Introduction to Cryptography*

Guy E. Blelloch

October 12, 2000

Contents

1 Introduction **3**
1.1 Definitions 3
1.2 Primitives 5

2 Protocols **7**
2.1 Digital Signatures 7
2.2 Key Exchange 8
2.3 Authentication 9
2.4 Some Other Protocols 9

3 Some Number Theory **10**
3.1 Groups: Basics 10
3.2 Integers modulo n 10
3.3 Generators (primitive elements) 11
3.4 Discrete logarithms 11

4 Private-key Algorithms **11**
4.1 Block ciphers 12
4.1.1 Feistel networks 12
4.1.2 Some Building Blocks 13
4.1.3 Security and Block Cipher Design Goals 14
4.2 DES (Data Encryption Standard) 14
4.2.1 E-box and P-box 14
4.2.2 S-box 15
4.2.3 Weaknesses of DES 16
4.2.4 DES in the Real World 16
4.3 Differential Cryptanalysis 17
4.4 Linear Cryptanalysis 18
4.5 IDEA 19

*These notes are a rearrangement and adaption of scribe notes taken by Tzu-Yi Chen, David Oppenheimer and Marat Boshernitsan when I taught this class in the fall of 97.

4.6 Block Cipher Encryption Speeds 21
4.7 Stream ciphers 21
4.8 RC4 22

5 Public-Key Algorithms **22**
5.1 Merkle-Hellman Knapsack Algorithm 23
5.2 RSA 24
5.3 Factoring 25
5.4 ElGamal Algorithm 25

6 Probabilistic Encryption **26**
6.1 Generation Random Bits 26
6.2 Blum-Goldwasser 26

7 Quantum Cryptography **27**

8 Kerberos (A case study) **28**
8.1 How Kerberos Works 28
8.2 Tickets and Authenticators 28
8.3 Kerberos messages 29

1 Introduction

Like compression, this is a large field in its own right. In this class we focus on the algorithmic aspects of the field, paying little or no attention to other issues such as politics.

We begin with a series of basic definitions, building up to some primitives.

1.1 Definitions

Cryptography General term referring to the field.

Cryptology The mathematics behind cryptography.

Cryptanalysis The breaking of codes.

Steganography The hiding of messages. This is an emerging field which studies the hiding of messages inside messages. A simple example is the hiding of a watermark in a piece of paper.

Cipher A method or algorithm for encrypting and decrypting.

A few more terms are best described with the aid of Figure 1. Referring to Figure 1,

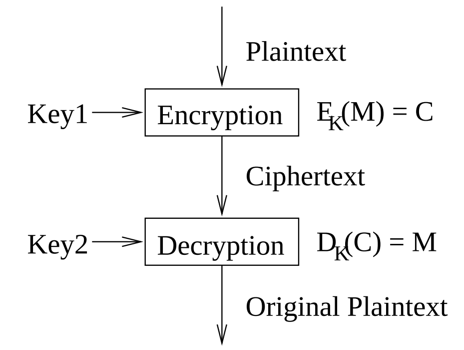


Figure 1: Flowchart showing the encryption and decryption of a message.

algorithms in which the two keys **key1** and **key2** are the same are often called *private-key* or *symmetric* algorithms (since the key needs to be kept private) and algorithms in which the two keys are different are often called *public-key* or *asymmetric* algorithms (since either **key1** or **key2** can be made public depending on the application).

Next we list some commonly used names . . . each name represents a specific role.

Alice initiates a message or protocol.

Bob second participant.

Carol third participant.

Trent trusted middleman.

Eve eavesdropper.

Mallory malicious active attacker.

Finally we define security by asking ourselves what it means to be secure. There are two definitions.

Unconditional Security. An algorithm is unconditionally secure if encrypted messages cannot be decoded without the key, even with infinite computational resources. In 1943 Shannon proved that for an algorithm to be unconditionally secure, the key must be as "long" as the message.

An example of a unconditionally secure algorithm is the *one-time-pad* in which the encoder and decoder both have the same random key which is the same length as the message. The encoder `XOR`s the message with the key, the decoder then decodes the message by `XOR`ing what he receives with the same random key. Each random key can only be used once. Such one-time pads were apparently used in the second-world war. Soldiers would carry around notepads full of "random" numbers. Once a page was used once, it had to be thrown out. Of course this is not a true one-time-pad since the random numbers were not truly random. In fact, as we will see, generating pseudo-random bits and `XOR`ing them with a message is one of the standard encryption techniques. Since the initial state of the generator is comparatively small, however, this is not unconditionally secure.

Computational Security. Here an algorithm is secure if it is "infeasible" to break—meaning that no (probabilistic) polynomial time (PPT) algorithm can decode a message. Unfortunately even under this weaker definition, no current cryptographic algorithm is provably secure. However, there are many systems that are conjectured to be computationally secure and no evidence has disproved this yet.

Notice that if $\mathcal{P} = \mathcal{NP}$, nothing is secure based on computational cost since we could effectively nondeterministically try all keys (although we might have to recognize when we have the right answer).

Cryptoanalytic Attacks There are several types of attacks that can be mounted against a particular encryption algorithm or protocol. We always assume that the cryptanalyst has complete knowledge of the encryption/decryption algorithms but does not know the decryption key. The cryptanalyst is trying to either decrypt the current message or figure out the key so she can decrypt future messages (or any other means of decrypting future messages).

Ciphertext-only attack. The cryptanalyst has the ciphertext for many messages all encrypted with the same algorithm and key. She knows nothing a priori about the plaintext.

Known-plaintext attack. The cryptanalyst has the ciphertext generated from know plaintext. For example, Alice sends Bob a standard file format with a common header. Eve is listening in and knows the plaintext for the header and sees the ciphertext as it goes by. In designing such attacks it is often assumed that the plaintext is random.

