2
Conventional Cryptography

Content

DES: Feistel Scheme, S-boxes
Modes of operation: ECB, CBC, OFB, CFB, CTR, UNIX passwords
Classical designs: IDEA, SAFER K-64, AES
⋆**Case study:** FOX, CS-CIPHER
Stream ciphers: RC4, A5/1, E0
Brute force attacks: exhaustive search, tradeoffs, meet-in-the-middle

In Chapter 1 we saw the foundations of cryptography. Shannon formalized secrecy with the notion of entropy coming from information theory, and proved that secrecy was not possible unless we used (at least) the Vernam cipher. Except for the red telephone application, this is not practical. We can however do some cryptography by changing the model and relying on a computational ability. Before carefully formalizing computability in Chapter 8 we use an intuitive notion of complexity. Indeed, the need of an industry for practical cryptographic solutions pushed toward adopting an empirical notion of secrecy: a cryptographic system provides secrecy until someone finds an attack against it.

We recall that symmetric encryption relies on three algorithms:

- a key generator which generates a secret key in a cryptographically random or pseudorandom way;
- an encryption algorithm which transforms a plaintext into a ciphertext using a secret key;
- a decryption algorithm which transforms a ciphertext back into the plaintext using the secret key.

Symmetric encryption is assumed to enable confidential communications over an insecure channel assuming that the secret key is transmitted over an extra secure channel. Fig. 2.1 represents one possible use of this scheme. Here the secret key is transmitted from the receiver to the sender in a confidential way, and the adversary tries to get information from the ciphertext only.

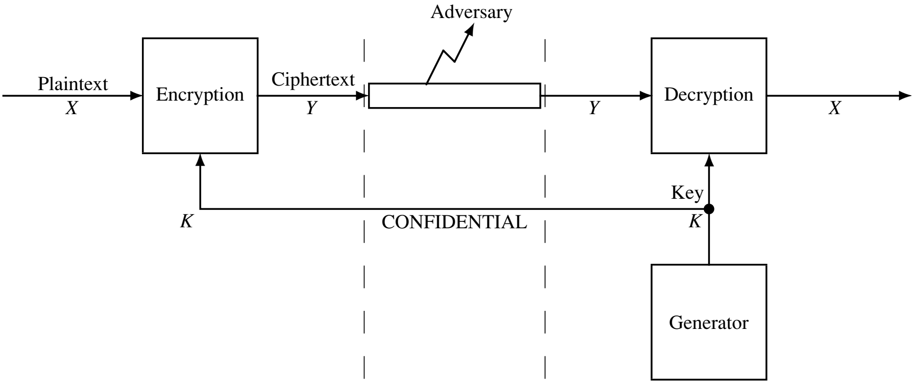

Figure 2.1. Symmetric encryption.

2.1 The Data Encryption Standard (DES)

DES is the Data Encryption Standard which was originally published by the NBS—National Bureau of Standards—a branch of the Department of Commerce in the USA.[1] After a call for proposals, DES was originally proposed as a standard by IBM, based on a previous cipher called LUCIFER developed by Horst Feistel. The US Government (and particularly the NSA) contributed to the development. DES was adopted as a standard and published in 1977 as a FIPS—Federal Information Processing Standard.[2] It was developed based on the need for security in electronic bank transactions. NIST did not renew the standard in 2004 and so the standard is now over. It is however still widely used.

In the seventies, the information technology business was mostly driven by hardware industry (and not by software and service companies). DES was intended for hardware implementations.

DES is a block cipher: it enables the encryption of 64-bit block plaintexts into 64-bit block ciphertexts by using a secret key. It is thus a family of permutations over the set of 64-bit block strings. Encryption of messages of arbitrary length is done through a mode of operation which is separately standardized (see Section 2.2).

The secret key is also a 64-bit string, but eight of these bits are not used at all. Therefore, we usually say that DES uses secret keys of length 56 bits.[3]

DES consists of a 16-round Feistel scheme. A Feistel scheme (named from the author of the previous cipher LUCIFER which was based on a similar structure) is a ladder structure which creates a permutation from a function. Actually, the input string is split into two parts of equal length, and the image of one part through a round function

[1] The NBS is now replaced by the NIST—National Institute of Standards and Technology.

[2] The standard has been updated several times. The 1999 version is available as Ref. [5].

[3] More precisely, the 64-bit key is represented as 8 bytes, and the most significant bit of every byte may be used for parity check.

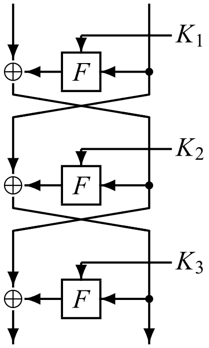

Figure 2.2. Function $\Psi(F^{K_1}, F^{K_2}, F^{K_3})$.

is XORed to the other part. We obtain two parts which are then exchanged (except in the final round). The round function uses subkeys derived from a secret key.

This elementary process is iterated, and the number of round function applications is called the number of rounds. We usually denote $\Psi(F_1, \ldots, F_r)$ the permutation obtained from an r-round Feistel scheme in which the round functions are $F_1, \ldots, F_r$. All F_i may come from a single function F with a parameter K_i defined by a subkey. We denote $F_i = F^{K_i}$. Fig. 2.2 illustrates a 3-round Feistel scheme. DES consists of 16 rounds.

More precisely, DES starts by a bit permutation IP, performs the Feistel cipher using subkeys generated by a key schedule, and finally performs the inverse of the IP permutation. This is illustrated in Fig. 2.3.

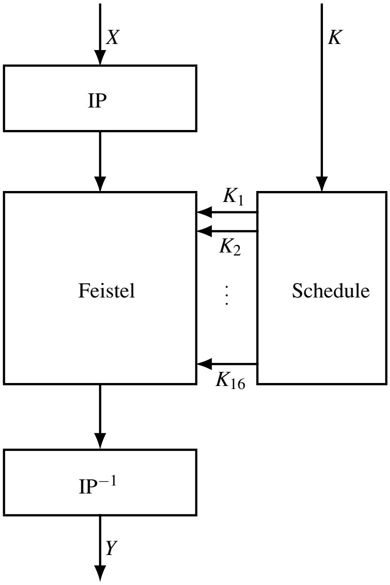

Figure 2.3. DES architecture.

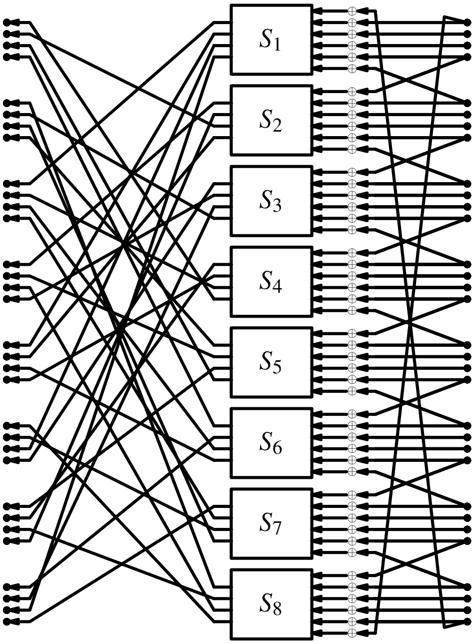

Figure 2.4. DES round function.

The round function of DES has a main 32-bit input, a 48-bit subkey parameter input, and a 32-bit output. For every round, the 48-bit subkey is generated from the secret key by a key schedule. Basically, every 48-bit subkey consists of a permutation and a selection of 48 out of the 56 bits of the secret key. As illustrated in Fig. 2.4, the round function consists of

- an expansion of the main input (one out of two input bits is duplicated) in order to get 48 bits,
- a XOR with the subkey,
- eight substitution boxes which transform a 6-bit input into a 4-bit output,
- a permutation of the final 32 bits (which can be seen as a kind of transposition).

The substitution boxes (called S-boxes) have a 6-bit input and a 4-bit output. We have eight S-boxes called $S_1, S_2, \ldots, S_8$. They are defined by tables in the standard. The tables however need to be read in a special way. For instance, S_3 is defined by

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

The 6-bit input $b_1b_2b_3b_4b_5b_6$ is split into two parts b_1b_6 and $b_2b_3b_4b_5$. The first part b_1b_6 indicates which line to read: 00 is the first line, 01 the second, 10 the third, and 11 the fourth. The second part $b_2b_3b_4b_5$ indicates which column to read in binary. For instance, 0101 is column 5. The entry is the 4-bit output in decimal, to be converted in binary. Hence the image of 001011 by S_3 is 0100 (4).

The DES key schedule is done by the following algorithm. We use two registers C and D of 28 bits. The 56 key bits from K are first split into C and D following a fixed bit selection table PC1. Each round then rotates the bits in C and D by r_i positions depending on the round number i. (The r_i's are also defined by a table.) Then another bit selection table PC2 takes 24 bits from each of the two registers and concatenates them in order to make a round key.

```
$K \xrightarrow{\text{PC1}} (C, D)$
for $i = 1$ to 16 do
    $C \leftarrow \text{ROL}_{r_i}(C)$
    $D \leftarrow \text{ROL}_{r_i}(D)$
    $K_i \leftarrow \text{PC2}(C, D)$
end for
```

Here ROL_r is a circular rotation of r bits to the left. The r_i's are defined by

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

Note that the sum of all r_i's is 28 so that we can generate the round keys in the decryption ordering by starting with the same C and D and by running the loop backwards.

2.2 DES Modes of Operation

DES enables the encryption of 64-bit blocks. In order to encrypt plaintexts of arbitrary length, we have to use DES in a mode of operation. Several modes have been standardized for DES (ECB, CBC, OFB, CFB, CTR) in Ref. [6].

2.2.1 Electronic Code Book (ECB)

The plaintext x is split into 64-bit blocks $x_1, \ldots, x_n$, and the ciphertext y is the concatenation of encrypted blocks.

$$\begin{aligned} x &= x_1||x_2||\cdots||x_n \\ y &= C(x_1)||C(x_2)||\cdots||C(x_n) \end{aligned}$$

(See Fig. 2.5.) There are a few security problems.

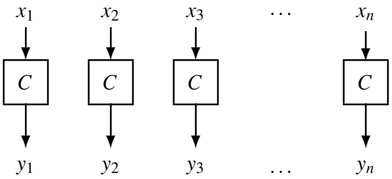

Figure 2.5. ECB mode.

Information Leakage by Block Collisions

If two plaintext blocks are equal (say $x_i = x_j$), then the two corresponding ciphertext blocks are equal. The equality relation is an information which leaks.

This would not be a problem if the plaintext blocks were totally random as the probability of equalities would be reasonably low. However, real plaintexts have lots of redundancy in practice, so equalities are frequent.

Integrity Issues

Although encryption is assumed to protect confidentiality, and not integrity, a third party can intercept the ciphertext and permute two blocks. The legitimate recipient of the modified ciphertext will decrypt the message correctly and obtain two permuted plaintext blocks.

Similarly, a block can be deleted, replaced by another one, *etc.* The plaintext is thus easily malleable by an adversary.

2.2.2 Cipher Block Chaining (CBC)

The plaintext x is split into 64-bit blocks $x_1, \ldots, x_n$, and the ciphertext y is the concatenation of blocks which are obtained iteratively. We have an initial vector IV which is a fake initial block. As illustrated in Fig. 2.6, encryption is performed by the following rules.

$$\begin{aligned} x &= x_1||x_2||\cdots||x_n \\ y_0 &= \mathrm{IV} \\ y_i &= C(y_{i-1} \oplus x_i) \\ y &= y_1||y_2||\cdots||y_n \end{aligned}$$

CBC decryption is easily performed by the following rules.

$$\begin{aligned} y &= y_1||y_2||\cdots||y_n \\ y_0 &= \mathrm{IV} \\ x_i &= y_{i-1} \oplus C^{-1}(y_i) \\ x &= x_1||x_2||\cdots||x_n \end{aligned}$$

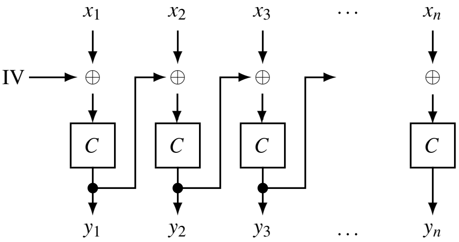

Figure 2.6. CBC mode.

