even rotational values are shown by the table:

position	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
rotation	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30

For the s-boxes, which have odd rotational values the table are shown here:

position	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
rotation	7	5	3	1	31	29	27	25	23	21	19	17	15	13	11	9

To fill the right half of the 64 s-boxes, we use the factor 229459604. This is the decryption factor of 0x25F1CDB or the multiplicatve inverse. The rotational values for s-boxes with even rotational are shown in the table:

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

For the s-boxes, which have odd rotational values the table are shown here:

position	33	35	37	39	41	43	45	47	49	51	53	55	57	59	61	63
rotation	17	15	13	11	9	7	5	3	1	31	29	27	25	23	21	19

All the circular shifts are taken to the left.

For further insight study the functions encryption_factors and decryption_factors at the beginning of the reference implementation.

2.5 Addition modulo 2^{256}

Addition modulo 2^{256} was introduced to give an upper bound for linear cryptanalysis. If we take [10], we can have an upper bound for linear cryptanalysis without being forced to examine the diffusion layer or the s-boxes. See subsection Key Schedule for further details.

2048XKS has a bit length of 2048 bits. This means addition modulo 2^{256} is applied eight times, from left to right, sometimes after the s-boxes, sometimes before the s-boxes. Since there are 64 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.

2.6 The Diffusion Layer

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

1. 64 blocks instead of eight.
2. Four bytes instead of one byte as primitive unit. See [15].
3. Before the addition primitive units are beeing rotated.

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. [15] 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 [11] 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 compenents of a. For example $a = 0x0F$ 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 [15] 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 [17, 19, 20]. Note the the natural unit of the diffusion layer of SAFER is a 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.

2.7 The Key Schedule

The key schedule of 1024 resembles that of IDEA. The round key of the first round is the user key. Note that 1024XKS has a 2048 bit round key, one half is applied before the s-boxes, one half after the s-boxes. The next round key is the previous round key rotated by 455 bits to the left. Obviously this is a linear function. The key schedule does prohibit linear cryptanalysis (see [13]). However, if a part of the key bits are known, then a part of all round keys are known. The key

schedule of 1024 is not one-way, but the key schedule of 2048XKS is one-way.

The round key generation of 2048XKS is as follows: First calculate the round keys in the 1024 manner, i.e. do the rotation by 455 bit to left. Then take a 2048 bit all zero string and let it pass through the algorithm. The resulting bit string is the first half of the first round key. Let the algorithm work in Output Feedback Mode (OFB). Each time the bit string has passed through the algorithm, a round key is assigned that bit string in ascending order. Given the number of primary rounds, secondary rounds and the middle transformation, the Output Feedback Mode (OFB) is applied 34 times. This key schedule was inspired by Blowfish [14].

However, that "forward mode" has a disadvantage: The first half of the first round key is assigned the bit string of the first OFB round. When encryption is applied, the first half of the first round key and the bit string have the same value. When they are added modulo 2 (XOR), the result is the all zero string. As the s-boxes left the zeroes unchanged, the first non-zero input is the second half of the first round key, or the second half of the user key. However, this is the only "error" that occurs in the "forward mode".

To avoid the "error", 2048XKS has a mode of key scheduling which I describe as the "backword mode". This means that Output Feedback Mode is still employed, but the round keys are assigned the value in descending order, i.e the last round key of the last secondary round is assigned the value first. This "backward mode" has not the same error as the "forward mode".

To distinguish the modes in the reference implementation, there is variable for the preprocessor named #define FORWARD. When the #define statement is true, then the key scheduling is in "forward mode". If the #define statement is not true, then the key scheduling is in "backward mode". To accomplish that, you could erase the #define statement or leave it as a commentary, i.e. to the beginning of the #define statement insert /* und the end of the statement insert */. The code of the reference implementation, which are influenced by the #define statement, are the functions encrypt and decrypt quite at the end of the reference implementation.

2.8 Decryption

For encryption and decryption the same algorithm is used. However, the multiplicative inverse must be used in the s-boxes.

