Notes by David Oppenheimer

Outline

- Why do DES and other block ciphers work?
- Differential and linear cryptanalysis
- Sample block cipher: IDEA
- Sample stream cipher: RC4
- Public-key algorithms
  - Merkle-Hellman Knapsack Algorithm
  - RSA

1 Block ciphers

Block ciphers encrypt plaintext in blocks and decrypt ciphertext in blocks.

1.1 Feistel networks

Many block ciphers take the form of *Feistel networks*. The structure of a Feistel network is illustrated in Figure 25.

In a Feistel network, the input is divided evenly into a left and right block, and the output of the i'th round of the cipher is calculated as

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1}, K_i) \end{aligned}$$

The good property of Feistel networks is that f can be made arbitrarily complicated, yet the input can always be recovered from the output as

$$\begin{aligned} R_{i-1} &= L_i \\ L_{i-1} &= R_i \oplus f(L_i, K_i) \end{aligned}$$

Therefore even if f is not invertible, the algorithm itself will be invertible, allowing for decryption (if you know the K_i's, of course).

Feistel networks are part of DES, Lucipher, FEAL, Khufu, LOKI, GOST, Blowfish, and other block cipher algorithms. You can see the Feistel structure in one round of DES, which is depicted in Figure 26.

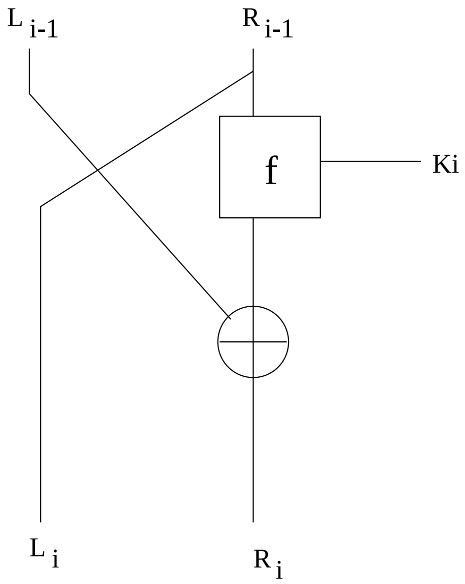


Figure 25: Feistel Network

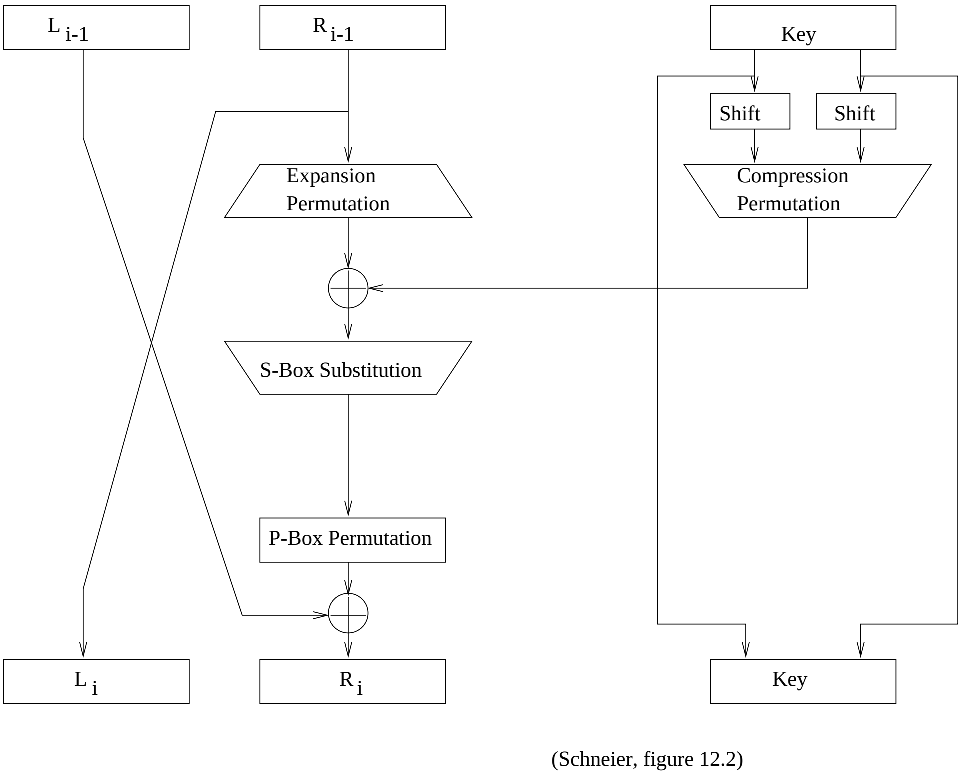


Figure 26: One round of DES (there are 16 rounds all together).

