3rd Class

2016-2017

Block Cipher

التشفير الكتلي

أستاذ المادة : أ. م. د. علاء كاظم

Block Cipher

Security Information Branch
Third Class

2017-2016

cryptography arose as a means to enable parties to maintain privacy of the information they send to each other, even in the presence of an adversary with access to the communication channel. While providing privacy there remains a central goal, the field has expanded to encompass many others, including not just other goals of communication security, such as guaranteeing, integrity and authenticity of communications, but many more sophisticated and fascinating goals, Cryptography is a discipline of mathematics and computer science concerned with information security and related issues, particularly encryption, authentication, and such applications as access Control. Cryptography, as an interdisciplinary subject, draws on several fields. Prior to the early 20th century, cryptography was chiefly concerned with Linguistic patterns. Since then, the emphasis has shifted, and Cryptography now makes extensive use of mathematics, including topics from information theory, computational complexity, statistics, combinatory, and especially number theory

University of Technology
Computer
Computer Science Department
Information security Branch
Assist. Prof. Dr. Alaa K.

BLOCK CIPHER

Lecture 1:

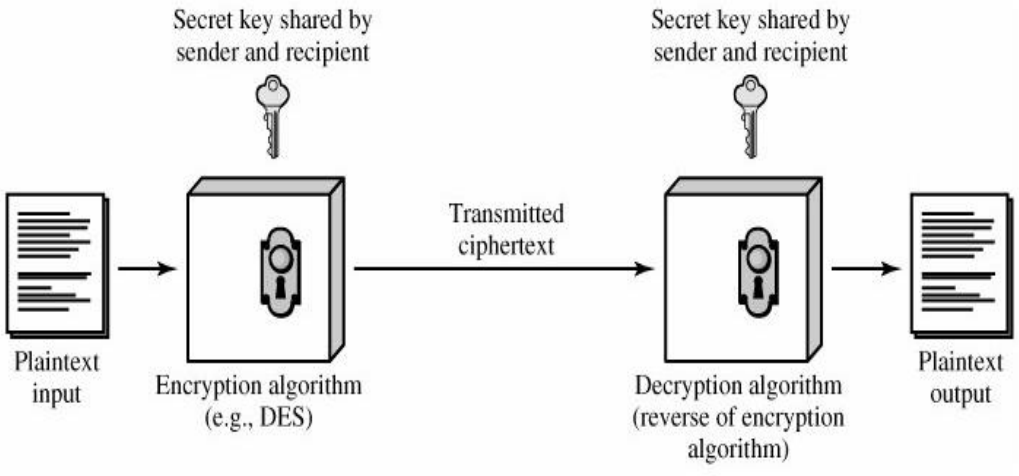
Historically: cryptography arose as a means to enable parties to maintain privacy of the information they send to each other, even in the presence of an adversary with access to the communication channel. While providing privacy there remains a central goal, the field has expanded to encompass many others, including not just other goals of communication security, such as guaranteeing, integrity and authenticity of communications, but many more sophisticated and fascinating goals, Cryptography is a discipline of mathematics and computer science concerned with information security and related issues, particularly encryption, authentication, and such applications as access Control. Cryptography, as an interdisciplinary subject, draws on several fields. Prior to the early 20th century, cryptography was chiefly concerned with Linguistic patterns. Since then, the emphasis has shifted, and Cryptography now makes extensive use of mathematics, including topics from information theory, computational complexity, statistics, combinatory, and especially number theory . Security has many facets. For a system to be secure, many factors must combined. For example, it should not be possible for hackers to exploit bugs, break into a system, and use an account. They shouldn't be able to buy off your system administrator. They shouldn't be able to steal your back-up tapes. These things lie in the realm of system security. The cryptographic protocol is just one piece of the puzzle. If it is poorly designed, the attacker will exploit that. For example, suppose the protocol transmits a password in the clear (that is, in a way that anyone watching can understand what it is), that is a protocol problem, not a system problem. In addition, it will certainly be exploited.

The security of the system is only as strong as its weakest link. This is a big part of the difficulty of building a secure system. To get security we need to address all the problems: how do we secure our machines against intruders? how do we administer machines to maintain security? how do we design good protocols? and so on. All of these problems are important, but we will not address all of these problems here. We usually have to assume that the rest of the system is competent at doing its job. We make this assumption because it provides a natural abstraction boundary in dealing with the enormous task of providing security. Information system security is a domain of a different nature, requiring different tools and expertise.

Symmetric Cipher Model

A symmetric encryption scheme has five ingredients:

- Plaintext: This is the original intelligible message or data that is fed into the algorithm as input.
- Encryption algorithm: The encryption algorithm performs various substitutions and transformations on the plaintext.
- Secret key: The secret key is also input to the encryption algorithm. The key is a value independent of the plaintext and of the algorithm. The algorithm will produce a different output depending on the specific key being used at the time. The exact substitutions and transformations performed by the algorithm depend on the key.
- Ciphertext: This is the scrambled message produced as output. It depends on the plaintext and the secret key. For a given message, two different keys will produce two different ciphertexts. The ciphertext is an apparently random stream of data and, as it stands, is unintelligible.
- Decryption algorithm: This is essentially the encryption algorithm run in reverse. It takes the ciphertext and the secret key and produces the original plaintext.

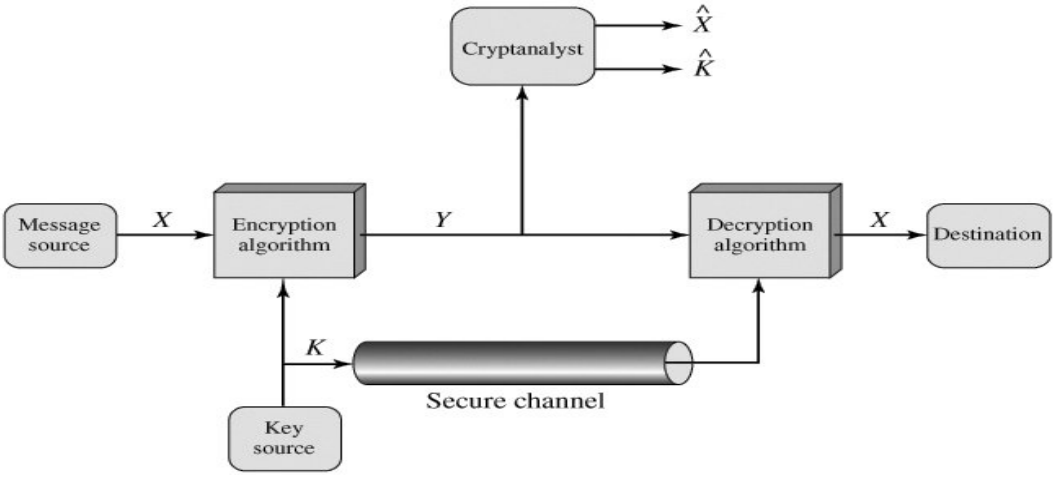


There are two requirements for secure use of conventional encryption:

1. **We need a strong encryption algorithm**. At a minimum, we would like the algorithm to be such that an opponent who knows the algorithm and has access to one or more cipher texts would be unable to decipher the ciphertext or figure out the key. This requirement is usually

stated in a stronger form: The opponent should be unable to decrypt ciphertext or discover the key even if he or she is in possession of a number of ciphertexts together with the plaintext that produced each ciphertext.

2. **Sender and receiver must have obtained copies of the secret key** in a secure fashion and must keep the key secure. If someone can discover the key and knows the algorithm, all communication using this key is readable.



NOTE: They are IDEA (1992), RC5 (1995), RC6 (1996), DES (1977) and AES (2001). The Advanced Encryption Standard (AES) specifies a FIPS-approved symmetric block cipher which will soon come to be used in lieu of Triple DES or RC6.

Feistel Mode

In cryptography, a Feistel cipher is a symmetric structure used in the construction of block ciphers, named after the German IBM cryptographer Horst Feistel; it is also commonly known as a Feistel network. A large proportion of block ciphers use the scheme, including the Data Encryption Standard (DES). The Feistel structure has the advantage that encryption and decryption operations are very similar, even identical in some cases, requiring only a reversal of the key schedule. Therefore the size of the code or circuitry required to implement such a cipher is nearly halved. Feistel networks and

similar constructions are product ciphers, and so combine multiple rounds of repeated operations, such as:

- Bit-shuffling (often called permutation boxes or P-boxes)
- Simple non-linear functions (often called substitution boxes or S-boxes)
- Linear mixing (in the sense of modular algebra) using XOR to produce a function with large amounts of what Claude Shannon described as "confusion and diffusion". Bit shuffling creates the diffusion effect, while substitution is used for confusion.

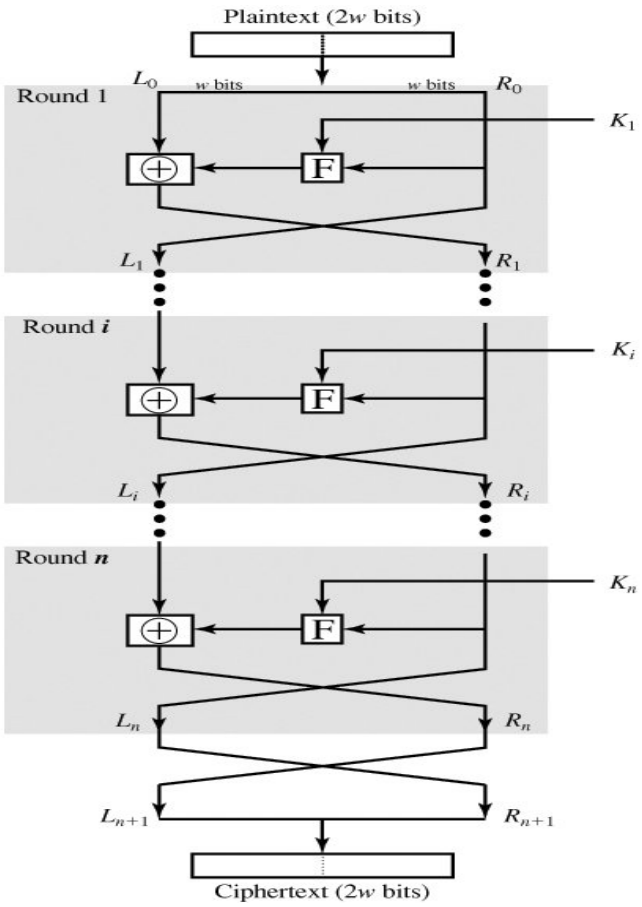


Figure: Feistel Mode

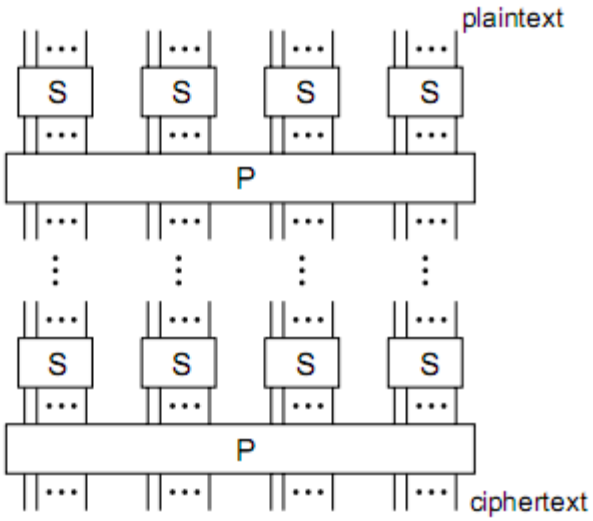
Confusion and Diffusion

Successful block cipher designs often integrate the concepts of *confusion* and *diffusion*. These ideas were introduced by Shannon . *Confusion* is a measure of the statistical properties of the input with relation to the output. Essentially, looking at the output should give little or no information about the input; in short, the transformation should complicate the input such that the output bears little statistical

relationship with the input. *Diffusion*, on the other hand, attempts to extend the influence of the input symbols over a wide range of output symbols in order to disguise the tendencies of the input. It must be noted that is not mandatory for both characteristics to be utilized to achieve secrecy. Indeed, the Vernam stream cipher achieves perfect secrecy with confusion alone. Since each plaintext symbol is combined with completely random data, there is no need to mix adjacent symbols of plaintext to achieve additional randomness. Unlike stream ciphers, block cipher design depends heavily on both principles of confusion and diffusion. Since the symbol length of a typical block cipher (64 bits) is often longer than the corresponding symbol in a stream cipher (8 or 32 bits), there are more possible bits positions, which necessitate and assist diffusion. A successful diffusion is one in which each plaintext bit and each key bit affects each and every ciphertext bit (in the case of encryption). This diffusion can be applied using a permutation which exchanges individual bit locations or sequential algebraic functions which combine and spread the influence of the inputs. A well diffused cipher will satisfy the strict avalanche criteria whereby if a single bit changes in the input, then half of the output bits will change in a random manner.

Definition : A product cipher combines two more transformations in manner intending that the resulting cipher is more secure than the individual components.

Definition : A substitution-permutation (SP) network is a product cipher composed of a number of stages each involving substitutions and permutations .



Substitution Operation a binary word is replaced by some other binary word the whole substitution function forms the key if use n bit words, the key is 2^(n)!bits, grows rapidly

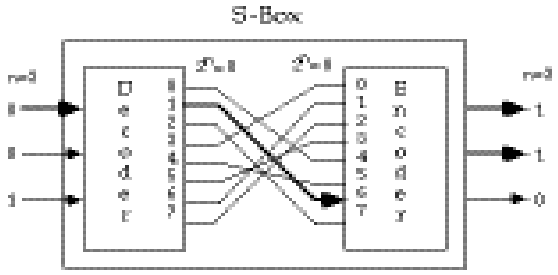
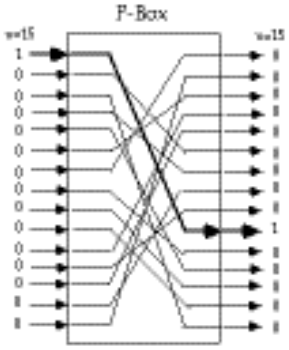


Fig 2.1 Substitution Operation

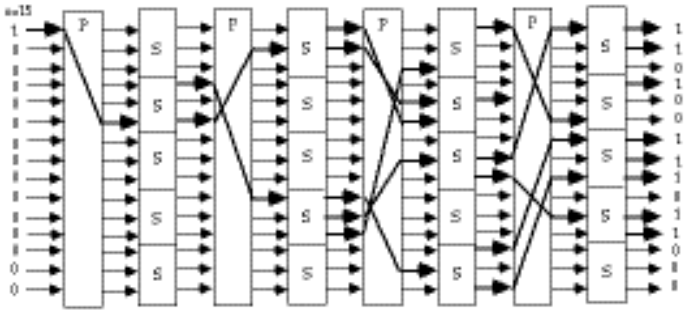
can also think of this as a large lookup table, with n address lines (hence 2^(n) addresses), each n bits wide being the output value will call them S-boxes

Permutation Operation a binary word has its bits reordered (permuted) the re-ordering forms the key if use n bit words, the key is n!bits, which grows more slowly, and hence is less secure than substitution



this is equivalent to a wire-crossing in practise (though is much harder to do in software) will call these P-boxes

Substitution-Permutation Network Shannon combined these two primitives he called these mixing transformations



Shannons mixing transformations are a special form of product ciphers where

- **S-Boxes**

provide confusion of input bits

- **P-Boxes**

  provide diffusion across S-box inputs in general these provide the following results.

Lecture 2:

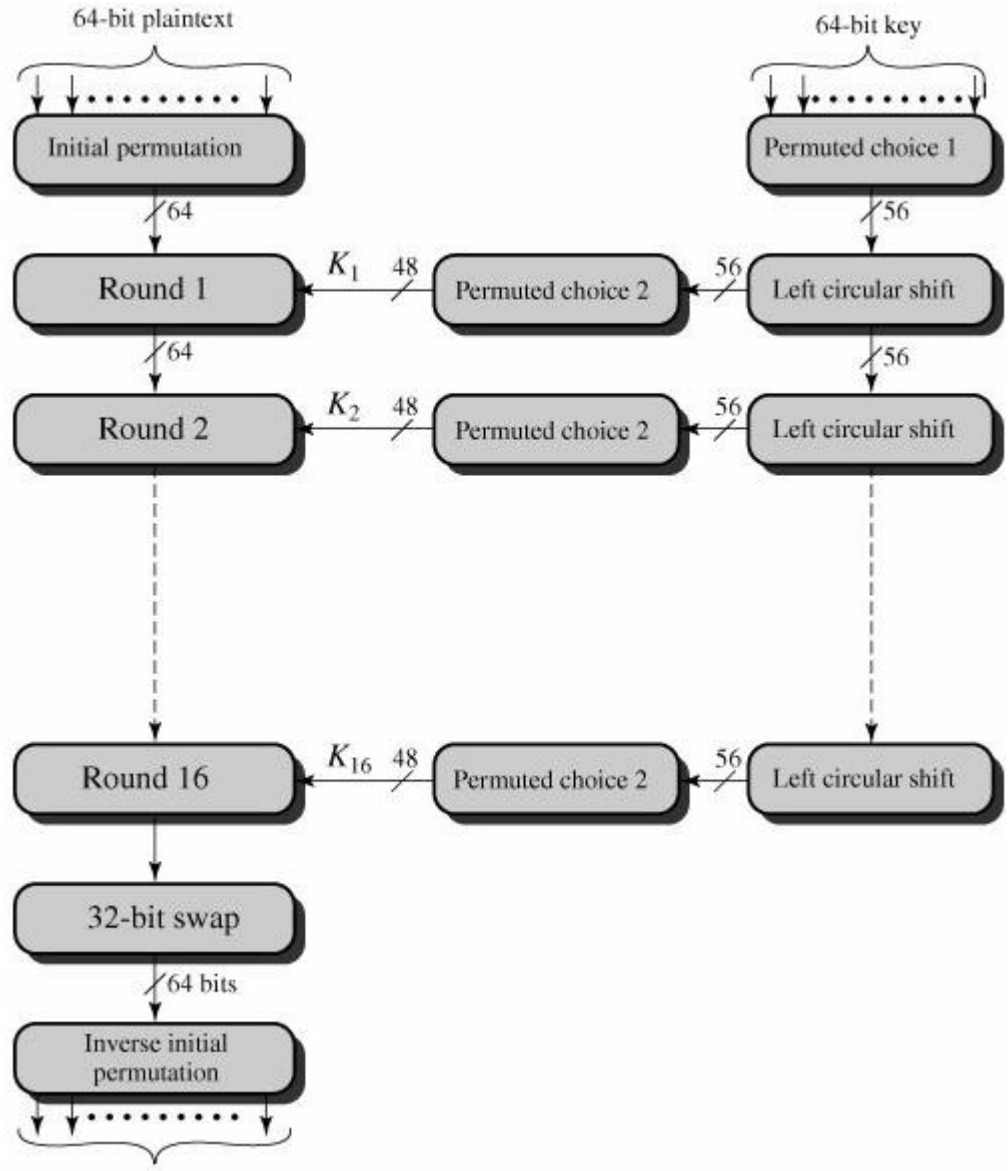
Data Encryption Standard DES

The origins of DES go back to the early 1970s. In 1972, after concluding a study on the US government's computer security needs, the US standards body NBS (National Bureau of Standards) now named NIST (National Institute of Standards and Technology) identified a need for a government-wide standard for encrypting unclassified, sensitive information. Accordingly, on 15 May 1973, after consulting with the NSA, NBS solicited proposals for a cipher that would meet rigorous design criteria. None of the submissions, however, turned out to be suitable. A second request was issued on 27 August 1974. This time, IBM submitted a candidate which was deemed acceptable a cipher developed during the period 1973–1974 based on an earlier algorithm, Horst Feistel's Lucifer cipher. The team at IBM involved in cipher design and analysis included Feistel, Walter Tuchman, Don Coppersmith, Alan

Konheim, Carl Meyer, Mike Matyas, Roy Adler, Edna Grossman, Bill Notz, Lynn Smith, and Bryant Tuckerman.