To fill the left half of the 64 s-boxes, we use the the factor 229459604. There is an increase by two for the rotational values for each step rang-

ing from 0 (left most s-box) to 30. The rotational values for the s-boxes with even rotational values are shown by the table:

position	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
rotation	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30

For the s-boxes, which have odd rotational values the table are shown here:

position	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
rotation	7	5	3	1	31	29	27	25	23	21	19	17	15	13	11	9

To fill the right half of the 64 s-boxes, we use the factor 0x25F1CDB. This is the decryption factor of 229459604 or the multiplicative inverse. The rotational values for s-boxes with even rotational are shown in the table:

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

For the s-boxes, which have odd rotational values the table are shown here:

position	33	35	37	39	41	43	45	47	49	51	53	55	57	59	61	63
rotation	17	15	13	11	9	7	5	3	1	31	29	27	25	23	21	19

All the circular shifts are taken to the right.

See the function decryption_factors of the reference implementation for details.

Also the keys have to be inverted. While XOR is self-inverse 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.

3 Implementation Consideration

The reference implementataion is programmed in the language C. Unfornately, this lacks instructions, which in assembler (processor language) are quite common. This makes the reference implemenation a bit clumsy. In the reference implementaion you must have 64 bit variable to allow for the carry. This 64 bit variables are then shifted

to right by 32 digits to get added in the next round of calculations. This is true for the addtion modulo 2^{256} and the s-boxes (see functions modmult and crypt). Also the rotations (see function pht and ipht) are made up with the shift to left, a shift to right, and and OR (see the definitions of the reference implementation). For example, in the Intel Architecture for 32 bit microprocessors (IA32), the s-boxes are pogrammed in this way:

```
MOV EAX,data
MOV EBX,factor
MUL EBX
ADD EAX,EDX
ADC EAX,0
MOV data,EAX
```

Also the addition modulo 2^{256} looks like this:

```
MOV EAX,data0
MOV EBX,key0
ADD EAX,BX
MOV data0,EAX
MOV EAX,data1
MOV EBX,key1
ADC EAX,EBX
MOV data1,EAX
          .
          .
          .
MOV EAX,data7
MOV EBX,key7
ADC EAX,EBX
MOV data7,EAX
```

The rotations of the diffusion layer are a part of the IA32. For the first values of the function pht (a_0 and a_1), the programming looks like this:

```
MOV EAX,a0
MOV EBX,a1
MOV ECX,EAX
ROL ECX,1
ADD EBX,ECX
MOV EDX,EBX
```

```
ROL EDX,2
ADD EAX,EDX
MOV a1,EBX
MOV a0,EAX
```

In the ideal case, the programming of 2048XKS should be all in assembler.

4 Intellectual Property

2048XKS is free. The reference implementation is covered by the GNU General Public License.

References

[1] Bamford, James: *The Shadow Factory. The Ultra-Secret NSA from 9/11 to the Eavesdropping on America*, p. 138, First Anchor Books Edition, New York, 2009

[2] Daemen, Joan; Luc von Linden, Rene Govaerts and Joos Vandewalle: Propagation Properties of Multiplication Modulo $2^n - 1$, *Proceedings of the 13th Symposium on Information Theory in the Benelux*, Werkgemeenschaft vaar Informatie- en Communicatietheory, available from `http://www.cosic.esat.kuleuven.be/publication/static/1992.html`, 1992

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

[4] Daemen, Joan: *Cipher and Hash Function Design, Strategies based on linear and differential Cryptanalysis*,Ph.D. Thesis, KU Leuven, Belgium, 1995

[5] Lai, Xuejia and James Massey: A Proposal for a New Block Encryption Standard, in Ivan Damgård (Ed.): *Advances in Cryptology - EUROCRYPT '90*, Springer Verlag, Berlin, 1991

[6] Lai, Xuejia, James Massey and Sean Murpy: Markov Ciphers and Differential Cryptanalysis, in Donald Davies (Ed.): *Advances in Cryptology - EUROCRYPT '91*, Springer Verlag, Berlin, 1991

[7] Lai, Xuejia: *On the Design and Security of Block Ciphers*, Ph.D. thesis at ETH Zürich, Switzerland, Hartung-Gorre Verlag, Konstanz, Germany, 1992

