# 1024XKS - A High Security Software Oriented Block Cipher Revisited

Dieter Schmidt*

March 27, 2010

**Abstract**

The block cipher 1024 has a key schedule that somehow resembles that of IDEA. The user key is cyclicly shifted by a fixed amount to form the round keys. In the key schedule of IDEA this has lead to weak keys. The primitive key schedule from 1024 may lead also to attacks with related keys. Although to the knowlegde of the author weak keys or attacks with related keys have not been published, there is a need to put things right. The new one-way key schedule of 1024XKS (eXtended Key Schedule) has pseudo-random round keys, which are obtained by using the cipher as randomizer. Apart from that, the user key has now two sizes, 2048 bit and 4096 bit. Also the order of the s-boxes have been changed to thwart attacks based on symmetry.

## 1 Introduction

When Claude Shannon, once a U.S. government cryptanalyst, [49] passed World War II in review, he noted that cryptographic machines like the ENIGMA enciphered only one letter of the plaintext to become ciphertext. He also showed, that when the key had the same size like the plaintext or exceeded it, an attacker even with unlimited computing power could not decipher the ciphertext. This was good news for government agencies in charge of information security. They quickly build apparatuses, mainly based on radioactive decay, to produce the lengthy bit sequences.

But back to Shannons other observation. He demanded, that a letter of the ciphertext should be dependent on as much plaintext as possible. He called it "diffusion". Shannon demanded also, that a

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

letter of the ciphertext be dependent on the plaintext in a complicated manner. He called this "confusion". I can see no reason that also the key is subject to "confusion" with regard to the ciphertext.

The rest of the article is organized as follows: Section 2 lists other block cipher algorithms and their key schedule. Section 3 looks into the block cipher 1024 and its original key schedule and s-box order. It is in fact a partial reprint of [45]. Section 4 shows the differencies of 1024 and 1024XKS, especially the new key schedule, the different key sizes and the different order of the s-boxes. Section 5 gives a report on intellectual property of 1024 and 1024XKS.

# 2 Block Ciphers and Key Schedules

## 2.1 Lucifer

But while government agencies were keen to use his first remark, the totally failed on his last remark, at least to my knowlegde. But along came Horst Feistel in 1973, an employee of IBM. He saw that the ever increasing traffic of computer data would be an easy target for espionage. He designed Lucifer [18, 64], a block cipher with 128 bits block size and 128 bits key length and 16 rounds. Lucifer is a product cipher [66], the data is substituted by a 32 4 bit * 4 bit s-boxes. The output of the s-box is permuted or transposed to form the next input for a s-box. The permutation is not used after the last s-box. The key is used to select one of the two s-boxes. Lucifer is a Substitution-Permutation-Network (SPN) [67]. It should be noted, that the s-box of Lucifer must be bijective, i.e. a permutation. The permutation of a SPN could be called a transposition. The F-function in Feistel ciphers [59], where substitution and permutation take place, must not be bijective, but can be a function, see for example [1, 46, 43].

## 2.2 DES

But not only the employees of a computer firm had woken up. In the early seventiehs the U.S. National Bureau of Standards (NBS, now National Institute of Standards and Technologie, NIST) had come to the same conclusion as the cryptographers from IBM. In two solicitations the NBS asked in the Federal Register from 15th of May 1973 and the Federal Register from 27th of August 1974 for a cryptographic algorithm. Finally, the cryptographers from IBM came with a tender. Although IBM had a patent pending on the algorithm, IBM was ready to share the patent with others. In two solicitations from 17th of March 1975 and on 1st August 1975 NBS asked for comments.

In 1976 NBS held two workshops on the cryptographic algorithm and despite the criticism NBS declared Data Encryption Standard (DES) on 23th November 1976 a federal standard and fit for the encryption of sensitive, but not classified, data. The criticism of the DES had largely two points:

1. A key length of 56 bits, compared with Lucifer's 128 bits.
2. The design criteria of a crucial part of the DES, the s-boxes, the only non-linear part of the algorithm, remained secret.

For the whole story about DES the reader is referred to literature, especially the role of the National Security Agency (NSA), see [2, 47].