Overall Structure

The algorithm's overall structure is shown in Figure (2-1) there are 16 identical stages of processing, termed *rounds*. There is also an initial and final permutation, termed *IP* and *FP*, which are inverses (IP "undoes" the action of FP, and vice versa). IP and FP have almost no cryptographic significance, but were apparently included in order to facilitate loading blocks in and out of mid- 1970s hardware, as well as to make DES run slower in software. Before the main rounds, the block is divided into two 32-bit halves and processed alternately; this crisscrossing is known as the Feistel scheme. The Feistel structure ensures that decryption and encryption are very similar processes the only difference is that the subkeys are applied in the reverse order when decrypting. The rest of the algorithm is identical. This greatly simplifies implementation, particularly in hardware, as there is no need for separate encryption and decryption algorithms. The red symbol denotes the exclusive-OR (XOR) operation. The *F-function* scrambles half a block together with some of the key. The output from the F-function is then combined with the other half of the block, and the halves are swapped before the next round. After the final round, the halves are not swapped; this is a feature of the Feistel structure, which makes encryption and decryption similar processes.


64-bit plaintext
64-bit key
Initial permutation
Permuted choice 1
64
56
Round 1
K_1
48
Permuted choice 2
56
Left circular shift
64
56
Round 2
K_2
48
Permuted choice 2
56
Left circular shift
Round 16
K_{16}
48
Permuted choice 2
56
Left circular shift
32-bit swap
64 bits
Inverse initial permutation

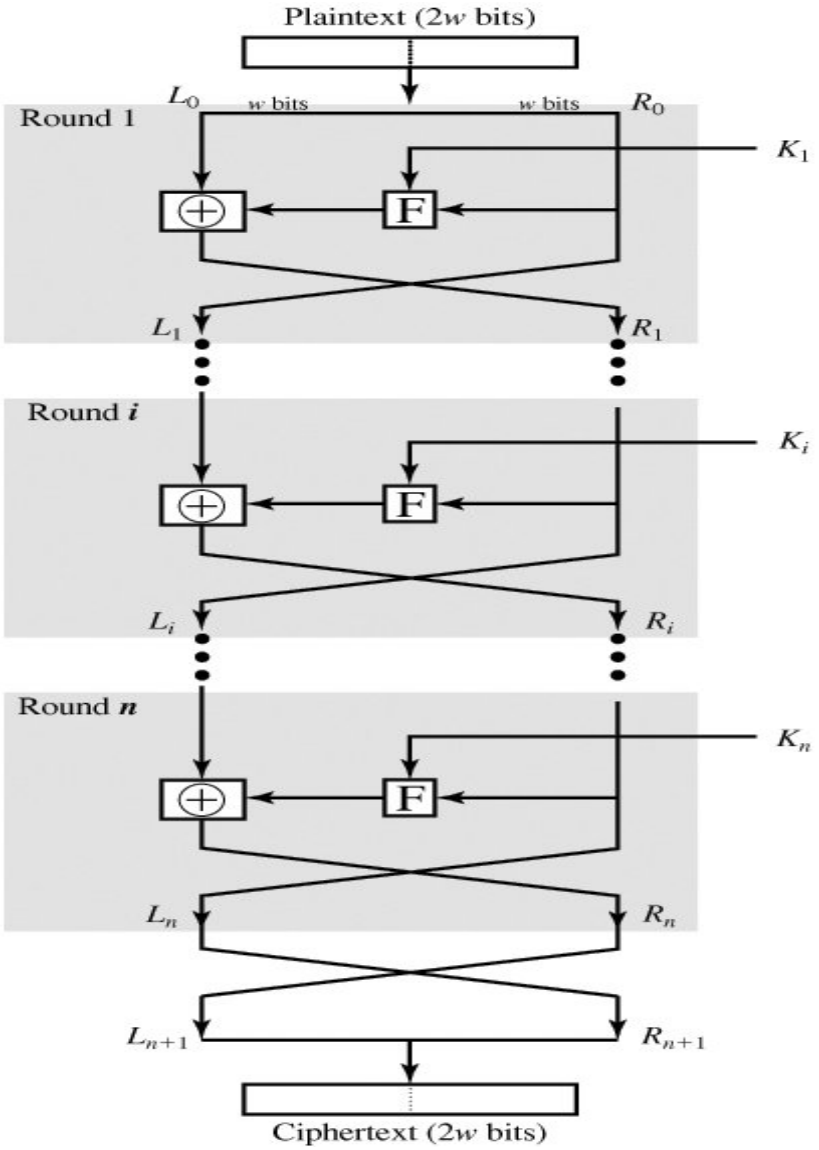


Figure :Structure of DES Block Cipher

The Feistel (F) Function

The F-function, depicted in Figure (2-2), operates on half a block (32 bits) at a time and consists of four stages:

1. *Expansion* the 32-bit half-block is expanded to 48 bits using the *expansion permutation*, denoted *E* in the diagram, by duplicating some of the bits.

2. *Key mixing* the result is combined with a *subkey* using an XOR operation. Sixteen 48-bit subkeys one for each round are derived from the main key using the *key schedule* (described below).

3. *Substitution* after mixing in the subkey, the block is divided into eight 6-bit pieces before processing by the *S-boxes*, or *substitution boxes*. Each of the eight S-boxes replaces its six input bits with four output bits according to a **non-linear transformation**, provided in the form of a look up table. The S-boxes provide the core of the security of DES without them; the cipher would be linear, and trivially breakable.

4. *Permutation* finally, the 32 outputs from the S-boxes are rearranged according to a fixed permutation, the *P-box*. The alternation of substitution from the S-boxes, and permutation of bits from the P-box and Expansion provides so-called "confusion and diffusion" respectively, a concept identified by Claude Shannon in the 1940s as a necessary condition for a secure yet practical cipher.

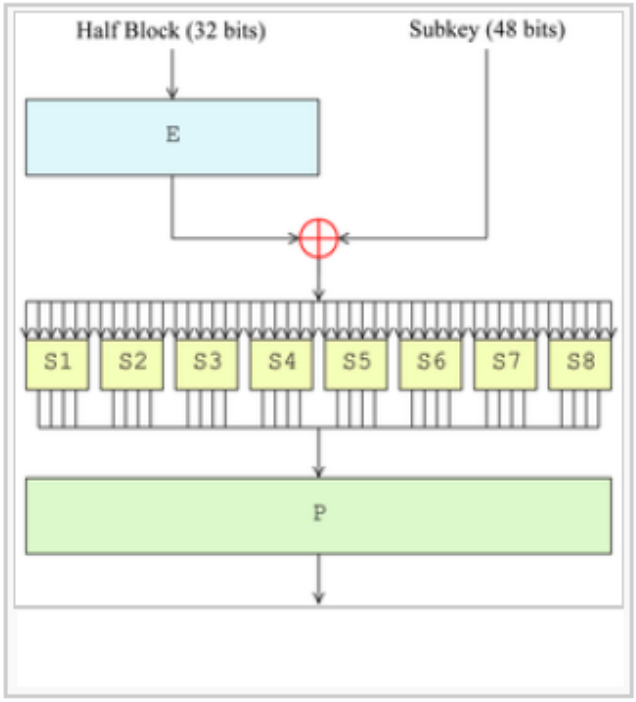
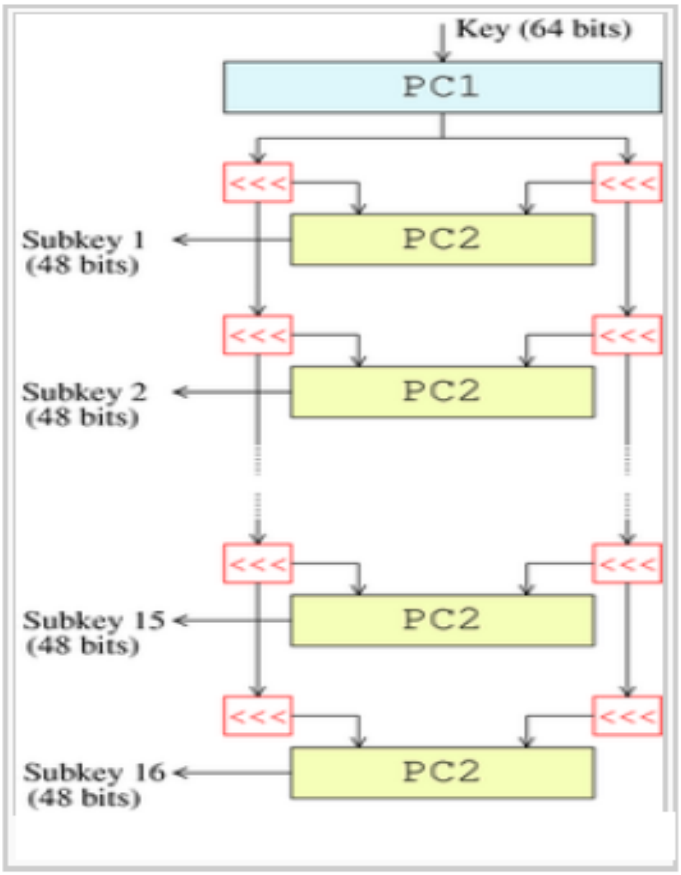


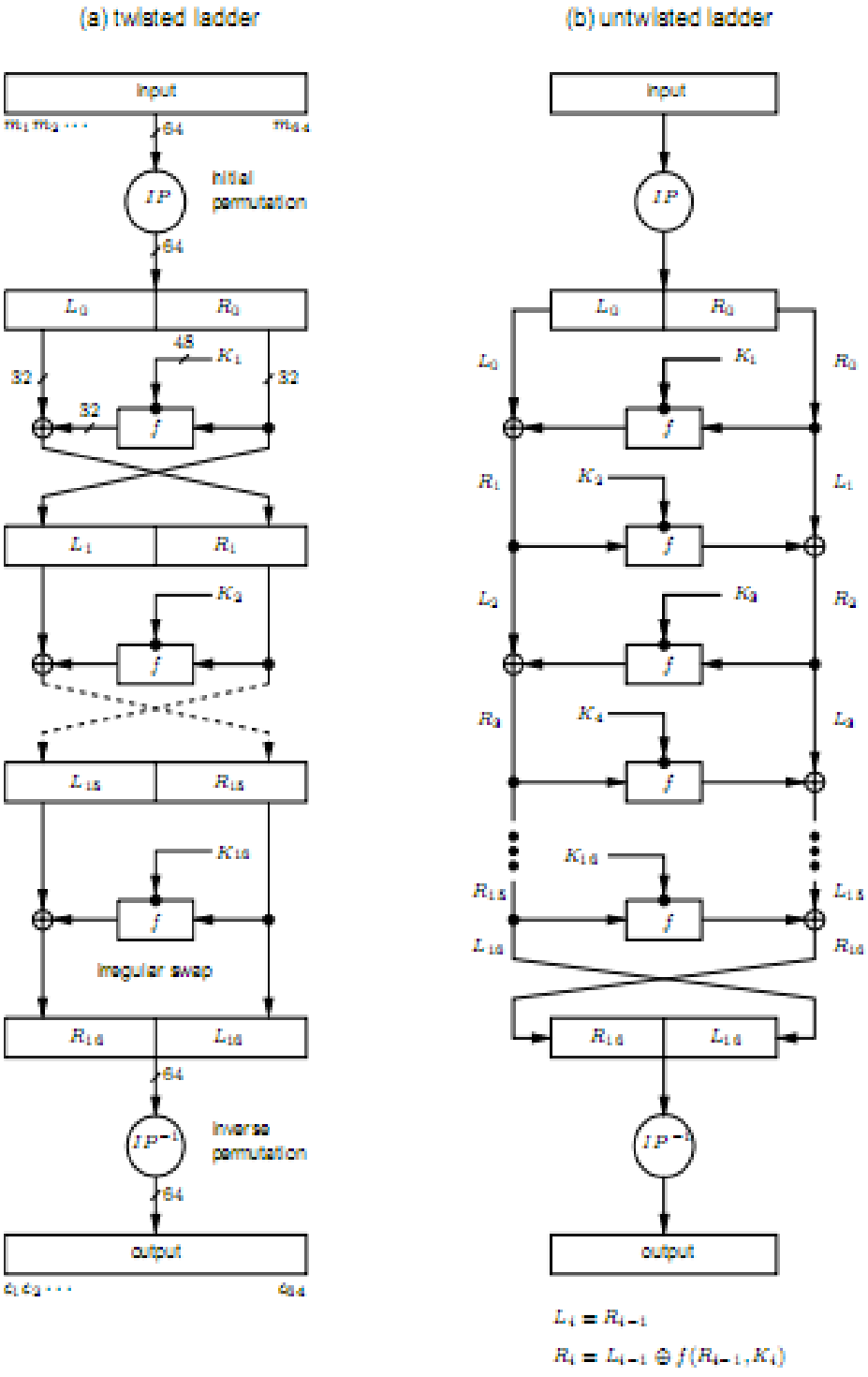
Figure (2-2) The Feistel Function (F-function) of DES

Key Schedule

Figure (2-3) illustrates the *key schedule* for encrypting the algorithm which generates the subkeys. Initially, 56 bits of the key are selected from the initial 64 by *Permuted Choice 1* (*PC-1*) the remaining eight bits are either discarded or used as parity check bits. The 56 bits are then divided into two 28-bit halves; each half is thereafter treated separately. In successive rounds, either halves are rotated left by one or two bits (specified for each round), and then 48 subkey bits are selected by

Permuted Choice 2 (*PC-2*) 24 bits from the left half, and 24 from the right. The rotations (denoted by "<<<" in the diagram) mean that a different set of bits is used in each subkey; The key schedule for decryption is similar the subkeys are in reverse order compared to encryption. A part from that change, the process is the same as for encryption.





Decrypting DES

After all the substitutions, permutations, XORs, and shifting around, you might think that the decryption algorithm is completely different and just as confusing as the encryption algorithm. On the contrary, the various operations were chosen to produce a very useful property: The same algorithm works for both encryption and decryption. With DES it is possible to use the same function to encrypt **or decrypt a block.**

The only difference is that the keys must be used in the reverse order. That is, if the encryption keys for each round are K1 K2 K3,..., K16 then the decryption keys are K16 K15 K14,..., K1. The algorithm that generates the key used for each round is circular as well. The key shift is a right shift and the number of positions shifted is 0,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1.

Example:

Key init(5b5a5767, 6a56676e)

PC1(Key) C=00ffd820, D=ffec9370

KeyRnd01 C1=01ffb040, D1=ffd926f0, PC2(C,D)=(38 09 1b 26 2f 3a 27 0f)

KeyRnd02 C2=03ff6080, D2=ffb24df0, PC2(C,D)=(28 09 19 32 1d 32 1f 2f)

KeyRnd03 C3=0ffd8200, D3=fec937f0, PC2(C,D)=(39 05 29 32 3f 2b 27 0b)

KeyRnd04 C4=3ff60800, D4=fb24dff0, PC2(C,D)=(29 2f 0d 10 19 2f 1d 3f)

KeyRnd05 C5=ffd82000, D5=ec937ff0, PC2(C,D)=(03 25 1d 13 1f 3b 37 2a)

KeyRnd06 C6=ff608030, D6=b24dfff0, PC2(C,D)=(1b 35 05 19 3b 0d 35 3b)

KeyRnd07 C7=fd8200f0, D7=c937ffe0, PC2(C,D)=(03 3c 07 09 13 3f 39 3e)

KeyRnd08 C8=f60803f0, D8=24dfffb0, PC2(C,D)=(06 34 26 1b 3f 1d 37 38)

KeyRnd09 C9=ec1007f0, D9=49bfff60, PC2(C,D)=(07 34 2a 09 37 3f 38 3c)

KeyRnd10 C10=b0401ff0, D10=26fffd90, PC2(C,D)=(06 33 26 0c 3e 15 3f 38)

KeyRnd11 C11=c1007fe0, D11=9bfff640, PC2(C,D)=(06 02 33 0d 26 1f 28 3f)

KeyRnd12 C12=0401ffb0, D12=6fffd920, PC2(C,D)=(14 16 30 2c 3d 37 3a 34)

KeyRnd13 C13=1007fec0, D13=bfff6490, PC2(C,D)=(30 0a 36 24 2e 12 2f 3f)

KeyRnd14 C14=401ffb00, D14=fffd9260, PC2(C,D)=(34 0a 38 27 2d 3f 2a 17)

KeyRnd15 C15=007fec10, D15=fff649b0, PC2(C,D)=(38 1b 18 22 1d 32 1f 37)

KeyRnd14 C14=401ffb00, D14=fffd9260, PC2(C,D)=(34 0a 38 27 2d 3f 2a 17)

Table 6.1

Initial Permutation

58, 50, 42, 34, 26, 18, 10, 2, 60, 52, 44, 36, 28, 20, 12, 4,
62, 54, 46, 38, 30, 22, 14, 6, 64, 56, 48, 40, 32, 24, 16, 8,
57, 49, 41, 33, 25, 17, 9, 1, 59, 51, 43, 35, 27, 19, 11, 3,
61, 53, 45, 37, 29, 21, 13, 5, 63, 55, 47, 39, 31, 23, 15, 7.