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

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

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

[10] Mukhopadhyay, Debdeep and Dipanwita RoyChowdhury: *Key Mixing in Block Cipher through Addition modulo* 2^n, ePrint Archive of the IACR, Report 2005/383, available from `http://eprint.iacr.org`

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

[12] Schmidt, Dieter: *1024 - A High Security Software Oriented Block Cipher*, ePrint Archive of the IACR, Report 2009/104, available from `http://eprint.iacr.org`

[13] Schmidt, Dieter: *1024XKS - A High Security Software Oriented Block Cipher*, ePrint Archive of th IACR, Report 2010/162, available from`http://eprint.iacr.org`

[14] Schneier, Bruce: Description of a New Variable Length Key 64-Bit Block Cipher, in Ross Anderson (Ed.): *Fast Sofware Encryption - Cambrigde Security Workshop*, Springer Verlag, Berlin, 1994

[15] 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`

[16] Shannon, Claude Elmwood: *Communication Theory of Secrecy Systems* Bell Systems Technical Journal, v. 28, n. 4, 1949, pp. 656-715, Reprint in Sloane, N.J.A. and A. Wyner (Eds.): *Claude Elwood Shannon: Collected Papers*, IEEE Press, Picataway, USA, 1993

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

[18] Vaudenay, Serge: On the Weak Keys of Blowfisch, in Dieter Gollmann (Ed.): *Fast Software Encryption*, Third International Workshop, Cambridge, Springer Verlag, Berlin, 1996

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

[20] Vaudenay, Serge: On the Security of CS-Cipher, in Lars Knudsen(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 64
#define ROTROUND 455

#define FORWARD
#define BIG_KEY

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

  e_factors[0]=0x025F1CDB;
  for(i=0;i<(WIDTH/4);i++){
    if(i!=0) e_factors[2*i]=ROL(e_factors[0],2*i);
    e_factors[2*i+1]=ROL(e_factors[0],(WIDTH/2+7-2*i)%(WIDTH/2));
  }
  e_factors[WIDTH/2]=229459604;
  for(i=0;i<(WIDTH/4);i++){
    if(i!=0) e_factors[WIDTH/2+2*i]=ROL(e_factors[WIDTH/2],2*i);
    e_factors[WIDTH/2+2*i+1]=\
    ROL(e_factors[WIDTH/2],(WIDTH/2+17-2*i)%(WIDTH/2));
  }
}