Chosen-plaintext attack. The cryptanalyst can select plaintext to encrypt and can see the corresponding encrypted ciphertext. For example, Eve might send a message to Alice to forward to Bob. Alice encrypts it, as she does with all her mail, and forwards it to Bob. Eve listens in.

Chosen-ciphertext attack. The cryptanalyst can select ciphertext to decrypt and can see the decrypted plaintext. For example, Alice is sending Bob an encrypted message to forward to Mallory. Mallory intercepts the message and replaces it with his own ciphertext. Bob decodes the message with his key, and forwards it to Mallory. This attack can be combined with a chosen-plaintext attack.

1.2 Primitives

Here we define three primitives, one-way functions, one-way trapdoor functions, and one way hashing functions. These form the basis of almost all cryptographic protocols.

One-way Functions A one-way function $y = f(x)$ is a function where it is easy to compute y from x, but "hard" to compute x from y. It turns out that the existence of one-way functions is a necessary condition for the existence of most primitives and protocols in cryptography, including the existence of secure private and public-key algorithms. Consider the following example of the relevance of one-way functions in a private-key protocol. If x is the encryption key, y is the ciphertext, m is the plaintext, and $y = f(x) = E_x(m)$, then we want it to be easy to compute y (the ciphertext) for a particular key x but hard to figure out the key x using a known-plaintext attack. Recall that in a know-plaintext attack we know at least one (y, m) pair. Knowing m defines f, and finding the key x is then the problem of finding $f^{-1}(y)$. Consider another example in the context of a public-key protocol. Interpret $y = f(x) = E_k(x)$ where k is the public key and x is the plaintext, and y is the ciphertext. Since k is public, we again know f, and finding $f^{-1}(y)$ would allow us to decrypt the ciphertext y. Note that the role of key and plaintext have been swapped in these two examples.

In defining one-way functions, we have to be very careful by what we mean by hard. For example we might try to define one-way functions as those in which computing x from y is NP-hard, but computing y from x is polynomial time (in the size of x). This might sound like the right definition, and it is in fact not difficult to come up with such functions. However, this is not a good definition. The problem is that computing x from y could only be hard in very few cases. For most cases it could be trivial—only once in a rare while would a message be secure. The flaw is that the theory of NP-hard problems is based on worst case performance while instead we would like best-case performance, or at least some form of average-case. Unfortunately the theory of NP-hard problems does not help us much here.

Rather than trying to find functions that are always hard to invert, it is sufficient to find functions that are hard to invert in most cases. With such functions we can make the likelihood of finding an easy case vanishingly small by composing the function. To define what we mean by "most" we say that a function $v(k) : N \to R$ is negligible if it vanishes faster than the inverse of any polynomial in k ($v(k) < 1/p(k)$ for sufficiently large k).

Definition 1.2.1 *A function* $f : \{0,1\}^* \to \{0,1\}^*$ *is* one-way *if:*

1. *There exists a PPT (Probabilistic Polynomial Time) algorithm that calculates* $f(x)$, *and*

2. *for every PPT algorithm* A *there is a negligible function* $v_A(k)$ *such that for sufficiently large* k,
$$P\left[f(z) = y : x \in \{0,1\}^k; y = f(x); z = A(1^k, y)\right] \leq v_a(k) \ .$$

This basically says that for any PPT algorithm the probability of being able to invert a particular y is asymptotically vanishingly small. There is also a notion of a weak one-way function in which the conditions are not as strong.

Unfortunately even assuming that $\mathcal{P} \neq \mathcal{NP}$ there is no proof that one-way functions exist. Since the security of just about all cryptographic protocols rely on their existence, we are left having to assume they exist. Even though we don't know they exist, there are many functions which are both conjectured to be one-way, and after many years of very smart people trying to disprove this, have still not been broken. Here we list some examples.

Factoring $y = u \cdot v$, where u and v are primes. If u and v are primes, it is hard to generate them from y.

Discrete Log $y = g^x \bmod p$, where p is a prime and g is a generator (i.e. $g^1, g^2, g^3, \ldots$ generate all i such that $0 \leq i < p$. This is believed to be about as hard as factoring.

DES with a fixed message $y = \mathrm{DES}_x(M)$. We will describe DES later. This assumes that DES is actually a family of functions with increasing key size but the same basic structure. For a fixed size key, as the real DES is defined, it makes no sense talking about asymptotics.

One-way Trapdoor Functions This is a one-way function with a "trapdoor". The trapdoor is a key that makes it easy to invert the function.

An example is RSA, where the function is $y = x^e \bmod N$, where $N = p \cdot q$. p,q, and e are prime, p and q are trapdoors (only one is needed). So, in public-key algorithms, $f(x)$ is the public key (e.g. e, N), whereas the trapdoor is the private key (e.g. p or q).

One-way Hashing Functions A one-way hashing function is written $y = H(x)$. H is a many-to-1 function, and y is often defined to be a fixed length (number of bits) independent of the length of x. The function H is chosen so that calculating y from x is easy, but given a y calculating any x such that $y = H(x)$ is "hard".

As in the case of one-way functions, one-way trapdoor functions and one-way hashing functions can be defined with similar formality.

2 Protocols

Here we describe a number of basic protocols, showing possible implementations. We note that in general a protocol which can be implemented using a private-key cryptosystem can also be implemented using a public-key cryptosystem—a primary difference is that the former may require a trusted middleman.

2.1 Digital Signatures

A digital signature is supposed to replace a normal signature. The properties we would like of such a signature include:

1. Convince a recipient or arbitrator of the author of a message (*i.e.*, make sure that Mallory cannot forge Alice's signature, assuming Alice is not malicious).

2. Do not allow repudiation (*i.e.*, make sure that Alice after signing a document does not come back and claim she did not sign it).