The initial vector does not have to be secret. There are actually four different ways to use the IV.

1. Generate a pseudorandom IV which is given in clear with the ciphertext.
2. Generate a pseudorandom IV which is transmitted in a confidential way.
3. Use a fixed IV which is a known constant.
4. Use a fixed IV which is another part of the secret key.

The US standards recommend one of the two first solutions.

There are a few security problems.

Information Leakage by First Block Collisions

If for two different plaintexts the first blocks x_1 are the same and the IV is fixed, then there is still a leakage of the equality of these blocks. This is why we prefer having a random IV.

Integrity Issues

A third party can replace ciphertext blocks so that all but a few plaintext blocks will decrypt well. This may be an integrity problem.

2.2.3 Output Feedback (OFB)

The plaintext x is split into ℓ-bit blocks $x_1, \ldots, x_n$, and the ciphertext y is the concatenation of blocks which are obtained iteratively. We still have an initial vector IV. As depicted in Fig. 2.7, the encryption obeys the following rules.

$$\begin{aligned} x &= x_1||x_2||\cdots||x_n \\ s_1 &= \mathrm{IV} \\ r_i &= \mathrm{truncL}_\ell(C(s_i)) \\ s_{i+1} &= \mathrm{truncR}_{64}(s_i||r_i) \end{aligned}$$

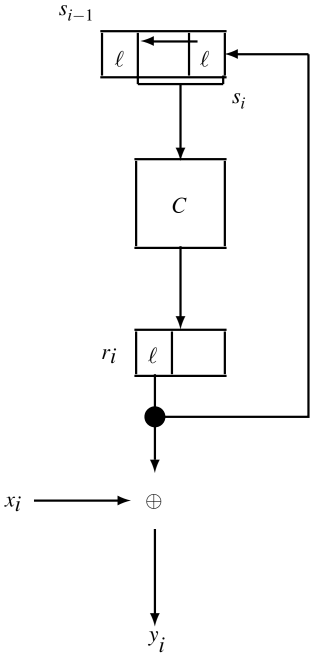

Figure 2.7. OFB mode.

$$y_i = x_i \oplus r_i$$
$$y = y_1 || y_2 || \cdots || y_n$$

Here truncL$_\ell$ truncates the ℓ leftmost bits, and truncR$_{64}$ truncates the 64 rightmost bits. When ℓ is set to the full block length (here 64 bits), the description of the OFB mode is quite simple as illustrated in Fig. 2.8. Note that it is not recommended to use ℓ smaller than the block length due to potential short cycles (see Ref. [57]).

Actually, the OFB mode can be seen as a pseudorandom generator mode which is followed by the one-time pad. Here IV must be used only once (otherwise the cipher is equivalent to a one-time pad with a key used several times). The IV does not have to be secret.

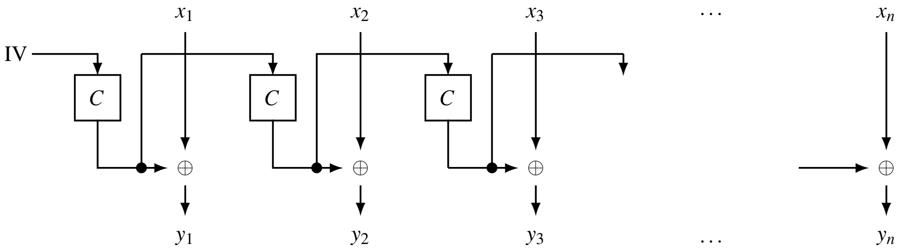

Figure 2.8. OFB mode with ℓ set to the block length.

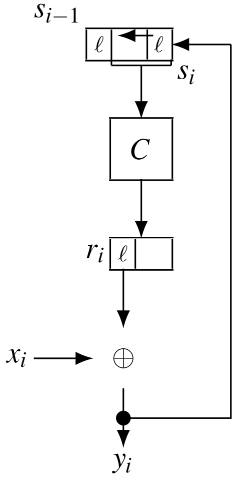

Figure 2.9. CFB mode.

2.2.4 *Cipher Feedback (CFB)*

The plaintext x is split into ℓ-bit blocks $x_1, \ldots, x_n$, and the ciphertext y is the concatenation of blocks which are obtained iteratively. We still have an initial vector IV. As depicted in Fig. 2.9, the encryption is according to the following rules.

$$\begin{aligned} x &= x_1 || x_2 || \cdots || x_n \\ s_1 &= \text{IV} \\ r_i &= \text{truncL}_\ell(C(s_i)) \\ y_i &= x_i \oplus r_i \\ s_{i+1} &= \text{truncR}_{64}(s_i || y_i) \\ y &= y_1 || y_2 || \cdots || y_n \end{aligned}$$

The simple version of the CFB mode with ℓ set to the block length (here 64 bits) is depicted in Fig. 2.10. As for the OFB mode and since the first block is encrypted by a one-time pad, IV need not be secret, but must be fresh (i.e. used only once).

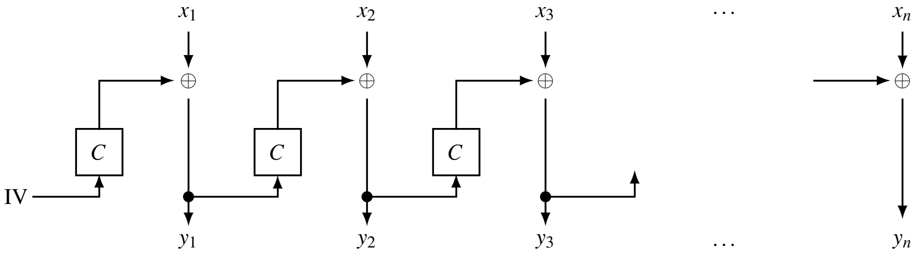

Figure 2.10. CFB mode with ℓ Set to the block length.

2.2.5 Counter Mode (CTR)

The plaintext x is split into ℓ-bit blocks $x_1, \ldots, x_n$, and the ciphertext y is the concatenation of blocks which are obtained iteratively. We use a sequence $t_1, \ldots, t_n$ of counters and the encryption is performed by

$$y_i = x_i \oplus \text{truncL}_\ell(C(t_i)).$$

For a given key, all counters must be pairwise different. For this we can, for instance, let t_i be equal to the binary representation of $t_1 + (i - 1)$ so that each t_i "counts" the block sequence. The initial counter t_1 can either be equal to the latest used counter value stepped by one unit or include a nonce which is specific to the plaintext. In the latter case nonces must be pairwise different.

In Fig. 2.11 the CTR mode with ℓ set to the block length of C is depicted.

2.3 Multiple Encryption

DES relies on a secret key of 56 effective bits, which is rather short. To strengthen its security, people suggested to use multiple DES encryption with several keys.

2.3.1 Double Mode

A first proposal was to use a double mode following the regular product cipher:

$$\text{Enc} = C_{k_1} \circ C_{k_2}$$

One security problem is that we may face meet-in-the-middle attacks (see Section 2.9.5). For this reason double modes are not recommended.

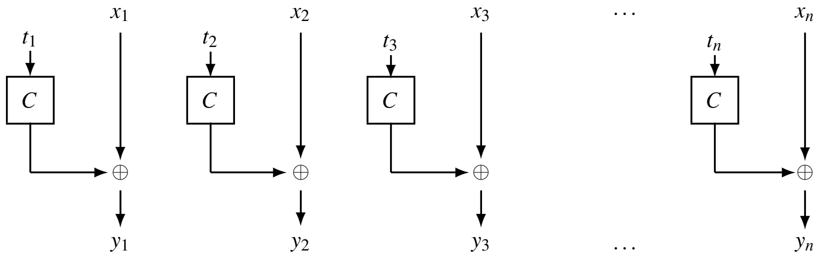

Figure 2.11. CTR mode with ℓ set to the block length.

2.3.2 Triple Mode

Since the double mode does not improve security so much, we use a standard triple mode. The regular triple-DES is defined by three 56-bit keys k_1, k_2, and k_3 by

$$\text{Enc} = C_{k_3} \circ C_{k_2}^{-1} \circ C_{k_1}.$$

(See Ref. [5].) In addition, three keying options are defined:

1. k_1, k_2, k_3 are three independent keys (the key length is thus 168 bits);
2. $k_1 = k_3$ and k_2 are two independent keys (the key length is thus 112 bits);
3. $k_1 = k_2 = k_3$ which is equivalent to DES in simple mode. (This is a kind of retrocompatibility with DES.)

Note that some advanced brute force attacks against triple modes exist as well.

2.4 An Application of DES: UNIX Passwords

A famous application of DES is the old UNIX CRYPT algorithm.[4] It is used for access control of users based on passwords.

Basically, the "encrypted" version of passwords is stored in a database `/etc/passwd` whose confidentiality was not originally meant to be ensured. Whenever a user types his login name and password, the system "encrypts" the password and compares it to the "encrypted" password stored in the corresponding record of the database. The encryption is based on modified DES due to the following observations (see Fig. 2.12).

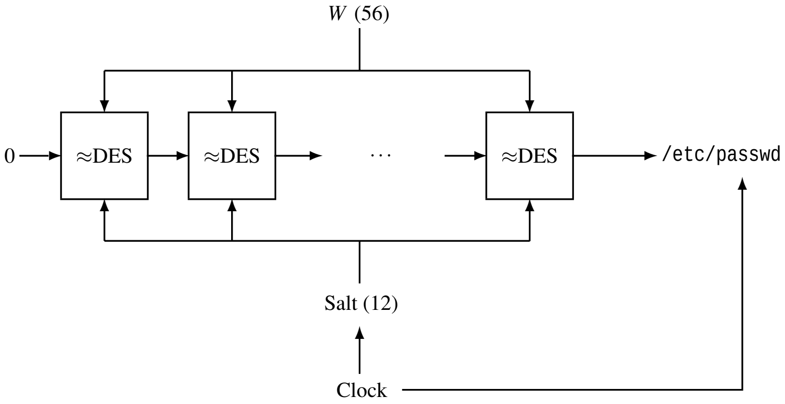

Figure 2.12. UNIX passwords.

[4] A summary of the specification can be found in Ref. [128], pp. 393–394.

- We do not want the security to rely on the secrecy of a critical secret key. Thus we want to make the decryption impossible even with full knowledge. Thus we use DES in a kind of a one-way mode: instead of computing $C(W)$ for a password W used as a plaintext, we compute $C_W(0)$ on the null plaintext with W used as a key. (W is truncated onto its first eight characters. It must consist of ASCII characters, thus 7-bit long, which makes 56 bits.)
- In order to make the exhaustive search more lengthy, we use a more complicated encryption. This can be tolerated for human user authentication as long as it does not require more than a fraction of a second. Thus we use 25 iterations of DES with the password used as a secret key.
- In order to prevent brute force attacks based on mass manufactured DES chips, we modify DES in order to make these chips unusable.
- In order to thwart attacks based on precomputed tables, the modification of DES involves a random 12-bit salt which is stored in clear with the encrypted password. Actually, some of the 48 bits of the expanded block are swapped depending on the salt. The 12-bit salt is generated from the system clock when the password is set up.

2.5 Classical Cipher Skeletons

Many block ciphers are described in the literature. We survey classical design skeletons.

2.5.1 Feistel Schemes

The Feistel scheme is the most popular block cipher skeleton. It is fairly easy to use a random round function in order to construct a permutation. In addition, encryption and decryption hardly need separate implementations.

An example is DES, described in Section 2.1.

Here are some possible generalizations of the Feistel scheme.

- We can add invertible substitution boxes in the two branches of the Feistel scheme (as done in the BLOWFISH cipher).
- We can replace the XOR by any other addition law. We do not necessarily need commutativity nor associativity: only regularity (like $a * x = a * y$ implies $x = y$).
- We do not need to have balanced branches. We may also have unbalanced ones (like in the BEAR and LION cipher).
- We can generalize the scheme so that it has more than two branches:
  (a) round functions with one input and several outputs (like in MARS),
  (b) round functions with several inputs and one output (like in MD4),
  (c) round functions with several inputs and outputs.