Table 6.2

Key Permutation (PC-1)

57, 49, 41, 33, 25, 17, 9, 1, 58, 50, 42, 34, 26, 18,
10, 2, 59, 51, 43, 35, 27, 19, 11, 3, 60, 52, 44, 36,
63, 55, 47, 39, 31, 23, 15, 7, 62, 54, 46, 38, 30, 22,
14, 6, 61, 53, 45, 37, 29, 21, 13, 5, 28, 20, 12, 4.

Table 6.3

Number of Key Bits Shifted per Round

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

Table 6.4

Compression Permutation (PC-2)

14, 17, 11, 24, 1, 5, 3, 28, 15, 6, 21, 10,
23, 19, 12, 4, 26, 8, 16, 7, 27, 20, 13, 2,
41, 52, 31, 37, 47, 55, 30, 40, 51, 45, 33, 48,
44, 49, 39, 56, 34, 53, 46, 42, 50, 36, 29, 32.

Table 6.5

Expansion Permutation

32, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9,
8. 9, 10, 11, 12, 13, 12, 13, 14, 15, 16, 17,
16, 17, 18, 19, 20, 21, 20, 21, 22, 23, 24, 25,
24, 25, 26, 27, 28, 29, 28, 29, 30, 31, 32, 1

The S-Box Substitution

After the compressed key is XORed with the expanded block, the 48-bit result moves to a substitution operation. The substitutions are performed by eight substitution boxes, or S

Table 6.6

S-Boxes

S-box 1:

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

S-box 2:

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

S-box 3:

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

S-box 4:

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

S-box 5:

2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9,
14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
41, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3,

S-box 6:

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

S-box 7:

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

S-box 8:

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

The P-Box Permutation

The 32-bit output of the S-box substitution is permuted according to a P-box. This permutation maps each input bit to an output position; no bits are used twice and no bits are ignored. Table 12.7 shows the position to which each bit moves. For example, bit 21 moves to bit 4. while bit 4 moves to bit 3 1.

Table 6.7

P-Box Permutation

16, 7, 20, 21, 29, 12, 28, 17, 1, 15, 23, 26, 5, 18, 31, 10,
2, 8, 24, 14, 32, 27, 3, 9, 19, 13, 30, 6, 22, 11, 4, 25

Finally, the result of the P-box permutation is XORed with the left half of the initial 64-bit block. Then the left and right halves are switched and another round begins.

The Final Permutation

The final permutation is the inverse of the initial permutation and is described in Table 6.8. Note that the left and right halves are not exchanged after the last round of DES; instead the concatenated block R16L16 is used as the input to the final permutation. There's nothing going on here; exchanging the halves and shifting around the permutation would yield exactly the same result. This is so that the algorithm can be used to both encrypt and decrypt.

Summary

DES is a widely used symmetric block cipher. It encrypts 64-bit data, and uses 56-bit key with 16 48-bit sub-keys. The basic process in enciphering a 64-bit data block using the DES
consists of:

- **an initial permutation (IP).**
- **16 rounds of a complex key dependent calculation f**
- **A final permutation, being the inverse of IP.**

Hardware and Software Implementations of DES Much has been written on efficient hardware and software implementations of the algorithm [997,81,533,534,437,738,1573,176,271,1572]. At this writing, the record holder for the fastest DES chip is a prototype developed at Digital Equipment

Corporation [512]. It supports ECB and CBC modes and is based on a GaAs gate array of 50,000 transistors. Data can be encrypted and decrypted at a rate of 1 gigabit per second, which translates to 16.8 million blocks per second. This is impressive. Table 12.9 gives the specifications for some commercial DES chips. Seeming discrepancies between clock speed and data rate are due to pipelining within the chip; a chip might have multiple DES engines working in parallel.

The most impressive DES chip is VLSI's 6868 (formerly called "Gatekeeper"). Not only can it perform DES encryption in only 8 clock cycles (prototypes in the lab can do it in 4 clock cycles), but it can also do ECB triple-DES in 25 clock cycles, and OFB or CBC triple-DES in 35 clock cycles. This sounds impossible to me, too, but I assure you it works. The most impressive DES chip is VLSI's 6868 (formerly called "Gatekeeper"). Not only can it perform DES encryption in only 8 clock cycles (prototypes in the lab can do it in 4 clock cycles), but it can also do ECB triple-DES in 25 clock cycles, and OFB or CBC triple-DES in 35 clock cycles. This sounds impossible to me, too, but I assure you it works.

A software implementation of DES on an IBM 3090 mainframe can perform 32,000 DES encryptions per second. Most microcomputers are slower, but impressive nonetheless. Table 12.10 [603,793] gives actual results and estimates for various Intel and Motorola microprocessors.

The Real Design Criteria

After differential cryptanalysis became public, IBM published the design criteria for the S-boxes and the P-box . The **criteria** for the S-boxes are:

- Each S-box has 6 input bits and 4 output bits. (This was the largest size that could be accommodated in a single chip with 1974 technology.)
- No output bit of an S-box should be too close to a linear function of the input bits.
- If you fix the left-most and right-most bits of an S-box and vary the 4 middle bits, each possible 4-bit output is attained exactly once.
- If two inputs to an S-box differ in exactly 1 bit, the outputs must differ in at least 2 bits.
- If two inputs to an S-box differ in the 2 middle bits exactly, the outputs must differ in at least 2 bits.
- If two inputs to an S-box differ in their first 2 bits and are identical in their last 2 bits, the two outputs must not be the same.

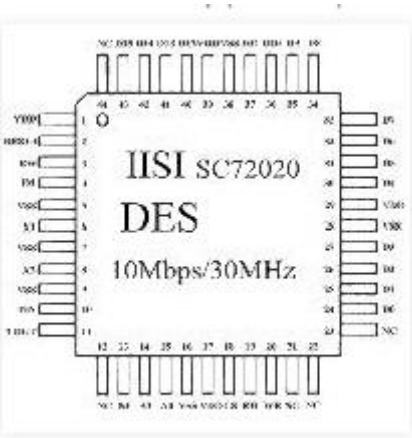
- For any nonzero 6-bit difference between inputs, no more than 8 of the 32 pairs of inputs exhibiting that difference may result in the same output difference.
- A criterion similar to the previous one, but for the case of three active S-boxes.

The criteria for the P-box are:

- The 4 output bits from each S-box in round i are distributed so that 2 of them affect the middle-bits of S-boxes at round*i* + 1 and the other 2 affect end bits.
- The 4 output bits from each S-box affect six different S-boxes; no 2 affect the same S-box.
- If the output bit from one S-box affects a middle bit of another S-box, then an output bit from that other S-box cannot affect a middle bit of the first S-box.

The paper goes on to discuss the criteria. Generating S-boxes is pretty easy today, but was a complicated task in the early 1970s. Tuchman has been quoted as saying that they ran computer programs for months cooking up the S-boxes

Supports ANSI X3.92 Data Encryption Algorithm - DES



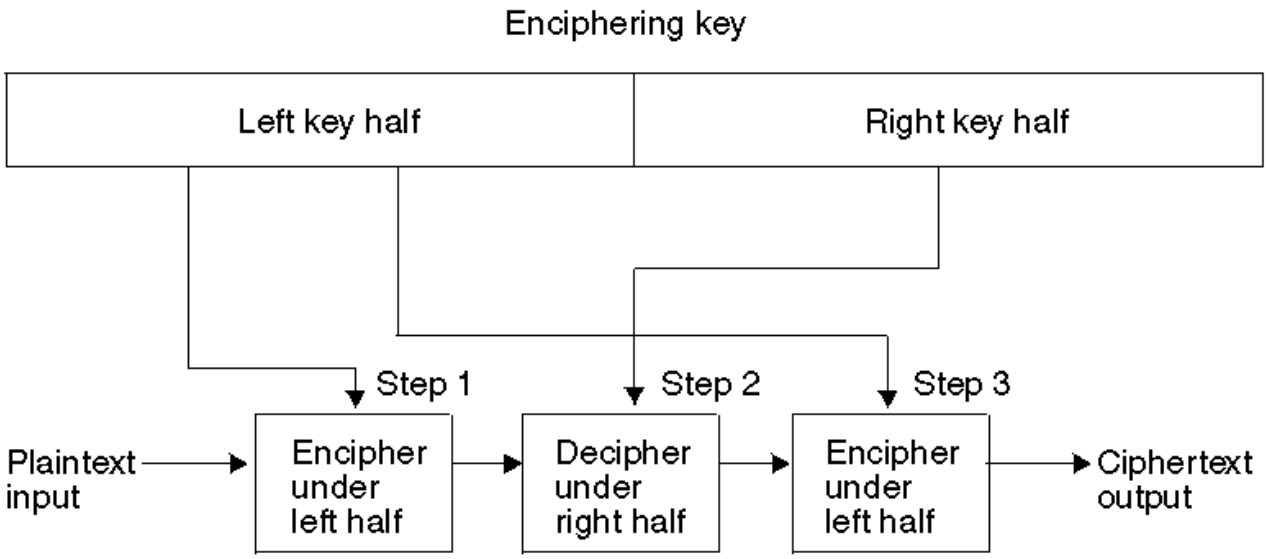
Specifications:

- Size: 18 mm x 18 mm x 3 mm
- Materials: 44-pin plastic QFP
- Transfer Rate: 10 Mbps with 30Mz clock, 667n sec for DES kernel
- 56-bit key: Encrypt/Decrypt 64-bit data words using 56-bit key word
- Byte/Word: Supports BYTE(8-bit)/WORD(16-bit) microprocessor interface
- Clock rate: Up to 30MHz

- Type I/Type II: Type I format(Mortorola chip) and Type II format(Intel Chip)
- Package: Plastic QFP6-44pin
- Features: .Export clearance from R.O.C. required Main Products:

Multiple DES

Some DES implementations use triple-DES (see Figure 12.10) [55]. Since DES is not a group, then the resultant cipher text is much harder to break using exhaustive search: 2^{112} attempts instead of 2^{56} attempts. See Section 15.2 for more details.



DESX

DESX is a DES variant from RSA Data Security, Inc. that has been included in the MailSafe electronic mail security program since 1986 and the BSAFE toolkit since 1987. DESX uses a technique called whitening (see Section 15.6) to obscure the inputs and outputs to DES. In addition to a 56-bit DES key, DESX has an additional 64-bit whitening key. These 64 bits are XORed to the plaintext before the first round of DES. An additional 64 bits, computed as a one-way function of the entire 120-bit DES key, is XORed to the ciphertext after the last round [155]. Whitening makes DESX much stronger than DES

against a brute-force attack; the attack requires $(2^{120})/n$ operations with n known plaintexts. It also improves security against differential and linear cryptanalysis; the attacks require 2^{61} chosen plaintexts and 2^{60} known plaintexts, respectively .

Generalized DES

Generalized DES (GDES) was designed both to speed up DES and to strengthen the algorithm. The overall block size increases while the amount of computation remains constant.

Figure 12.11 is a block diagram of GDES. GDES operates on variable-sized blocks of plaintext. Encryption blocks are divided up into q 32-bit sub-blocks; the exact number depends on the total block size (this was variable in the design, but must be fixed for each implementation). In general, q equals the block size divided by 32.

Function f is calculated once per round on the right-most block. The result is XORed with all the other parts, which are then rotated to the right. GDES has a variable number of rounds, n. There is a slight modification to the last round, so that the encryption and decryption processes differ only in the order of the subkeys (just like DES). In fact, if $q = 2$ and $n = 16$, this *is* DES.

Biham and Shamir [167,168] showed that, using differential cryptanalysis, GDES with $q = 8$ and $n = 16$ is breakable with only six chosen plaintexts. If independent subkeys are also used, 16 chosen plaintexts are required. GDES with $q = 8$ and $n = 22$ is breakable with 48 chosen plaintexts, and GDES with $q = 8$ and $n = 31$ requires only 500,000 chosen plaintexts to break. Even GDES with $q = 8$ and $n = 64$ is weaker than DES; 2^{49} chosen plaintexts are required to break it. In fact, any GDES scheme that is faster than DES is also less secure (see Table 12.15).

A variant of this scheme recently appeared [1591]. It is probably no more secure than the original GDES. In general, any large block DES variant that is faster than DES is probably also less secure than DES.

DES with Alternate S-Boxes

Other DES modifications centered around the S-boxes. Some designs made the order of the S-boxes variable. Other designers varied the contents of the S-boxes themselves. Biham and Shamir showed that the design of the S-boxes, and even the order of the S-boxes themselves, were optimized against differential cryptanalysis:

The replacement of the order of the eight DES S-boxes (without changing their value) also makes DES much weaker: DES with 16 rounds of a particular replaced order is breakable in about 2^{38} steps.... DES with random S-boxes is shown to be very easy to break. Even a minimal change of one entry of one of the DES S-boxes can make DES easier to break.

The DES S-boxes were not optimized against linear cryptanalysis. There are better S-boxes than the ones that come with DES, but blindly choosing new S-boxes isn't a good idea.

Lecture 3:

CAST

CAST was designed in Canada by Carlisle Adams and Stafford Tavares. They claim that the name refers to their design procedure and should conjure up images of randomness, but note the authors' initials. The example CAST algorithm uses a 64-bit block size and a 64-bit key.

The structure of CAST should be familiar. The algorithm uses six S-boxes with an 8-bit input and a 32-bit output. Construction of these S-boxes is implementation-dependent and complicated.

To encrypt, first divide the plaintext block into a left half and a right half. The algorithm has 8 rounds. In each round the right half is combined with some key material using function f and then XORed with **the left half to form the new right half**. The original right half (before the round) becomes the new left half. After 8 rounds (don't switch the left and right halves after the eighth round), the two halves are concatenated to form the ciphertext.

Function f is simple:

(1) Divide the 32-bit input into four 8-bit quarters: *a*, *b*, *c*, *d*.

(2) Divide the 16-bit subkey into two 8-bit halves: *e*, *f*.

(3) Process *a* through S-box 1, *b* through S-box 2, *c* through S-box 3, *d* through S-box 4, *e* through S-box 5, and *f* through S-box 6.

(4) XOR the six S-box outputs together to get the final 32-bit output.

Alternatively, the 32-bit input can be XORed with 32 bits of key, divided into four 8-bit quarters, processed through the S-boxes, and then XORed together. *N* rounds of this appears to be as secure as *N* + 2 rounds of the other option.

The 16-bit subkey for each round is easily calculated from the 64-bit key. If k_1, k_2,..., k_8 are the 8 bytes of the key, then the subkeys for each round are:

Round 1: k_1, k_2

Round 2: k_3, k_4

Round 3: k_5, k_6

Round 4: k_7, k_8

Round 5: k_4, k_3

Round 6: k_2, k_1

Round 7: k_8, k_7

Round 8: k_6, k_5

The strength of this algorithm lies in its S-boxes. CAST does not have fixed S-boxes; new ones are constructed for each application. Design criteria are in; bent functions are the S-box columns, selected for a number of desirable S-box properties. Once a set of S-boxes has been constructed for a given implementation of CAST, they are fixed for all time. The S-boxes are implementation-dependent, but not key-dependent.

The CAST is resistant to differential cryptanalysis , CAST is resistant to linear cryptanalysis. There is no known way to break CAST other than brute force.

Northern Telecom is using CAST in their Entrust security software package for Macintoshes, PCs, and UNIX workstations. The particular S-boxes they chose are not public. The Canadian government is evaluating CAST as a new encryption standard. CAST is patent-pending.

Lecture 4:

GOST

GOST 28147-89

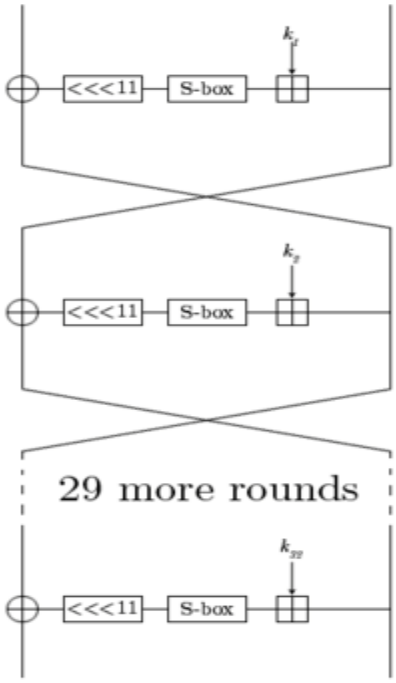


Diagram of GOST

General

GOST

GOST is a block algorithm from the former Soviet Union. "GOST" is an acronym for "Gosudarstvennyi Standard," or Government Standard, sort of similar to a FIPS, except that it can (and does) refer to just about any kind of standard. (Actually, the full name is Gosudarstvennyi Standard Soyuza SSR, or Government Standard of the Union of Soviet Socialist Republics.) This standard is number 28147-89. The Government Committee for Standards of the USSR authorized the standard, whoever they were.

I don't know whether GOST 28147-89 was used for classified traffic or just for civilian encryption. A remark at its beginning states that the algorithm "satisfies all cryptographic requirements and not limits the grade of information to be protected." I have heard claims that it was initially used for very high-grade communications, including classified military communications.