3. Make it impossible to tamper with a signed message without invalidating the signature (*i.e.*, make sure that Mallory cannot change the contents of a message that Alice has signed).

Digital signatures can be implemented using either private-key and public-key cryptosystems as illustrated below. It should be pointed out, however, that neither of these schemes practically address the second goal. This is because Alice could claim that her private-key had been compromised (*e.g.*, posted on the web) without her knowledge. This problem can be reduced but not solved by using time stamps. Also the security of the protocols is no more secure than the components they use, so none of the goals strictly hold.

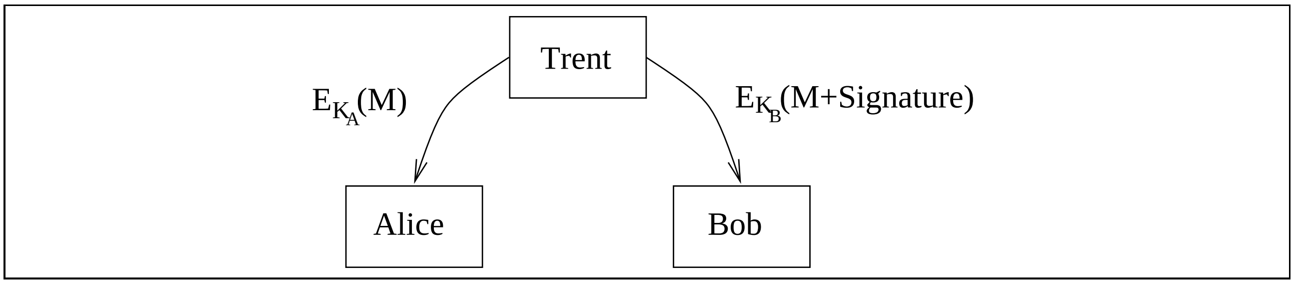


Figure 2: Flowchart showing an implementation of digital signatures using a private-key cryptosystem.

Using a private-key Cryptosystem: In Figure 2 Alice and Trent share the private key K_A and Bob and Trent share the private key K_B. Trent (a trusted middleman) receives a message from Alice which he can decrypt using K_A. Because he can decrypt it using K_A, he knows Alice sent the message. He now encrypts the decoded message plus a signature saying that Alice really sent this message using K_B. He sends this new version to Bob, who can now decode it using the key K_B.

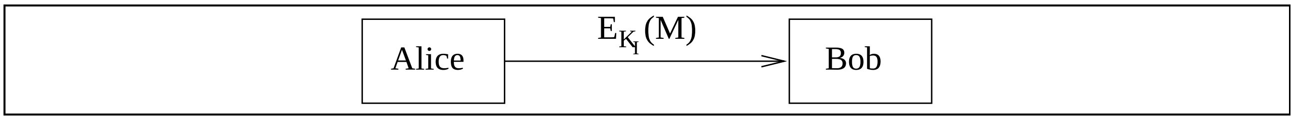


Figure 3: Flowchart showing an implementation of digital signatures using a public-key cryptosystem.

Using a public-key Cryptosystem: In Figure 3 K_I is Alice's private key. Bob decrypts the message using her public key. The act of encrypting the message with a private key that only Alice knows represents her signature on the message.

The following is a more efficient implementation using a public-key cryptosystem. In

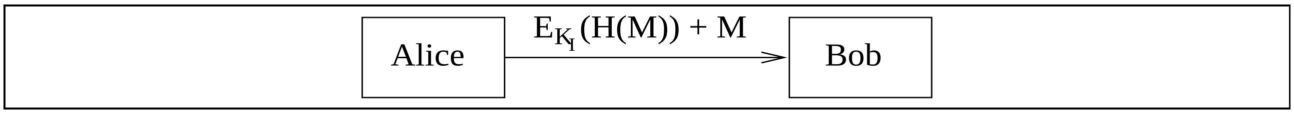


Figure 4: Flowchart showing a more efficient implementation of digital signatures using a public-key cryptosystem.

Figure 4 $H(M)$ is a one-way hash of M. Upon receiving M, Bob can verify that it indeed come from Alice by decrypting $H(M)$ using Alice's public key K_I and checking that M does in fact hash to the value encrypted.

2.2 Key Exchange

These are protocols for exchanging a key between two people. Again, they can be implemented both with private-key and public-key cryptosystems as follows:

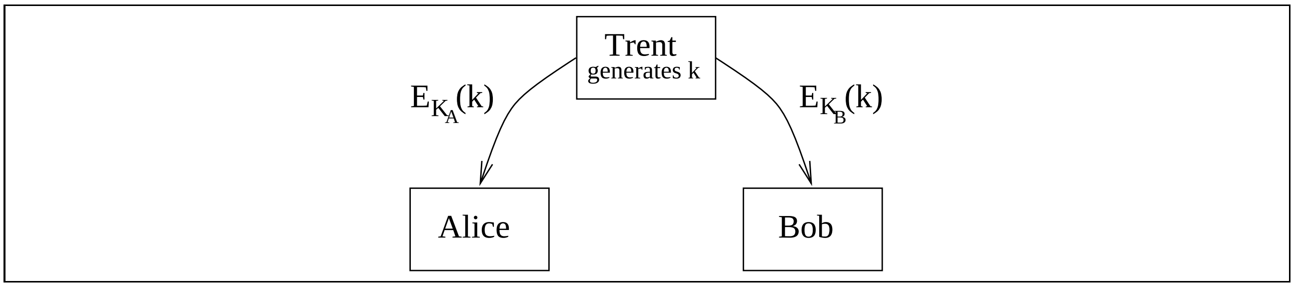


Figure 5: Flowchart showing an implementation of key exchange using a private-key cryptosystem.