The first three variants are illustrated in Fig. 2.13.

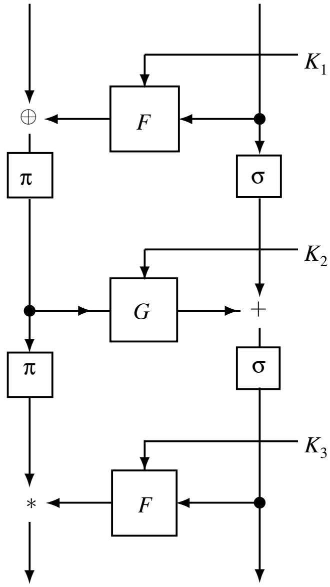

Figure 2.13. Variants of the Feistel scheme with two branches.

2.5.2 Lai–Massey Scheme

A famous block cipher which is not based on the Feistel scheme is the IDEA cipher. IDEA stands for International Data Encryption Algorithm. It follows two previous versions called PES (Proposed Encryption Standard) and IPES (Improved Proposed Encryption Standard). It was developed during the PhD studies of Xuejia Lai under the supervision of James Massey at the ETH Zürich. IDEA was published in Lai's thesis (Ref. [110]) in 1992. It is patented by Ascom and made freely available for noncommercial use.[5]

Like DES, IDEA is a block cipher for 64-bit blocks. IDEA uses much longer keys than DES as it allows for 128-bit keys. In the same way that DES was dedicated to hardware, IDEA was dedicated to software implementation on 16-bit microprocessors (which used to be a luxurious architecture in the early nineties). It makes an extensive use of the XOR, the addition modulo 2^{16}, and the product of nonzero residues modulo $2^{16} + 1$.

IDEA uses a structure similar to the Feistel scheme which can be called the Lai–Massey scheme. It also enables making a permutation from a function. It however requires a two-branch balanced structure and a commutative and associative law like the XOR operation. As depicted in Fig. 2.14, it simply consists of adding to both branches the output of a round function whose input is the difference of the two branches: a

[5] The commercial development of IDEA is currently managed by the company MediaCrypt.

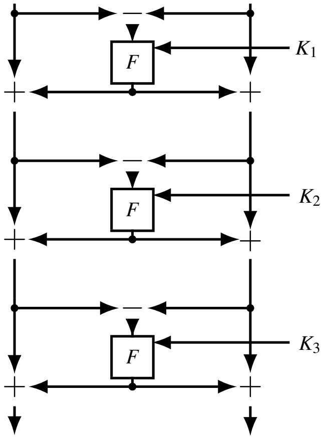

Figure 2.14. The Lai–Massey scheme.

(x_L, x_R) pair is mapped onto the (y_L, y_R) pair defined by

$$y_L = x_L + t$$
$$y_R = x_R + t$$

where $t = F(x_L - x_R)$. Note that this scheme is invertible by replacing the $+$ by $-$ and changing the order of the subkeys. Unfortunately, the Lai–Massey scheme cannot be used as is since the pair difference is an invariance: we have

$$x_L - x_R = y_L - y_R.$$

This is obviously an unsuitable property for security. For this reason we must insert (at least) a permutation σ as depicted in Fig. 2.15 and have

$$y_L = \sigma(x_L + t)$$
$$y_R = x_R + t$$

as it will be detailed in Section 2.6.1 for the FOX algorithms. When the permutation σ is such that $z \mapsto \sigma(z) - z$ is also a permutation, we say that σ is an *orthomorphism* for the $+$ law. We can demonstrate that when σ is an orthomorphism, then the Lai–Massey scheme provides security properties which are similar to those for the Feistel scheme. So the invariance of the basic Lai–Massey scheme is no longer a problem. In IDEA, key-dependent permutations (namely, products and additions) are used instead of a fixed σ.

IDEA consists of eight rounds. One round is as represented in Fig. 2.16. The $\cdot$ represents the multiplication modulo $2^{16} + 1$ to a subkey, the $+$ is the regular addition modulo 2^{16} to a subkey, $\oplus$ is the bitwise XOR, and MA is the Multiplication–Addition structure, which is depicted in Fig. 2.17. The MA structure also requires multiplication to subkeys. The addition law which is used in the Lai–Massey scheme of IDEA is the XOR.

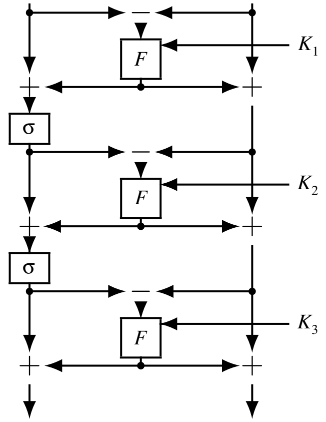

Figure 2.15. The Lai–Massey scheme with orthomorphism σ.

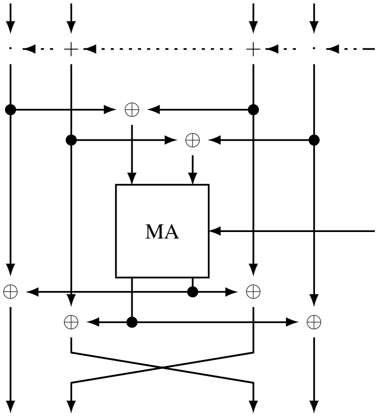

Figure 2.16. One round of IDEA.

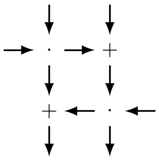

Figure 2.17. The MA structure in IDEA.

2.5.3 Substitution–Permutation Network

Shannon originally defined the encryption as a cascade of substitutions (like the Caesar cipher, or like the S-boxes in DES) and permutations (or transpositions, like the Spartan scytales, or the bit permutation after the S-boxes in DES). Therefore, many block ciphers fit to the category of substitution–permutation networks. However, this term was improperly used in order to refer to cascade on invertible layers made from invertible substitutions of coordinate permutations. Feistel schemes and Lai–Massey schemes are not considered to belong to this category in general.

SAFER K-64 is an example of a substitution–permutation network. It was made by James Massey for Cylink and was published in 1993 (see Refs. [121, 122]). It encrypts 64-bit blocks with 64-bit keys and is dedicated to 8-bit microprocessors (which are widely used in embedded system, for instance in smart cards). It uses XORs and additions modulo 2^8. It also uses exponentiation in basis 45 in the set of residues modulo 257 and its inverse which are implemented with lookup tables.

SAFER K-64 is a cascade of six rounds which consists of

- a layer of XOR or addition to subkeys,
- a layer of substitutions (exponentiation or logarithms as above),
- a layer of XOR or addition to subkeys,
- three layers of parallel linear diffusion boxes which make an overall transformation similar to the fast Fourier transform.

(See Fig. 2.18.) Diffusion boxes consist of mappings denoted by 2-PHT which are linear with respect to the $\mathbf{Z}_{256}$ structure. They are represented with their inverse in Fig. 2.19.

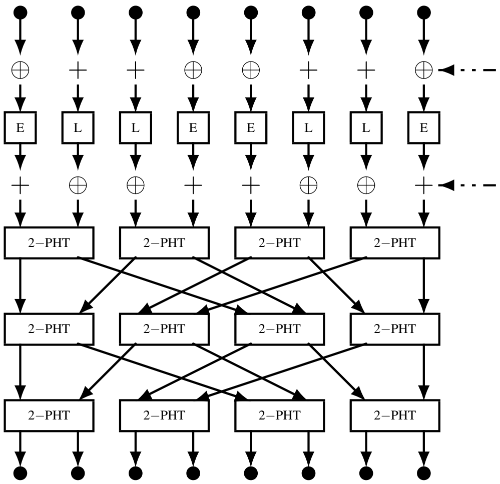

Figure 2.18. One round of SAFER.

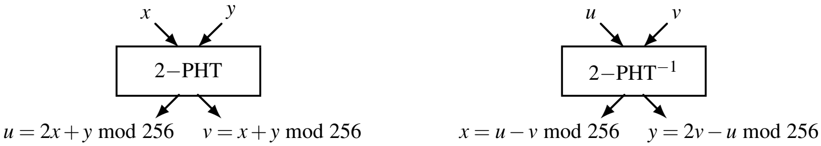

Figure 2.19. Diffusion in SAFER.

2.6 Other Block Cipher Examples

2.6.1 ⋆FOX: A Lai–Massey Scheme

FOX is a family of block ciphers which was released in 2003 (see Refs. [96, 97]). It was designed by Pascal Junod and Serge Vaudenay for the MediaCrypt company. The family includes block ciphers with 64-bit and 128-bit blocks. Round numbers and key sizes are flexible. We use an integral number r of rounds between 12 and 255 and a key of k bits with an integral number of bytes, up to 256 bits. The name FOX64/k/r refers to the block cipher of the family characterized by 64-bit blocks, r rounds, and keys of k bits. Similarly, FOX128/k/r refers to the block cipher with 128-bit blocks. The nominal choices denoted by FOX64 and FOX128 refer to FOX64/128/16 and FOX128/256/16 respectively. Namely, we use $r = 16$ as a nominal number of rounds and a key length which corresponds to two blocks.

A key schedule processes the key K and a direction (either "encrypt" or "decrypt") and produces a sequence $\text{RK}_1, \ldots, \text{RK}_r$ of r round keys in this ordering if the direction is "encrypt" or the opposite if the direction is "decrypt." Encryption is performed through r rounds as depicted in Fig. 2.20. Every round processes a data block and a round key RK (whose size consists of two blocks) and produces another data block. The $r - 1$ first rounds have identical structure but the last round is a little different.

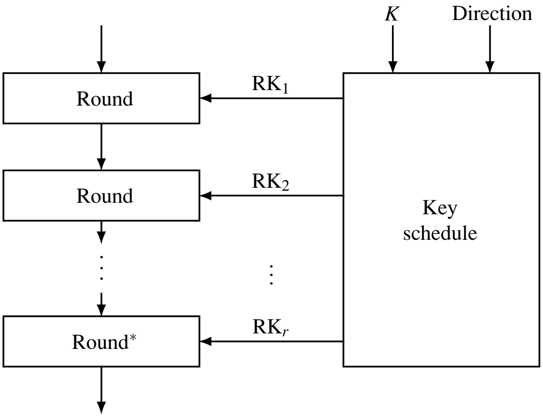

Figure 2.20. The FOX skeleton.

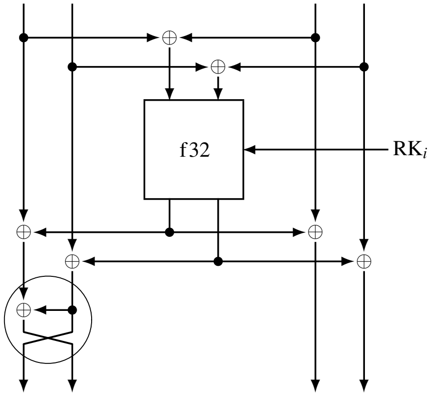

Figure 2.21. One round of FOX64 with an orthomorphism.

The FOX64 round is a Lai–Massey scheme as defined in Section 2.5.2 with the XOR as the addition law and an orthomorphism as depicted in Fig. 2.21. Note that branches in the Lai–Massey scheme are split into two in the figure, leading us to four branches in total. The orthomorphism appears in the bottom left branches (circled in the figure). It maps (a, b) onto $(b, a \oplus b)$ for the encryption and onto $(a \oplus b, a)$ for the decryption. The last round of FOX64 is the same Lai–Massey scheme without the orthomorphism. The FOX128 round is an extended Lai–Massey scheme with two orthomorphisms as depicted in Fig. 2.22. The last round omits the orthomorphisms. With this design we easily demonstrate that flipping the key schedule direction effects two permutations which are the inverse of each other.