Description of GOST

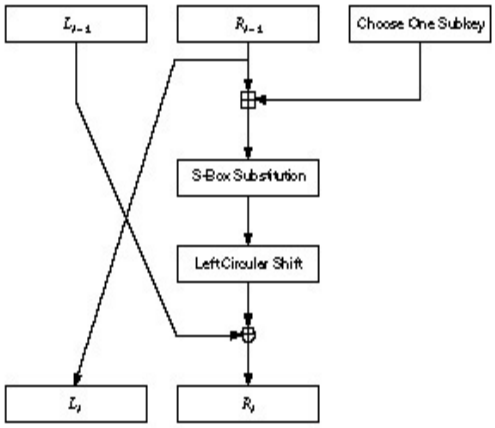
GOST is a 64-bit block algorithm with a 256-bit key. GOST also has some additional key material that will be discussed later. The algorithm iterates a simple encryption algorithm for 32 rounds.

To encrypt, first break the text up into a left half, *L*. and a right half, *R*. The subkey for round *i* is K_i. A round, *i*, of GOST is:

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

Figure is a single round of GOST. Function f is straightforward. First, the right half and the *i*th subkey are added modulo 2^{32}. The result is broken into eight 4-bit chunks, and each chunk becomes the input to a different S-box. There are eight different S-boxes in GOST; the first 4 bits go into the first S-box, the second 4 bits go into the second S-box, and so on. Each S-box is a permutation of the numbers 0 through 15. For example, an S-box might be:



One round of GOST.

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

In this case, if the input to the S-box is 0, the output is 7. If the input is 1, the output is 10, and so on. All eight S-boxes are different; these are considered additional key material. The S-boxes are to be kept secret.

The outputs of the eight S-boxes are recombined into a 32-bit word, then the entire word undergoes an **11-bit left circular shift**. Finally, the result XORed to the left half to become the new right half, and the right half becomes the new left half. Do this 32 times and you're done.

The subkeys are generated simply. The 256-bit key is divided into eight 32-bit blocks: k_1, k_2,..., k_8. Each round uses a different subkey, as shown in following Table. Decryption is the same as encryption with the order of the k_is reversed.

The GOST standard does not discuss how to generate the S-boxes, only that they are somehow supplied. This has led to speculation that some Soviet organization would supply good S-boxes to those organizations it liked and bad S-boxes to those organizations it wished to eavesdrop on. This may very well be true, but further conversations with a GOST chip manufacturer within Russia offered

Use of GOST Subkeys in Different Rounds

Round:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Subkey:	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
Round:	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Subkey:	1	2	3	4	5	6	7	8	8	7	6	5	4	3	2	1

More recently, a set of S-boxes used in an application for the Central Bank of the Russian Federation surfaced. These S-boxes are also used in the GOSTone-way hash function (see section 18.11) [657].They are listed in Table 14.2.

Cryptanalysis of GOST

These are the major differences between DES and GOST.

- DES has a complicated procedure for generating the subkeys from the keys. GOST has a very simple procedure.
- DES has a 56-bit key; GOST has a 256-bit key. If you add in the secret S-box permutations,.

Table
GOST S-Boxes

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

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

- The S-boxes in DES have 6-bit inputs and 4-bit outputs; the S-boxes in GOST have 4-bit inputs and outputs. Both algorithms have eight S-boxes, but an S-box in GOST is one-fourth the size of an S-box in DES.
- DES has an irregular permutation, called a P-box; GOST uses an 11-bit left circular shift.
- DES has 16 rounds; GOST has 32 rounds.

If there is no better way to break GOST other than brute force, it is a very secure algorithm. GOST has a 256-bit key—longer if you count the secret S-boxes. Against differential and linear cryptanalysis, GOST is probably stronger than DES. Although the random S-boxes in GOST are probably weaker than the fixed S-boxes in DES, their secrecy adds to GOST's resistance against differential and linear attacks. Also, both of these attacks depend on the number of rounds: the more rounds, the more difficult the attack. GOST has twice as many rounds as DES; this alone probably makes both differential and linear cryptanalysis infeasible.

The other parts of GOST are either on par or worse than DES. GOST doesn't have the same expansion permutation that DES has. Deleting this permutation from DES weakens it by reducing the avalanche effect; it is reasonable to believe that GOST is weaker for not having it. GOST's use of addition instead is no less secure than DES's XOR.

The greatest difference between them seems to be GOST's cyclic shift instead of a permutation. The DES permutation increases the avalanche effect. In GOST a change in one input bit affects one S-box

in one round, which then affects two S-boxes in the next round, three the round after that, and so on. GOST requires 8 rounds before a single change in an input affects every output bit; DES only requires 5 rounds. This is certainly a weakness. But remember: GOST has 32 rounds to DES's 16.

GOST's designers tried to achieve a balance between efficiency and security. They modified DES's basic design to create an algorithm that is better suited for software implementation. They seem to have been less sure of their algorithm's security, and have tried to compensate by making the key length very large, keeping the S-boxes secret, and doubling the number of iterations. Whether their efforts have resulted in an algorithm more secure than DES remains to be seen.

Lecture 5:

RC5

RC5

The RC5 encryption algorithm was designed by Ronald Rivest of Massachusetts Institute of Technology (MIT) and it first appeared in December 1994. RSA Data Security, Inc. estimates that RC5 and its successor, RC6, are strong candidates for potential successors to DES. RC5 analysis (RSA Laboratories) is still in progress and is periodically updated to reflect any additional findings.

Description of RC5

RC5 is a symmetric block cipher designed to be suitable for both **software** and **hardware** implementation . It is a parameterized algorithm, with a ***variable block size***, a ***variable number of rounds*** and a ***variable-length key***. This provides the opportunity for great **flexibility** in both performance characteristics and the level of security.

A particular RC5 algorithm is designated as RC5-**W/R/B**. The number of bits in a word, W, is a parameter of RC5. Different choices of this parameter result in different RC5 algorithms. RC5 is iterative in structure, with a variable number of rounds. The number of rounds, R, is a second parameter of RC5. RC5 uses a variable-length secret key. The key length **B** (in bytes) is a third parameter of RC5. These parameters are summarized as follows:

W: The word size, in bits. The standard value is ***32bits***; allowable values are 16, 32 and 64. RC5 encrypts two-word blocks so that the plaintext and ciphertext blocks are each 2w bits long.

R: The number of rounds. Allowable values of R are 0, 1, ..., 255. Also, the expanded key table S contains **T = 2 (r + 1) words**.

B: The number of bytes in the secret key K. Allow able values of b are 0, 1, ..., 255 byte.

K:The b-byte secret key; K[0], K[1], ...,K[b − 1]

RC5 consists of three components: a **key expansion algorithm**, an **encryption algorithm** and a **decryption algorithm**. These algorithms use the following three primitive operations:

- ⊞ Addition of words modulo 2^w
- ⊕ Bit-wise exclusive-OR of words
- <<< Rotation symbol: the rotation of x to the left by y bits is denoted **by x <<< y.**

One design feature of RC5 is its simplicity, which makes RC5 easy to implement. Another feature of RC5 is its heavy use of data-dependent rotations in encryption; this feature is very useful in preventing both differential and linear cryptanalysis.

Key Expansion

The key-expansion algorithm expands the user's key K to fill the expanded key table S, so that S resembles an array of t = 2(r + 1) random binary words determined by K. It uses two word-size magic constants P_w and Q_w defined for arbitrary w as shown below:

$$P_w = \text{Odd}((e - 2)2^w)$$

$$Q_w = \text{Odd}((\varphi - 1)2^w)$$

where

e = 2.71828 ... (base of natural logarithms)

$\Phi = (1 + \sqrt{5})/2 = 1.61803$ ... (golden ratio)

Odd(x) is the odd integer nearest to x.

First algorithmic step of key expansion: This step is to copy the secret key K[0, 1, ..., b − 1]into an array L[0, 1, ..., c − 1]of c =[b/u] words, where u = w/8 is the number of bytes/word.

This first step will be achieved by the following pseudocode operation**:**

for i = b – 1 down to 0 do

L[i/u] = (L[i/u] <<< 8) + K[i]

where all bytes are unsigned and the array L is initially zeroes.

Second algorithmic step of key expansion: This step is to initialize array **S** to a particular fixed pseudo-random bit pattern, using an arithmetic progression modulo $\mathbf{2^w}$ determined by two constants $\mathbf{P_w}$ **and** $\mathbf{Q_w}$**.**

$S[0] = P_w$:

for i = 1 to t − 1 do

S[i] = S[i − 1] + Q_w.

Third algorithmic step of key expansion: This step is to mix in the user's secret key in three passes over the arrays **S** and **L**. More precisely, due to the potentially different sizes of S and L, the larger array is processed three times, and the other array will be handled more after.

i = j = 0;

A = B = 0;

do

3∗ max (t,c) times:

A = S[i] = (S[i] + A + B) <<< 3

B = L[j] = (L[j] + A + B) <<< (A + B);

i = (i + 1)(mod t);

j = (j + 1)(mod c).

Note that with the key-expansion function it is not so easy to determine K from S**.**

Example. Since w = 32,r = 12 and b=16

u = w/8 = 32/8 = 4 bytes/word

c =[b/u]=[16/4]= 4 words

t = 2(r + 1) = 2(12 + 1) = 26 words

The plaintext and the user's secret key are given as follows:

Plaintext = eedba521 6d8f4b15

Key = 91 5f 46 19 be 41 b2 51 63 55 a5 01 10 a9 ce 91.

1. **Key expansion**

Two magic constants

P32 = 3084996963 = 0xb7e15163

Q32 = 2654435769 = 0x9e3779b9

Step 1

For i = b − 1 down to 0 do

L[i/u] = (L[i/u] <<< 8) + K[i] where b = 16,u = 4 and L is initially 0.

L[i/4] = L[3] for i = 15, 14, 13 and 12.

L[3] = (L[3] <<< 8) + K[15] = 00 + 91 = 91

L[3] = (L[3] <<< 8) + K[14] = 9100 + ce = 91ce

L[3] = (L[3] <<< 8) + K[13] = 91ce00 + a9 = 91cea9

***L[3] = (L[3] <<< 8) + K[12] = 91cea900 + 10 = 91cea910**

L[i/4] = L[2] for i = 11, 10, 9 and 8.

L[2] = (L[2] <<< 8) + K[11] = 00 + 01 = 01

L[2] = (L[2] <<< 8) + K[10] = 0100 + a5 = 01a5

L[2] = (L[2] <<< 8) + K[9] = 01a500 + 55 = 01a555

***L[2] = (L[2] <<< 8) + K[8] = 01a55500 + 63 = 01a55563**

L[i/4] = L[1] for i = 7, 6, 5 and 4.

L[1] = (L[1] <<< 8) + K[7] = 00 + 51 = 51

L[1] = (L[1] <<< 8) + K[6] = 5100 + b2 = 51b2

L[1] = (L[1] <<< 8) + K[5] = 51b200 + 41 = 51b241

***L[1] = (L[1] <<< 8) + K[4] = 51b24100 + be = 51b241be**

L[i/4] = L[0] for i = 3, 2, 1 and 0.

L[0] = (L[0] <<< 8) + K[3] = 00 + 19 = 19

L[0] = (L[0] <<< 8) + K[2] = 1900 + 46 = 1946

L[0] = (L[0] <<< 8) + K[1] = 194600 + 5f = 19465f

***L[0] = (L[0] <<< 8) + K[0] = 19465f00 + 91 = 19465f91**

Thus, converting the secret key from bytes to words (*) yields:

L[0] = 19465f91

L[1] = 51b241be

L[2] = 01a55563

L[3] = 91cea910

Step 2

S[0] = P_{32}.

For i = 1 to 25 do

S[i] = S[i − 1] + Q_{32}:

S[0] = b7e15163

S[1] = S[0] + Q32 = b7e15163 + 9e3779b9 = 5618cb1c

S[2] = S[1] + Q32 = 5618cb1c + 9e3779b9 = f45044d5

S[3] = S[2] + Q32 = f45044d5 + 9e3779b9 = 9287be8e

When the iterative processes continue up to t − 1 = 2(r + 1) − 1 = 25, we can obtain the expanded key table S as shown below:

S[0] = b7e15163 S[09] = 47d498e4 S[18] = d7c7e065

S[1] = 5618cb1c S[10] = e60c129d S[19] = 75ff5a1e

S[2] = f45044d5 S[11] = 84438c56 S[20] = 1436d3d7

S[3] = 9287be8e S[12] = 227b060f S[21] = b26e4d90

S[4] = 30bf3847 S[13] = c0b27fc8 S[22] = 50a5c749

S[5] = cef6b200 S[14] = 5ee9f981 S[23] = eedd4102.

S[6] = 6d2e2bb9 S[15] = fd21733a S[24] = 8d14babb

S[7] = 0b65a572 S[16] = 9b58ecf3 **S[25]** = 2b4c3474

S[8] = a99d1f2b S[17] = 399066ac

Step 3

i = j = 0;A = B = 0;

3 × max(**t**, c) = 3 × **26** = 78 times

A = S[i] = (S[i] + A + B) <<< 3

B = L[j] = (L[j] + A + B) <<< (A + B)

i = i + 1(mod 26)

j = j + 1(mod 4)

A = S[0] = (b7e15163 + 0 + 0) <<< 3

= b7e15163 <<< 3 = bf0a8b1d

B = L[0] = (19465f91 + bf0a8b1d) <<< (A + B)

= d850eaae <<< bf0a8b1d = db0a1d55

A = S[1] = (5618cb1c + bf0a8b1d + db0a1d55) <<< 3

= f02d738e <<< 3 = 816b9c77

B = L[1] = (51b241be + 816b9c77 + db0a1d55) <<< (A + B)

= ae27fb8a <<< 5c75b9cc = 7fb8aae2

A = S[2] = (f45044d5 + 816b9c77 + 7fb8aae2) <<< 3

= f5748c2e <<< 3 = aba46177

B = L[2] = (01a55563 + aba46177 + 7fb8aae2) <<< (A + B)

= 2d0261bc <<< 2b5d0c59 = 785a04c3

A = S[3] = (9287be8e + aba46177 + 785a04c3) <<< 3

= b68624c8 <<< 3 = b4312645

B = L[3] = (91cea910 + b4312645 + 785a04c3) <<< (A + B)

= be59d418 <<< 2c8b2b08 = 59d418be

...

A = S[25] = (4e0d4c36 + f66a1aaf + 6d7f672f) <<< 3

= b1f6ce14,<<< 3 = 8fb670a5,

B = L[1] = (cdfc2657 + 8fb670a5 + 6d7f672f) <<< (A + B)

= cb31fe2b <<< fd35d7d4 = e2bcb31f

Encryption

The input block to RC5 consists of two w-bit words given in two registers, A and B. The output is also placed in the registers A and B. Recall that RC5 uses an expanded key table, S[0, 1,...,t − 1], consisting of t = 2(r + 1) words. The key-expansion algorithm initializes S from the user's given secret key parameter K. However, the S table in RC5 encryption is not like an S-box used by DES. The encryption algorithm is given in the pseudo code as shown below:

A = A + S[0];

B = B + S[1];

for i = 1 to r do

A = ((A ⊕ B) <<< B) + S[2i];

B = ((B ⊕ A) <<< A) + S[2i + 1];

The output is in the registers A and B.

To encrypt the 64-bit input block, use of the following steps:

- Use the expanded key table S[0, 1, ..., 25]
- Input the plaintext in **two 32-bit registers, A and B.**
- Compute the ciphertext using the RC5 encryption algorithm

Decryption

RC5 decryption is given in the pseudocode as shown below.