Using a private-key Cryptosystem: In Figure 5, Alice and Trent share the private key K_A and Bob and Trent share the private key K_B. Trent generates a key k, then encodes k with K_A to send to Alice and encodes k with K_B to send to Bob.

Using a public-key Cryptosystem: In Figure 6 K_I is Bob's public key. Alice generates a key k, which she encrypts with Bob's public key. Bob decrypts the message using his private key.

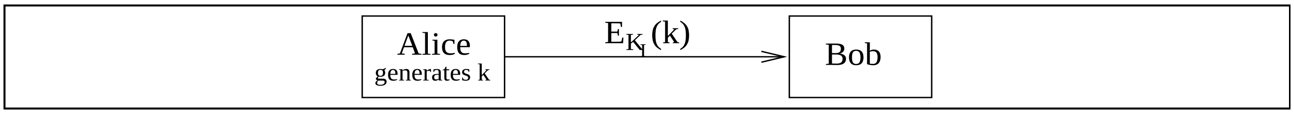


Figure 6: Flowchart showing an implementation of key exchange using a public-key cryptosystem.

2.3 Authentication

These protocols are often used in password programs (e.g. the Unix `passwd` function) where a host needs to authenticate that a user is who they say they are.

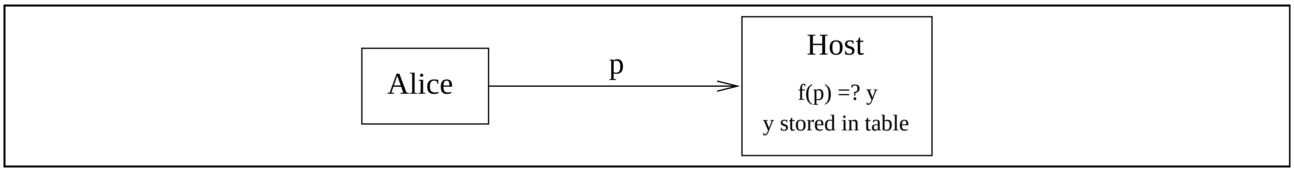


Figure 7: Flowchart showing a basic implementation of an authentication protocol.

Figure 7 shows a basic implementation of an authentication protocol. Alice sends p to the host. The host computes $f(p)$. The host already has a table which has y and checks to see that $f(p) = y$. f is a one-way function.

To make it less likely that Mallory will be able to pretend to be Alice to the host, sequences of passwords can be used. Sequences of passwords are used by having Alice give the host a random number R. The host computes $x_0 = R, x_1 = f(R), x_2 = f(f(R)), \ldots x_n = f^n(R)$ and gives all but x_n to Alice. The host keeps x_n. Now, the first time Alice logs in, she uses x_{n-1} as the password. The host computes $f(x_{n-1})$ and compares it to the x_n which it has. If the two are equivalent, Alice has been authenticated. Now the host replaces x_n with x_{n-1}. The next time Alice logs in, she uses x_{n-2}. Because each password is only used once by Alice, and f is a one-way function, sequences of passwords are more secure.

2.4 Some Other Protocols

Some other protocols which are not discussed here are:

- Secret sharing
- Timestamping services
- Zero-knowledge proofs
- Blind-signatures
- Key-escrow
- Secure-elections
- Digital Cash

3 Some Number Theory

3.1 Groups: Basics

A group is a set G with binary operation $*$ defined on G, such that the following properties are satisfied:

1. **Closure.** For all $a, b \in G$, we have $a * b \in G$.
2. **Associativity.** For all $a, b, c \in G$, we have $(a * b) * c = a * (b * c)$.
3. **Identity.** There is an element $I \in G$, such that for all $a \in G$, $a * I = I * a = a$.
4. **Inverse.** For every $a \in G$, there exists a unique element $b \in G$, such that $a * b = b * a = I$.

For example, integers $\mathbf{Z}$ under the operation of addition form a group: identity is 0 and the inverse of a is $-a$.

3.2 Integers modulo n

For prime p, Z_p is a group of integers modulo p:

- $Z_p \equiv \{1, 2, 3, ..., p-1\}$
- $* \equiv$ multiplication mod p
- $I \equiv 1$
- $a^{-1} \equiv a * a^{-1} = 1 \pmod{p}$

Example: $Z_7 = \{1, 2, 3, 4, 5, 6\}$. Note that if p is **not** a prime, Z_p is not a group under $*$, because there isn't necessarily an inverse. All encryption algorithms assume existence of an inverse. So, for n not prime, we define:

- $Z_n^* \equiv \{m \ : \ 1 \leq m < n, \ gcd(n, m) = 1\}$ (i.e. integers that are relatively prime to n)
- $* \equiv$ multiplication mod n
- $I \equiv 1$
- $a^{-1} \equiv a * a^{-1} = 1 \pmod{p}$

Example: $Z_{10}^* = \{1, 3, 7, 9\}$, $1^{-1} = 1$, $3^{-1} = 7$, $7^{-1} = 3$, $9^{-1} = 9$. The number of elements in the group Z_n^* is called the *Euler Phi* function, and denoted as $\phi(n)$. Lets say that n factors into the product of primes $p_1^{k1} p_2^{k2} \ldots p_l^{kl}$, where the ki are the powers of each prime. It is not hard to show that $\phi(n) = \prod_{i=1}^{l} p_i^{ki-1}(p_i - 1)$. For example $Z_{12} = \{1, 5, 7, 11\}$, $12 = 2^2 3$, and $\phi(12) = 2^{2-1}(2-1)3^{1-1}(3-1) = 4$. If n is a product of two primes p and r, then $\phi(n) = (p-1)(r-1)$.