DES has a block size of 64 bits and a keylength of 56 bits and is a balanced Feistel Network. At the beginning, all the data will go through an initial permutation, which has no cryptographic relevance, but is merely there to thwart a software implementation. At the end of the encryption process, all the data must pass through the reverse initial permutation. For the encryption process, take the two halves of the data block. The lefthand block (32 bits) is copied as the input for the F-function. The data of the F-function is then expanded to 48 bits and added modulo 2 (XOR) to the round key. After that the data is digested through eight 6 bit * 4 bit s-boxes. A 32 bit permutation (transposition) is performed. The F-function is now completed and its output is added modulo 2 (XOR) to the right half of the data block. Now the encryption process takes the right half of the data block and makes it the input of the F-function. The output of the F-function is addded modulo 2 (XOR) to the left half of the data block. The F-function (round) is repeated 16 times, eight times by using the left half of data block as input and eight times by using the right half of data bock as input. After the last round, the left and right halves of the data block are interchanged. The output of the F-function is added modulo 2 (XOR) to the half block that was not the input. The whole algorithm in detail is here [38, 47, 57].

The key space of DES is 64 bits. However, only 56 bits are the effective key space, since 8 bits of the 64 bits are parity bits. The effective key space undergoes a permutation or transposition. The key is then split in halves, i.e. two keys with 28 bits. These keys are shifted cyclicly every round by 1 bit or 2 bits. From each of the two keys 24 bits are extracted und together they form the round key. The round key is added modulo 2 (XOR) to the expanded data bits of the F-function.

The operations in DES are linear, except the s-boxes. They key schedule of DES due to its linearity has some interesting properties:

four keys are weak keys. Encrypting with these four keys is an involution. When $E_{ki}$ is encryption with key $ki$ and $P$ is the plaintext,then

$$E_{ki}(E_{ki}(P)) = P \tag{1}$$

Six of the keys are semi-weak. Let $D_{k1}$ decryption with key $k1$. Encryption with one pair of semi-weak keys $k2$, operates idencitally to decryption with another $k1$.

$$E_{k1}(E_{k2}(P)) = P \tag{2}$$
$$E_{k2} = D_{k1} \tag{3}$$

And finally: Let $C$ the be ciphertext. Let $\overline{C}$ the bitwise complement of $C$. Then:

$$E_{\overline{k}}(\overline{P}) = \overline{C} \tag{4}$$
$$E_k(P) = C \tag{5}$$

This is the complementation property of DES. As a consequence, a chosen-plaintext-attack needs only half of the keys to check, i.e. $2^{55}$ instead of $2^{56}$ computational complexity.

As I mentioned earlier, the 56 bits of the DES user key are split into halves of 28 bits. The 28 bits are rotated by 1 bit or 2 bits. For all the 16 rounds of the DES, the shift in total is 28 bits. For encryption, the shifts are to the left, for decryption, the shifts are to the right. Obviously this makes computation of decryption und encryption easier, since gates in 1977 were scarce and expensive. Of each 28 bits, 24 bits are selected, 48 bits in total, to form the key of the F-function or the round key. The shifts in dependance of the rounds are as follows:

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

I can see no obvious pattern of the shifts depending on the rounds. [5] clames, that DES is not susceptible to attacks with related keys. Also [22] makes no mention of the DES. However [4] makes use that the key schedule of DES is not one-way, i.e. whole or partial knowlegde of one round key gives partial knowledge of another round key.

However, it is easy to find symmetries of shifts vs. rounds distribution, for example:

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

or

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

These examples of the shifts vs. rounds distribution have a symmetry and should be susceptible to related key attacks. It seems to me, that the cryptographers at IBM or the cryptographers at the NSA or both had some knowledge about related keys in 1975, but they kept silent. Open research had to wait for 18 years for [5]. It is intereseting that Don Coppersmith of IBM, when Eli Biham and Adi Shamir reinvented differential cryptanalysis in [4] at the beginning of the 1990ties, broke the silence and said, they were aware of differential cryptanalysis in 1975, but remained silent for national security reasons. See the last paragraph of the preface in [4].

## 2.3 FEAL

The first block cipher with a one-way key schedule, at least to my knowledge, is the FEAL cipher [50, 58]. The FEAL Cipher is a balanced Feistel cipher. It uses addition modulo 2 (XOR), a rotation by 2 bits to left and addition modulo 256. The only non-linear operation in FEAL is the addition modulo 256. The key schedule uses a modified F-function of FEAL to create pseudo-random round keys and is one-way.

As the only non-linear operation is addition modulo 256 or one byte addition, FEAL became an easy prey for Eli Biham and Adi Shamir, who were at that time developing differential cryptanalysis [3]. The FEAL inventors tried to make the algorithm immune from differential cryptanalysis [35, 36] and introduced a 128 bit user key and a new key schedule. However, differential cryptanalysis delivers the round keys, and the attempted strengthening against differential cryptanalysis of FEAL was in vain. Another interesting book on FEAL is [47].