Round functions are denoted f32 and f64 for FOX64 and FOX128 respectively. Those functions process a data of 32 and 64 bits respectively and a round key RK$_i$ which is split into two halves RK$_{i0}$ and RK$_{i1}$. As depicted in Figs. 2.23 and 2.24, RK$_{i0}$ is first XORed to the input data. Then a byte-wise substitution is performed using a substitution box denoted S-box followed by a linear transform denoted mu4 and mu8

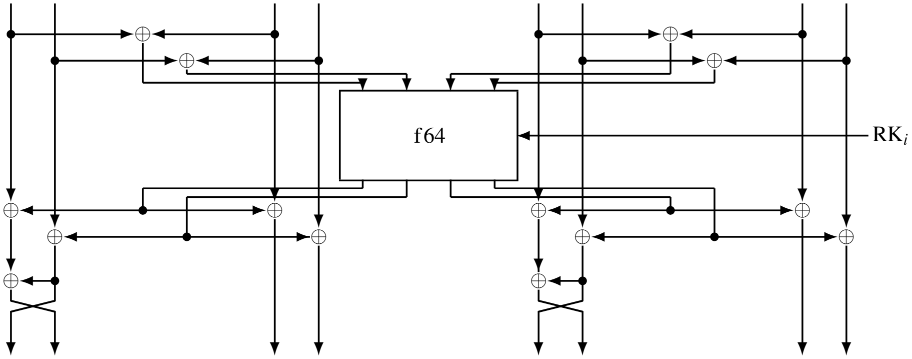

Figure 2.22. One round of FOX128 with orthomorphisms.

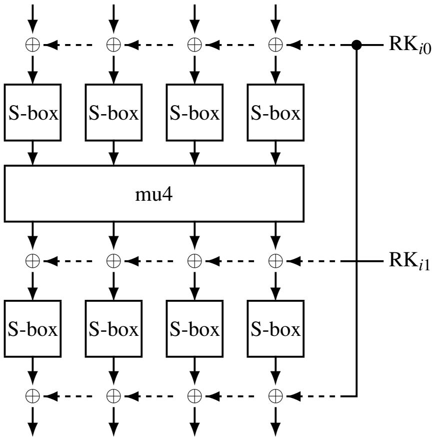

Figure 2.23. Round function f32 of FOX64.

respectively. Then RK$_{i1}$ is XORed with the output of mu4 (or mu8) and another bytewise substitution takes place. Finally, a last XOR to RK$_{i0}$ is performed. Functions mu4 and mu8 are linear in the sense that they process vectors of bytes that are considered as elements of the finite field GF(2^8) by multiplying them with a constant matrix.

The key schedule of FOX highly depends on the parameters. The main idea, as depicted in Fig. 2.25, consists of first padding the key with some constant in order to get a 256-bit key, then mixing those bytes in order to avoid trailing constant bytes, and obtain a 256-bit main key. This key is XORed to constants which are generated by a linear feedback shift register (LFSR) which can be clocked in one direction or the other. The XOR is then processed through a nonlinear (NL) function which produces a round key. There are some subtleties depending on the parameters.

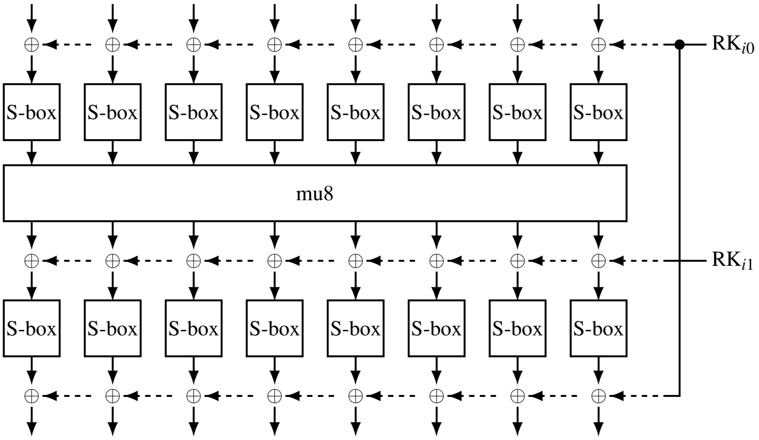

Figure 2.24. Round function f4 of FOX128.

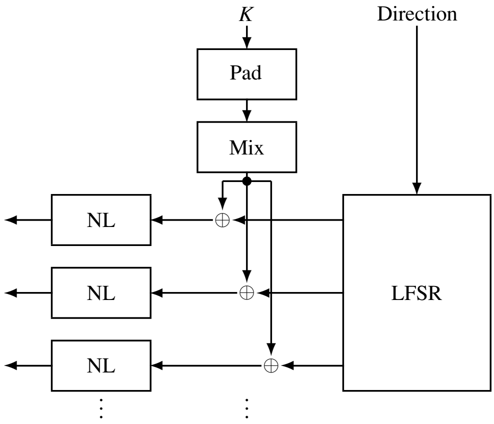

Figure 2.25. Key schedule of FOX.

- For FOX64 with key of length up to 128 bits, there is no real need for having a 256-bit main key and this actually induces a penalty for the implementation performances. Indeed we use a 128-bit main key and LFSR and NL functions updated accordingly.
- When the key has a "full size," i.e. $k = 256$, or $k = 128$ with FOX64, there is no need for padding and byte mixing. Indeed, we omit them. In order to avoid key schedule interference between several kinds of keys, we slightly modify NL.

It should be noted that NL is defined by using functions which are similar to encryption rounds. It was designed in order to be "one way" and to generate unpredictable round keys.[6]

2.6.2 ⋆CS-CIPHER: A Substitution–Permutation Network

Another example is the CS-CIPHER (CSC) which was developed by Jacques Stern and Serge Vaudenay at the Ecole Normale Supèrieure for the company Communication & Systems. It was published in 1998 (see Refs. [176, 181]). It encrypts 64-bit blocks with keys of variable length from 0 to 128 bits and is dedicated to 8-bit microprocessors, and consists of eight rounds of fast Fourier transform (FFT)-like layers (see Fig. 2.26). The difference with SAFER is that this transform is not linear.

One round of CSC is an FFT-like layer with a mixing box M as an elementary operation. M has two input bytes and two output bytes. It includes a one-position bitwise rotation to the left (denoted ROTL), XORs (denoted with the $\oplus$ notation), a nonlinear permutation P defined by a table, and a special linear transform φ defined by

$$\varphi(x) = (\mathrm{ROTL}(x) \text{ AND } 55) \oplus x$$

[6] See Ref. [96] for a complete description.

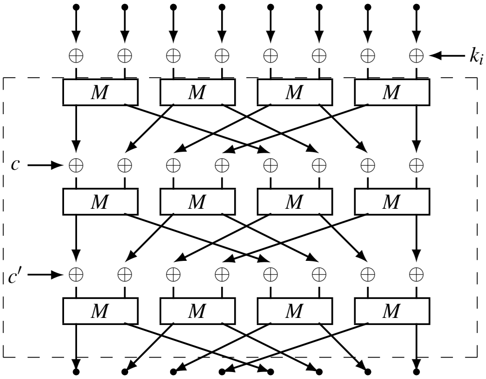

Figure 2.26. One round of CS-CIPHER.

where AND is the bitwise logical AND and 55 is an hexadecimal constant which is 01010101 in binary. We notice that φ is linear, and actually an involution since

$$\begin{aligned}\varphi(\varphi(x)) &= (\mathrm{ROTL}(\varphi(x)) \text{ AND } \mathtt{55}) \oplus \varphi(x) \\ &= x.\end{aligned}$$

Thus φ is a linear permutation. The permutation P is defined in order to be a nonlinear involution:

$$P(P(x)) = x.$$

We can then finally define M. Fig. 2.27 represents M with the XOR with subkey bytes at the input. It is easy to see that Fig. 2.28 represents the inverse transform where φ' is defined by

$$\varphi'(x) = (\mathrm{ROTL}(x) \text{ AND } \mathtt{aa}) \oplus x.$$

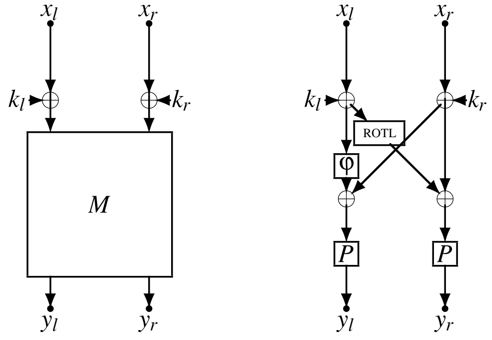

Figure 2.27. The mixing box of CSC.

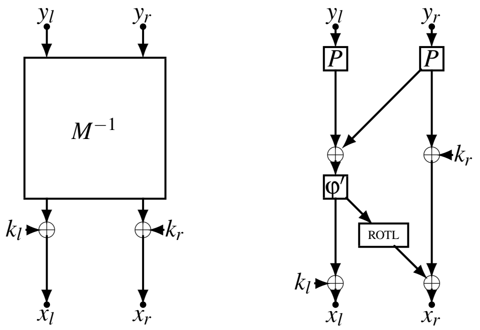

Figure 2.28. The invert mixing box of CSC.

For completeness we also provide a complete view of CSC in Fig. 2.29. We see that the key schedule is actually defined by a Feistel scheme.

2.7 The Advanced Encryption Standard (AES)

With the improvement of computer technology due to the Moore law, the security of DES is no longer appropriate for electronic commerce. The US Government decided to restart a standardization process called the Advanced Encryption Standard (AES)

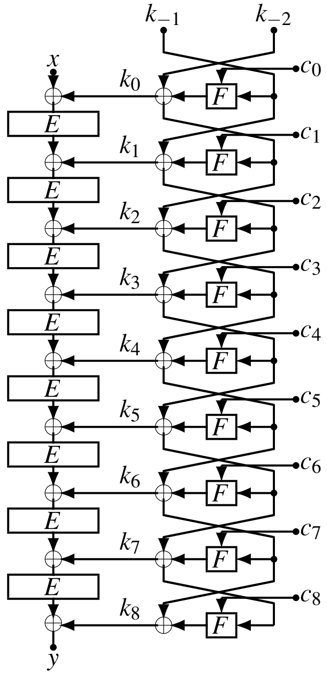
k_{-1}
k_{-2}
x
k_0 k_1 k_2 k_3 k_4 k_5 k_6 k_7 k_8
c_0 c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8
E
F
y

Figure 2.29. External view of CSC.

in 1997. This process was open: anyone was invited to submit a candidate and to send public comments. Fifteen candidates were accepted (a few other submissions did not meet the requirements and were rejected) in 1998. Based on public comments (and apparently on popularity), this pool was downsized to five finalists in 1999. In October 2000, one of these five algorithms was selected as the forthcoming standard: Rijndael (see Refs. [1, 54]).

Rijndael was designed by Joan Daemen (from the Belgium company Proton World International) and Vincent Rijmen. They both originated from the Catholic University of Leuven. Rijndael was designed for the AES process. Following the AES requirements, it encrypts 128-bit blocks with keys of size 128, 192, or 256. It is dedicated to 8-bit microprocessors. It consists of several rounds of a simple substitution–permutation network.

AES is based on the structure of SQUARE.[7] This design simply consists of writing the 128-bit message block as a 4×4 square matrix of bytes. (Formally, Rijndael tolerates other block sizes, but 128-bit was the target block size for AES.) Encryption is performed through 10, 12, or 14 rounds depending on whether the key size is 128, 196, or 256 bits. The number of rounds is denoted by Nr. Each round (but the final one) consists of four simple transformations:

1. SubBytes, a byte-wise substitution defined by a single table of 256 bytes,
2. ShiftRows, a circular shift of all rows (row number i of the matrix is rotated by i positions to the left for $i = 0, 1, 2, 3$),
3. MixColumns, a linear transformation performed on each column and defined by a 4×4 matrix of GF(2^8) elements,
4. AddRoundKey, a simple bitwise XOR with a round key defined by another matrix.

The final round is similar, except for MixColumns which is omitted. The round keys are generated by a separate key schedule.

More formally, one block s is encrypted by the following process, in which w is the output subkey sequence from the key schedule algorithm.

```
AES encryption($s$, $W$)
  AddRoundKey($s$, $W_0$)
  for $r = 1$ to $\mathrm{Nr} - 1$ do
      SubBytes($s$)
      ShiftRows($s$)
      MixColumns($s$)
      AddRoundKey($s$, $W_r$)
  end for
  SubBytes($s$)
  ShiftRows($s$)
  AddRoundKey($s$, $W_{\mathrm{Nr}}$)
```

[7] SQUARE was designed by the same authors and Lars Knudsen in 1997 (see Ref. [55]).