For i = r downto1do

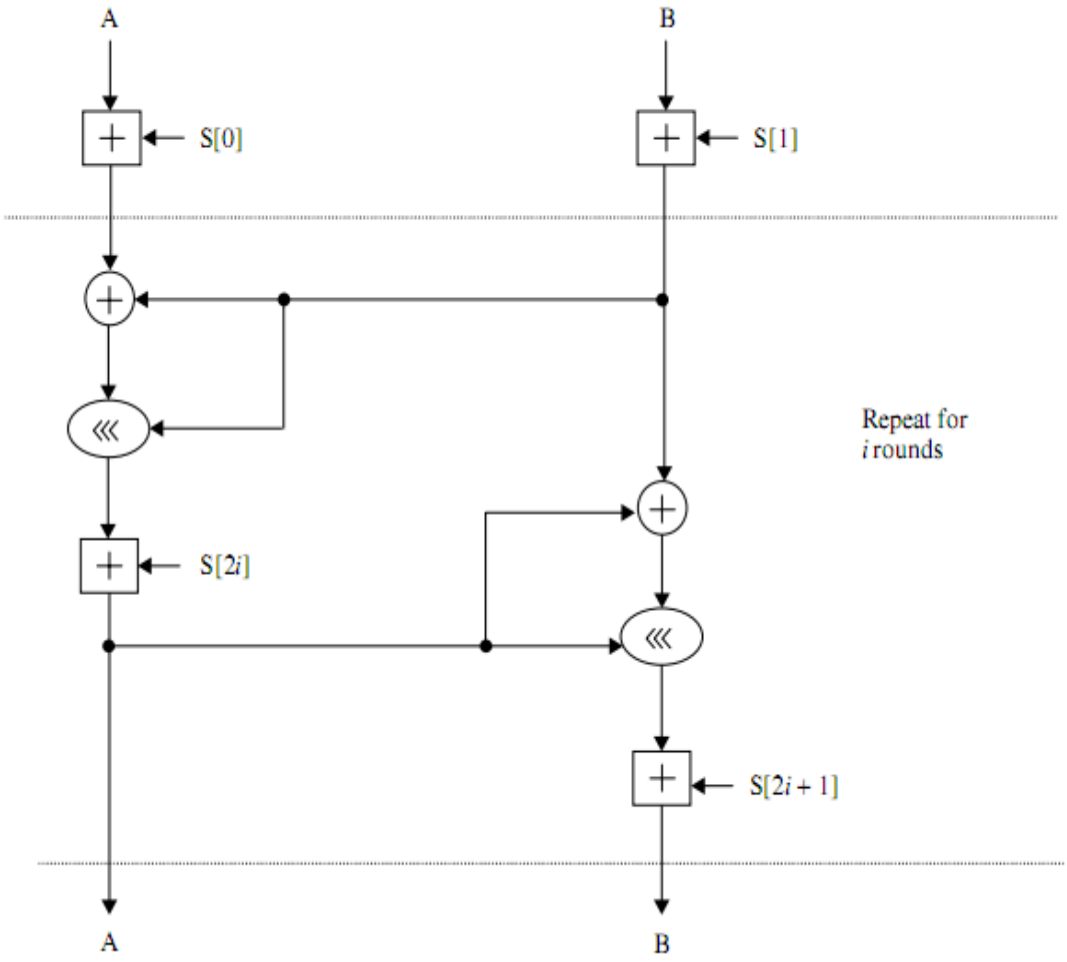
$B = ((B - S[2i + 1]) >>> A) \oplus A$

$A = ((A - S[2i]) >>> B) \oplus B$

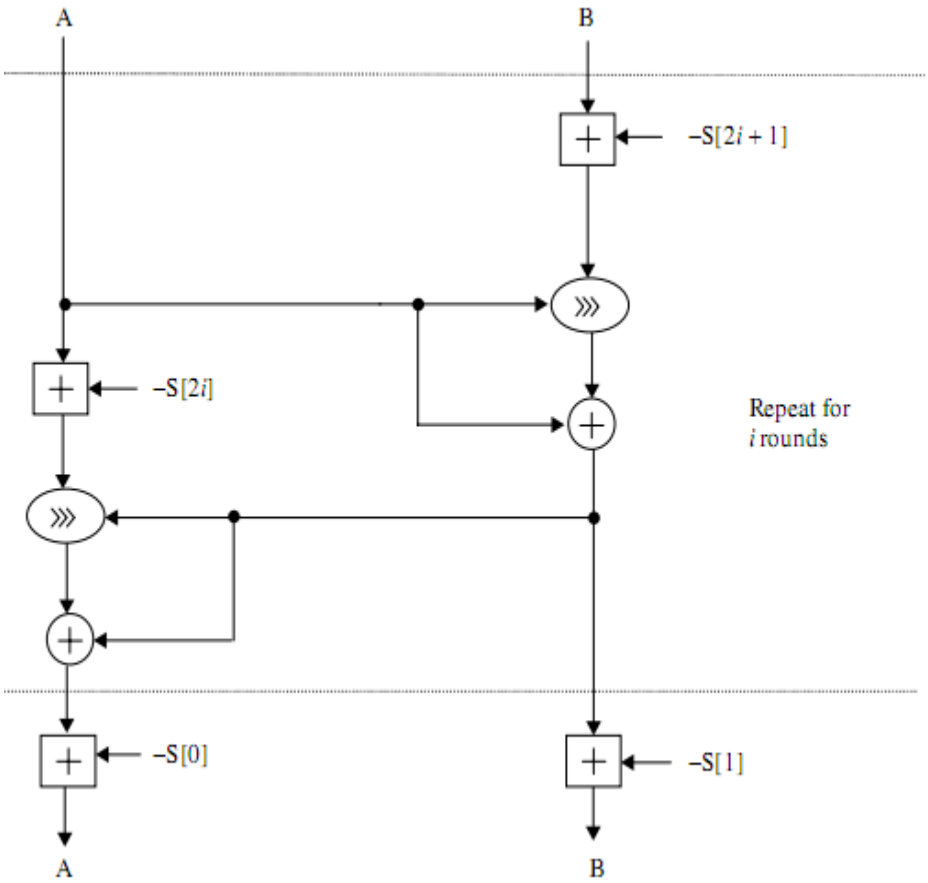
$B = B - S[1]$

$A = A - S[0]$

The decryption routine is easily derived from the encryption routine. The RC5 encryption/decryption algorithms are illustrated as shown in Figures 3.10 and 3.11, respectively.



RC5 encryption algorithm.



RC5 decryption algorithm.

A = ((A ⊕ B) <<< B) + S[2i];

B = ((B ⊕ A) <<< A) + S[2i + 1];

Lecture 6:

Blowfish

Blowfish

Blowfish is optimized for applications where the key does not change often, like a communications link or an automatic file encryptor. It is significantly faster than DES when implemented on 32-bit microprocessors with large data caches, such as the Pentium and the PowerPC. Blowfish is not suitable for applications, such as packet switching, with frequent key changes, or as a one-way hash function. Its large memory requirement makes it infeasible for smart card applications.

1. Fast. Blowfish encrypts data on 32-bit microprocessors at a rate of 26 clock cycles per byte.
2. Compact. Blowfish can run in less than 5K of memory.
3. Simple. Blowfish uses only simple operations: addition, XORs, and table lookups on 32-bit operands. Its design is easy to analyze which makes it resistant to implementation errors.
4. Variably Secure. Blowfish's key length is variable and can be as long as 448 bits.

Description of Blowfish

Blowfish is a 64-bit block cipher with a variable-length key. The algorithm consists of two parts: key expansion and data encryption. Key expansion converts a key of up to 448 bits into several subkey arrays totaling 4168 bytes.

Data encryption consists of a simple function iterated 16 times. Each round consists of a key-dependent permutation, and a key- and data-dependent substitution. All operations are additions and XORs on 32 bit words. The only additional operations are four indexed array data lookups per round.

Blowfish uses a large number of subkeys. These keys must be precomputed before any data encryption or decryption.

The P-array consists of 18 32-bit subkeys:

$$P_1, P_2, ..., P_{18}$$

Four 32-bit S-boxes have 256 entries each:

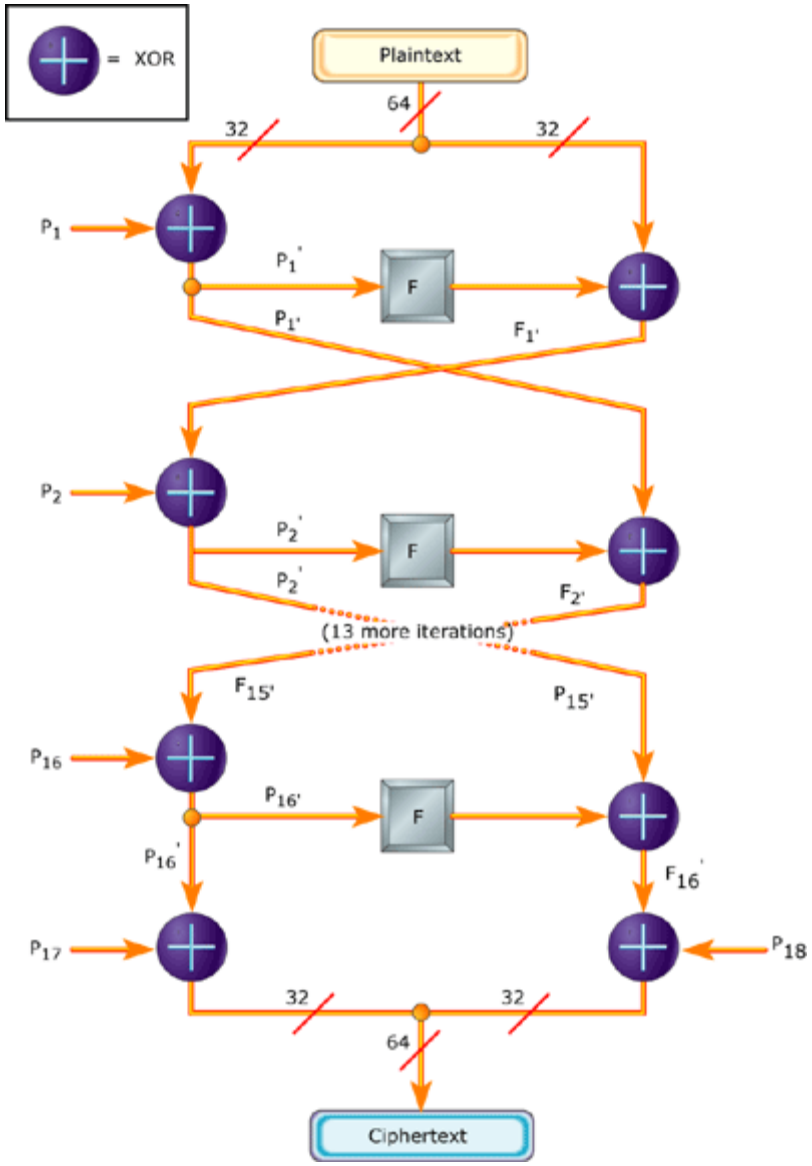
$$S_{1,0}, S_{1,1}, ..., S_{1,255}$$

$$S_{2,0}, S_{2,1}, ..., S_{2,255}$$

$$S_{3,0}, S_{3,1}, ..., S_{3,255}$$

$$S_{4,0}, S_{4,1}, ..., S_{4,255}$$

The exact method used to calculate these subkeys will be described later in this section.



Blowfish is a Feistel network (see Section 14.10) consisting of 16 rounds. The input is a 64-bit data element, x. To encrypt:

Divide x into two 32-bit halves: x_L, x_R

For i = 1 to 16:

$x_L = x_L \oplus P_i$

$x_R = F(x_L) \oplus x_R$

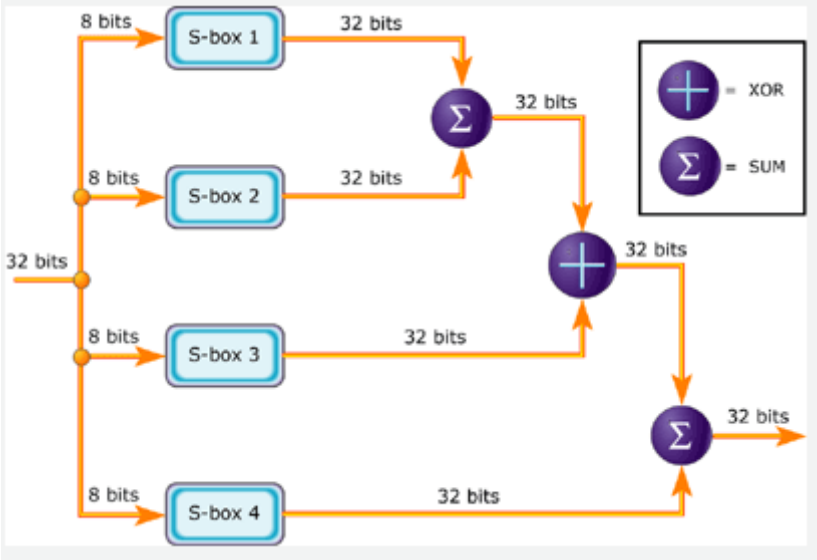
Swap x_L and x_R

Swap x_L and x_R (Undo the last swap.)

$x_R = x_R \oplus P_{17}$

$x_L = x_L \oplus P_{18}$

Recombine x_L and x_R



Function F

Function F is as follows (see Figure 14.3):

Divide x_L into four eight-bit quarters:

a, *b*, *c*, and *d* $F(x_L) = ((S_{1,a} + S_{2,b} \bmod 2^{32}) \oplus S_{3,c}) + S_{4,d} \bmod 2^{32}$

Decryption is exactly the same as encryption, except that P_1, P_2,..., P_{18} are used in the reverse order. Implementations of Blowfish that require the fastest speeds should unroll the loop and ensure that all subkeys are stored in cache.

The subkeys are calculated using the Blowfish algorithm. The exact method follows.

(1) Initialize first the P-array and then the four S-boxes, in order, with a fixed string. This string consists of the hexadecimal digits of p.

(2) XOR P_1 with the first 32 bits of the key, XOR P_2 with the second 32-bits of the key, and so on for all bits of the key (up to P_{18}). Repeatedly cycle through the key bits until the entire P-array has been XORed with key bits.

(3) Encrypt the all-zero string with the Blowfish algorithm, using the subkeys described in steps (1) and (2).

(4) Replace P_1 and P_2 with the output of step (3).

(5) Encrypt the output of step (3) using the Blowfish algorithm with the modified subkeys.

(6) Replace P_3 and P_4 with the output of step (5).

(7) Continue the process, replacing all elements of the P-array, and then all four S-boxes in order, with the output of the continuously changing Blowfish algorithm.

In total, 521 iterations are required to generate all required subkeys. Applications can store the subkeys—there's no need to execute this derivation process multiple times.

NOTE:

P=18 32-bit to byte (div 8)

P=18*4=72

S=256 32-bit to byte (div 8)

S=256*4(byte)*4(number s-box)=4096

The total= 4168

In each round we replace 2 p then 8 round to replace all p

Then

521(number Round)* 8(Number P-)=4168 byte as subkey

Lecture 7:

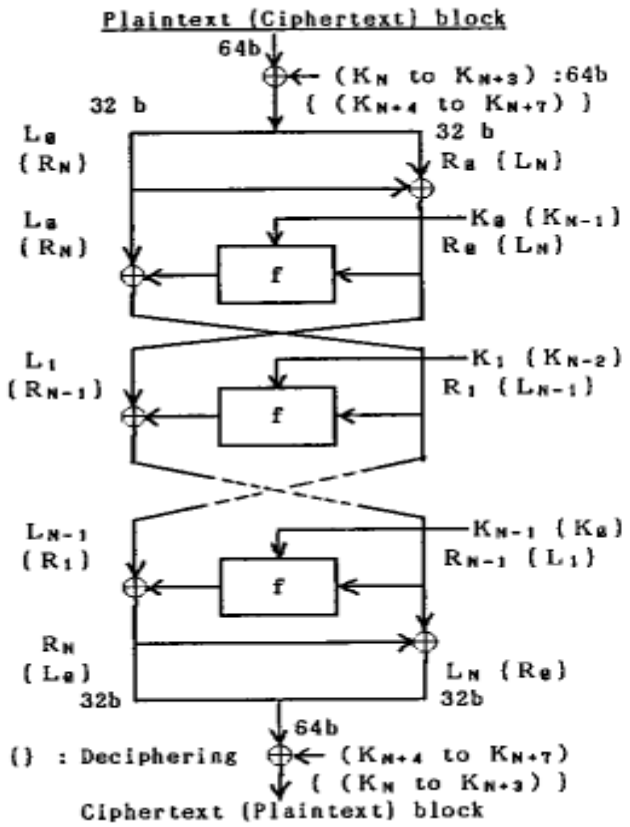
FEAL

FEAL

FEAL was designed by Akihiro Shimizu and Shoji Miyaguchi from NTT Japan. It uses a 64-bit block and a 64-bit key. The idea was to make a DES-like algorithm with a stronger round function. Needing fewer rounds, the algorithm would run faster. Unfortunately, reality fell far short of the design goals.

Description of FEAL

In the following Figure is a block diagram of one round of FEAL. The encryption process starts with a 64-bit block of plaintext. First, the data block is XORed with 64 key bits. The data block is then split into a left half and a right half. The left half is XORed with the right half to form a new right half. The left and new right halves go through *n* rounds (four, initially). In each round the right half is combined with 16 bits of key material (using function f) and XORed with the left half to form the new right half. The original right half (before the round) forms the new left half. After *n* rounds (remember not to switch the left and right halves after the *n*th round) the left half is again XORed with the right half to form a new right half, and then the left and right halves are concatenated together to form a 64-bit whole. The data block is XORed with another 64 bits of key material, and the algorithm terminates.



One round of FEAL.

Function f takes the 32 bits of data and 16 bits of key material and mixes them together. First the data block is broken up into 8-bit chunks, then the chunks are XORed and substituted with each other. Figure 13.4 is a block diagram of function f. The two functions S_0 and S_1, are defined as:

$S_0(a,b)$ = rotate left two bits $((a + b) \bmod 256)$

$S_1(a,b)$ = rotate left two bits $((a + b + 1) \bmod 256)$

The same algorithm can be used for decryption. The only difference is: When decrypting, the key material must be used in the reverse order.

In the following Figure is a block diagram of the key-generating function. First the 64-bit key is divided into two halves. The halves are XORed and operated on by function f_k, as indicated in the diagram. Figure (f_k) is a block diagram of function f_k. The two 32-bit inputs are broken up into 8-bit blocks and combined and substituted as shown. S_0 and S_1 are defined as just shown. The 16-bit key blocks are then used in the encryption/decryption algorithm.

On a 10 megahertz 80286 microprocessor, an assembly-language implementation of FEAL-32 can encrypt data at a speed of 220 kilobits per second. FEAL-64 can encrypt data at a speed of 120 kilobits per second.

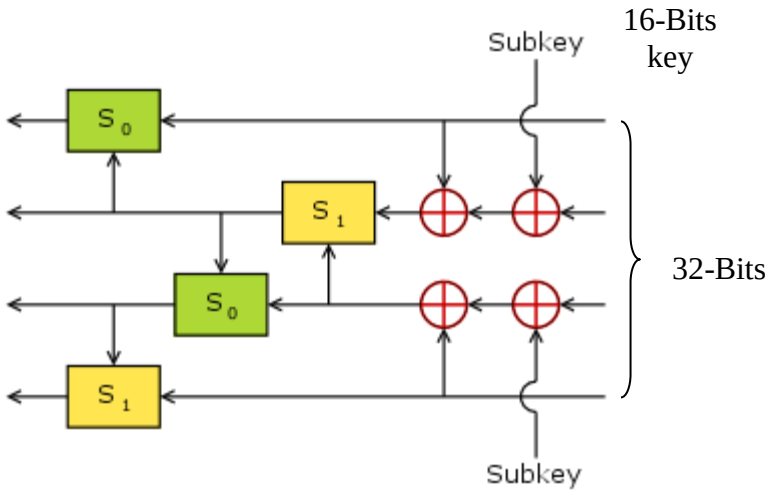


Figure :*Function f.*

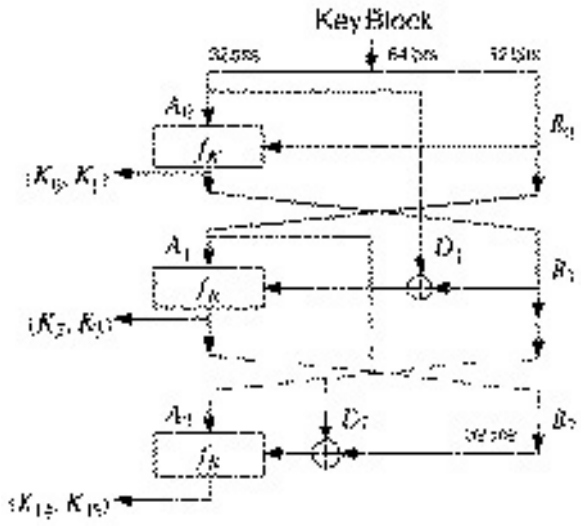


Figure :*Key processing part of FEAL.*

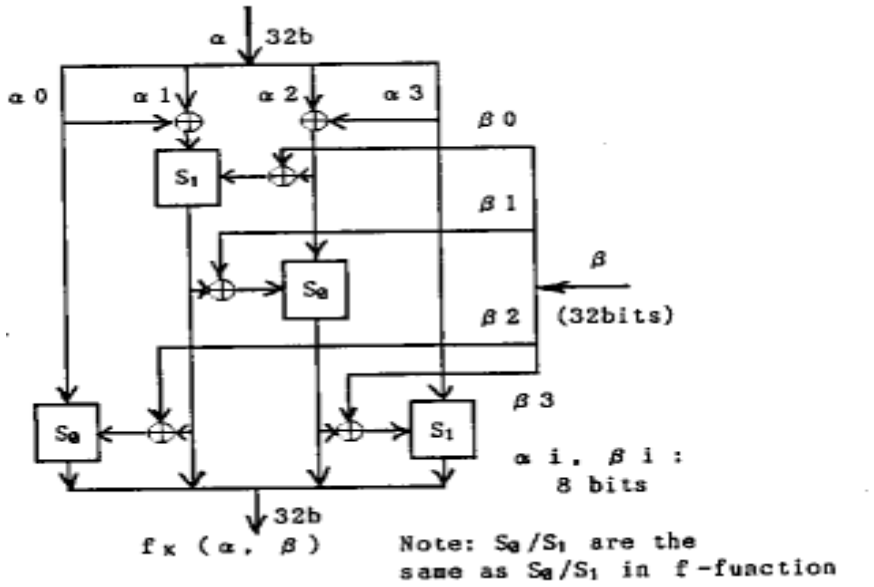


Figure :*Function* f_K.