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

```
  d_factors[0]=229459604;
  for(i=0;i<(WIDTH/4);i++){
    if(i!=0) d_factors[2*i]=ROR(d_factors[0],2*i);
    d_factors[2*i+1]=ROR(d_factors[0],(WIDTH/2+7-2*i)%(WIDTH/2));
  }
  d_factors[WIDTH/2]=0x025F1CDB;
  for(i=0;i<(WIDTH/4);i++){
    if(i!=0) d_factors[WIDTH/2+2*i]=ROR(d_factors[WIDTH/2],2*i);
    d_factors[WIDTH/2+2*i+1]=\
    ROR(d_factors[WIDTH/2],(WIDTH/2+17-2*i)%(WIDTH/2));
  }
}

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;
    }
  }
}

#if defined(BIG_KEY)

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

  for(i=0;i<4;i++){
    for(j=0;j<WIDTH;j++){
      key[i][j]=user_key[i][j];
    }
  }
  for(i=1;i<NUM_ROUNDS;i++){
    for(j=0;j<WIDTH;j++) {
      key[4*i+3][j]=(key[4*(i-1)+((j+((WIDTH*INT_LENGTH-ROTROUND)\
      /INT_LENGTH))/WIDTH*3)]\
      [(j+((WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH))\
      %WIDTH]<<(ROTROUND%INT_LENGTH))|\
      (key[4*(i-1)+((j+(WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH+1)/WIDTH*3)]\
      [(j+(WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH+1)%WIDTH]>>\
      (INT_LENGTH-ROTROUND%INT_LENGTH));

      key[4*i+2][j]=(key[4*(i-1)+3-(j+((WIDTH*INT_LENGTH-ROTROUND)\
      /INT_LENGTH))/WIDTH]\
      [(j+((WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH))\
      %WIDTH]<<(ROTROUND%INT_LENGTH))\
      |(key[4*(i-1)+3-(j+(WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH+1)/WIDTH]\
      [(j+(WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH+1)%WIDTH]>>\
```

```
        (INT_LENGTH-ROTROUND%INT_LENGTH));

        key[4*i+1][j]=(key[4*(i-1)+2-(j+((WIDTH*INT_LENGTH-ROTROUND)\
        /INT_LENGTH))/WIDTH]\
        [(j+((WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH))\
        %WIDTH]<<(ROTROUND%INT_LENGTH))\
        |(key[4*(i-1)+2-(j+(WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH+1)/WIDTH]\
        [(j+(WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH+1)%WIDTH]>>\
        (INT_LENGTH-ROTROUND%INT_LENGTH));


        key[4*i][j]=(key[4*(i-1)+1-(j+((WIDTH*INT_LENGTH-ROTROUND)\
        /INT_LENGTH))/WIDTH]\
        [(j+((WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH))\
        %WIDTH]<<(ROTROUND%INT_LENGTH))\
        |(key[4*(i-1)+1-(j+(WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH+1)/WIDTH]\
        [(j+(WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH+1)%WIDTH]>>\
        (INT_LENGTH-ROTROUND%INT_LENGTH));

      }
    }
    for(j=0;j<WIDTH;j++){
        key[33][j]=(key[30-(j+((WIDTH*INT_LENGTH-ROTROUND)\
        /INT_LENGTH))/WIDTH]\
        [(j+((WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH))\
        %WIDTH]<<(ROTROUND%INT_LENGTH))\
        |(key[30-(j+(WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH+1)/WIDTH]\
        [(j+(WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH+1)%WIDTH]>>\
        (INT_LENGTH-ROTROUND%INT_LENGTH));


        key[32][j]=(key[29-(j+((WIDTH*INT_LENGTH-ROTROUND)\
        /INT_LENGTH))/WIDTH]\
        [(j+((WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH))\
        %WIDTH]<<(ROTROUND%INT_LENGTH))\
        |(key[29-(j+(WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH+1)/WIDTH]\
        [(j+(WIDTH*INT_LENGTH-ROTROUND)/INT_LENGTH+1)%WIDTH]>>\
        (INT_LENGTH-ROTROUND%INT_LENGTH));
    }
}

#else
```

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

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

#endif

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);

a[33]+=ROL(a[32],1);
a[32]+=ROL(a[33],2);
a[35]+=ROL(a[34],7);
a[34]+=ROL(a[35],16);
a[37]+=ROL(a[36],13),
a[36]+=ROL(a[37],30);
a[39]+=ROL(a[38],19);
a[38]+=ROL(a[39],12);
a[41]+=ROL(a[40],25);
a[40]+=ROL(a[41],26);
a[43]+=ROL(a[42],31);
a[42]+=ROL(a[43],8);
a[45]+=ROL(a[44],5);
a[44]+=ROL(a[45],22);
a[47]+=ROL(a[46],11);
a[46]+=ROL(a[47],4);
a[49]+=ROL(a[48],17);
a[48]+=ROL(a[49],18),
```

```
a[51]+=ROL(a[50],23);
a[50]+=a[51];
a[53]+=ROL(a[52],29);
a[52]+=ROL(a[53],14);
a[55]+=ROL(a[54],3);
a[54]+=ROL(a[55],28);
a[57]+=ROL(a[56],9);
a[56]+=ROL(a[57],10);
a[59]+=ROL(a[58],15);
a[58]+=ROL(a[59],24);
a[61]+=ROL(a[60],21);
a[60]+=ROL(a[61],6);
a[63]+=ROL(a[62],27);
a[62]+=ROL(a[63],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);