The block s is also called state and represented as a matrix of terms $s_{i,j}$ for $i, j \in \{0, 1, 2, 3\}$. Terms are bytes, i.e. elements of a set Z of cardinality 256. SubBytes is defined as follows.

```
SubBytes($s$)
for $i = 0$ to 3 do
    for $j = 0$ to 3 do
        $s_{i,j} \leftarrow$ S-box($s_{i,j}$)
    end for
end for
```

Here S-box is the substitution table. Mathematically, it is a permutation of $\{0, 1, \ldots, 255\}$. ShiftRows is defined as follows.

```
ShiftRows($s$)
1: replace $[s_{1,0}, s_{1,1}, s_{1,2}, s_{1,3}]$ by $[s_{1,1}, s_{1,2}, s_{1,3}, s_{1,0}]$
     {rotate row 1 by one position to the left}
2: replace $[s_{2,0}, s_{2,1}, s_{2,2}, s_{2,3}]$ by $[s_{2,2}, s_{2,3}, s_{2,0}, s_{2,1}]$
     {rotate row 2 by two positions to the left}
3: replace $[s_{3,0}, s_{3,1}, s_{3,2}, s_{3,3}]$ by $[s_{3,3}, s_{3,0}, s_{3,1}, s_{3,2}]$
     {rotate row 3 by three positions to the left}
```

We define the set Z as the set of all the 256 possible combinations

$$a_0 + a_1.x + a_2.x^2 + \cdots + a_7.x^7$$

where $a_0, a_1, a_2, \ldots, a_7$ are either 0 or 1 and x is a formal term. Elements of Z are thus defined as polynomials of degree at most 7. AddRoundKey is defined as follows.

```
AddRoundKey($s$, $k$)
for $i = 0$ to 3 do
    for $j = 0$ to 3 do
        $s_{i,j} \leftarrow s_{i,j} \oplus k_{i,j}$
    end for
end for
```

Here the $\oplus$ operation over Z is defined as an addition modulo 2, i.e.

$$\left(\sum_{i=0}^{7} a_i.x^i\right) \oplus \left(\sum_{i=0}^{7} b_i.x^i\right) = \sum_{i=0}^{7}(a_i + b_i \bmod 2).x^i.$$

A multiplication × in Z is further defined as follows.

1. We first perform the regular polynomial multiplication.
2. We make the Euclidean division of the product by the $x^8 + x^4 + x^3 + x + 1$ polynomial and we take the remainder.
3. We reduce all its terms modulo 2.

Later in Chapter 6 we will see that this provides Z with the structure of the unique finite field of 256 elements. This finite field is denoted by GF(2^8). This means that we can add, multiply, or divide by any nonzero element of Z with the same properties that we have with regular numbers. We can further define matrix operations with terms in Z. We can then define MixColumns as follows.

```
MixColumns(s)
1: for $i = 0$ to 3 do
2:     let $v$ be the 4-dimensional vector with coordinates $s_{0,i}, s_{1,i} s_{2,i} s_{3,i}$
3:     replace $s_{0,i}, s_{1,i} s_{2,i} s_{3,i}$ by the coordinates of $M \times v$
4: end for
```

Here M is a 4×4-matrix over Z defined by

$$M = \begin{pmatrix} x & x+1 & 1 & 1 \\ 1 & x & x+1 & 1 \\ 1 & 1 & x & x+1 \\ x+1 & 1 & 1 & x \end{pmatrix}.$$

The substitution table S-box is defined by the inversion operation x^{-1} (except for $x = 0$ which is mapped to zero) in the finite field GF(2^8). This operation has good nonlinear properties. In order to "break" the algebraic structure of this table, an affine transformation is added on this function.

The linear transformation in MixColumns is defined by a matrix following principles similar to the mixing box of CSC (see Section 2.6.2): whenever i input bytes of this linear transformation are modified, we make sure that this induces a modification of at least $5 - i$ output bytes.[8]

We complete the description of AES by outlining the key expansion. It is easier to consider W as a row sequence (i.e. four bytes) of length 4Nr starting by w_0 and up to $w_{4\mathrm{Nr}-1}$. Hence

$$W_i = [w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}].$$

The key expansion proceeds with a key described as a sequence of Nk rows (i.e. Nk is either 4, 6, or 8) starting from key_0. The expansion works as follows.

[8] Equivalently, the set of all $(x, M(x))$ 8-byte vectors is an MDS code if M denotes the linear transformation, or in other words, M is a multipermutation.

```
**KeyExpansion**(key, Nk)
for $i = 0$ to Nk $- 1$ do
    $w_i \leftarrow \text{key}_i$
end for
for $i =$ Nk to $4 \times (\text{Nr} + 1) - 1$ do
    $t \leftarrow w_{i-1}$
    if $i$ mod Nk $= 0$ then
        replace $[t_0, t_1, t_2, t_3]$ by $[t_1, t_2, t_3, t_0]$ in $t$
        apply S-box to the four bytes of $t$
        XOR $x^{i/\text{Nk}-1}$ (raise the polynomial $x$ to the power $i/\text{Nk} - 1$
          in GF($2^8$)) to the first byte of $t$
    else if Nk $= 8$ and $i$ mod Nk $= 4$ then
        apply S-box to the four bytes of $t$
    end if
    $w_i \leftarrow w_{i-\text{Nk}} \oplus t$
end for
```

2.8 Stream Ciphers

2.8.1 Stream Ciphers versus Block Ciphers

All conventional encryption schemes that we have seen so far are block ciphers in the sense that they encrypt blocks of plaintexts. They are often opposed to stream ciphers which encrypt streams of plaintext on the fly. A stream cipher often encrypts streams of plaintext bits, or streams of plaintext bytes. This distinction is often misleading since block ciphers are used as well in a mode of operation so that they can encrypt streams of blocks. Nevertheless, we will call block cipher an encryption scheme in which the underlying primitive is defined on a large finite set (of “blocks”) which cannot be enumerated exhaustively in practice. With this definition we cannot assimilate a bit or a byte to a block. Conversely, we call stream cipher an encryption scheme which can encrypt streams of information in a smaller finite set.

First of all we notice that we can transform a pseudorandom generator into a stream cipher. Stream ciphers are indeed often defined by a key-stream generator which is used as a one-time pad: instead of having a large random key for the Vernam cipher, we use a pseudorandom key which is generated as a key-stream.

We also notice that we can transform a block cipher into a stream cipher by using the CFB, or CTR mode, with a small parameter ℓ[9] (see Sections 2.2.3, 2.2.4, and 2.2.5).

2.8.2 RC4

RC4 is an encryption algorithm which was designed in 1987 by Ronald Rivest at MIT. It was kept as a commercial secret until it was disclosed in 1994. In particular there is no

[9] It is not recommended to do the same for OFB.

patent on RC4, but RC4 is a registered trademark of RSA Data Security. RC4 is widely used, for instance in SSL/TLS (see Section 12.3). In particular, some Internet browsers and servers may use RC4 as a default encryption algorithm for protected transactions.

RC4 works as a finite automaton with an internal *state*. It reads a plaintext as a byte stream and produces a ciphertext as a byte stream. Its heart is actually a key-stream generator which is used for the one-time pad algorithm. In an initialization stage, a secret key is processed without producing keys. The automaton ends up in an internal state which is thus uniquely derived from the secret key only. Then, every time unit, the automaton updates its internal state and produces a key byte which is XORed to a plaintext byte in order to lead to a ciphertext byte.

We consider the set $\{0, 1, \ldots, 255\}$ of all bytes. The internal state consists of two bytes i and j and a permutation S of this set which is encoded as an array $S[0], S[1], \ldots, S[255]$. All operations are done on bytes (i.e. additions are taken modulo 256).

In the initialization, we process a key which is represented as a sequence $K[0], K[1], \ldots, K[\ell - 1]$ of ℓ bytes. The internal state is first initialized as follows. Byte j is set to 0, and the permutation S is set to the identity, i.e. $S[i] = i$ for $i = 0, 1, \ldots, 255$. Key bytes are then iteratively processed, and the bytes i and j are reset to 0.

```
$j \leftarrow 0$
for $i = 0$ to $255$ do
    $S[i] \leftarrow i$
end for
for $i = 0$ to $255$ do
    $j \leftarrow j + S[i] + K[i \bmod \ell]$
    swap $S[i]$ and $S[j]$
end for
$i \leftarrow 0$
$j \leftarrow 0$
```

The key size ℓ is typically between 5 and 16 bytes (i.e. between 40 and 256 bits).

It is important that we never use the same state twice. Thus, plaintexts are iteratively encrypted, which means that the initial state for a new plaintext is equal to the final state for the previous plaintext.

The key-stream generator works as follows. Every time unit, we perform the following sequence of instructions.

```
$i \leftarrow i + 1$
$j \leftarrow j + S[i]$
swap $S[i]$ and $S[j]$
output $S[S[i] + S[j]]$
```

Thus we update i, j, and S, and we output a byte which is given by S at index $S[i] + S[j]$.

2.8.3 *A5/1: GSM Encryption*

A5/1 is another stream cipher which is part of the A5 family. It is used in the GSM mobile telephone networks. It is used in order to secure phone calls in the radio link from the mobile telephone to the base station. It was designed by the SAGE group of ETSI. The description of A5/1 is another trade secret, but the algorithm was reverse-engineered and published in the Internet. It is commonly admitted that this description is similar to the ETSI one.

A5/1 is also based on a finite automaton with an internal state. As depicted in Fig. 2.30, A5/1 is based on three LFSRs with a mutual clock control. The three registers R_1, R_2, R_3 contain 19, 22, and 23 bits respectively. The internal state thus has $19 + 22 + 23 = 64$ bits. Every time unit, some registers are clocked and some may not be clocked at all. When a register is clocked, it means that its content is shifted by one bit position and that a new bit is pushed. This new bit is the XOR of a few bits of the involved LFSRs.

More precisely, R_1 has 19 bits $R_1[0], \ldots, R_1[18]$. When R_1 is clocked, the content $R_1[0], \ldots, R_1[18]$ is replaced by b, $R_1[0], \ldots, R_1[17]$, i.e. R_1 is shifted by inserting a new bit b which is computed by

$$b = R_1[13] \oplus R_1[16] \oplus R_1[17] \oplus R_1[18].$$

R_2 has 22 bits $R_2[0], \ldots, R_2[21]$. When R_2 is clocked, it is similarly shifted by inserting a new bit

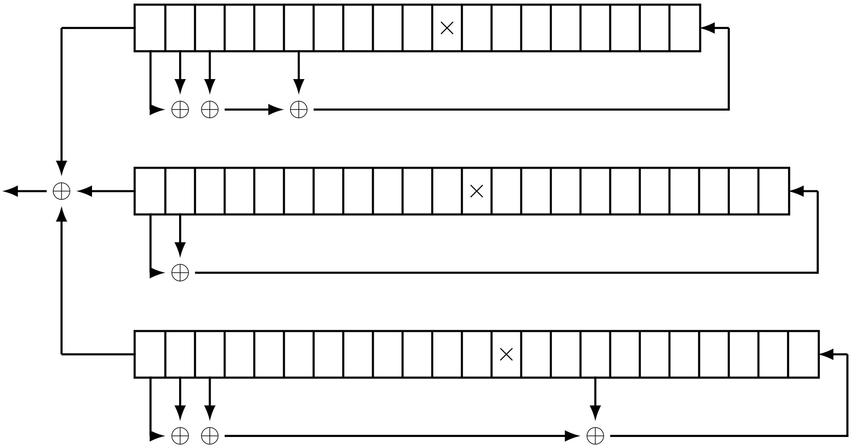

Figure 2.30. A5/1 automaton.

$$b = R_2[20] \oplus R_2[21].$$

R_3 has 23 bits $R_3[0], \ldots, R_3[22]$. When R_3 is clocked, it is similarly shifted by inserting a new bit

$$b = R_3[7] \oplus R_3[20] \oplus R_3[21] \oplus R_3[22].$$

In order to determine which registers to clock, we use three special bits called "clocking taps" from every register, namely $R_1[8]$, $R_2[10]$, and $R_3[10]$. We compute the majority bit among those three bits, and registers whose clocking tap agree with the majority are clocked. Consequently, we are ensured that at least two registers are clocked. All registers are clocked if the three clocking taps agree on the same bit.