Cryptanalysis of FEAL

FEAL-4, FEAL with four rounds, was successfully cryptanalyzed with a chosen-plaintext attack in [201] and later demolished. This later attack, by Sean Murphy, was the first published differential-cryptanalysis attack and required only 20 chosen plaintexts. The designers retaliated with 8-round FEAL [1436,1437,1108] which Biham and Shamir cryptanalyzed at the SECURICOM '89 conference[1427]. Another chosen-plaintext attack, using only 10,000 blocks, against FEAL-8 [610] forced the designers to throw up their hands and define FEAL-*N* , with a variable number of rounds (greater than 8, of course).

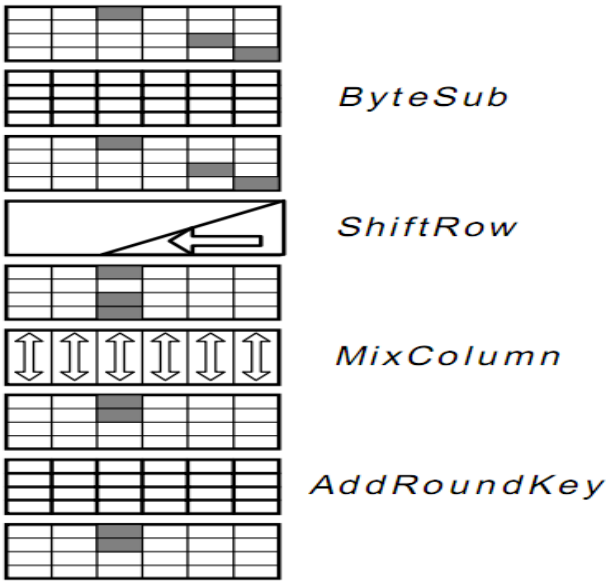
Biham and Shamir used differential cryptanalysis against FEAL-*N*; they could break it more quickly than by brute force (with fever than 2^{64} chosen plaintext encryptions) for *N* less than 32 . FEAL-16 required 2^{28} chosen plaintexts or $2^{46.5}$ known plaintexts to break. FEAL-8 required 2000 chosen plaintexts or $2^{37.5}$ known plaintexts to break. FEAL-4 could be broken with just eight carefully selected chosen plaintexts.

The FEAL designers also defined FEAL-*NX*, a modification of FEAL, that accepts 128-bit keys. Biham and Shamir showed that FEAL-*NX* with a 128-bit key is just as easy to break as FEAL-*N* with a 64-bit

key, for any value of *N*. Recently FEAL-*N*(X)S has been proposed, which strengthens FEAL with a dynamic swapping function.

There's more. Another attack against FEAL-4, requiring only 1000 known plaintexts, and against FEA^{o}L-8, requiring only 20,000 known plaintexts. The best attack is by Mitsuru Matsui and Atshuiro Yamagishi . This is the first use of linear cryptanalysis, and can break FEAL-4 with 5 known plaintexts, FEAL-6 with 100 known plaintexts and FEAL-8 with 2^{15} known plaintexts. Differential-linear cryptanalysis can break FEAL-8 with only 12 chosen plaintexts . Whenever someone discovers a new cryptanalytic attack, he always seems to try it out on FEAL first.

Lecture 8:



RIJNDEAL

1. Design rationale

The three criteria taken into account in the design of Rijndael are the following:

- Resistance against all known attacks;
- Speed and code compactness on a wide range of platforms;
- Design simplicity.

In most ciphers, the round transformation has the Feistel Structure. In this structure typically part of the bits of the intermediate State are simply transposed unchanged to another position. The round transformation of Rijndael does not have the Feistel structure. Instead, the round transformation is composed of three distinct invertible uniform transformations, called layers.

By "uniform", we mean that every bit of the State is treated in a similar way. The specific choices for the different layers are for a large part based on the application of the Wide Trail Strategy, a design method to provide resistance against linear and differential cryptanalysis

- The linear mixing layer: guarantees high diffusion over multiple rounds.
- The non-linear layer: parallel application of S-boxes that have optimum worst-case nonlinearity properties.

- The key addition layer: A simple EXOR of the Round Key to the intermediate State.

Before the first round, a key addition layer is applied. The motivation for this initial key addition is the following. Any layer after the last key addition in the cipher (or before the first in the context of known-plaintext attacks) can be simply peeled off without knowledge of the key and therefore does not contribute to the security of the cipher. (e.g., the initial and final permutation in the DES). Initial or terminal key addition is applied in several designs, e.g., IDEA, SAFER and Blowfish.

In order to make the cipher and its inverse more similar in structure, the linear mixing layer of the last round is different from the mixing layer in the other rounds. It can be shown that this does not improve or reduce the security of the cipher in any way. This is similar to the absence of the swap operation in the last round of the DES

2.Specification

Rijndael is an iterated block cipher with a variable block length and a variable key length. The block length and the key length can be independently specified to 128, 192 or 256 bits.

Note: this section is intended to explain the cipher structure and not as an implementation guideline. For implementation aspects, we refer to Section 5.

3.The State, the Cipher Key and the number of rounds

The different transformations operate on the intermediate result, called the State: Definition: the intermediate cipher result is called the State. The State can be pictured as a rectangular array of bytes. This array has four rows, the umber of columns is denoted by Nb and is equal to the block length divided by 32

The Cipher Key is similarly pictured as a rectangular array with four rows. The number of columns of the Cipher Key is denoted by Nk and is equal to the key length divided by 32. These representations are illustrated in Figure 1.

In some instances, these blocks are also considered as one-dimensional arrays of 4-byte vectors, where each vector consists of the corresponding column in the rectangular array representation. These arrays

hence have lengths of 4, 6 or 8 respectively and indices in the ranges 0..3, 0..5 or 0..7. 4-byte vectors will sometimes be referred to as words.

Where it is necessary to specify the four individual bytes within a 4-byte vector or word the notation (a, b, c, d) will be used where a, b, c and d are the bytes at positions 0, 1, 2 and 3 respectively within the column, vector or word being considered.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Figure 1: Example of State (with Nb = 6) and Cipher Key (with Nk = 4) layout.

The input and output used by Rijndael at its external interface are considered to be one- dimensional arrays of 8-bit bytes numbered upwards from 0 to the **4*Nb−1**. These blocks hence have lengths of **16**, **24** or **32** bytes and array indices in the ranges 0..15, 0..23 or 0..31.

The Cipher Key is considered to be a one-dimensional arrays of 8-bit bytes numbered upwards from 0 to the 4*Nk−1. These blocks hence have lengths of 16, 24 or 32 bytes and array indices in the ranges **0..15, 0..23 or 0..31.**

The cipher input bytes (the "plaintext" if the mode of use is ECB encryption) are mapped onto the state bytes in the order $a_{0,0}$, $a_{1,0}$, $a_{2,0}$, $a_{3,0}$, $a_{0,1}$, $a_{1,1}$, $a_{2,1}$, $a_{3,1}$, $a_{4,1}$ ... , and the bytes of the Cipher Key are mapped onto the array in the order **k0,0, k1,0, k2,0, k3,0, k0,1, k1,1, k2,1, k3,1, k4,1** ... At the end of the cipher operation, the cipher output is extracted from the state by taking the state bytes in the same order.

Hence if the one-dimensional index of a byte within a block is n and the two dimensional index is (i ,j), we have:

$$i = 4 \bmod n; \qquad j=[n/4] \ ; \qquad n=i+j*4;$$

Moreover, the index i is also the byte number within a 4-byte vector or word and j is the index

for the vector or word within the enclosing block.

The number of rounds is denoted by **Nr** and depends on the values **Nb** and **Nk**. It is given in Table 1.

Nr	Nb = 4	Nb = 6	Nb = 8
Nk = 4	10	12	14
Nk = 6	12	12	14
Nk = 8	14	14	14

Table 1: Number of rounds (Nr) as a function of the block and key length.

4. The round transformation

The round transformation is composed of four different transformations. In pseudo C notation we have:

Round(State,RoundKey)

{

ByteSub(State);

ShiftRow(State);

MixColumn(State);

AddRoundKey(State,RoundKey);

}

The final round of the cipher is slightly different. It is defined by:

FinalRound(State,RoundKey)

{

ByteSub(State) ;

ShiftRow(State) ;

AddRoundKey(State,RoundKey);

}

In this notation, the "functions" (**Round, ByteSub, ShiftRow**, …) operate on arrays to which pointers (**State, RoundKey**) are provided.

It can be seen that the final round is equal to the round with the MixColumn step removed. The component transformations are specified in the following subsections.

4.1The ByteSub transformation

The ByteSub Transformation is a non-linear byte substitution, operating on each of the State bytes independently. The substitution table (or S-box) is invertible and is constructed by the composition of two transformations:

1. First, taking the multiplicative inverse in GF(2^8),.
2. Then, applying an affine (over GF(2)) transformation defined.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

The application of the described S-box to all bytes of the State is denoted by:

ByteSub(State) .

Figure 2 illustrates the effect of the **ByteSub** transformation on the State.

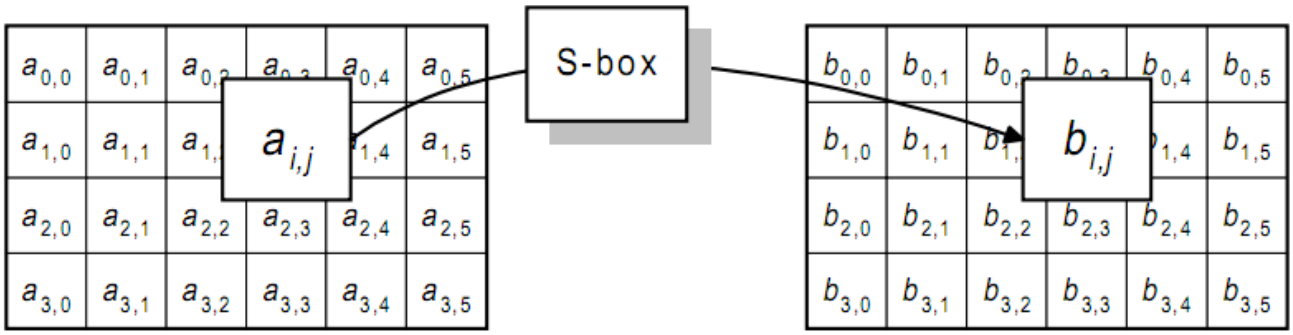


Figure 2: ByteSub acts on the individual bytes of the State.

The inverse of ByteSub is the byte substitution where the inverse table is applied. This is obtained by the inverse of the affine mapping followed by taking the multiplicative inverse in GF(2^8).

4.2 The ShiftRow transformation

In **ShiftRow**, the rows of the State are cyclically shifted over different offsets. *Row 0* is not shifted, *Row 1* is shifted over C1 bytes, *row 2* over C2 bytes and *row 3* over C3 bytes.

The shift offsets C1, C2 and C3 depend on the block length Nb. The different values are specified in Table 2

Nb	C1	C2	C3
4	1	2	3
6	1	2	3
8	1	3	4

Table 2: Shift offsets for different block lengths.

The operation of shifting the rows of the State over the specified offsets is denoted by:

ShiftRow(State) .

Figure 3 illustrates the effect of the ShiftRow transformation on the State.

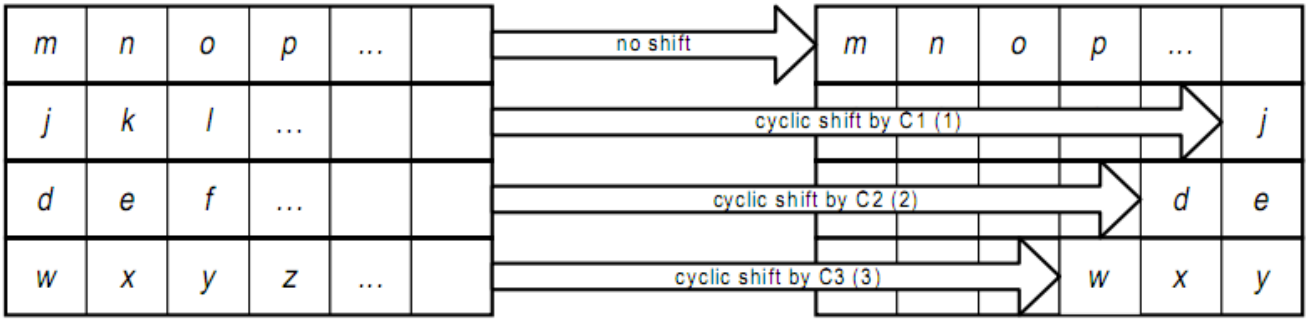


Figure 3: ShiftRow operates on the rows of the State

The inverse of **ShiftRow** is a cyclic shift of the 3 bottom rows over ***Nb-C1, Nb-C2*** and ***Nb-C3*** bytes respectively so that the byte at position j in row i moves to position ***(j + Nb-Ci)*** mod Nb.

4.3 The MixColumn transformation

In MixColumn, the columns of the State are considered as polynomials over GF(2^8) and multiplied modulo $\mathbf{x^4 + 1}$ with a fixed polynomial c(x), given by

$$c(x) = \text{'03'}\, x^3 + \text{'01'}\, x^2 + \text{'01'}\, x + \text{'02'}.$$

This polynomial is coprime to $x^4 + 1$ and therefore invertible. this can be written as a matrix multiplication. Let **b(x) = c(x) ⊗ a(x),**

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

The application of this operation on all columns of the State is denoted by

MixColumn(State).

$b0 = 2a_0 + 3a_1 + 1a_2 + 1a3$

$b1 = 1a_0 + 2a_1 + 3a_2 + 1a_3$

$b2 = 1a_0 + 1a_1 + 2a_2 + 3a_3$

$b3 = 3a_0 + 1a_1 + 1a_2 + 2a_3$

Implementation

r[0] = b[0] ^ a[3] ^ a[2] ^ b[1] ^ a[1]; /* 2 * a0 + a3 + a2 + 3 * a1 */

r[1] = b[1] ^ a[0] ^ a[3] ^ b[2] ^ a[2]; /* 2 * a1 + a0 + a3 + 3 * a2 */

r[2] = b[2] ^ a[1] ^ a[0] ^ b[3] ^ a[3]; /* 2 * a2 + a1 + a0 + 3 * a3 */

r[3] = b[3] ^ a[2] ^ a[1] ^ b[0] ^ a[0]; /* 2 * a3 + a2 + a1 + 3 * a0 */

InverseMixColumns

$$\begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Or:

$$r_0 = 14a_0 + 9a_3 + 13a_2 + 11a_1$$
$$r_1 = 14a_1 + 9a_0 + 13a_3 + 11a_2$$
$$r_2 = 14a_2 + 9a_1 + 13a_0 + 11a_3$$
$$r_3 = 14a_3 + 9a_2 + 13a_1 + 11a_0$$

Figure 4 illustrates the effect of the MixColumn transformation on the State.

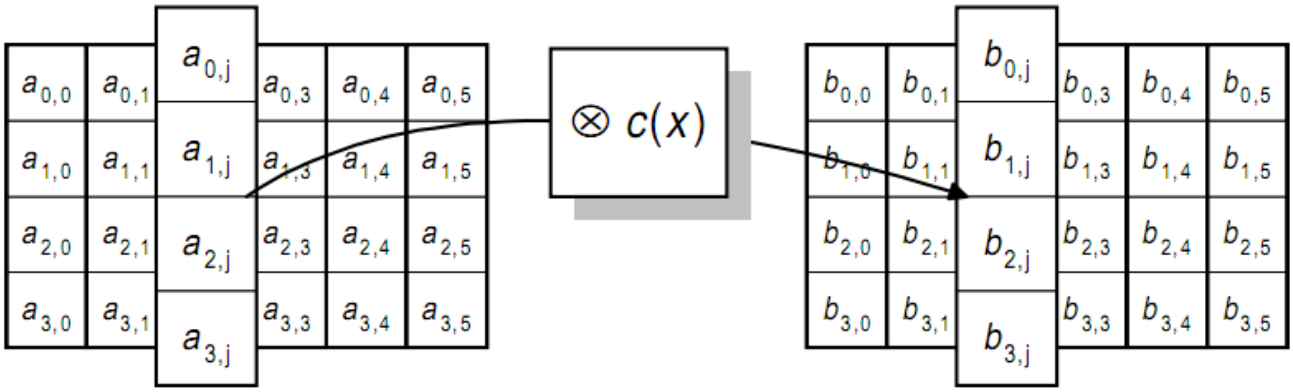


Figure 4: MixColumn operates on the columns of the State.