1.2 Security and Block Cipher Design Goals

This raises the question of how to design f so that the cipher is secure.

Confusion, diffusion, and the avalanche effect are three goals of block cipher design. *Confusion* decorrelates the output from the input. This can be achieved with *substitutions*. *Diffusion* spreads (diffuses) the correlations in the input over the output. This can be achieved with *permutations*: the bits from the input are mixed up and spread over the output. A good block cipher will be designed with an *avalanche effect*, in which one input bit rapidly affects all output bits, ideally in a statistically even way. By "rapidly" we mean after only a few rounds of the cipher.

The ideal f for a block cipher with 64-bit blocks would be a totally random 64-bit $\rightarrow$ 64-bit key-dependent substitution. Unfortunately this would require a table with $2^{64} \cdot 2^{56} = 2^{120}$ entries for a cipher with 56-bit keys (like DES): for each of the 2^{56} keys you'd need a mapping from each of the 2^{64} possible inputs to the appropriate 64-bit output.

Since this would require too much storage, we approximate the ideal by using a pseudorandom mapping with substitutions on smaller blocks. An algorithm that repeatedly performs *substitutions* on smaller blocks (so the storage requirements for the mapping are feasible) and *permutations* across blocks is called a *substitution-permutation network*. This is an example of a *product cipher* because it iteratively generates confusion and diffusion. A cipher with the same structure in each *round*, repeated for multiple rounds, is called an *iterated block cipher*. DES is an example of an iterated block cipher, a substitution-permutation network, and a product cipher.

1.3 DES Security

DES was designed with the above goals in mind. It was also designed to resist an attack called *differential cryptanalysis*, which was not publicly known at the time.

1.3.1 E-box and P-box

The DES *E-box* (labeled as the expansion permutation in Figure 26) takes 32 bits as input and produces 48 bits as output as shown in Figure 27. The 48 output bits are the 32 input bits with some of the input bits repeated. In particular, for each block of 4 input bits, the first and fourth bits appear twice in the output while the two middle bits appear once. The E-box therefore enhances the avalanche effect, because 4 outputs from an S-box in round R will become the input to 6 S-boxes in round $R+1$. The E-box also enables the use of 48 bits of the key in the XOR operation that follows the E-box.

The DES *P-Box* is a straight 32-bit permutation (every input bit is used exactly once in the output)

1.3.2 S-box

There are 8 S-boxes; each takes six bits as input and produces 4 bits as output. Some important security properties of the S-boxes are

- The output bits are not a linear function of the inputs, nor close to a linear function