Every time unit, a bit is output from this scheme. This output bit is the XOR of the leading bits, namely

$$R_1[18] \oplus R_2[21] \oplus R_3[22].$$

We use the generated key stream as in the one-time pad.

A5/1 also includes an initialization which generates the initial internal state from an encryption key and some GSM parameters. It is required that a new key is set up for any new frame of 114 bits. More precisely, the key is set up from a 64-bit secret key KC and a 22-bit frame number Count. This is indeed a kind of CTR mode. The usage of a secret key KC is thus limited. Because of the structure of the frame number in the GSM standard we can have at most 2.7 million frames for a single key, which corresponds to 4 hours of GSM communication.

The A5/1 initialization works as follows. The three registers are first set to zero. Then every bit of KC is processed in 64 clock cycles by XORing them to the first register cells and stepping all registers (i.e. the clock control is disabled). Every bit of the frame number Count is then processed in a similar way and the A5/1 automaton is run for 100 clock cycles with its clock control enabled (but output bits are discarded).

```
set all registers to zero
for $i = 0$ to 63 do
    $R_1[0] \leftarrow R_1[0] \oplus \mathrm{KC}[i]$
    $R_2[0] \leftarrow R_2[0] \oplus \mathrm{KC}[i]$
    $R_3[0] \leftarrow R_3[0] \oplus \mathrm{KC}[i]$
    clock all registers
end for
for $i = 0$ to 21 do
    $R_1[0] \leftarrow R_1[0] \oplus \mathrm{Count}[i]$
    $R_2[0] \leftarrow R_2[0] \oplus \mathrm{Count}[i]$
```

```
  $R_3[0] \leftarrow R_3[0] \oplus \text{Count}[i]$
  clock all registers
end for
for $i = 0$ to 99 do
  clock the A5/1 automaton
end for
```

2.8.4 E0: Bluetooth Encryption

E0 is another stream cipher which is used in the Bluetooth standard (see Ref. [18]). As in A5/1, E0 is an automaton which generates keystreams which are simply XORed to the plaintext as in the Vernam cipher. E0 generates one bit every clock cycle and frames of 2745 bits. After a frame is generated, the E0 automaton is reset to another state.

The state of the E0 automaton is described by the content of four linear feedback shift registers $\text{LFSR}_1, \text{LFSR}_2, \text{LFSR}_3, \text{LFSR}_4$ of length $25, 31, 33, 39$ respectively, and the content of two 2-bit registers c_{t-1} and c_t. Every clock cycle, the four registers are clocked, c_t, is moved to c_{t-1}, and c_t is updated by using c_{t-1}, c_t, and the registers.

More precisely, one bit x_t^i is output from LFSR_i at every clock cycle. A summator computes $y_t = x_t^1 + x_t^2 + x_t^3 + x_t^4$ and represents it in binary using three bits $y_t^2 y_t^1 y_t^0$. The automaton outputs a new keystream bit $z_t = y_t^0 \oplus c_t^0$ where $c_t = c_t^1 c_t^0$. The new value $c_{t+1} = c_{t+1}^1 c_{t+1}^0$ of c_t is computed by

$$\begin{aligned} c_{t+1}^1 &= s_{t+1}^1 \oplus c_t^1 \oplus c_{t-1}^0 \\ c_{t+1}^0 &= s_{t+1}^0 \oplus c_t^0 \oplus c_{t-1}^1 \oplus c_{t-1}^0 \end{aligned}$$

where

$$s_{t+1} = \left\lfloor \frac{y_t + c_t}{2} \right\rfloor .$$

E0 is actually a little more complicated. E0 is based on two levels of the above automaton as depicted in Fig. 2.31. Formally, the encryption algorithm E0 takes the logical address BD_ADDR of the master (Bluetooth is based on master-slave protocols) which is represented on 48 bits, the clock value of the master CLK which is represented on 26 bits, and an encryption key K_c of 128 bits. The first level is used in order to initialize the automaton for every frame. The second level generates frames with the initialized automaton. Concretely, the encryption key K_c is first linearly shrunk and then expanded into a 128-bit key so that the *effective* key length can be lowered for regulation purposes. Then, the reexpanded encryption key, the master address, and the master clock enter into the LFSR of the automaton which is clocked. After running the automaton, 128 bits are output, which serve as the initial value of the second-level

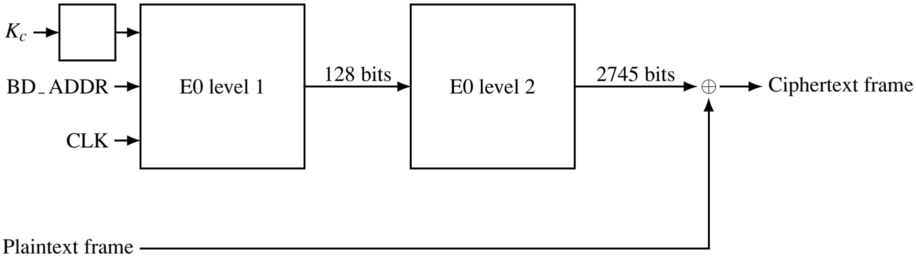

Figure 2.31. The E0 keystream generator.

automaton for producing the keystream frame. Frames are finally used as a keystream in the Vernam cipher.

E0 is very efficient in hardware. Its circuit is depicted in Fig. 2.32. The division by two at the output from the second summator simply consists of dropping one bit. Internal memory bit pairs for c_t and c_{t-1} are represented by the z^{-1} symbol as a clock time delay.

2.9 Brute Force Attacks

We now wonder what kind of security we can expect from symmetric encryption from a generic point of view. Namely, we wonder about performances of generic attacks using brute force which can apply to any encryption algorithm considered as a black box. Security will be measured in terms of the encryption parameters such as the key length and the message space size.

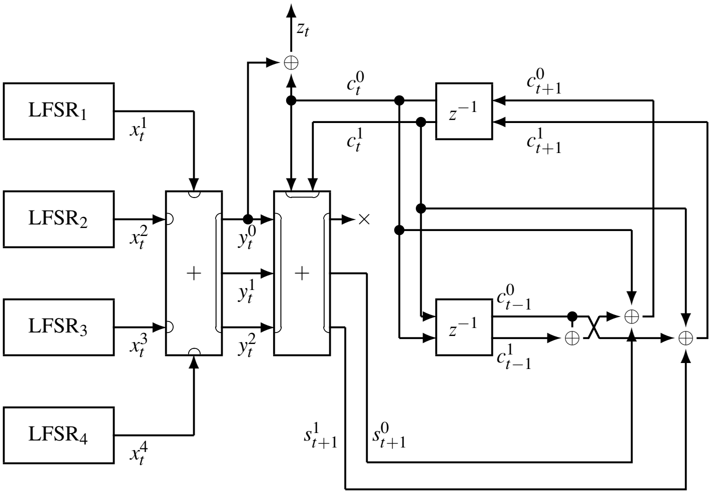

Figure 2.32. One level of the E0 keystream generator.

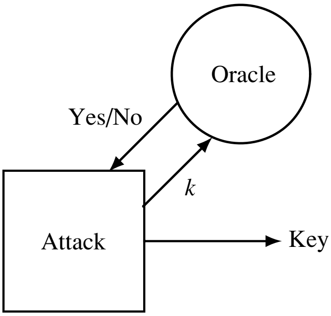

Figure 2.33. Key recovery with a stop test oracle.

2.9.1 Exhaustive Search

Exhaustive search consists of trying all possible keys exhaustively until it is the correct key. We can simplify the attack model by assuming that we have an oracle, which for each key K answers if it is correct or not. The attack is thus a machine which plays with the oracle in order to get some information out of it by sending queries (see Fig. 2.33). Without any more formal notion of a "machine," we can still define the complexity in terms of number of oracle calls.[10] In a cipher with a key space of N possible keys which are uniformly generated, we can prove that the best attack has a worst case complexity of N oracle calls and an average complexity of (about) $\frac{N}{2}$ oracle calls. Clearly, the best attack can be assumed to be deterministic (otherwise, we just take the machine which simulates the best deterministic behavior of the probabilistic machine), and does not query the oracle with twice the same question. Therefore, the best attack can be defined as an ordering of the candidates for the right key. The optimal ordering is defined by the *a priori* distribution of the target key.[11]

In practice we do not know the *a priori* distribution of the target. Thus, if we take a fixed ordering in order to try key candidates, we may fall into the worst case complexity. For this we randomize the ordering of the key candidates. Fig. 2.34 shows a program draft for this attack. In the worst case, the right key is the last one which is sent to the oracle. If the right key is the i-th candidate, the complexity is equal to i. Since the permutation σ is randomly chosen, the probability that the right key is the i-th queried one is $\frac{1}{N}$. Thus the average complexity is

$$\sum_{i=1}^{N} i\frac{1}{N} = \frac{N+1}{2}.$$

The ultimate security goal of a cipher with keys of n bits is to have a best attack of complexity $\Omega(2^n)$ with a "not-too-small" work factor.[12]

[10] In Chapter 8 we define a formal notion of machine and the notion of complexity of the computation.

[11] See Exercise 2.9 on p. 62.

[12] This Ω notation means that there exists a constant $c > 0$ called work factor such that for any n, the complexity is at least $c2^n$ elementary operations. The notion of complexity is formally defined in Chapter 8.

There is a trick in order to strengthen the security against exhaustive search: make a key schedule complicated. In general, the key schedule is legitimately used only once for many encryptions. Exhaustive search uses the key schedule many times. For instance, BLOWFISH has a complicated key schedule.

```
Input: an oracle $\mathcal{O}$, a set of possible keys $\mathcal{K} = \{k_1, \ldots, k_N\}$
Oracle interface: input is an element of $\mathcal{K}$, output is Boolean
pick a random permutation $\sigma$ of $\{1, \ldots, N\}$
for all $i = 1$ to $N$ do
    if $\mathcal{O}\ (k_{\sigma(i)})$ then
        yield $k_{\sigma(i)}$ and stop
    end if
end for
search failed
```

Figure 2.34. Exhaustive search algorithm.

We can wonder now what the hypothesis about the availability of the oracle really means in practice. We may only have some hints about the key. Namely, we can have some equations that the key must satisfy. Typically, we can assume that we have a plaintext–ciphertext pair so that the key must solve the equation which says that the plaintext encrypts into the ciphertext. Another typical situation is when we intercept a ciphertext and we know that the plaintext has some redundant information, e.g. the plaintext is an English text. If the equations are characteristic enough for the key, the unique solution is the right key candidate so that we can simulate the oracle by equation solution checking.

2.9.2 Dictionary Attack

Another brute force attack is the dictionary attack. A dictionary is a huge table which has been precomputed in order to speed up a key search. In practice, when one looks for the definition of a word, it is too expensive to do it by exhaustive search! For this a dictionary sorts all definitions by using an ordering on a key.

Here is a way to formalize a dictionary attack. We first precompute a list of many $(C_K(x), K)$ pairs for a fixed plaintext x. The first entry $C_K(x)$ is used as an index in order to sort the list as in a dictionary, and the second entry K is the "definition." Then, if we obtain a $C_K(x)$ value (by chosen plaintext attack), we directly have a list of suggested K values. Let M be the number of entries in the dictionary. If K is in a set of N possible keys, and uniformly distributed, the probability of success, i.e. the probability that it is in the dictionary, is M/N. When K is not uniformly distributed, we can focus on a dictionary of the most likely K values. This is done in practice when we try to crack passwords.