3.3 Generators (primitive elements)

For a group G, element $a \in G$ is a **generator of** G if $G = \{a^n \ : \ n \in Z\}$ (i.e. G is "generated" by the powers of a). For Z_7 we have:

x	x^2	x^3	x^4	x^5	x^6
1	1	1	1	1	1
2	4	1	2	4	1
3	**2**	**6**	**4**	**5**	**1**
4	2	1	4	2	1
5	**4**	**6**	**2**	**3**	**1**
6	1	6	1	6	1

Here, 3 and 5 are the generators of Z_7 while 1, 2, 4 and 6 are not since none of them will generate 3. For Z_{10}^* we have:

x	x^2	x^3	x^4
1	1	1	1
3	**9**	**7**	**1**
7	**9**	**3**	**1**
9	1	9	1

So, 3 and 7 are the generators of Z_{10}^*. Note that any element to the power of the size of the group gives identity. In fact, there's a theorem stating that this is always true:

Theorem 3.3.1 *For all finite groups G and for any $a \in G$, $a^{|G|} = I$, where I is the identity element in G.*

From this theorem, it follows that for $a \in Z_p$, $a^{|Z_p|} = a^{p-1} = 1$, and for two primes p and q and $a \in Z_{pq}^*$, $a^{|Z_{pq}^*|} = a^{(p-1)(q-1)} = I$. The latter fact is essential for RSA.

3.4 Discrete logarithms

If g is a generator of Z_p (or Z_p^* if p is not prime), then for all y there is a unique x such that $y = g^x \pmod p$. We call this x the **discrete logarithm** of y, and use the notation $\log_g(y) = x \pmod p$.

For Example: in Z_7, $\log_3(6) = 3$, but $\log_4(5) = \emptyset$ (no inverse), and $\log_4(2) = \{2, 5\}$ (multiple inverses) because 4 is not a generator of Z_7.

4 Private-key Algorithms

We discuss block ciphers and then briefly introduce the well known and widely used block cipher DES. We begin discussion of how DES works, saving much for the next lecture.

4.1 Block ciphers

Block ciphers divide the message to be encrypted into blocks (e.g. 64 bits each) and then convert one block at a time. Hence:

$$C_i = f(k, M_i) \qquad\qquad M_i = f'(k, C_i)$$

Notice that if $M_i = M_j$, $C_i = C_j$. This is undesirable because if someone (Eve or Mallory) notices the same block of code going by multiple times, they know that this means identical blocks were coded. Hence they've gained some information about the original message.

There are various ways to avoid this. For example, something called *cipher block chaining (CBC)* encrypts as follows.

$$C_i = E_k(C_{i-1} \oplus M_i)$$

This has the effect that every code word depends on the code works that appear before it and are therefore not of any use out of context. Cipher block chaining can be decrypted as follows.

$$M_i = C_{i-1} \oplus D_k(C_i)$$

Unfortunately, this could still cause problems across messages. Note that if two messages begin the same way, their encodings will also begin the same way (but once the two messages differ, their encodings will be different from then on). Because many messages that people want to send begin with standard headers (e.g. specifying the type of file being sent), this means someone could gain information about the message being sent by creating a table of all the encrypted headers that they have seen. One way to fix this problem is to attach a random block to the front of every message.

4.1.1 Feistel networks

Many block ciphers take the form of *Feistel networks*. The structure of a Feistel network is illustrated in Figure 8.

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).

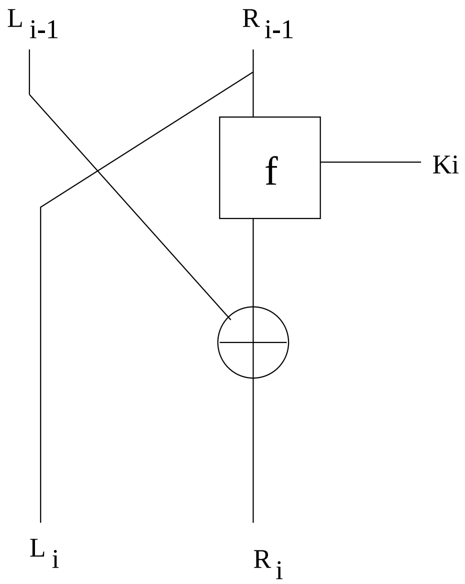


Figure 8: Feistel Network

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 9.

4.1.2 Some Building Blocks

There are several building-blocks that are commonly used within each round of block ciphers. Here we briefly describe some of them.

Substitution: A fixed mapping. For example:

$$\begin{array}{ccc} 000 & \rightarrow & 011 \\ 001 & \rightarrow & 101 \\ 010 & \rightarrow & 000 \\ 011 & \rightarrow & 001 \\ 100 & \rightarrow & 100 \\ & \vdots & \end{array}$$

This is often stored as a table. Since storing a large substitution is impractical (for b-bits it would require 2^b table entries), substitutions are most often made on subsets of the bits.

Permutation: A rearrangement of order. For example:

$$b_0b_1b_2b_3b_4 \quad \rightarrow \quad b_2b_4b_3b_0b_1$$

This is also often stored as a table.

Expansion Permutation: A rearrangement of order that may repeat some bits of the input in the output. For example:

$$b_0b_1b_2b_3 \quad \rightarrow \quad b_2b_1b_2b_0b_3b_0$$

4.1.3 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.

4.2 DES (Data Encryption Standard)

DES was designed by IBM for NIST (the National Institute of Standards and Technology) with "consultation" from the NSA in 1975.

DES uses 64-bit blocks and 56-bit keys. It is still used heavily, for example by banks when they encrypt the PIN that you type in at an ATM machine. DES was designed with the goals mentioned in Section 4.1.3 in mind. It was also designed to resist an attack called *differential cryptanalysis*, which was not publicly known at the time. Because of its short key, however, DES on its own is no longer very safe—supercomputers can crack it in a few hours. Based on DES, one can build an algorithm called 3-DES which is conjectured to be considerably safer.