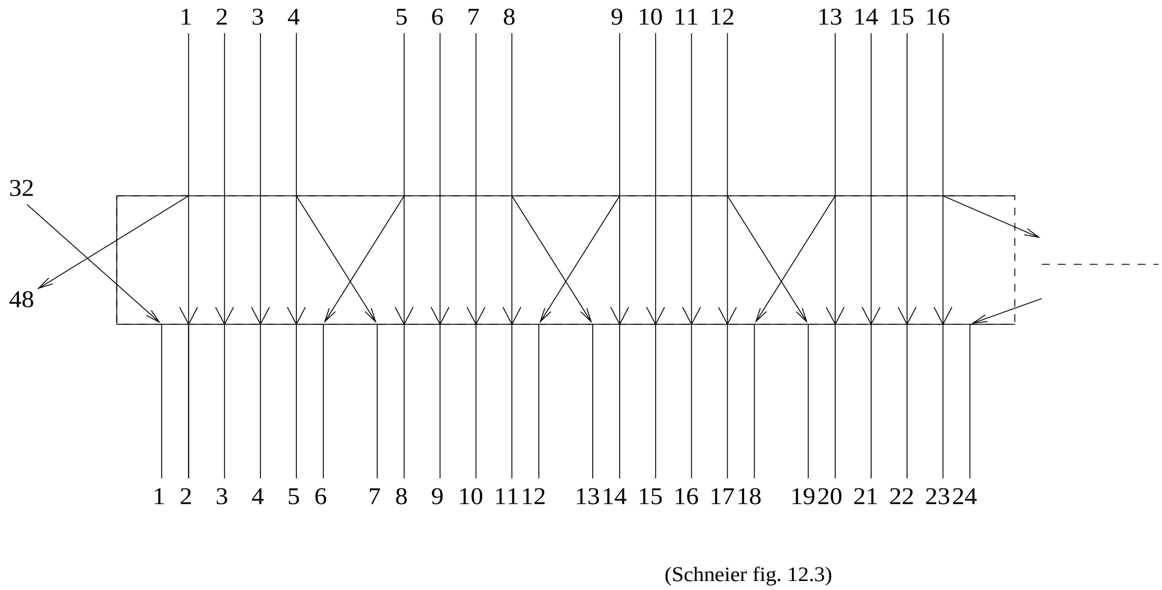


Figure 27: The DES E-box

- A difference of 1 bit in the input to an S-box affects at least 2 bits in the output
- For any 6-bit input difference $i_1 \oplus i_2$, no more than 8 of the 32 possible input pairs with difference $i_1 \oplus i_2$ may result in the same output difference.

We will see that this last property is important in making DES resistant to differential cryptanalysis.

1.3.3 Weaknesses of DES

- Some keys are insecure. Of the 2^{56} possible DES keys, 64 of them are poor choices. Luckily this is few enough that a DES encryption program can check to see if the key you want to use is one of these bad keys, and if so tell you not to use it.
  - Weak keys: There are four *weak keys*: 0x00000000000000, 0x0000000FFFFFFF, 0xFFFFFFF0000000, and 0xFFFFFFFFFFFFFF. When you use a weak key, the subkey used in each DES round is identical. This makes cryptanalysis a lot easier. It also makes $E_k(X) = D_k(X)$ since DES *decryption* follows the same algorithm as *encryption* but uses the subkeys in the reverse order. (If the subkeys are all the same, then obviously the forward and reverse order of them are identical.)
  - Semiweak keys: There are twelve *semiweak keys*. When you use a semiweak key, there are only two different subkeys used among the 16 DES rounds (each unique subkey is used 8 times). This makes cryptanalysis easier.
  - Possibly weak keys: There are forty-eight *possibly weak keys*. When you use a possibly weak key, there are only four unique subkeys generated; each one is used 4 times. Again, this makes cryptanalysis easier.

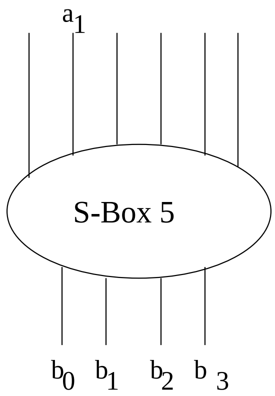


Figure 28: S-Box 5

- S-box 5 is strongly biased. Refer to Figure 28. $b_0 \oplus b_1 \oplus b_2 \oplus b_3 = a_1$ for 12 of the 64 possible inputs to S-box 5. Ideally this should be true for exactly half (32) of the possible inputs. Linear cryptanalysis can take advantage of this bias.

1.3.4 DES in the Real World

DES is used frequently in the "real world."

- Banks
  - bank machines
  - inter-bank transfers
- ISDN (some)
- Kerberos
- Unix password authentication (almost)
- Privacy-enhanced mail (PEM)

DES is not used in many newer systems because of the small 56-bit keysize. However, *triple-DES* (or *3-DES*) provides significantly more security while still using the DES algorithm. The basic idea is to encrypt multiple times using different keys. In 3-DES,

$$\begin{aligned} C &= E_{K1}(D_{K2}(E_{K3}(P))) \\ P &= D_{K3}(E_{K2}(D_{K1}(C))) \end{aligned}$$

Note that you *cannot* simply use $C = E_{K1}(E_{K2}(P))$ for DES (or any other block algorithm) and get much better security than simply using one key. If the algorithm is a group, then this is completely equivalent to saying $C = E_{K3}(P)$ for some K_3 and you have gained nothing. If the algorithm is not a group, there is a special attack which lets you find the key in 2^{n+1} encryptions for an algorithm with an n-bit key, instead of the expected 2^{2n} encryptions. DES is not a group.