## 2.4 Khufu

Khufu is a block cipher with 64 bit block length and 512 bit key [34, 62]. Khufu uses whitening at the beginning and the end of the cipher, but the main target of the user key is a s-box with 8 bit input and 32 bit output. Khufu is the first algorithm (according to my knowledge), that has key-dependent s-boxes. Khufu is defined for 8, 16, 24, 32, 40, 48, 56, 64 rounds. Every eight rounds (a so called octet) a new s-box is used. Khufu is a Feistel-Cipher with whitening.

Khufu encrypts the user key of 512 bit/64 bytes with an initial s-box in Cipher Block Chaining Mode to create the whitening keys and the s-box entries. In order to have a bijective s-box, Khufu constructs the 8 bit to 32 bit s-box out of four 8 bit to 8 bit s-boxes. This could be omitted, since the 256 entries in the 8 bit to 32 bit s-box are well below $2^{16}$, the square root from $2^{32}$. The probability to make a collision is low, according to the birthday paradox [56]. The key schedule is one-way.

The encryption of the data proceeds as follows. At first the data passes through the beginning whitening, i.e. the whole block of 64 bits data is added modulo 2 (XOR) with an auxiliary key. The data block is split into halves of 32 bit. The most significant byte of the left half is used as the input of the s-box. The output of the s-box is added modulo 2 (XOR) to the right half. The left half is rotated by eight bit and the left half and right half are interchanged. Then the same procedure is done with the new left half. After eight rounds, all the eight bytes in the data block have been used as input into the s-box. That is why the number of rounds must be a multiple of 8. Khufu with 16 rounds was successfully attacked using differential cryptanalysis [19].

## 2.5 IDEA

The block cipher IDEA (International Data Encryption Algorithm, IDEA is a registered trademark) was originally known as PES (Proposed Encryption Standard) and published by Xuejia Lai and James Massey [26, 60]. But differential cryptanalysis, which were at that time emerging, prompted the authors to make some changes to the algorithm [27, 28]. IDEA has a 64 bit block size and a key length of 128 bits and 8.5 rounds. The design concept of IDEA is "mixing operations from different algebraic groups." Buildings blocks of IDEA are addition modulo 2 (XOR), addition modulo $2^{16}$ and multiplication by $2^{16}+1$. Note that $2^{16}+1$ is a prime. The whole architecture from IDEA is especially useful for 16-bit microprocessors.

The key schedule of IDEA is quite simple: The user key is the first subkey. The other subkeys are derived as follows: The actual subkey is generated by its predecessor shifted to the left by 25 bits.

Criticism on IDEA's key schedule came prompt, see for example [12, 13, 33, 22, 8], but attacks, that could seriously threaten the security of IDEA did not take place. But as time went by, new attacks have been published, especially in the new millenium. Papers like [7, 6] show, that attacks on 6-Round IDEA are faster then exhaustive search. After all the block length and the key size of IDEA are yesterdays business. Today a block cipher should have a block size of at least 128 bit and a key length of at least 256 bit. In most countries the patents of IDEA will expire 2010 or 2011.

## 2.6 Blowfish and Kaweichel

Blowfish has a 64 bit block length and a key size with up 448 bits and 16 rounds [46, 55]. It is a Feistel cipher, however the subkeys are not injected into the F-function. The subkeys are applied to the left half before the left half is used as the input to the F-function. The subkeys are added modulo 2 (XOR) to the left half of the data block. In addition, the s-boxes are pseudo-random. The four s-boxes have an input of 8 bit and an output of 32 bit. Four s-boxes form the F-function. The outputs of the s-boxes are added modulo 2 (XOR) or added modulo $2^{32}$ to form the output of the F-function. Finally there two subkeys added modulo 2 (XOR), when the last of the Feistel rounds are over. Blowfish needs a minimum 4 Kbyte L1-Cache to be real fast in software. Blowfish F-function is indeed a function and not a permutation, i.e. it is non-surjective.

The key schedule of Blowfish is quite complicated: At first assign all the subkeys and the s-boxes a pseudo-random values. In Blowfish, the hexadecimal values of $\pi$ less the initial 3 are used. The user key with maximal 448 bits is added modulo 2 (XOR) to the subkeys. If the user key is to short, repeat it until all the subkeys are added modulo 2 (XOR) to the user key.

Let the 64 bit 0-String be encrypted. The output of the cipher is encrypted again, but before that it is assigned the first two subkeys. Continue the Output FeedBack mode of cipher (OFB) until all the subkeys and all the s-boxes have received new values. Obviously this key schedule is one-way.

There is not much literature on Blowfish. Serge Vaudenay [52] published a report on weak keys of Blowfish. The same did Orhun Kara and Cevat Manap [21] with the reflection attack. Vincent Rijmen [41] published an attack on 4-round Blowfish with second-order differential.