DES uses a 16-round Feistel network. A single round is shown in Figure 9 and we will describe each of the components of the round.

4.2.1 E-box and P-box

The DES *E-box* (labeled as the expansion permutation in Figure 9) takes 32 bits as input and produces 48 bits as output as shown in Figure 10. The 48 output bits are the 32 input

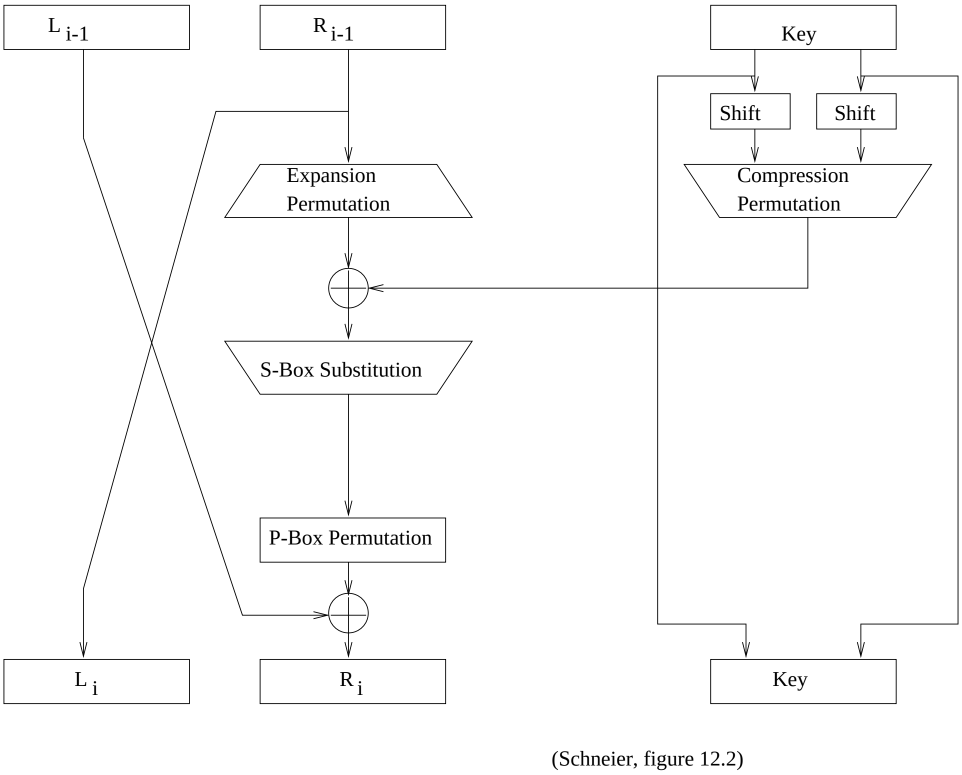


Figure 9: One round of DES (there are 16 rounds all together).

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)

4.2.2 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
- 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.

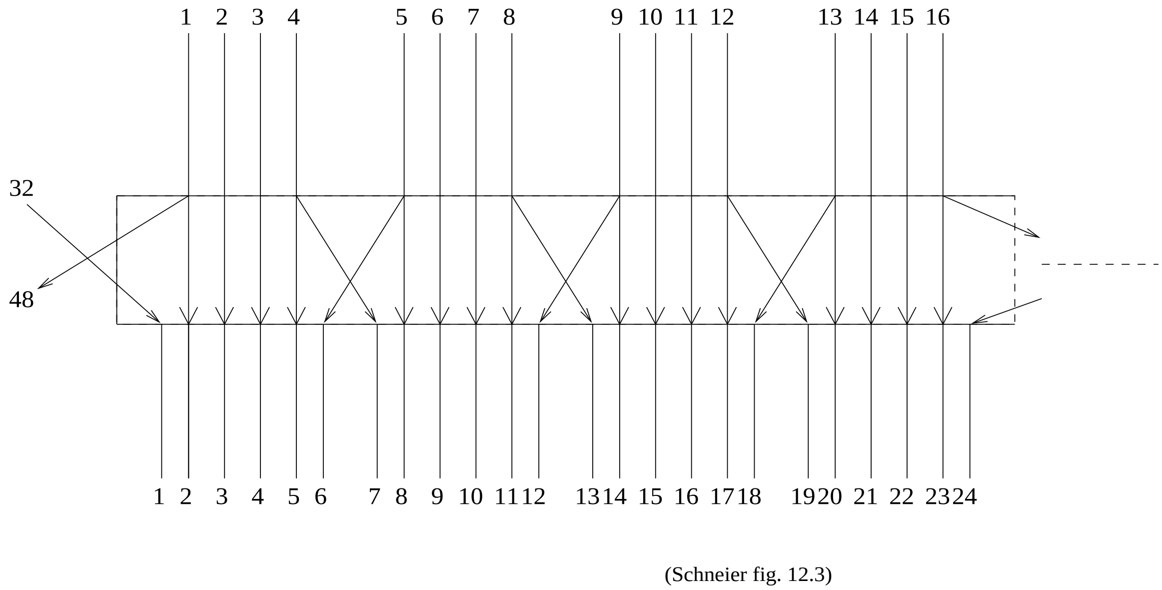


Figure 10: The DES E-box

4.2.3 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.
- S-box 5 is strongly biased. Refer to Figure 11. $b_0 \oplus b_1 \oplus b_2 \oplus b_3 = a_1$ 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.

4.2.4 DES in the Real World

DES is used frequently in the "real world."

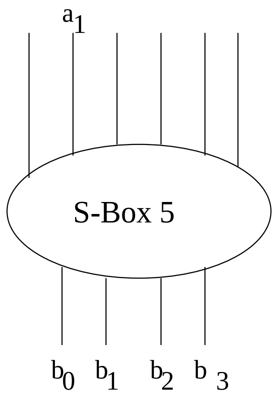


Figure 11: S-Box 5

- 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.