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

a[33]+=ROL(a[32],1);
a[32]+=ROL(a[33],2);
a[35]+=ROL(a[34],15);
```

```
a[34]+=ROL(a[35],24);
a[37]+=ROL(a[36],29);
a[36]+=ROL(a[37],14);
a[39]+=ROL(a[38],11);
a[38]+=ROL(a[39],4);
a[41]+=ROL(a[40],25);
a[40]+=ROL(a[41],26);
a[43]+=ROL(a[42],7);
a[42]+=ROL(a[43],16);
a[45]+=ROL(a[44],21);
a[44]+=ROL(a[45],6);
a[47]+=ROL(a[46],3);
a[46]+=ROL(a[47],28);
a[49]+=ROL(a[48],17);
a[48]+=ROL(a[49],18);
a[51]+=ROL(a[50],31);
a[50]+=ROL(a[51],8);
a[53]+=ROL(a[52],13);
a[52]+=ROL(a[53],30);
a[55]+=ROL(a[54],27);
a[54]+=ROL(a[55],20);
a[57]+=ROL(a[56],9);
a[56]+=ROL(a[57],10),
a[59]+=ROL(a[58],23);
a[58]+=a[59];
a[61]+=ROL(a[60],5);
a[60]+=ROL(a[61],22);
a[63]+=ROL(a[62],19);
a[62]+=ROL(a[63],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);

b[33]+=ROL(b[32],1);
b[32]+=ROL(b[33],2);
b[35]+=ROL(b[34],19);
b[34]+=ROL(b[35],12);
b[37]+=ROL(b[36],5);
b[36]+=ROL(b[37],22);
b[39]+=ROL(b[38],23);
b[38]+=b[39];
b[41]+=ROL(b[40],9);
b[40]+=ROL(b[41],10);
b[43]+=ROL(b[42],27);
b[42]+=ROL(b[43],20);
b[45]+=ROL(b[44],13);
b[44]+=ROL(b[45],30);
b[47]+=ROL(b[46],31);
b[46]+=ROL(b[47],8);
b[49]+=ROL(b[48],17);
b[48]+=ROL(b[49],18);
b[51]+=ROL(b[50],3);
```

```
b[50]+=ROL(b[51],28);
b[53]+=ROL(b[52],21);
b[52]+=ROL(b[53],6);
b[55]+=ROL(b[54],7);
b[54]+=ROL(b[55],16);
b[57]+=ROL(b[56],25);
b[56]+=ROL(b[57],26);
b[59]+=ROL(b[58],11);
b[58]+=ROL(b[59],4);
b[61]+=ROL(b[60],29);
b[60]+=ROL(b[61],14);
b[63]+=ROL(b[62],15);
b[62]+=ROL(b[63],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);

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

b[33]+=ROL(b[32],1);
b[32]+=ROL(b[33],2);
b[35]+=ROL(b[34],27);
b[34]+=ROL(b[35],8);
b[37]+=ROL(b[36],21);
b[36]+=ROL(b[37],14);
b[39]+=ROL(b[38],15);
```

```
  b[38]+=ROL(b[39],20);
  b[41]+=ROL(b[40],9);
  b[40]+=ROL(b[41],26);
  b[43]+=ROL(b[42],3);
  b[42]+=b[43];
  b[45]+=ROL(b[44],29);
  b[44]+=ROL(b[45],6);
  b[47]+=ROL(b[46],23);
  b[46]+=ROL(b[47],12);
  b[49]+=ROL(b[48],17);
  b[48]+=ROL(b[49],18);
  b[51]+=ROL(b[50],11);
  b[50]+=ROL(b[51],24);
  b[53]+=ROL(b[52],5);
  b[52]+=ROL(b[53],30);
  b[55]+=ROL(b[54],31);
  b[54]+=ROL(b[55],4);
  b[57]+=ROL(b[56],25);
  b[56]+=ROL(b[57],10);
  b[59]+=ROL(b[58],19);
  b[58]+=ROL(b[59],16);
  b[61]+=ROL(b[60],13);
  b[60]+=ROL(b[61],22);
  b[63]+=ROL(b[62],7);
  b[62]+=ROL(b[63],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);

a[32]-=ROL(a[33],2);
a[33]-=ROL(a[32],1);
a[34]-=ROL(a[35],8);
a[35]-=ROL(a[34],27);
a[36]-=ROL(a[37],14);
a[37]-=ROL(a[36],21);
a[38]-=ROL(a[39],20);
a[39]-=ROL(a[38],15);
a[40]-=ROL(a[41],26),
a[41]-=ROL(a[40],9);
a[42]-=a[43];
a[43]-=ROL(a[42],3);
a[44]-=ROL(a[45],6);
a[45]-=ROL(a[44],29);
a[46]-=ROL(a[47],12);
a[47]-=ROL(a[46],23);
a[48]-=ROL(a[49],18);
a[49]-=ROL(a[48],17);
a[50]-=ROL(a[51],24);
a[51]-=ROL(a[50],11);
```