The inverse of MixColumn is similar to MixColumn. Every column is transformed by multiplying it with a specific multiplication polynomial d(x), defined by

$$(\text{'03'}\ x^3 + \text{'01'}\ x^2 + \text{'01'}\ x + \text{'02'}) \otimes d(x) = \text{'01'}.$$

It is given by:

$$d(x) = \text{'0B'}\ x^3 + \text{'0D'}\ x^2 + \text{'09'}\ x + \text{'0E'}.$$

4.4 The Round Key addition

In this operation, a Round Key is applied to the State by a simple bitwise EXOR. The Round Key is derived from the Cipher Key by means of the key schedule. The Round Key length is equal to the block length **Nb**.

The transformation that consists of EXORing a Round Key to the State is denoted by:

AddRoundKey(State,RoundKey).

This transformation is illustrated in Figure 5.

Figure 5: In the key addition the Round Key is bitwise EXORed to the State.

AddRoundKey is its own inverse.

5. Key schedule

The Round Keys are derived from the Cipher Key by means of the key schedule. This consists of two components: the Key Expansion and the Round Key Selection. The basic principle is the following:

- The total number of Round Key bits is equal to the block length multiplied by the number of rounds plus 1. (e.g., for a block length of 128 bits and 10 rounds, 1408 Round Key bits are needed).
- The Cipher Key is expanded into an Expanded Key.
- Round Keys are taken from this Expanded Key in the following way: the first Round Key consists of the first Nb words, the second one of the following Nb words, and so on.

5.1 Key expansion

The Expanded Key is a linear array of **4-byte** words and is denoted by **W[Nb*(Nr+1)].** The first **Nk** words contain the Cipher Key. All other words are defined recursively in terms of words with smaller indices. The key expansion function depends on the value of **Nk**: there is a version for **Nk** equal to or below **6**, and a version for **Nk** above **6.**

For Nk ⩽6, we have:

KeyExpansion(byte Key[4*Nk] word W[Nb*(Nr+1)])

{

```
    for(i = 0; i < Nk; i++)
   W[i] = (Key[4*i],Key[4*i+1],Key[4*i+2],Key[4*i+3]);
        for(i = Nk; i < Nb * (Nr + 1); i++)
        {
        temp = W[i - 1];
       if (i % Nk == 0)
       temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
      W[i] = W[i - Nk] ^ temp;
      }
}
```

In this description, **SubByte(W)** is a function that returns a **4-byte** word in which each byte is the result of applying the **Rijndael S-box** to the byte at the corresponding position in the input word. The function **RotByte(W)** returns a word in which the bytes are a cyclic permutation of those in its input such that the input word *(a,b,c,d*) produces the output word *(b,c,d,a).*

It can be seen that the first **Nk** words are filled with the Cipher Key. Every following word **W[i]** is equal to the EXOR of the previous word **W[i-1]** and the word **Nk** positions earlier **W[i-Nk].** For words in positions that are a multiple of **Nk**, a transformation is applied to **W[i-1]** prior to the EXOR and a round constant is EXORed. This transformation consists of a cyclic shift of the bytes in a word (**RotByte**), followed by the application of a table lookup to all four bytes of the word (**SubByte**).

5.2 Round Key selection

Round key i is given by the Round Key buffer words **W[Nb*i]** to **W[Nb*(i+1)]**. This is illustrated in Figure 6.

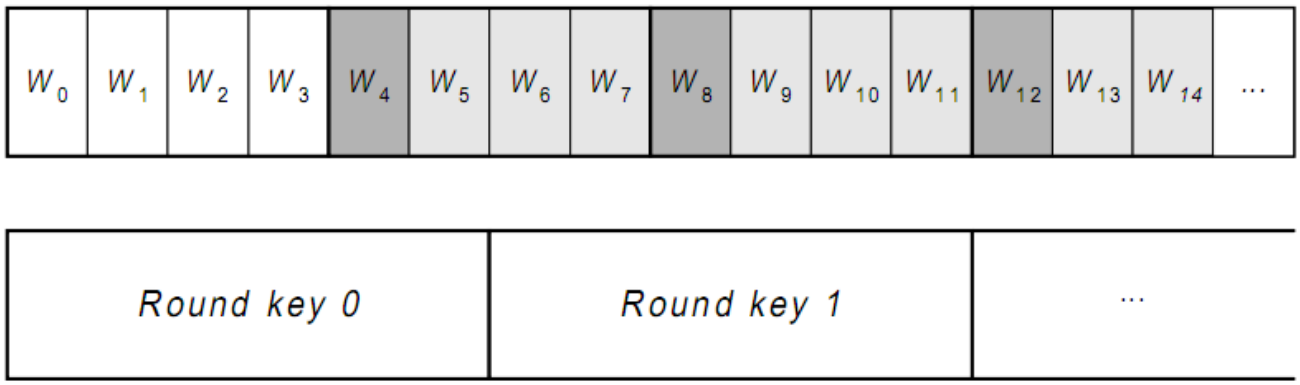


Figure 6: Key expansion and Round Key selection for Nb = 6 and Nk = 4.

Note: The key schedule can be implemented without explicit use of the array W[Nb*(Nr+1)]. For implementations where RAM is scarce, the Round Keys can be computed on-the-fly using a buffer of Nk words with almost no computational overhead.

6.The cipher

The cipher Rijndael consists of

- an initial Round Key addition;
- Nr-1 Rounds;
- a final round.

In pseudo C code, this gives:

```
Rijndael(State,CipherKey)
{
KeyExpansion(CipherKey,ExpandedKey) ;
AddRoundKey(State,ExpandedKey);
For( i=1 ; i<Nr ; i++ ) Round(State,ExpandedKey + Nb*i) ;
FinalRound(State,ExpandedKey + Nb*Nr);
}
```

The key expansion can be done on beforehand and Rijndael can be specified in terms of the Expanded Key.

```
Rijndael(State,ExpandedKey)
{
AddRoundKey(State,ExpandedKey);
For( i=1 ; i<Nr ; i++ ) Round(State,ExpandedKey + Nb*i) ;
FinalRound(State,ExpandedKey + Nb*Nr);
}
```

Note: the Expanded Key shall always be derived from the Cipher Key and never be specified directly. There are however no restrictions on the selection of the Cipher Key itself.

Lecture 9

Serpent Block Cipher Algorithms

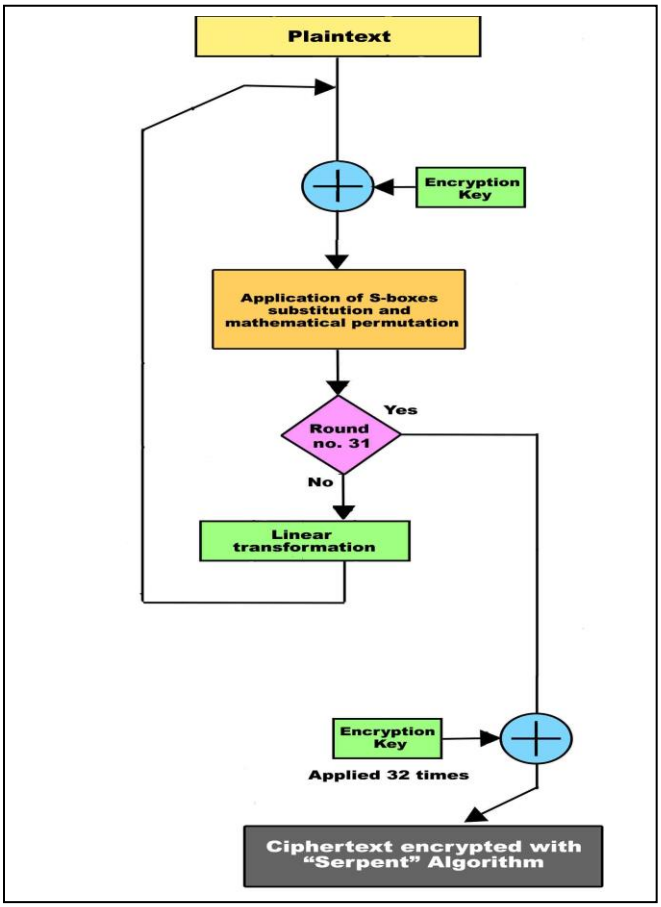
Serpent is a 32-round SP-network operating on four 32-bit words, thus giving a block size of 128 bits. All values used in the cipher are represented as bit streams. The indices of the bits are counted from 0 to bits 31 in one 32-bit word, 0 to bits 127 in 128-bit blocks, 0 to bits 255 in 256-bit keys, and so on. For internal computation, all values are represented in little-endian, where the first word (word 0) is the least significant word, and the last word is the most significant, and where bit 0 is the least significant bit of word 0. Externally, we write each block as a plain 128-bit hex number. Serpent encrypts a 128-bit plaintext *P* to a 128-bit cipher text *C* in 32 rounds under the control of 33 128-bit sub keys *K*0,.....,*K*32. The user key length is variable, but for the purposes of this submission we fix it at 128, 192 or 256 bits, short keys with less than 256 bits are mapped to full-length keys of 256 bits by appending one "1" bit to the MSB end, followed by as many "0" bits as required to make up 256 bits. This mapping is designed to map every short key to a full-length key, with no two short keys being equivalent. The serpent algorithm cipher consists of [31]:

-**Initial permutation** IP.

-**Rounds** :32 rounds, each consisting of a key mixing operation, a pass through S- boxes,and (in all but the last round) a linear transformation. In the last round, this linear transformation is replaced by an additional key mixing operation,

-**Final permutation** FP.

Can explain the algorithm of serpent in the following figure:



The initial and final permutations do not have any cryptographic significance. They are used to simplify an optimized implementation of the cipher, both these two permutations and the linear transformation are to add more complexity .can explain the initial permutation and final permutation in the following

0	32	64	96	1	33	65	97	2	34	66	98	3	35	67	99
4	36	68	100	5	37	69	101	6	38	70	102	7	39	71	103
8	40	72	104	9	41	73	105	10	42	74	106	11	43	75	107
12	44	76	108	13	45	77	109	14	46	78	110	15	47	79	111
16	48	80	112	17	49	81	113	18	50	82	114	19	51	83	115
20	52	84	116	21	53	85	117	22	54	86	118	23	55	87	119
24	56	88	120	25	57	89	121	26	58	90	122	27	59	91	123
28	60	92	124	29	61	93	125	30	62	94	126	31	63	95	127

Figure (2.9): Initial Permutation

0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	124
1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61
65	69	73	77	81	85	89	93	97	101	105	109	113	117	121	125
2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62
66	70	74	78	82	86	90	94	98	102	106	110	114	118	122	126
3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63
67	71	75	79	83	87	91	95	99	103	107	111	115	119	123	127

Figure (2.10): Final Initial Permutation

The initial permutation *IP* is applied to the plaintext *P* giving ^*B*0, which is the input to the first round. The rounds are numbered from 0 to 31, where the first round is round 0 and the last is round 31. The output of the first round (round 0) is ^*B*1, the output of the second round (round 1) is ^*B*2, the output of round *i* is ^*Bi*+1, and so on, until the output of the last round (in which the linear transformation is replaced by an additional key mixing) is denoted by ^*B*32. The final permutation *FP* is now applied to give the cipher text *C*.

Each round function R_i (i € {0… 31*}* uses only a single replicated S-box. For example, *R*0 uses *S*0, 32 copies of which are applied in parallel. Thus the first copy of *S*0 takes bits 0, 1, 2 and 3 of (^*B*0 *XOR* ^*K*0) as its input and returns as output the first four bits of an intermediate vector, the next copy of *S*0 inputs bits 4-7 of(^*B*0 *XOR* ^*K*0) and returns the next four bits of the intermediate vector, and so on. The intermediate vector is then transformed using the linear transformation, giving ^*B1*. Similarly, *R*1 uses 32 copies of *S*1 in parallel on ^*B*1 *XOR* ^*K*1 and transforms their output using the linear transformation, giving ^*B*2.

The set of eight S-boxes is used four times. Thus after using *S*7 in round 7, we use *S*0 again in round 8, then *S*1 in round 9, and so on. The last round *R*31 is slightly different from the others, apply *S*7 on ^*B*31 *XOR* ^*K*31, and XOR the result with ^*K*32 rather than applying the linear transformation. The result ^*B*32 is then permuted by *FP*, giving the cipher text.

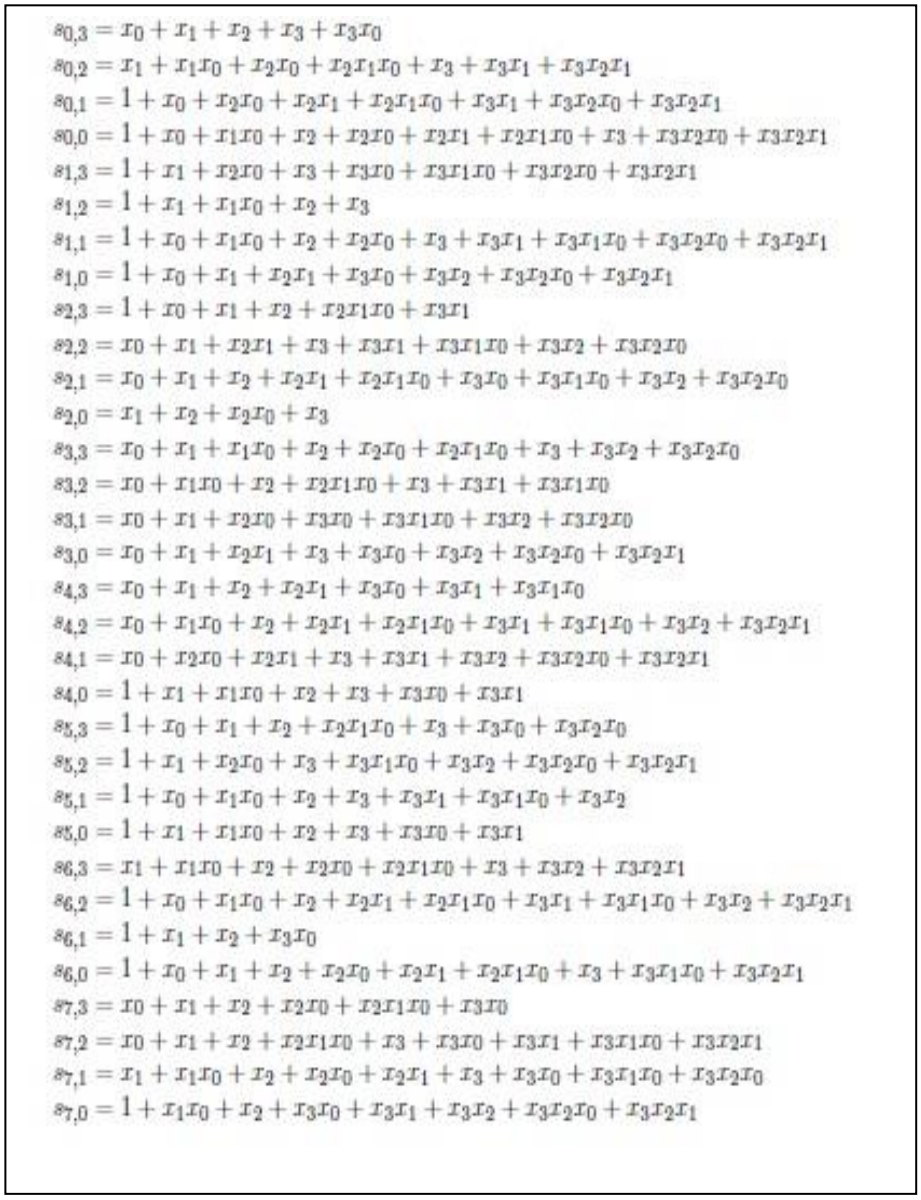
Thus the 32 rounds use 8 different S-boxes each of which maps four input bits to four output bits. Each S-box is used in precisely four rounds, and in each of these it is used 32 times in parallel. Can explain the in the following figure

InvS0:	13	3	11	0	10	6	5	12	1	14	4	7	15	9	8	2
InvS1:	5	8	2	14	15	6	12	3	11	4	7	9	1	13	10	0
InvS2:	12	9	15	4	11	14	1	2	0	3	6	13	5	8	10	7
InvS3:	0	9	10	7	11	14	6	13	3	5	12	2	4	8	15	1
InvS4:	5	0	8	3	10	9	7	14	2	12	11	6	4	15	13	1
InvS5:	8	15	2	9	4	1	13	14	11	6	5	3	7	12	10	0
InvS6:	15	10	1	13	5	3	6	0	4	9	14	7	2	12	8	11
InvS7:	3	0	6	13	9	14	15	8	5	12	11	7	10	1	4	2

The S-boxes used in Serpent for encryption from S_0 to S_7 are given below:

S0:	3	8	15	1	10	6	5	11	14	13	4	2	7	0	9	12
S1:	15	12	2	7	9	0	5	10	1	11	14	8	6	13	3	4
S2:	8	6	7	9	3	12	10	15	13	1	14	4	0	11	5	2
S3:	0	15	11	8	12	9	6	3	13	1	2	4	10	7	5	14
S4:	1	15	8	3	12	0	11	6	2	5	4	10	9	14	7	13
S5:	15	5	2	11	4	10	9	12	0	3	14	8	13	6	7	1
S6:	7	2	12	5	8	4	6	11	14	9	1	15	13	3	10	0
S7:	1	13	15	0	14	8	2	11	7	4	12	10	9	3	5	6