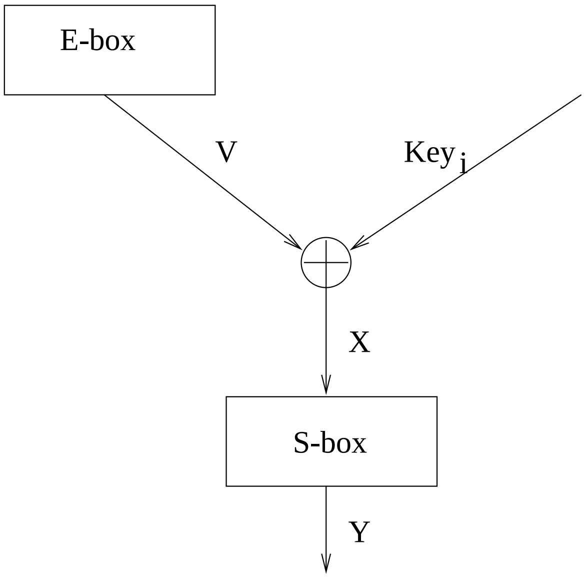


Figure 29: Differential Cryptanalysis

1.4 Differential Cryptanalysis

Define $\Delta V = V_1 \oplus V_2$ and $\Delta Y = Y_1 \oplus Y_2$.

Refer to Figure 29 which shows part of one round of DES. The basic idea behind differential cryptanalysis is that for any given ΔV, not all values of ΔY are equally likely. If a cryptanalyst is able to force someone to encrypt a pair of plaintext messages with a ΔV that he knows, he can look at the corresponding pair of encrypted messages, compute their ΔY, and use ΔV, ΔY, and the V's to get some information about Key_i. Of course, you need lots of pairs of messages before the deltas become statistically significant enough to give you information about the key.

This idea is applied one round at a time. The more rounds there are in the cipher, the harder it is to derive the original key using this technique because the statistical bias decreases with each round.

Differential cryptanalysis makes 14-round DES significantly easier to break than brute force, but has little effect on the security of 16-round DES. The designers of DES knew about differential cryptanalysis and designed the S-boxes (and chose the number of rounds) to resist this attack.

1.5 Linear Cryptanalysis

Linear cryptanalysis allows an attacker to gain some information about one bit of a key if he knows a plaintext and its corresponding ciphertext. The idea is that (the XOR of some plaintext bits) $\oplus$ (the XOR of some ciphertext bits), a one-bit by one-bit XOR that gives you a one-bit result, gives you information about the XOR of some of the key bits. This works when there is a bias in the linear approximation for an S-box. S-box 5 is the most biased S-box for DES.