Kaweichel [42, 43, 44] is the natural extension of Blowfish for 64 bit microprocessor. Its block length is 128 bits and key space is up to 1920 bit, depending on the number of rounds. Kaweichel is a Feistel cipher and comes with 16, 24 and 32 rounds. To be real fast in software, Kaweichel needs a L1-Cache of at least 16 kBytes. The subkeys are in the same place as with Blowfish. However, there is some difference: The subkeys are added modulo $2^{64}$. The s-boxes have an input of 8 bit and output of 64 bit. Eight s-boxes form the round function or F-function. The way, in which the outputs of the s-boxes are treated is not the same as in Blowfish, because I want to thwart the attack by Vincent Rijmen [41]. The attacks by Serge Vaudenay [52] and Orhun Kara and Cevat Manap [21] are impossible, since addition modulo $2^{64}$ and addition modulo 2 (XOR) do not commutate. Kaweichel has like Blowfish,a F-function and not a permutation as a round function, i.e. it is non surjective. After the output from the F-function is added modulo 2 (XOR) to the right half, the right half is rotated by 11 bits to the left. The key schedule is the same as in Blowfish, only even subkeys are assigned at first. The second difference to the Blowfish key schedule is that shorter keys are not appended by itself. As in Blowfish, the key schedule is one-way.

## 2.7 MISTY and KASUMI

Misuru Matsui, the doyen of linear cryptanalysis, invented MISTY1 and MISTY2. The description of both block ciphers is rather complicated, so I refer the reader to the literature [32] or Wikipedia [65]. The block ciphers have a 64 bit block length and a key size of 128 bit. MISTY1 and MISTY2 are Feistel ciphers and the recommended number of rounds are 8, although the number of rounds could be $4 \cdot i, i = 1, 2, 3, \ldots$. Despite being 13 years old, open cryptanalysis has not found an attack faster then exhaustive search. The best attack is by Dunkelman and Keller [16], which analysis 6 round of MISTY1 in a time complexity with more than $2^{123}$.

MISTY was paid a lot of attention, when is successor KASUMI [61], also known as A5/3, became the block cipher of the GSM Association for the third generation of mobile phones (UMTS). However, recent research showes [17] that MISTY offers more security than KASUMI. In KASUMI, the key schedule has been changed, the subkeys off all rounds are linear functions of the user key. A related key boomerang attack by the Israeli researchers has complexity $2^{36}$ data, $2^{30}$ memory and $2^{32}$ time with 4 related keys. The whole attack lasts on a modern PC two hours.

The GSM Association stated before the attack, that "removing all the FI functions in key scheduling part makes the hardware smaller and/or reduces key set-up time. We expect that related key attacks do not work for this structure". Usually, the subkeys are stored in memory be it hardware or software, so they can be accessed at once. The key scheduling is not time-critical, but encryption/decryption is.

This is good example that the key schedule should support the block cipher. This is also good example that changing the key schedule of an existing block cipher can do much harm to the security of the block cipher.

### 2.8 Other block ciphers

The LOKI variants (LOKI89 & LOKI91), which were contrived by Australian cryptographers [9, 10], were also susceptible to related key analysis [23, 24, 5]. The round keys are linear functions (rotation of the left half and the right of half of the user key) of the user key. For more on LOKI, try [63]

Square, which is the predecessor of Rijndael (now the Advanced Encryption Standard AES ) was recently shown to be also susceptible to related key analysis [25]. The attackers used the fact, that the derivation of the round keys is affine function of the user key. The full Square block cipher can be broken with time complexity $2^{123}$, while the key length is $2^{128}$.

## 3 The Algorithm of 1024

### 3.1 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 $\lfloor \div \rfloor$ ,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. [11, 12, 14]. The studied function is:

$$f^a(x) = \begin{cases} a \times x & \text{if} \quad x < 2^n - 1 \\ 2^n - 1 & \text{if} \quad x = 2^n - 1 \end{cases} \tag{6}$$

The calculation is easy:

$$a * b \ \ mod(2^n - 1) = (a * b \ \ mod(2^n) + \lfloor \frac{a*b}{2^n} \rfloor)(1 + \lfloor \frac{1}{2^n} \rfloor) \tag{7}$$

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 [11] gives a wrong formula. It has been corrected in chapter 11 of Joan Daemens Ph.D. thesis [14]. 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 [12].

In [12] 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 the encryption factor without rotation. The next block is assigned the encryption_factors $\lll 1$ and so on. The last block (right most) is assigned the 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 $2^{-9}$.