4.3 Differential Cryptanalysis

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

Refer to Figure 12 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

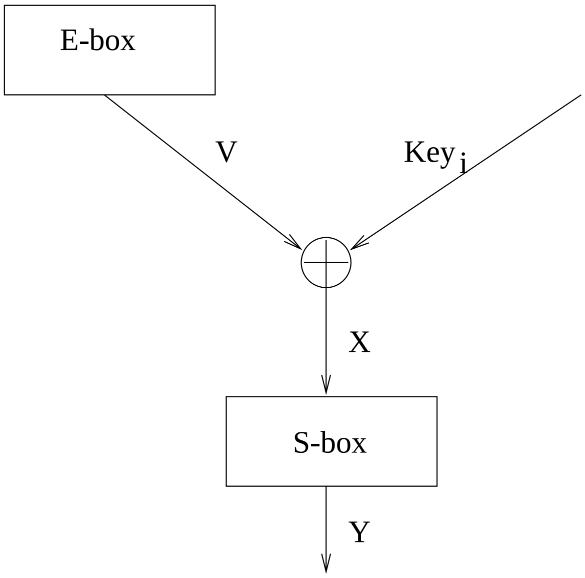


Figure 12: Differential Cryptanalysis

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.

4.4 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.

Refer to Figure 13 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

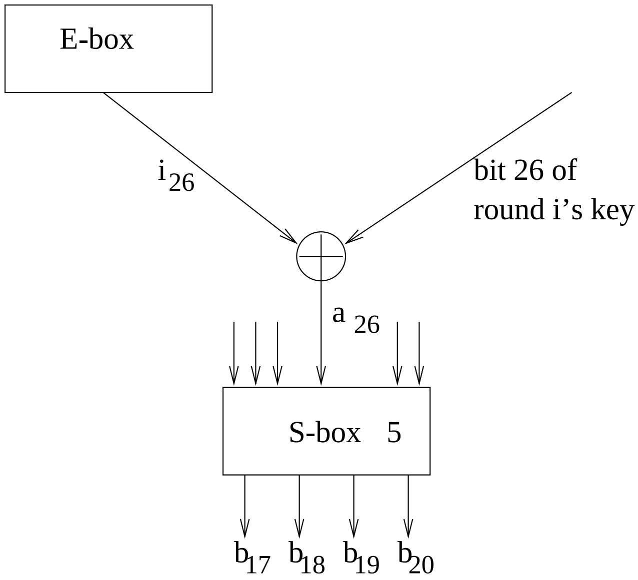


Figure 13: Linear Cryptanalysis using S-Box 5

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.

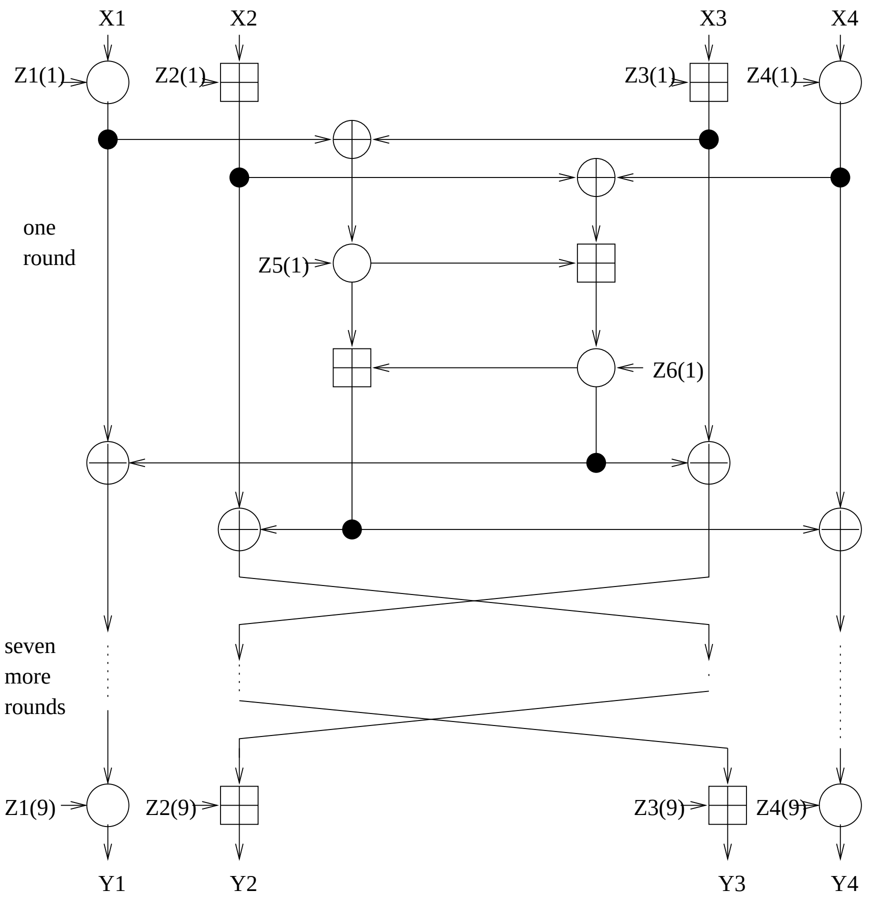
4.5 IDEA

IDEA, illustrated in Figure 14, 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 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.