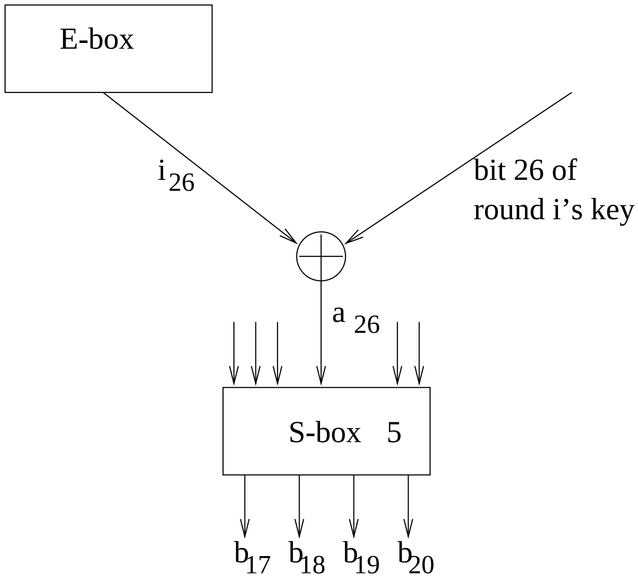


Figure 30: Linear Cryptanalysis using S-Box 5

Refer to Figure 30 which shows the operation of S-box 5. If you use linear cryptanalysis, knowing something about i_{26} (the second input bit to S-box 5, if we number input bits to the array of S-boxes starting with bit 1 on the left) and b_{17}, b_{18}, b_{19}, and b_{20} (the seventeenth through twentieth output bits from the array of S-boxes) will tell you something about the 26'th bit of the key used in that round. This is because (1) i_{26} is the XOR of the 26'th bit of the key used in that round, with the 26'th bit of the output from the expansion permutation for that round (which you know if you know the input bits to that round), and (2) S-box 5 has the following statistical bias: $i_{26} = b_{17} \oplus b_{18} \oplus b_{19} \oplus b_{20}$ for only 12 of the 64 possible inputs to S-box 5. Armed with enough plaintext-ciphertext pairs, linear approximations for the S-boxes, and a model of the S-boxes' biases, you can do statistical analysis to learn about the key.

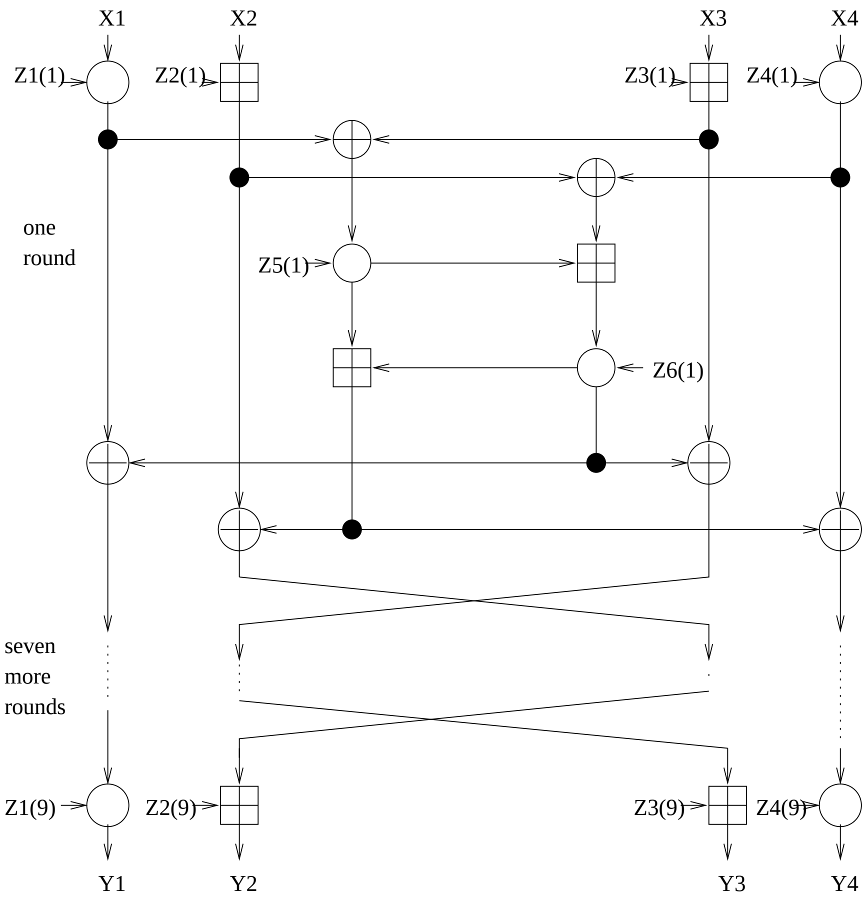
Linear cryptanalysis is the most successful attack on DES. Still, it's not much better than a brute force key search.

1.6 IDEA

IDEA, illustrated in Figure 31, is another block cipher. It operates on 64-bit blocks and uses 128-bit keys. IDEA is a product cipher. It composes three algebraic groups:

- Addition (which provides the diffusion property)
- Multiplication (which provides the confusion property)
- XOR

IDEA has some weak keys, but they're not weak in the same sense as DES keys. (If you use a weak IDEA key, an attacker can figure out which key you used by using a chosen-plaintext attack.) There are 2^{32} weak IDEA keys, but since there are so many more possible



Xi = 16-bit plaintext subblock

Yi = 16 bit ciphertext subblock

Zi(r) = 16-bit key subblock

$\oplus$ = bit-by-bit XOR of 16-bit subblocks

$\boxplus$ = addition modulo 2^{16} of 16-bit integers

$\bigcirc$ = multiplication modulo 2^{16} + 1 of 16-bit integers with the zero subblock corresponding to 2^{16}

(Schneier fig. 13.9)

Figure 31: The IDEA cipher

IDEA keys than DES keys, the chances of picking one of these randomly is smaller than the chance of picking one of the weak DES keys randomly.