## 3.2 Diffusion Layer

The diffusion layer has as parent the diffusion layer from SAFER [29, 30]. However, there are four modifications:

1. 32 blocks instead of eight.
2. Four bytes instead of one byte as primitive unit. See [48].
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:

$$b_1 = 2a_1 + a_2 \tag{8}$$
$$b_2 = a_1 + a_2 \tag{9}$$

It can be rewritten:

$$b_2 = a_1 + a_2 \tag{10}$$
$$b_1 = a_1 + b_2 \tag{11}$$

The Pseudo-Hadamard-Transform has one disadvantage. The least significant bit of $b_1$ is not dependent on $a_1$. Schneier et. al. [48] 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 [40] a branch number for invertible linear mappings was introduced. It is defined as

$$B(\theta) = \min_{a \neq 0}(\omega_h(a) + \omega_h(\theta(a))) \tag{12}$$

where $\omega_h$ 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. $\theta$ is the linear mapping. The branch number of the linear mapping $\theta$ 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 [48] 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 [51, 53, 54]. 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.

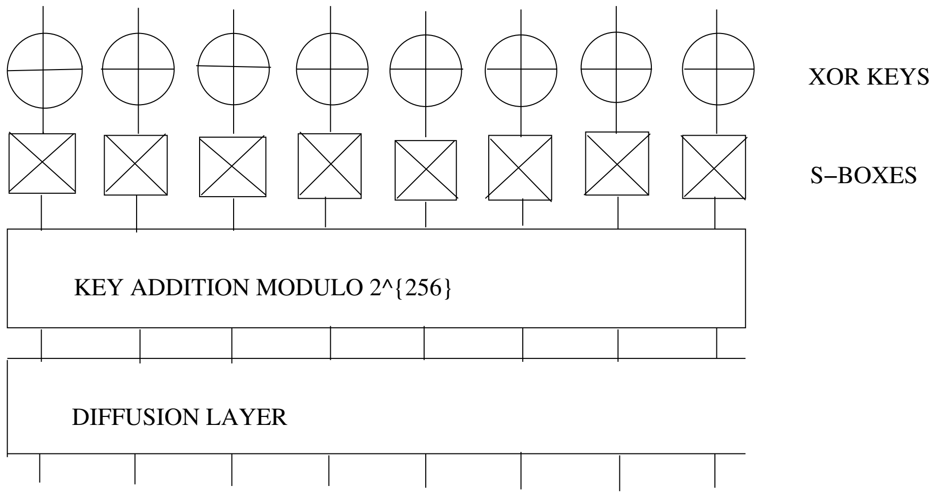


Figure 1: The left quarter of a primary round

### 3.3 Addition modulo $2^{256}$

Addition modulo $2^{256}$ was introduced to give an upper bound for linear cryptanalysis. If we take [37], 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 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.

### 3.4 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

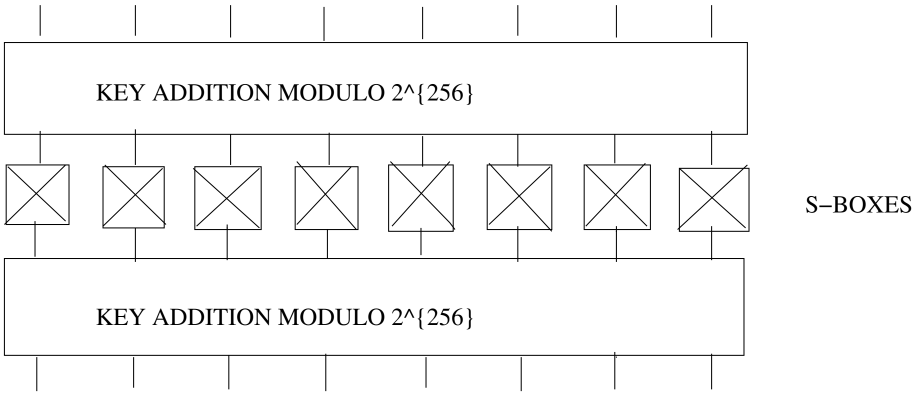


Figure 2: A left quarter of the middle transform

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. 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.