Algebraic Relations of Serpent S-Boxes Let x3, x2, x1, x0 be the input bits to a S-box and let $s_{i,3}$, $s_{i,2}$, $s_{i,1}$, $s_{i,0}$ denote the output bits of the i^{th} S-box S_i. Then, we can represent each of the output bit as a function of the input bits as following figures



$$
\begin{aligned}
s_{0,3} &= x_0 + x_1 + x_2 + x_3 + x_3x_0 \\
s_{0,2} &= x_1 + x_1x_0 + x_2x_0 + x_2x_1x_0 + x_3 + x_3x_1 + x_3x_2x_1 \\
s_{0,1} &= 1 + x_0 + x_2x_0 + x_2x_1 + x_2x_1x_0 + x_3x_1 + x_3x_2x_0 + x_3x_2x_1 \\
s_{0,0} &= 1 + x_0 + x_1x_0 + x_2 + x_2x_0 + x_2x_1 + x_2x_1x_0 + x_3 + x_3x_2x_0 + x_3x_2x_1 \\
s_{1,3} &= 1 + x_1 + x_2x_0 + x_3 + x_3x_0 + x_3x_1x_0 + x_3x_2x_0 + x_3x_2x_1 \\
s_{1,2} &= 1 + x_1 + x_1x_0 + x_2 + x_3 \\
s_{1,1} &= 1 + x_0 + x_1x_0 + x_2 + x_2x_0 + x_3 + x_3x_1 + x_3x_1x_0 + x_3x_2x_0 + x_3x_2x_1 \\
s_{1,0} &= 1 + x_0 + x_1 + x_2x_1 + x_3x_0 + x_3x_2 + x_3x_2x_0 + x_3x_2x_1 \\
s_{2,3} &= 1 + x_0 + x_1 + x_2 + x_2x_1x_0 + x_3x_1 \\
s_{2,2} &= x_0 + x_1 + x_2x_1 + x_3 + x_3x_1 + x_3x_1x_0 + x_3x_2 + x_3x_2x_0 \\
s_{2,1} &= x_0 + x_1 + x_2 + x_2x_1 + x_2x_1x_0 + x_3x_0 + x_3x_1x_0 + x_3x_2 + x_3x_2x_0 \\
s_{2,0} &= x_1 + x_2 + x_2x_0 + x_3 \\
s_{3,3} &= x_0 + x_1 + x_1x_0 + x_2 + x_2x_0 + x_2x_1x_0 + x_3 + x_3x_2 + x_3x_2x_0 \\
s_{3,2} &= x_0 + x_1x_0 + x_2 + x_2x_1x_0 + x_3 + x_3x_1 + x_3x_1x_0 \\
s_{3,1} &= x_0 + x_1 + x_2x_0 + x_3x_0 + x_3x_1x_0 + x_3x_2 + x_3x_2x_0 \\
s_{3,0} &= x_0 + x_1 + x_2x_1 + x_3 + x_3x_0 + x_3x_2 + x_3x_2x_0 + x_3x_2x_1 \\
s_{4,3} &= x_0 + x_1 + x_2 + x_2x_1 + x_3x_0 + x_3x_1 + x_3x_1x_0 \\
s_{4,2} &= x_0 + x_1x_0 + x_2 + x_2x_1 + x_2x_1x_0 + x_3x_1 + x_3x_1x_0 + x_3x_2 + x_3x_2x_1 \\
s_{4,1} &= x_0 + x_2x_0 + x_2x_1 + x_3 + x_3x_1 + x_3x_2 + x_3x_2x_0 + x_3x_2x_1 \\
s_{4,0} &= 1 + x_1 + x_1x_0 + x_2 + x_3 + x_3x_0 + x_3x_1 \\
s_{5,3} &= 1 + x_0 + x_1 + x_2 + x_2x_1x_0 + x_3 + x_3x_0 + x_3x_2x_0 \\
s_{5,2} &= 1 + x_1 + x_2x_0 + x_3 + x_3x_1x_0 + x_3x_2 + x_3x_2x_0 + x_3x_2x_1 \\
s_{5,1} &= 1 + x_0 + x_1x_0 + x_2 + x_3 + x_3x_1 + x_3x_1x_0 + x_3x_2 \\
s_{5,0} &= 1 + x_1 + x_1x_0 + x_2 + x_3 + x_3x_0 + x_3x_1 \\
s_{6,3} &= x_1 + x_1x_0 + x_2 + x_2x_0 + x_2x_1x_0 + x_3 + x_3x_2 + x_3x_2x_1 \\
s_{6,2} &= 1 + x_0 + x_1x_0 + x_2 + x_2x_1 + x_2x_1x_0 + x_3x_1 + x_3x_1x_0 + x_3x_2 + x_3x_2x_1 \\
s_{6,1} &= 1 + x_1 + x_2 + x_3x_0 \\
s_{6,0} &= 1 + x_0 + x_1 + x_2 + x_2x_0 + x_2x_1 + x_2x_1x_0 + x_3 + x_3x_1x_0 + x_3x_2x_1 \\
s_{7,3} &= x_0 + x_1 + x_2 + x_2x_0 + x_2x_1x_0 + x_3x_0 \\
s_{7,2} &= x_0 + x_1 + x_2 + x_2x_1x_0 + x_3 + x_3x_0 + x_3x_1 + x_3x_1x_0 + x_3x_2x_1 \\
s_{7,1} &= x_1 + x_1x_0 + x_2 + x_2x_0 + x_2x_1 + x_3 + x_3x_0 + x_3x_1x_0 + x_3x_2x_0 \\
s_{7,0} &= 1 + x_1x_0 + x_2 + x_3x_0 + x_3x_1 + x_3x_2 + x_3x_2x_0 + x_3x_2x_1
\end{aligned}
$$

It can be seen from above figure that some of the output bits are functions of the input bits with nonlinear order 2, namely $s_{0,3}$, $s_{1,2}$, $s_{2,0}$, $s_{4,0}$, $s_{5,0}$, $s_{6,1}$. Also $s_{4,0}$ and $s_{5,0}$ are identical. Also, we can see that in the case of inverse S-boxes the nonlinear order of the output bits of the Serpent S-box is either 2 or 3. Let x 3, x2, x1, x0 be the input bits to an Inverse S-box and let invsi,3, invsi,2, invsi,2, invsi,0 denote the output bits of the i^{th} inverse S-box $InvS_i$.

$$invs_{0,3} = 1 + x_0 + x_2x_1 + x_3 + x_3x_1x_0 + x_3x_2 + x_3x_2x_0 + x_3x_2x_1$$
$$invs_{0,2} = 1 + x_0 + x_1 + x_1x_0 + x_2 + x_3$$
$$invs_{0,1} = x_0 + x_1 + x_2 + x_2x_0 + x_3x_1 + x_3x_2x_0 + x_3x_2x_1$$
$$invs_{0,0} = 1 + x_1x_0 + x_2 + x_2x_1 + x_3x_0 + x_3x_1 + x_3x_1x_0 + x_3x_2 + x_3x_2x_0 + x_3x_2x_1$$
$$invs_{1,3} = x_0 + x_2 + x_3 + x_3x_1$$
$$invs_{1,2} = 1 + x_0 + x_1 + x_2x_0 + x_2x_1 + x_2x_1x_0 + x_3 + x_3x_2x_0$$
$$invs_{1,1} = x_1 + x_2 + x_2x_1x_0 + x_3 + x_3x_0 + x_3x_1 + x_3x_2x_0 + x_3x_2x_1$$
$$invs_{1,0} = 1 + x_0 + x_1 + x_1x_0 + x_2x_1x_0 + x_3x_1 + x_3x_2x_0 + x_3x_2x_1$$
$$invs_{2,3} = 1 + x_1x_0 + x_2x_1 + x_2x_1x_0 + x_3 + x_3x_2x_0$$
$$invs_{2,2} = 1 + x_0 + x_1x_0 + x_2 + x_3 + x_3x_0 + x_3x_1 + x_3x_1x_0 + x_3x_2x_0$$
$$invs_{2,1} = x_1 + x_1x_0 + x_2 + x_3x_0 + x_3x_1x_0 + x_3x_2 + x_3x_2x_0$$
$$invs_{2,0} = x_0 + x_1 + x_2 + x_2x_1 + x_3x_1$$
$$invs_{3,3} = x_0 + x_1 + x_2 + x_2x_0 + x_2x_1x_0 + x_3x_0 + x_3x_1x_0 + x_3x_2$$
$$invs_{3,2} = x_1x_0 + x_2x_0 + x_2x_1 + x_3x_0 + x_3x_1 + x_3x_1x_0 + x_3x_2 + x_3x_2x_0$$
$$invs_{3,1} = x_1 + x_2 + x_2x_1 + x_2x_1x_0 + x_3 + x_3x_0 + x_3x_2x_0 + x_3x_2x_1$$
$$invs_{3,0} = x_0 + x_2 + x_2x_1 + x_3 + x_3x_0 + x_3x_1 + x_3x_2x_1$$
$$invs_{4,3} = x_1 + x_1x_0 + x_2 + x_3x_0 + x_3x_1x_0 + x_3x_2$$
$$invs_{4,2} = 1 + x_0 + x_1 + x_1x_0 + x_2 + x_2x_0 + x_2x_1x_0 + x_3 + x_3x_1 + x_3x_1x_0$$
$$invs_{4,1} = x_1x_0 + x_2 + x_2x_0 + x_3 + x_3x_0 + x_3x_2x_0$$
$$invs_{4,0} = 1 + x_0 + x_1 + x_2 + x_3 + x_3x_0 + x_3x_1x_0 + x_3x_2 + x_3x_2x_0$$
$$invs_{5,3} = 1 + x_1 + x_1x_0 + x_2 + x_2x_1x_0 + x_3x_0$$
$$invs_{5,2} = x_0 + x_1x_0 + x_2 + x_3x_1 + x_3x_1x_0 + x_3x_2x_0$$
$$invs_{5,1} = x_0 + x_1 + x_2x_0 + x_2x_1 + x_2x_1x_0 + x_3 + x_3x_0 + x_3x_1x_0$$
$$invs_{5,0} = x_0 + x_2x_1 + x_3 + x_3x_1x_0$$
$$invs_{6,3} = 1 + x_1 + x_1x_0 + x_2 + x_2x_1 + x_2x_1x_0 + x_3 + x_3x_0 + x_3x_1x_0 + x_3x_2 + x_3x_2x_1$$
$$invs_{6,2} = 1 + x_0 + x_1 + x_2x_1 + x_3x_1 + x_3x_1x_0 + x_3x_2 + x_3x_2x_1$$
$$invs_{6,1} = 1 + x_1 + x_2 + x_2x_0 + x_3$$
$$invs_{6,0} = 1 + x_0 + x_1x_0 + x_2x_0 + x_2x_1 + x_2x_1x_0 + x_3 + x_3x_1x_0 + x_3x_2x_1$$
$$invs_{7,3} = x_1x_0 + x_2 + x_2x_1x_0 + x_3x_0 + x_3x_1 + x_3x_1x_0$$
$$invs_{7,2} = x_1 + x_2x_0 + x_3 + x_3x_1x_0 + x_3x_2 + x_3x_2x_0$$
$$invs_{7,1} = 1 + x_0 + x_2 + x_2x_1 + x_3 + x_3x_0 + x_3x_1 + x_3x_2x_0 + x_3x_2x_1$$
$$invs_{7,0} = 1 + x_0 + x_1 + x_2x_1 + x_3x_1 + x_3x_1x_0 + x_3x_2 + x_3x_2x_1$$

It can be seen that from the above figure the output bits invs $_{0,2}$, invs$_{1,3}$, invs $_{2,0}$, invs $_{6,1}$ are functions of the input bits with nonlinear order 2 [34].

As with DES, the final permutation is the inverse of the initial permutation. Thus the cipher may be formally described by the following equations:

$$\mathbf{B}_0 = \text{IP (P)}$$
$$\text{\^{}}\mathbf{B_{i+1}} = \text{Ri (\^{}B}_i\text{)}$$
$$\mathbf{C} = \text{FP (\^{}B}_{32}\text{)}$$

Where

$$Ri(X) = L(\text{\^{}}S_i\ (X\ XOR\ \text{\^{}}K_i)) \quad i = \mathbf{0\ldots\ 30}$$
$$R_i(X) = \text{\^{}}S_i(X\ XOR\ \text{\^{}}K_i)\ \text{\^{}}K_{32} \qquad i = \mathbf{31}$$

Where ^ S_i is the application of the S-box $S_{i \bmod 8}$ 32 times in parallel, and L is the linear transformation.

Linear Transformation The 32 bits in each of the output words are linearly mixed, by the equations:

$$
\begin{aligned}
X_0, X_1, X_2, X_3 &:= \mathcal{S}_i(B_i \oplus K_i) \\
X_0 &:= X_0 <<< 13 \\
X_2 &:= X_2 <<< 3 \\
X_1 &:= X_1 \oplus X_0 \oplus X_2 \\
X_3 &:= X_3 \oplus X_2 \oplus (X_0 << 3) \\
X_1 &:= X_1 <<< 1 \\
X_3 &:= X_3 <<< 7 \\
X_0 &:= X_0 \oplus X_1 \oplus X_3 \\
X_2 &:= X_2 \oplus X_3 \oplus (X_1 << 7) \\
X_0 &:= X_0 <<< 5 \\
X_2 &:= X_2 <<< 22 \\
B_{i+1} &:= X_0, X_1, X_2, X_3
\end{aligned}
$$

Where <<< denotes rotation, and << denotes shift. In the last round, this linear transformation is replaced by an additional key mixing: $B_{32} = S7(B31\ XOR\ K31)\ XOR\ K_{32}$. Note that at each stage IP $(B_i) = \hat{}B_i$, and $IP\ (K_i) = \hat{}K_i$. can explain the linear transformation in the following figure:

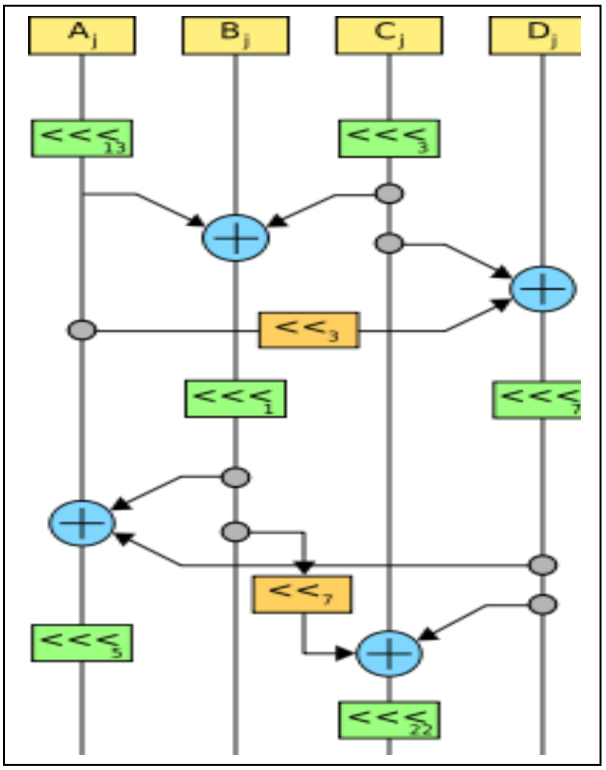


Figure (2.14): Linear Transformation Function

9.1 Key Schedule

As with the description of the cipher, can describe the key schedule in either standard or bitslice mode. Give the substantive description for the latter case.

Our cipher requires 132 32-bit words of key material. We first pad the user supplied key to 256 bits, if necessary. We then expand it to 33 128-bit sub keys $K0\ldots K32$, We write the key K as eight 32-bit words $w_{-8}\ldots w_{-1}$ and expand these to an intermediate key (which we call *pre key*) $w_0\ldots w_{131}$ by the following affine recurrence:

$$w_i = (w_{i-8}\ XOR\ w_{i-5}\ XOR\ w_{i-3}\ XOR\ w_{i-1}\ XOR\ \phi\ XOR\ i) <<< 11$$

Where ϕ is the fractional part of the golden ratio ($\sqrt{5} + 1$) =2 or 0x9e3779b9 in hexadecimal. The underlying polynomial $x^8 + x^7 + x^5 + x^3 + 1$ is primitive, which together with the addition of the round index is chosen to ensure an even distribution of key bits throughout the rounds, and to eliminate weak keys and related keys.

The round keys are now calculated from the pre keys using the S-boxes, again in bits lice mode .can used the S-boxes to transform the pre keys w_i into words k_i of round key in the following way:

$$\{k0, k1, K2, k3\} = S3\ (w0, w1, w2, w3)$$

$$\{k4, k5, k6, k7\} = S2\ (w4, w5, w6, w7)$$

$$\{k8, k9, k10, k11\} = S1\ (w8, w9, w10, w11)$$

$$\{k12, k13, k14, k15\} = S0\ (w12, w13, w14, w15)$$

$$\{k16, k17, k18, k19\} = S7\ (w16, w17, w18, w19)$$

$$: : :$$

$$\{k124, k125, k126, k127\} = S4\ (w124, w125, w126, w127)$$

$$\{k128, k129, k130, k131\} = S3\ (w128, w129, w130, w131)$$

Then renumber the 32-bit values k_j as 128-bit sub keys K_i (for i € {0 . . . r}) as follows:

$$K_i = \{k_{4i}, k_{4i+1}, k_{4i+2}, k_{4i+3}\}$$

2.4.2 Decryption in Serpent Algorithm

Decryption is different from encryption in that the inverse of the S-boxes must be used in the reverse order (inverse of LTF), as well as the inverse linear transformation and reverse order of the sub keys.

9.2 Security of Serpent Algorithm

In [35] the rectangle attack is applied to 256-bit key 10-round Serpent. The attack uses $2^{126.8}$ chosen plaintexts and has a time complexity of 2^{217} memory accesses.1 The 10-round rectangle attack is improved and the improved attack requires $2^{126.3}$ chosen plaintexts with time complexity of $2^{173.8}$ memory accesses. A similar boomerang attack which requires almost the entire code book is also presented in [36].

In [37] a linear attack on 11-round Serpent is presented. The attack exploits a 9-round linear approximation with bias of 2^{-58}. The attack requires data complexity of 2^{118} known plaintexts and time complexity of 2^{214} memory accesses. The linear approximation presented in [6] is combined with a differential in [38] to construct a differential-linear attack on 11-round Serpent. The data complexity of this attack is $2^{125.3}$ chosen plaintexts and the time complexity is about $2^{139.2}$ 11-round Serpent encryptions. The first attack on 10-round Serpent with 128-bit keys is also presented in [38]. The 10-round attack requires $2^{107.2}$ chosen plaintexts and $2^{125.2}$ 10-round Serpent encryptions.