Eight-round IDEA has stood up to all attacks to date. IDEA is used in PGP.

1.7 Block Cipher Encryption Speeds

Encryption Speeds of some Block Ciphers on a 33 MHz 486SX

algorithm	speed (KB/sec)	algorithm	speed (KB/sec)
Blowfish (12 rounds)	182	MDC (using MD4)	186
Blowfish (16 rounds)	135	MDC (using MD5)	135
Blowfish (20 rounds)	110	MDC (using SHA)	23
DES	35	NewDES	233
FEAL-8	300	REDOC II	1
FEAL-16	161	REDOC III	78
FEAL-32	91	RC5-32/8	127
GOST	53	RC5-32/12	86
IDEA	70	RC5-32/16	65
Khufu (16 rounds)	221	RC5-32/20	52
Khufu (24 rounds)	153	SAFER (6 rounds)	81
Khufu (32 rounds)	115	SAFER (8 rounds)	61
Luby-Rackoff (using MD4)	47	SAFER (10 rounds)	49
Luby-Rackoff (using MD5)	34	SAFER (12 rounds)	41
Luby-Rackoff (using SHA)	11	3-Way	25
Lucifer	52	Triple-DES	12

from Schneier, Table 14.3

2 Stream ciphers

2.1 Stream ciphers vs. Block cipher

Block ciphers operate on blocks of plaintext; for each input block they produce an output block. *Stream ciphers* usually encrypt/decrypt one bit at a time. The difference is illustrated in Figure 32, which depicts the operation of a typical block cipher and a typical stream cipher. Stream ciphers seem to be more popular in Europe, while block ciphers seem to be more popular in the United States.

A stream cipher uses a function (F in the picture) that takes a key and produces a stream of pseudorandom bits (b_i in the picture). One of these bits is XORed with one bit of the plaintext input (m_i), producing one bit of the ciphertext output (c_i).

A block cipher *mode* combines the cipher algorithm itself with feedback (usually). Figure 33 illustrates a block algorithm in CBC (Cipher Block Chaining) mode. Because the ciphertext for block *i-1* is used in calculating the ciphertext for block i, identical plaintext blocks will not always encrypt to the same ciphertext (as is true when a block algorithm is used directly without feedback).