We now assume that we have T targets $C_K(x)$ instead of one: there are T secret keys that we try to attack simultaneously and we are interested in getting at least

```
Input: an encryption scheme $C$, a fixed message $x$
Preprocessing
for $M$ different candidates $K$ do
    compute $C_K(x)$
    insert $(C_K(x), K)$ in a dictionary
end for
output the dictionary
Attack
Attack input: $T$ many values $y_i = C_{K_i}(x)$, a dictionary
for $i = 1$ to $T$ do
    look at $y_i$ in the dictionary
    for all $(C_K(x), K)$ with $C_K(x) = y_i$ do
        yield $i$, $K$
    end for
end for
```

Figure 2.35. Multitarget dictionary attack.

one. (Fig. 2.35 illustrates the program structure.) Let p be the probability of success. Assuming that the target K values are independent and uniformly distributed, we have $1 - p = (1 - M/N)^T$. Hence

$$p \approx 1 - e^{-MT/N}.$$

This becomes quite interesting when $M \approx T \approx \sqrt{N}$. Of course the probability of success increases substantially when the targets are not uniformly distributed and we focus on most likely candidates.

2.9.3 Codebook Attack

Yet another brute force attack is the codebook attack. . It consists of, first, collecting all $(C_K(x), x)$ pairs, then, upon reception of a y to decrypt, look for the entry in the collection for which $y = C_K(x)$. The whole collection of pairs is called the codebook.

2.9.4 $\star$Time–Memory Tradeoffs

An optimized way to perform exhaustive search is to use a large precomputed table and to make a compromise between time complexity and memory requirements. Here we must consider four parameters: the time to perform the precomputation, the size of the precomputed table, the probability of success when looking for a key, and the time complexity for looking for a key. For the latter parameter, we can also distinguish worst case complexity and average case complexity.

The original method for time-memory tradeoffs was proposed by Martin Hellman in 1980 in order to break DES (see Ref. [88]). We assume that we are looking for a key K for which we know a plaintext–ciphertext pair (x, y) with a fixed x. This fixed x is necessary to prepare the precomputed table. In practice this attack assumption makes sense when the adversary performs a chosen plaintext attack (in order to get y from x) or when the encryption protocol requires that the sender starts by encrypting this fixed message x. We further assume that for all k the bitstrings $C_k(x)$ have the same size which is larger than the key length so that it is likely that values of k for which $y = C_k(x)$ reduce to the unique solution $k = K$.

One first selects an arbitrary "reduction function" R which reduces a bitstring of the size of y to a key candidate $R(y)$. This leads to the definition of a function f

$$f(k) = R(C_k(x))$$

which maps a key candidate to another key candidate so that we can iterate it. The precomputation consists of making a table of m pairs $(k_{i,0}, k_{i,t})$ where all $k_{i,0}$ are randomly selected and $k_{i,t}$ is the t-th term of a sequence $k_{i,j}$ defined by

$$k_{i,j} = f(k_{i,j-1}).$$

(See Fig. 2.36.) In order to look for K once we are given y, we can apply R to y and repeatedly apply f until we find a value which matches one $k_{i,t}$ in the table. If we find one value $k_{i,t}$, we can get $k_{i,0}$ from the table and apply f repeatedly until we find $R(y)$ again. Note that if the total number of f applications reaches $t - 1$ without success we can give up and the attack fails. Otherwise we find a key k such that $f(k) = R(y)$. This may be due to $C_k(x) = y$, which leads to $K = k$. Otherwise the attack fails. Figs. 2.37 and 2.38 with ℓ set to 1 illustrate this method.

$$T_1 : \begin{array}{|c|ccccccccccc|c|cc}
\cline{1-1}\cline{13-13}
k^1_{1,0} & \overset{f_1}{\mapsto} & k^1_{1,1} & \overset{f_1}{\mapsto} & k^1_{1,2} & \overset{f_1}{\mapsto} & k^1_{1,3} & \overset{f_1}{\mapsto} & \cdots & \overset{f_1}{\mapsto} & k^1_{1,t-1} & \overset{f_1}{\mapsto} & k^1_{1,t} & & (k^1_{1,t},k^1_{1,0}) \\
k^1_{2,0} & \overset{f_1}{\mapsto} & k^1_{2,1} & \overset{f_1}{\mapsto} & k^1_{2,2} & \overset{f_1}{\mapsto} & k^1_{2,3} & \overset{f_1}{\mapsto} & \cdots & \overset{f_1}{\mapsto} & k^1_{2,t-1} & \overset{f_1}{\mapsto} & k^1_{2,t} & & (k^1_{2,t},k^1_{2,0}) \\
k^1_{3,0} & \overset{f_1}{\mapsto} & k^1_{3,1} & \overset{f_1}{\mapsto} & k^1_{3,2} & \overset{f_1}{\mapsto} & k^1_{3,3} & \overset{f_1}{\mapsto} & \cdots & \overset{f_1}{\mapsto} & k^1_{3,t-1} & \overset{f_1}{\mapsto} & k^1_{3,t} & \Rightarrow & (k^1_{3,t},k^1_{3,0}) \\
\vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
k^1_{m,0} & \overset{f_1}{\mapsto} & k^1_{m,1} & \overset{f_1}{\mapsto} & k^1_{m,2} & \overset{f_1}{\mapsto} & k^1_{m,3} & \overset{f_1}{\mapsto} & \cdots & \overset{f_1}{\mapsto} & k^1_{3,t-1} & \overset{f_1}{\mapsto} & k^1_{m,t} & & (k^1_{m,t},k^1_{m,0}) \\
\cline{1-1}\cline{13-13}
\end{array}$$

$$\vdots$$

$$T_\ell : \begin{array}{|c|ccccccccccc|c|cc}
\cline{1-1}\cline{13-13}
k^\ell_{1,0} & \overset{f_\ell}{\mapsto} & k^\ell_{1,1} & \overset{f_\ell}{\mapsto} & k^\ell_{1,2} & \overset{f_\ell}{\mapsto} & k^\ell_{1,3} & \overset{f_\ell}{\mapsto} & \cdots & \overset{f_\ell}{\mapsto} & k^\ell_{1,t-1} & \overset{f_\ell}{\mapsto} & k^\ell_{1,t} & & (k^\ell_{1,t},k^\ell_{1,0}) \\
k^\ell_{2,0} & \overset{f_\ell}{\mapsto} & k^\ell_{2,1} & \overset{f_\ell}{\mapsto} & k^\ell_{2,2} & \overset{f_\ell}{\mapsto} & k^\ell_{2,3} & \overset{f_\ell}{\mapsto} & \cdots & \overset{f_\ell}{\mapsto} & k^\ell_{2,t-1} & \overset{f_\ell}{\mapsto} & k^\ell_{2,t} & & (k^\ell_{2,t},k^\ell_{2,0}) \\
k^\ell_{3,0} & \overset{f_\ell}{\mapsto} & k^\ell_{3,1} & \overset{f_\ell}{\mapsto} & k^\ell_{3,2} & \overset{f_\ell}{\mapsto} & k^\ell_{3,3} & \overset{f_\ell}{\mapsto} & \cdots & \overset{f_\ell}{\mapsto} & k^\ell_{3,t-1} & \overset{f_\ell}{\mapsto} & k^\ell_{3,t} & \Rightarrow & (k^\ell_{3,t},k^\ell_{3,0}) \\
\vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
k^\ell_{m,0} & \overset{f_\ell}{\mapsto} & k^\ell_{m,1} & \overset{f_\ell}{\mapsto} & k^\ell_{m,2} & \overset{f_\ell}{\mapsto} & k^\ell_{m,3} & \overset{f_\ell}{\mapsto} & \cdots & \overset{f_\ell}{\mapsto} & k^\ell_{3,t-1} & \overset{f_\ell}{\mapsto} & k^\ell_{m,t} & & (k^\ell_{m,t},k^\ell_{m,0}) \\
\cline{1-1}\cline{13-13}
\end{array}$$

Figure 2.36. Table for Time–Memory tradeoff.

```
**Input**: an encryption scheme $C$, a fixed message $x$
**Parameter**: $\ell, m, t$
**Preprocessing**
for $s = 1$ to $\ell$ do
    pick a reduction function $R_s$ at random and define $f_s : k \mapsto R_s(C_k(x))$
    for $i = 1$ to $m$ do
        pick $k'$ at random
        $k \leftarrow k'$
        for $j = 1$ to $t$ do
            compute $k \leftarrow f_s(k)$
        end for
        insert $(k, k')$ in table $T_s$
    end for
end for
```

Figure 2.37. Time–Memory tradeoff (Preprocessing).

```
**Attack**
**Attack input**: $y = C_K(x)$
for $s = 1$ to $\ell$ do
    set $i$ to 0
    set $k$ to $R_s(y)$
    while $T_s$ contains no $(k, .)$ entry and $i < t$ do
        increment $i$
        $k \leftarrow f_s(k)$
    end while
    if $T_s$ contains a $(k, .)$ entry then
        get the $(k, k')$ entry from table $T_s$
        while $C_{k'}(x) \neq y$ and $i < t$ do
            increment $i$
            $k' \leftarrow f_s(k')$
        end while
        if $C_{k'}(x) = y$ then
            yield $k'$
        end if
    end if
end for
abort: the attack failed
```

Figure 2.38. Time-Memory tradeoff (Attack).

We can estimate the number of pairwise different $k_{i,j}$ in one table as follows.

$$E\left(\#\{k_{i,j}; 1 \leq i \leq m, 0 \leq j < t\}\right) = \sum_{i=1}^{m}\sum_{j=0}^{t-1} \Pr\left[k_{i,j} \notin \{k_{i',j'}; i' < i \text{ or } j' < j\}\right].$$

(We do not count the end value of each chain.) Let $\text{Fresh}_{i,j}$ be the event $\left[k_{i,j} \notin \{k_{i',j'}; i' < i \text{ or } j' < j\}\right]$. Since $k_{i,j} = f(k_{i,j-1})$, we have $\text{Fresh}_{i,j} \subseteq \text{Fresh}_{i,j-1}$. Furthermore, we have

$$\Pr\left[\text{Fresh}_{i,j}|\text{Fresh}_{i,j-1}\right] \geq 1 - \frac{it}{2^n}.$$

Hence we have

$$\Pr\left[\text{Fresh}_{i,j}\right] \geq \left(1 - \frac{it}{2^n}\right)^{j+1}.$$

We deduce that the expected number of keys per table is greater than

$$\sum_{i=1}^{m}\sum_{j=0}^{t-1}\left(1 - \frac{it}{2^n}\right)^{j+1}.$$

The probability of success for looking for K is the probability that K is one of the keys in the table. Thus we have

$$p \geq 2^{-n}\sum_{i=1}^{m}\sum_{j=0}^{t-1}\left(1 - \frac{it}{2^n}\right)^{j+1}.$$

As the table gets larger, some chain will eventually collide and merge, which prevents the number of keys per table from increasing. So it may be preferable to keep many small tables. Figs. 2.37, and 2.38 illustrate the multitable version. Obviously the precomputation complexity is $P = \ell mt$ encryptions and reductions. The time complexity of the attack is $T = \ell t$ encryptions and reductions in the worst case, and $T_e = \frac{\ell}{2}t$ in the average case (for success only). The memory size is about $M = \ell m$ blocks. The probability p of success is such that

$$p \geq 1 - \left(1 - 2^{-n}\sum_{i=1}^{m}\sum_{j=0}^{t-1}\left(1 - \frac{it}{2^n}\right)^{j+1}\right)^{\ell}.$$

We let $x = 2^{\frac{n}{3}}$ and $1 - e^{-L}$ be the right-hand term of the p inequality. For $\ell = \lambda x$, $m = \mu x$, and $t = \tau x$, when $\mu\tau \ll x$, we have

$$L = -\lambda x \log\left(1 - x^{-3}\sum_{i=1}^{\mu x}\sum_{j=0}^{\tau x - 1}\left(1 - \frac{i\tau}{x^2}\right)^{j+1}\right)$$

$$\approx \frac{\lambda}{x^2}\sum_{i=1}^{\mu x}\sum_{j=0}^{\tau x-1}\left(1-\frac{i\tau}{x^2}\right)^{j+1}$$
$$= \frac{\lambda}{\tau}\sum_{i=1}^{\mu x}\frac{1}{i}\left(1-\left(1-\frac{i\tau}{x^2}\right)^{\tau x}\right)\left(1-\frac{i\tau}{x^2}\right)$$
$$\approx \frac{\lambda}{\tau}\sum_{i=1}^{\mu x}\frac{1}{i}\left(1-e^{-\frac{i\tau^2}{x}}\right)$$
$$\approx \frac{\lambda}{\tau}\int_0^{\mu\tau^2}\frac{1-e^{-t}}{t}\,dt.$$

(Note that this is equivalent to $\frac{\lambda}{\tau}\log(\mu\tau^2)$ when $\mu\tau^2$ increases, so that $p \geq 1-(\mu\tau^2)^{-\frac{\lambda}{\tau}}$ when μ and τ are large enough.) With $\ell \approx m \approx t \approx 2^{\frac{n}{3}}$ we obtain $P \approx 2^n$, $T \approx M \approx 2^{2n/3}$, and $p \geq 0.549$.