```
a[52]-=ROL(a[53],30);
a[53]-=ROL(a[52],5);
a[54]-=ROL(a[55],4);
a[55]-=ROL(a[54],31);
a[56]-=ROL(a[57],10);
a[57]-=ROL(a[56],25);
a[58]-=ROL(a[59],16);
a[59]-=ROL(a[58],19);
a[60]-=ROL(a[61],22);
a[61]-=ROL(a[60],13);
a[62]-=ROL(a[63],28);
a[63]-=ROL(a[62],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);

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

a[32]-=ROL(a[33],2);
a[33]-=ROL(a[32],1);
a[34]-=ROL(a[35],12);
a[35]-=ROL(a[34],19);
a[36]-=ROL(a[37],22);
a[37]-=ROL(a[36],5);
a[38]-=a[39];
```

```
a[39]-=ROL(a[38],23);
a[40]-=ROL(a[41],10),
a[41]-=ROL(a[40],9);
a[42]-=ROL(a[43],20);
a[43]-=ROL(a[42],27);
a[44]-=ROL(a[45],30);
a[45]-=ROL(a[44],13);
a[46]-=ROL(a[47],8);
a[47]-=ROL(a[46],31);
a[48]-=ROL(a[49],18);
a[49]-=ROL(a[48],17);
a[50]-=ROL(a[51],28);
a[51]-=ROL(a[50],3);
a[52]-=ROL(a[53],6);
a[53]-=ROL(a[52],21);
a[54]-=ROL(a[55],16);
a[55]-=ROL(a[54],7);
a[56]-=ROL(a[57],26);
a[57]-=ROL(a[56],25);
a[58]-=ROL(a[59],4);
a[59]-=ROL(a[58],11);
a[60]-=ROL(a[61],14);
a[61]-=ROL(a[60],29);
a[62]-=ROL(a[63],24);
a[63]-=ROL(a[62],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);

b[32]-=ROL(b[33],2);
b[33]-=ROL(b[32],1);
b[34]-=ROL(b[35],24);
b[35]-=ROL(b[34],15);
b[36]-=ROL(b[37],14);
b[37]-=ROL(b[36],29);
b[38]-=ROL(b[39],4);
b[39]-=ROL(b[38],11);
b[40]-=ROL(b[41],26);
b[41]-=ROL(b[40],25);
b[42]-=ROL(b[43],16);
b[43]-=ROL(b[42],7);
b[44]-=ROL(b[45],6);
b[45]-=ROL(b[44],21);
b[46]-=ROL(b[47],28);
b[47]-=ROL(b[46],3);
b[48]-=ROL(b[49],18);
b[49]-=ROL(b[48],17);
b[50]-=ROL(b[51],8);
b[51]-=ROL(b[50],31);
b[52]-=ROL(b[53],30);
b[53]-=ROL(b[52],13);
```

```
b[54]-=ROL(b[55],20);
b[55]-=ROL(b[54],27);
b[56]-=ROL(b[57],10);
b[57]-=ROL(b[56],9);
b[58]-=b[59];
b[59]-=ROL(b[58],23);
b[60]-=ROL(b[61],22);
b[61]-=ROL(b[60],5);
b[62]-=ROL(b[63],12);
b[63]-=ROL(b[62],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);

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

b[32]-=ROL(b[33],2);
b[33]-=ROL(b[32],1);
b[34]-=ROL(b[35],16);
b[35]-=ROL(b[34],7);
b[36]-=ROL(b[37],30);
b[37]-=ROL(b[36],13);
b[38]-=ROL(b[39],12);
b[39]-=ROL(b[38],19);
b[40]-=ROL(b[41],26);
b[41]-=ROL(b[40],25);
b[42]-=ROL(b[43],8),
```