## 3.5 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.

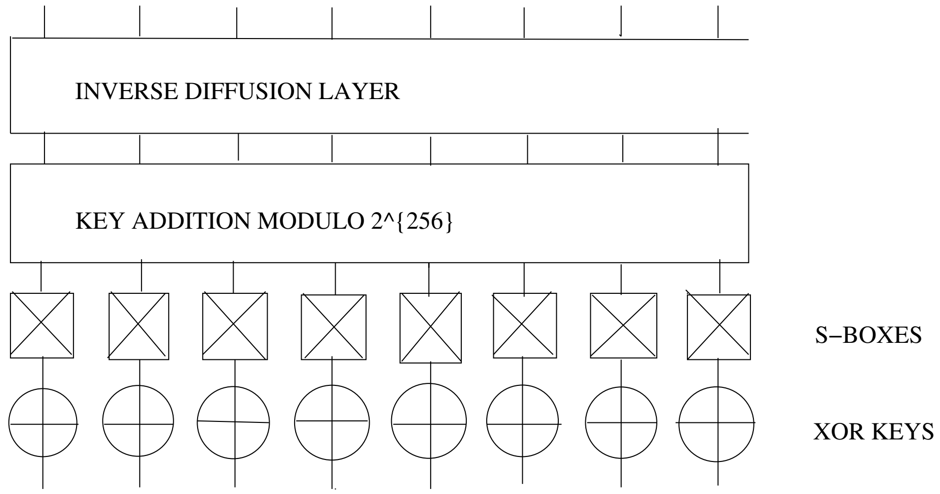


Figure 3: The left quarter of a secondary round

## 3.6 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 $2^{256}$ 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.

## 3.7 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 [31, 37]. 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 [31, 37] we have as average bias $\epsilon = 2^{-9*128} = 2^{-1152}$ (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).

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.

## 3.8 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 $2^{32} - 1$. [12] 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-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.

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

# 4 1024XKS vs. 1024

## 4.1 Different Key Schedule

The key schedule of 1024 resembles that of IDEA. The round key of the first round is the user key. Note that 1024 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 [45]). 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 1024XKS is one-way.

The round key generation of 1024XKS 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 1024 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 [46].

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", 1024XKS 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.

## 4.2 Different Key Size

1024XKS comes with two user key sizes, 2048 bit and 4096 bit. The round keys are obtained as follows: The first round keys are the user key. Then the first round keys are rotated by 455 bit to the left to form the next round keys. The process is iterated until all the round keys from the primary rounds, the middle transformation and the secondary rounds have their assigned values. The next part of the round key generation is described in the previous subsection (Different Keys Schedule). The first key schedule for the key size of 2048 of 1024 is described in the previous section, subsection Key Schedule. A look at the function key_schedule will give further insight.

To distinguish the key sizes in the reference implementation, there is variable for the preprocessor named #define BIG_KEY. When the #define statement is true, the key size is 4096 bit. If the #define statement is not true, then the key size is 2048 bit. 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, is the function key_schedule quite at the beginning of the reference implementation.

## 4.3 Different Order of the S-Boxes

The order of the s-boxes in 1024 is as follows: The left most s-box is assigned a value found in [12, 14]. It is the encryption factor, which can by expressed in C 0x0251F1CDB. The next s-box is assigned that value rotated to the left by 1. The third s-box is assigned that value from the left most s-box rotated to the left by 2. and so on. Since rotation to the left by one can be expressed by multiplication by 2 mod $2^n - 1$ there is a symmmetry, which can be used by a cryptanalyst. The same is true for the decryption factors. The left most s-box is assigned the value 0x0DAD4694. The rest of the decryption factors is assigned the value the predecessor had rotated by one to the right.

In 1024XKS the s-boxes, which contain the even rotation numbers, are same as in 1024. There is an increase by two for the rotational values for each step ranging from 0 (left most s-box) to 30 (second right most s-box). The rotational values for the s-boxes with even rotational values is 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 is 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 |

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

# 5 Intellectual Property

As far as I know, the predecessors of 1024 (MMB and SAFER) and 1024XKS (MMB, SAFER, Blowfish) have no legal protection like patents or a registered trademarks. I did not file for legal protection (patents, registered trademark or alike) for 1024 and 1024XKS and will never do. Thus, the block ciphers 1024 and 1024XKS can freely be used by anyone.

# References


[1] Adams, C.M. and S.E. Tavares: Designing S-Boxes for Ciphers Resistant to Differential Cryptanalysis, in: W. Wolfowicz (Ed.): *Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography*, Rome, Italy, 1993

[2] Bamford, James: *The Puzzle Palace*, Penguin Books, Harmondsworth, England, 1983

[3] Biham, Eli and Adi Shamir: Differential Cryptanalysis of FEAL and N-Hash, in Donald Davies (Ed.): *Advances in Cryptology - EUROCRYPT '91*, Springer Verlag, Berlin, 1991

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

[5] Biham, Eli: New Types of Cryptanalytic Attacks Using Related Keys, in Tor Helleseth (Ed.): *Advances in Cryptology - EUROCRYPT '93*, Springer Verlag, Berlin, 1994