There exist several improvements of this technique, e.g. using the notion of “distinguished points.” The most efficient so far uses the notion of “rainbow tables” and is due to Philippe Oechslin (see Ref. [142]). Here, instead of using many tables with one reduction function in each, we use a larger table with many reduction functions (see Fig. 2.39). We define

$$k_{i,j} = f_j(k_{i,j-1}).$$

The algorithm is depicted in Figs. 2.40, 2.41.

Here we make sure that no chains collide in the precomputation. This induces an overhead in the precomputation. At the i-th iteration the probability for a new chain to collide is approximately $1-e^{-t\frac{i-1}{2^n}}$, so the precomputation complexity is

$$P \approx \sum_{i=1}^{m} t e^{t\frac{i-1}{2^n}} = t\frac{e^{\frac{mt}{2^n}}-1}{e^{\frac{t}{2^n}}-1} \approx 2^n\left(e^{\frac{mt}{2^n}}-1\right).$$

The probability of a key being in a column is $m.2^{-n}$ so

$$p = 1-(1-m.2^{-n})^t$$

$$\begin{array}{c|c|ccccccccccc|c|cc}
\cline{2-2}\cline{14-14}
 & k_{1,0} & \overset{f_1}{\mapsto} & k_{1,1} & \overset{f_2}{\mapsto} & k_{1,2} & \overset{f_3}{\mapsto} & k_{1,3} & \overset{f_4}{\mapsto} & \cdots & \overset{f_{t-1}}{\mapsto} & k_{1,t-1} & \overset{f_t}{\mapsto} & k_{1,t} & & (k_{1,t},k_{1,0}) \\
 & k_{2,0} & \overset{f_1}{\mapsto} & k_{2,1} & \overset{f_2}{\mapsto} & k_{2,2} & \overset{f_3}{\mapsto} & k_{2,3} & \overset{f_4}{\mapsto} & \cdots & \overset{f_{t-1}}{\mapsto} & k_{2,t-1} & \overset{f_t}{\mapsto} & k_{2,t} & & (k_{2,t},k_{2,0}) \\
T: & k_{3,0} & \overset{f_1}{\mapsto} & k_{3,1} & \overset{f_2}{\mapsto} & k_{3,2} & \overset{f_3}{\mapsto} & k_{3,3} & \overset{f_4}{\mapsto} & \cdots & \overset{f_{t-1}}{\mapsto} & k_{3,t-1} & \overset{f_t}{\mapsto} & k_{3,t} & \Rightarrow & (k_{3,t},k_{3,0}) \\
 & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
 & k_{m,0} & \overset{f_1}{\mapsto} & k_{m,1} & \overset{f_2}{\mapsto} & k_{m,2} & \overset{f_3}{\mapsto} & k_{m,3} & \overset{f_4}{\mapsto} & \cdots & \overset{f_{t-1}}{\mapsto} & k_{3,t-1} & \overset{f_t}{\mapsto} & k_{m,t} & & (k_{m,t},k_{m,0}) \\
\cline{2-2}\cline{14-14}
\end{array}$$

Figure 2.39. Rainbow table.

```
Input: an encryption scheme $C$, a fixed message $x$
Parameter: $m, t$
Preprocessing
for $j = 1$ to $t$ do
    pick a reduction function $R_j$ at random and define $f_j : k \mapsto R_j(C_k(x))$
end for
for $i = 1$ to $m$ do
    repeat
        pick $k'$ at random
        $k \leftarrow k'$
        for $j = 1$ to $t$ do
            compute $k \leftarrow f_j(k)$
        end for
    until $T$ contains no $(k, .)$ entry
    insert $(k, k')$ in table $T$
end for
```

Figure 2.40. Time–Memory tradeoff with a rainbow table (Preprocessing).

$$\approx 1 - e^{-\frac{mt}{2^n}}.$$

The time complexity of the attack is $T = \frac{t^2}{2}$ encryptions and reductions in the worst case, and $T_e = \frac{t^2}{8}$ in the average case (for success only). The memory size is about $M = m$ blocks. With $m = 2^{\frac{2n}{3}}$ and $t = (\log 2)2^{\frac{n}{3}} \approx 0.693 \times 2^{\frac{n}{3}}$ we obtain $P \approx 2^n$, $M \approx 2^{\frac{2n}{3}}$, $T \approx 0.240 \times 2^{\frac{2n}{3}}$, and $p \approx \frac{1}{2}$. This is a substantial improvement.

To summarize brute force generic attacks, we list in the following table the *essential* complexity of each part of the attack for a probability of success close to one.

Strategy	Preprocessing	Memory	Time
Exhaustive search	1	1	N
Dictionary attack	N	N	1
Tradeoffs	N	$N^{\frac{2}{3}}$	$N^{\frac{2}{3}}$

2.9.5 Meet-in-the-Middle Attack

In order to strengthen DES, people first tried to make a simple product with itself, resulting in a double-DES (see Section 2.3.1). More generally, we have

$$C_{K_1K_2}(X) = C''_{K_2}\left(C'_{K_1}(X)\right).$$

```
Attack
Attack input: $y = C_K(x)$
for $i = t$ down to 1 do
    set $k$ to $R_i(y)$
    for $j = i$ to $t$ do
        $k \leftarrow f_j(k)$
    end for
    if $T$ contains one $(k, .)$ entry then
        get the $(k, k')$ entry from table $T$
        for $s = 1$ to $i - 1$ do
            $k' \leftarrow f_s(k')$
        end for
        if $C_{k'}(x) = y$ then
            yield $k'$
            abort: the attack succeeded
        end if
    end if
end for
abort: the attack failed
```

Figure 2.41. Time–Memory tradeoff with a rainbow table (Attack).

If C' and C'' have key spaces of N' and N'' keys respectively, the ultimate security goal is to have a security related to a minimal complexity of $N = N' \times N''$. This goal is not achieved as shown by the following attack. The attack is called "meet-in-the-middle attack" and was devised by Ralph Merkle and Martin Hellman in 1981 (see Ref. [133]).

As illustrated in Fig. 2.42, let us consider a known plaintext attack: we know a (x, y) pair such that $y = C_{K_1K_2}(x)$. The meet-in-the-middle attack consists of making a table of all possible $C_{K_1}(x)$ (which is of size N'), and for all possible K_2, in computing $C_{K_2}^{-1}(y)$ and looking for the value in the table. This will suggest a list of possible K_1K_2 keys which can be tried with other known plaintexts (see Fig. 2.43). The complexity is $N' + N''$ instead of N. Therefore doubling the structure is not a good paradigm for strengthening the security.

2.10 Exercises

Exercise 2.1. *Prove that*

$$\mathrm{DES}_{\overline{K}}(\overline{x}) = \overline{\mathrm{DES}_K(x)}.$$

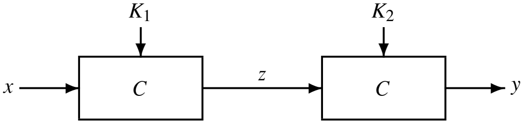

Figure 2.42. Meet-in-the-Middle.

```
Input: two encryption schemes $C'$ and $C''$ with two corresponding sets
    of possible keys $\mathcal{K}'$ and $\mathcal{K}''$, an $(x, y)$ pair with $y = C''_{K_2}(C'_{K_1}(x))$
for all $k_1 \in \mathcal{K}'$ do
    compute $z = C'_{k_1}(x)$
    insert $(z, k_1)$ in a hash table (indexed with the first entry)
end for
for all $k_2 \in \mathcal{K}''$ do
    compute $z = C''^{-1}_{k_2}(y)$
    for all $(z, k_1)$ in the hash table do
        yield $(k_1, k_2)$
    end for
end for
```

Figure 2.43. Meet-in-the-middle attack.

Deduce a brute force attack against DES with average complexity 2^{54} *DES encryptions.*[13]

Exercise 2.2. *We say that a DES key K is weak if DES_K is an involution. Exhibit four weak keys for DES.*

Exercise 2.3. *We say that a DES key K is semi-weak if it is not weak and if there exists K' such that*

$$\mathrm{DES}_K^{-1} = \mathrm{DES}_{K'}.$$

Exhibit four semi-weak keys for DES.[14]

Exercise 2.4. *In CBC Mode, show that an opponent can replace one ciphertext block so that the decryption will let all plaintext blocks but x_i and x_{i+1} unchanged.*

Exercise 2.5. *Describe how the OFB decryption is performed.*

Describe how the CFB decryption is performed.

Exercise 2.6. *Prove that the multiplication over nonzero residues modulo* $2^{16}+1$ *defines a group operation over a set of* 2^{16} *elements.*

We represent nonzero residues modulo $2^{16}+1$ *with their modulo* 2^{16} *reduction. (This means that "1" represents 1, "2" represents 2, etc., and "0" represents* 2^{16}*.)*

Explain how to efficiently implement this operation in software from regular CPU operations. (This operation is used in IDEA.)

Exercise 2.7. *Show that* $x \mapsto (45^x \bmod 257) \bmod 256$ *is a permutation over* $\{0, \ldots, 255\}$. *(This permutation is used as a substitution box in SAFER K-64.)*

[13] This exercise was inspired by Ref. [89].

[14] This exercise was inspired by Ref. [89].

Exercise 2.8. *We want to break a keyed cryptographic system. We assume we have access to an oracle which on each queried key answer whether or not the key is correct. We iteratively query the oracle with randomly selected keys (in an independent way). Compute the expected complexity (in term of oracle queries) in general, and when the distribution of the key is uniform. How can we improve the attack?*

Exercise 2.9. *We assume that we have an oracle which for any of the N possible keys K answers if it is correct or not.*

If the a priori distribution of the keys is not uniform but known by the adversary, what is the best algorithm for finding the key with the oracle? Prove that its complexity relates to the guesswork which is defined by

$$W(K) = \sum_{i=1}^{N} i. \Pr[K = k_i]$$

where all possible keys $k_1, \ldots, k_N$ are sorted such that the $\Pr[K = k_i]$ sequence is decreasing.[15]

As an example, let us consider a pseudorandom generator which generates an element x of $\{0, 1, \ldots, p-1\}$ uniformly for a given prime number p. In order to get a random n-bit string, we consider $K = x \bmod 2^n$.

1. *Compute $W(K)$ for $p > 2^n$. Example: $n = 64$ and $p = 2^{64} + 13$.*
2. *Compute $W(K)$ for $p < 2^n$. Example: $n = 64$ and $p = 2^{64} - 59$.*
3. *Let $\Delta = |p - 2^n|$. Express $W(K)$ in terms of Δ and compare both cases.*

Exercise 2.10. *Explain how to make a dictionary attack against UNIX passwords. What is the complexity?*

Exercise 2.11. *We assume that we have an oracle which for any of the N possible keys K answers if it is correct or not.*

If the a priori distribution of the keys is not uniform but known by the adversary, what is the best memoryless algorithm for finding the key with the oracle? Prove that its complexity relates to the Renyi entropy of coefficient $\frac{1}{2}$ which is defined by

$$H_{\frac{1}{2}}(K) = \left(\sum_{i=1}^{N} \sqrt{\Pr[K = k_i]} \right)^2$$

where $k_1, \ldots, k_N$ is the list of all possible keys.[16]

[15] This exercise was inspired by Ref. [123].

[16] This exercise was inspired by Ref. [39].