```
  b[43]-=ROL(b[42],31);
  b[44]-=ROL(b[45],22);
  b[45]-=ROL(b[44],5);
  b[46]-=ROL(b[47],4);
  b[47]-=ROL(b[46],11);
  b[48]-=ROL(b[49],18);
  b[49]-=ROL(b[48],17);
  b[50]-=b[51];
  b[51]-=ROL(b[50],23);
  b[52]-=ROL(b[53],14);
  b[53]-=ROL(b[52],29);
  b[54]-=ROL(b[55],28);
  b[55]-=ROL(b[54],3);
  b[56]-=ROL(b[57],10);
  b[57]-=ROL(b[56],9);
  b[58]-=ROL(b[59],24);
  b[59]-=ROL(b[58],15);
  b[60]-=ROL(b[61],6);
  b[61]-=ROL(b[60],21);
  b[62]-=ROL(b[63],20);
  b[63]-=ROL(b[62],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[][WIDTH],unsigned long long size,\
  unsigned long data[][WIDTH]){

  unsigned long key[4 *NUM_ROUNDS+2][WIDTH];
  unsigned long factors[WIDTH];
  unsigned long i,j,intermediate[1][WIDTH];


  key_schedule(userkey,key);
  encryption_factors(factors);
  for(i=0;i<WIDTH;i++){
    intermediate[0][i]=0;
  }
  #if defined(FORWARD)
  for(i=0;i<(4*NUM_ROUNDS+2);i++){
    crypt(key,factors,intermediate,1ULL);
    for(j=0;j<WIDTH;j++){
      key[i][j]=intermediate[0][j];
    }
  }
   #else
  for(i=(4*NUM_ROUNDS+2);i>0;i--){
    crypt(key,factors,intermediate,1ULL);
    for(j=0;j<WIDTH;j++){
      key[i-1][j]=intermediate[0][j];
    }
  }
  #endif

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

  unsigned long key[4*NUM_ROUNDS+2][WIDTH];
  unsigned long factors[WIDTH];
  unsigned long i,j,intermediate[1][WIDTH];

  key_schedule(userkey,key);
```

```
  encryption_factors(factors);
  for(i=0;i<WIDTH;i++){
    intermediate[0][i]=0;
  }
  #if defined(FORWARD)
  for(i=0;i<(4*NUM_ROUNDS+2);i++){
    crypt(key,factors,intermediate,1ULL);
    for(j=0;j<WIDTH;j++){
      key[i][j]=intermediate[0][j];
    }
  }
   #else
  for(i=(4*NUM_ROUNDS+2);i>0;i--){
    crypt(key,factors,intermediate,1ULL);
    for(j=0;j<WIDTH;j++){
      key[i-1][j]=intermediate[0][j];
    }
  }
  #endif
  invert_keys(key);
  decryption_factors(factors);
  crypt(key,factors,data,size);
}

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

#if defined(BIG_KEY)
  unsigned long userkey[4][WIDTH];
#else
  unsigned long userkey[2][WIDTH];
#endif

  for(i=0;i<WIDTH;i++) data[0][i]=i;
#if defined(BIG_KEY)
  for(i=0;i<WIDTH;i++){
    for(j=0;j<4;j++) userkey[j][i]=WIDTH*j+i;
  }
#else
  for(i=0;i<WIDTH;i++){
    for(j=0;j<2;j++) userkey[j][i]=WIDTH*j+1;
  }
```

```
#endif

  encrypt(userkey,1ULL,data);
  for(i=0;i<WIDTH;i++) printf("%lx\n",data[0][i]);
  scanf("%ld",&j);
#if defined(BIG_KEY)
  for(i=0;i<WIDTH;i++){
    for(j=0;j<4;j++) userkey[j][i]=WIDTH*j+i;
  }
#else
 for(i=0;i<WIDTH;i++){
    for(j=0;j<2;j++) userkey[j][i]=WIDTH*j+1;
  }
#endif
  decrypt(userkey,1ULL,data);
  for(i=0;i<WIDTH;i++) printf("%lx\n",data[0][i]);
  scanf("%ld",&j);
return(0);
}
```