[6] Biham, Eli; Orr Dunkelman and Nathan Keller: A New Attack on 6-Round IDEA, in Alex Biryukov (Ed.): *Fast Software Encryption 2007*, Springer Verlag, Berlin, 2007. Available from: `https://www.cosic.esat.kuleuven.be/publications/article-920.pdf`

[7] Biryukov, Alex; Jorge Nakahara, Bart Preneel and Joos Vandewalle: New Weak-Key Classes for IDEA, in F. Bao, R.H. Deng, S. Qing (Eds.): *Information and Communication Security - ICICS 2002*, Springer Verlag, Berlin, 2002. Available from: `https://www.cosic.esat.kuleuven.be/publications/article-189.pdf`

[8] Borst, J.; L.R. Knudsen, V. Rijmen: Two Attacks on Reduced IDEA, in Walter Fumy (Ed.): *Advances in Cryptology - EUROCRYPT '97*, Springer Verlag, Berlin, 1997

[9] Brown, L; Pieprzyk, Josef and Jennifer Seberry: LOKI - A Cryptographic Primitive for Authentication and Secrecy Application, in Pieprzyk, Josef and Jennifer Seberry (Eds.): *Advances in Cryptology - AUSCRYPT '90*, Springer Verlang, Berlin, 1990

[10] Brown, L; Kwan, M; Seberry, Jennifer and Josef Pieprzyk: Improving Resistance to Differential Cryptanalysis and the Redesign of LOKI, in Imai, H. et. al. (Eds.): *Advances in Cryptology - ASIACRYPT '91*, Springer Verlag, Berlin, 1993

[11] Daemen, Joan; Luc Van Linden; René Govaerts and Joos Vandewalle: Propagation Properties of Multiplication Modulo $2^n - 1$, appeared in the *Proceedings of the 13th Symposium on Information Theory in Benelux, Werkgemeenschap voor Informatie- en Communicatietheorie*, pp. 111-118, 1992, Available from: `https://www.cosic.esat.kuleuven.be/publications/article-136.pdf`

[12] Daemen, Joan; René Govaerts and Joos Vandewalle: Block Ciphers Based on Modular Arithmetic, in W. Wolfowicz (Ed.): *Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography*, Fondazione Ugo Bordoni, Rome, 1993. Available from: `https://www.cosic.esat.kuleuven.be/publications/article-277.pdf`

[13] Daemen, Joan; René Govaerts and Joos Vandewalle: Weak Keys for IDEA, in Douglas Stinson (Ed.): *Advances in Cryptology - CRYPTO '93*, Springer Verlag, Berlin, 1993

[14] Daemen, Joan: *Cipher and Hash Function Design, Strategies based on linear and differential Cryptanalysis*, Ph.D. thesis, KU Leuven, Belgium. Available from: `https://homes.esat.kuleuven.be/~cosicart/ps/JD-9500`, 1995

[15] Daeman, Joan; Knudsen, Lars and Vincent Rijmen: The Block Cipher Square, in Biham, Eli (Ed:): *Fast Software Encryption '97*, Springer Verlag, Berlin, 1997

[16] Dunkelman, Orr and Nathan Keller: An Improved Impossible Differential Attack on MISTY1, in *Advances in Cryptology - Asiacrypt 2008*, Springer Verlag, Berlin, 2008

[17] Dunkelman, Orr; Nathan Keller and Adi Shamir: *A Practical-Time Attack on the A5/3 Cryptosystem Used in Third Generation GSM Telephony.* Available from: `http://eprint.iacr.org/2010/013.pdf`

[18] Feistel, Horst: *Cryptography and Computer Privacy*, in Scientific American, Band 228, Nummer 5, USA, Mai 1973

[19] Gilbert, H. and P. Chauvaud: A chosen plaintext attack of the 16-round Khufu Cryptosystem, in Desmedt, Yvo (Ed.): *Advances in Cryptology - CRYPTO '94*, Springer Verlag, Berlin, 1994

[20] Grupen, Claus and Dieter Schmidt: *Beschreibung einer Blockchiffre -Kaweichel- (in German).* Available from: `http://www.infoserversecurity.org/itsec_infoserver_v0.5/sections/science/docs/1095771791/kaweichel.pdf`

[21] Kara, Orhun and Cevat Manap: A New Class of Weak Keys for Blowfish, in Alex Biryukov (Ed.): *Fast Software Encrytion 2007*, Springer Verlag, Berlin, 2007

[22] Kelsey, John; Bruce Schneier and David Wagner: Key Schedule Cryptanalysis of IDEA, G-DES, GOST, SAFER and Triple-DES, in Neal Koblitz (Ed.): *Advances in Cryptology - CRYPTO '96*, Springer Verlag, Berlin, 1996