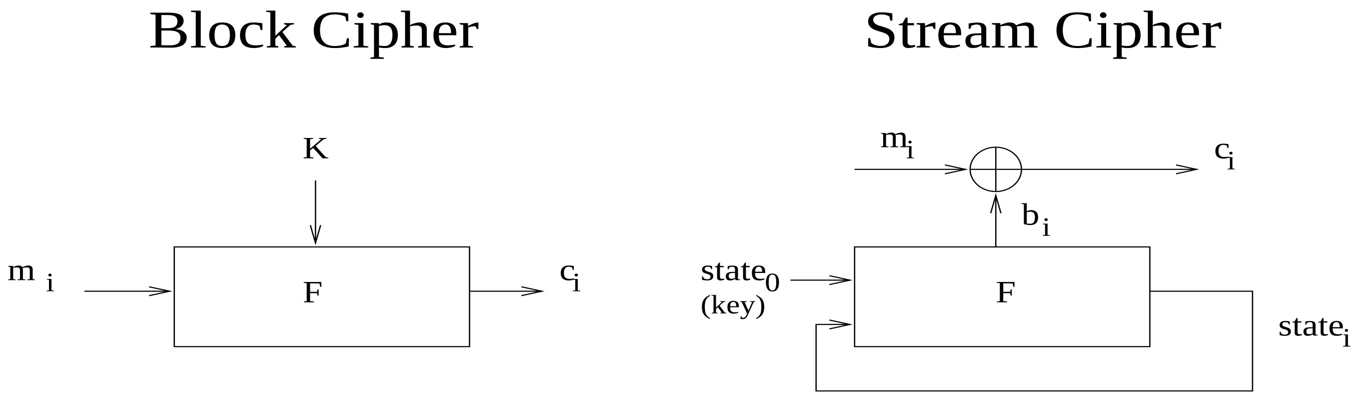


Figure 32: Block v. Stream Cipher

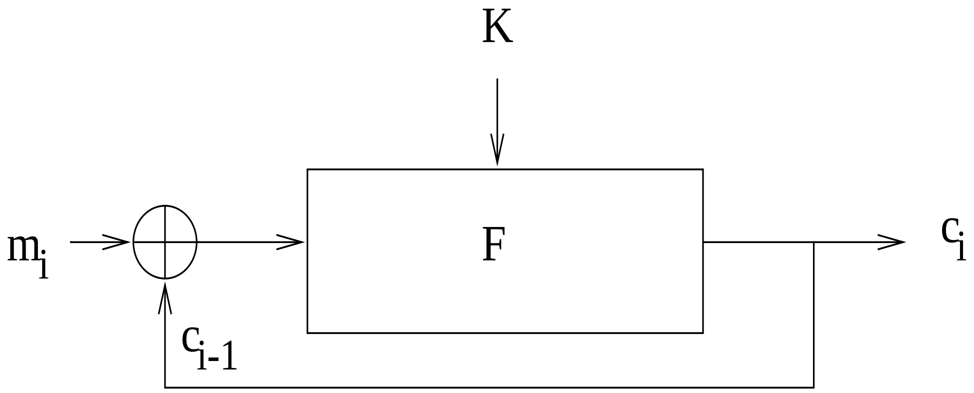


Figure 33: A block cipher in CBC mode

Block ciphers are typically more efficiently implemented in software than stream ciphers because block ciphers operate on many bits at a time.

2.2 RC4

RC4 is a stream cipher that generates one byte at a time. The key is used to generate a 256-element table $S_0, S_1, S_2, \ldots S_{255}$ which contains a permutation of the numbers 0 to 255. Once this table has been initialized, the following pseudocode for RC4 generates one byte on each round.

```
i=j=0
while (generating) {
        i = (i+1) mod 256
        j = (j+S_i) mod 256
        swap S_i and S_j
        t = (S_i + S_j) mod 256
        output = S_t
}
```

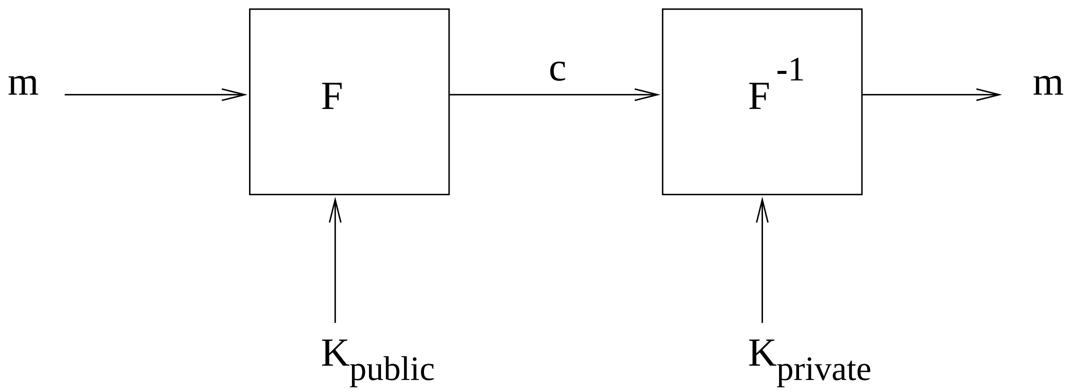


Figure 34: One-way trapdoor function

The byte-sized `output` is XORed with a byte of plaintext to produce a byte of ciphertext on each iteration.

RC4 is used in Lotus Notes, Oracle Secure SQL, and other products.

3 Public-Key Algorithms

Public-key algorithms are based on one-way trapdoor functions. As previously mentioned, a *one-way function* is "easy" to apply in the forward direction but "hard" to apply in the reverse direction. One-way functions with a *trapdoor* are "easy" to apply in the forward direction and "hard" to apply in the reverse direction *unless* you know a secret. If you know the secret, then the function is also "easy" to apply in the reverse direction.