[23] Knudsen, Lars: Cryptanalysis of LOKI, in Imai H. et.al. (Eds.): *Advances in Cryptology - ASIACRYPT '91*, Springer Verlag, Berlin, 1993

[24] Knudsen, Lars: Cryptanalysis of LKOI91, in Seberry,J. and Y.Zheng (Eds.): *Advances in Cryptology - ASIACRYPT '92*, Springer Verlag, Berlin, 1993

[25] Koo, Boonwok; Yeom, Yongjin and Junghwan Song: *Related Key Boomerang Attack on Block Cipher Square.* Available from `http://eprint.iacr.org/2010/073.pdf`

[26] 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

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

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

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

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

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

[32] Matsui, Mitsuru: New Block Encryption Algorithm MISTY, in Biham, Eli (Ed.): *Fast Software Encryption*, 4th International Workshop, Springer Verlag, Berlin, 1997

[33] Meier, Willy: On the security of the IDEA block cipher, in Tor Helleseth (Ed.). *Advances in Cryptology - EUROCRYPT '93*, Springer Verlag, Berlin, 1993

[34] Merkle, Ralph: Fast Software Encryption Function, in Alfred Menezes and Scott Vanstone (Eds.): *Advances in Cryptology - CRYPTO '90*, Springer Verlag, Berlin, 1991

[35] Miyaguchi, Shoji: The FEAL-8 Cryptosystem and a Call for Attack, in G. Brassard (Ed.): *Advances in Cryptology - CRYPTO '89*, Springer Verlag, Berlin, 1990

[36] Miyaguchi, Shoji: The FEAL Cipher Family, in Menezes, A.J. and S.A. Vanstone (Eds.): *Advances in Cryptology - CRYPTO '90*, Springer Verlag, Berlin, 1991

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

[38] National Bureau of Standards, *Data Encryption Standard*, U.S. Department of Commerce, FIPS publication 46, January 1977.

[39] National Bureau of Standards, *DES Modes of Operation*, U.S. Department of Commerce, FIPS publication 81, December 1980.

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

[41] Rijmen, Vincent: *Cryptanalysis and design of iterated block ciphers,* Doctoral Dissertation, Catholic University Leuven, Belgium, October 1997. Available from: `https://www.cosic.esat.kuleuven.be/publications/thesis-4.ps`

[42] Schmidt, Dieter: *Reference Implementation of the Block Cipher Kaweichel in C.* Available from: `http://www.infoserversecurity.org/itsec_infoserver_v0.5/sections/science/docs/1095771918/kaweichel.zip`

[43] Schmidt, Dieter: *Kaweichel, an Extension of Blowfish for 64-Bit Architecturs* Available from: `http://eprint.iacr.org/2005/144.pdf`

[44] Schmidt, Dieter: *On the Security of Kaweichel*, Available from: `http://eprint.iacr.org/2005/432.pdf`

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

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

[47] Schneier, Bruce: *Angewandte Kryptographie*, Addison-Wesley, Bonn, 1996. German Translation of *Applied Cryptography.*

[48] 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`

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

[50] Shimizu, Akihiro and Shoji Miyaguchi: Fast data encipherment algorithm FEAL, in David Chaum and Wyn Price (Eds.): *Advances in Cryptology - EUROCRYPT '87*, Springer Verlag, Berlin, 1987

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

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

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

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

[55] http://en.wikipedia.org/wiki/Blowfish_(cipher)

[56] http://en.wikipedia.org/wiki/Birthday_paradox

[57] http://en.wikipedia.org/wiki/Data_Encryption_Standard

[58] http://en.wikipedia.org/wiki/FEAL

[59] http://en.wikipedia.org/wiki/Feistel_network

[60] http://en.wikipedia.org/wiki/International_Data_Encryption_Algorithm

[61] http://en.wikipedia.org/wiki/KASUMI_(block_cipher)

[62] http://en.wikipedia.org/wiki/Khufu_and_Khafre

[63] http://em.wikipedia.org/wiki/LOKI

[64] http://en.wikipedia.org/wiki/Lucifer_(cipher)

[65] http://en.wikipedia.org/wiki/MISTY1

[66] http://en.wikipedia.org/wiki/Product_cipher

[67] http://en.wikipedia.org/wiki/SP-Network

# 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

#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/2);i++){
    if(i!=0) e_factors[2*i]=ROL(e_factors[0],2*i);
    e_factors[2*i+1]=ROL(e_factors[0],(WIDTH+7-2*i)%WIDTH);
```

```
  }

}

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

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

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