Public key algorithms use two keys: a *private key* and a *public key*. Alice generates a public-private keypair. She publishes her public key and keeps the corresponding private key private. To send Alice a message, Bob encrypts using Alice's public key. To decrypt the message, Alice uses her private key. In contrast to symmetric encryption, two parties wishing to communicate using a public-key algorithm don't need to meet in person to exchange a secret key – they just need to look up each other's public keys in the public key directory. Exactly how to implement a secure public key distribution system (e.g. a secure public key directory) is quite another story.

Encrypting a message with a public key can be thought of as applying the "forward" direction of a one-way function. The corresponding private key is the "trapdoor" that makes the one-way function easy to reverse, allowing the encrypted message to be decrypted. This is illustrated in Figure 34.

3.1 Merkle-Hellman Knapsack Algorithm

The *Merkle-Hellman Knapsack Algorithm* is a public-key algorithm that derives its "security" from the NP-complete *knapsack problem*. The designers of this algorithm believed that to decrypt a message encrypted using the algorithm, a person would need to solve the knapsack problem. Unfortunately they were wrong: while the knapsack problem itself is NP-complete, the instances of the problem which are used for encryption using their algorithm make it possible to decrypt without having to solve an NP-complete problem.

The *knapsack problem* says: Given a vector M of weights and a sum S, calculate the boolean vector B such that $\sum_i B_i M_i = S$.

How can this be turned into a public-key algorithm? To encrypt, first break the input into blocks of size $\|M\|$ bits. (M should be a very long vector.) For each block calculate a sum $C_i = \sum_i B_i M_i$, where B_i is the plaintext. The series of C_i values calculated is the ciphertext. The vector M used to encrypt each block is the public key.

The corresponding private key is a different vector of weights; we'll call it M'. M' has the following important properties: (1) it is a series of superincreasing weights (we assume M was not), and (2) $\sum_i B_i M_i' = S$ – in other words, M' can also be used to solve for B given S.

The facts that makes this a sensible public-key algorithm are that (1) one can find B given a superincreasing M' in polynomial time, but not in polynomial time if given only a non-superincreasing M, and (2) it is easy to turn a superincreasing knapsack sequence into a normal knapsack sequence. (2) Makes it easy to compute your public key once you've selected your private key. Unfortunately, the way the authors generate the public key from the private key makes it possible for an adversary to compute the private key from the public key; the message can then be decrypted in polynomial time. This is the weakness in the algorithm.

The U.S. patent on the Merkle-Hellman knapsack public key algorithm expired on August 19, 1997.

3.2 RSA

While Merkle-Hellman's security was based on the difficulty of solving the knapsack problem, the RSA algorithm's security is based on the difficulty of factoring large numbers (in particular, products of large primes).

To compute a public-private keypair:

1. Pick big, random primes p and q
2. Calculate $n = pq$
3. Choose e relatively prime to $(p-1)(q-1)$. Note that e does not need to be random.
4. Calculate $d = e^{-1} \bmod (p-1)(q-1)$

The *public key* is (e, n). The *private key* is (d, n).

To encrypt: $c = m^e \bmod n$
To decrypt: $m = c^d \bmod n$

We'll now show that $E_d(E_e(m)) = m$.

$$\begin{aligned} E_d(E_e(m)) &= (m^e)^d \\ &= m^{ed} \end{aligned}$$

$$
\begin{aligned}
&= m^{k(p-1)(q-1)+1} \\
&= m \cdot m^{k(p-1)(q-1)} \\
&= m \cdot m^{(p-1)(q-1)^k} \\
&= m \cdot 1^k \\
&= m
\end{aligned}
$$

(Note that all operations are performed *mod n.*)
The one tricky part of this proof is the step $m \cdot m^{(p-1)(q-1)^k}$ (mod n) $= m \cdot 1^k$ (mod n). This step is explained in the next lecture after a discussion of some group theory. Briefly, it is because $\|Z_{pq}\| = (p-1)(q-1)$ and a theorem that says *For all finite groups G, for every a in G,* $a^{\|G\|} = I$, where in this case $I = 1$.

If factoring is easy, then breaking RSA becomes easy.