Xi = 16-bit plaintext subblock

Yi = 16 bit ciphertext subblock

Zi(r) = 16-bit key subblock

(Schneier fig. 13.9)

$\oplus$ = 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}

Figure 14: The IDEA cipher

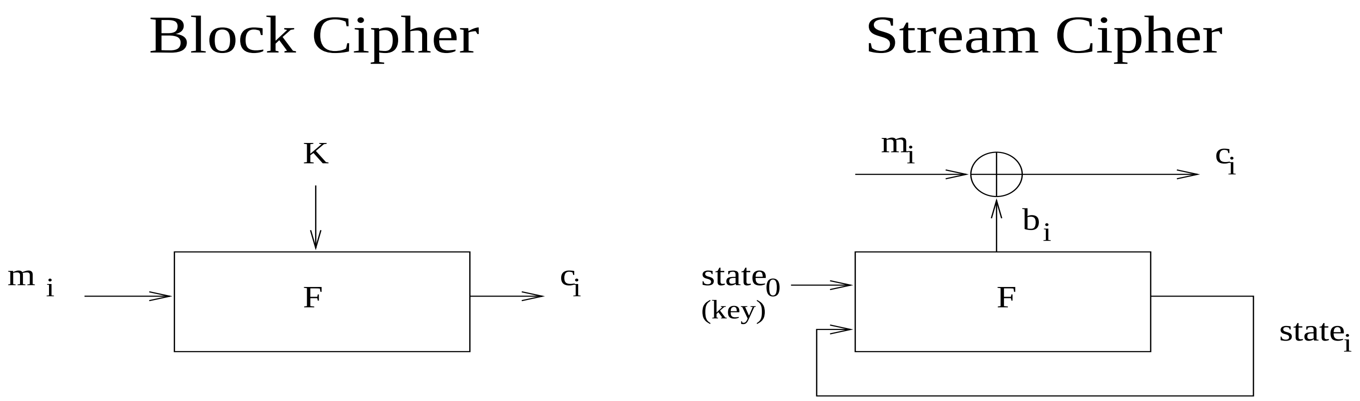


Figure 15: Block v. Stream Cipher

4.6 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

4.7 Stream ciphers

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 15, 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

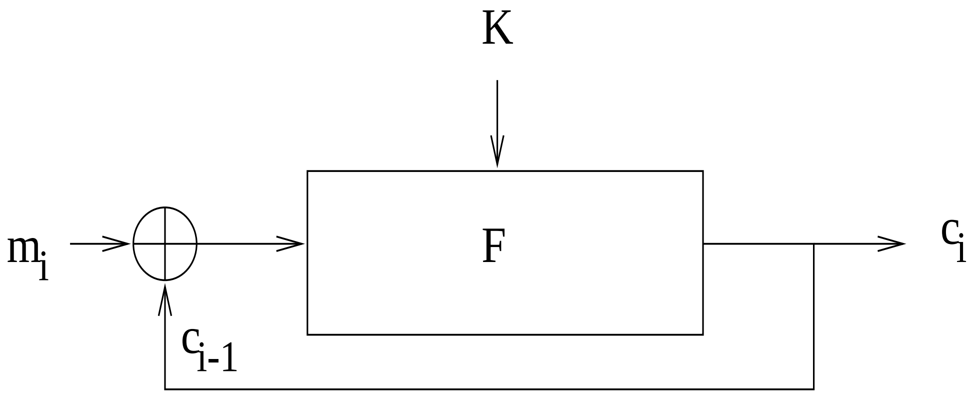


Figure 16: A block cipher in CBC mode

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 16 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).

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

4.8 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
}
```

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.

5 Public-Key Algorithms

Public-key algorithms are based on one-way trapdoor functions. As previously mentioned,

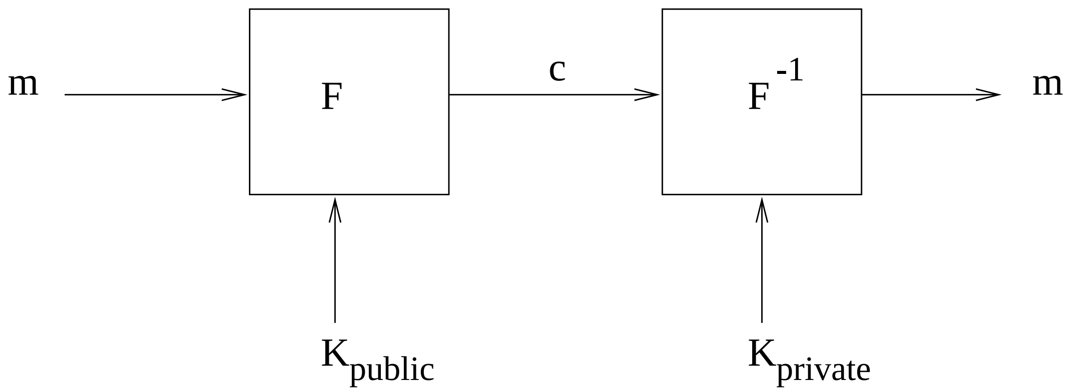


Figure 17: One-way trapdoor function

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 private-key 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 17.

5.1 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.

5.2 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). This means that if factoring is easy, then breaking RSA becomes easy. However, it has not been shown that RSA is equivalent to factoring (i.e. there might be another way to break RSA).

Here are the 5 steps of the RSA algorithms,

1. Pick two random large primes p and q (on the order of a few hundred digits). For maximum security, choose p and q to be of the same length.

2. Compute their product $n = pq$.

3. Randomly choose the encryption key e, such that e and $(p-1)(q-1)$ are relatively prime. Then, find the decryption key d, such that $ed \bmod (p-1)(q-1) = 1$. I.e., $ed = k(p-1)(q-1)+1$, for some integer k. The numbers e and n are the public key, and the numbers d and n are the private key. The original primes p and q are discarded.

4. To encrypt a message m: $c = m^e \pmod{n}$.

5. To decrypt a cipher c: $m = c^d \pmod{n}$.

Why does this work? Note that $m = c^d = (m^e)^d = m^{ed} = m^{k(p-1)(q-1)+1} = (m^{(p-1)(q-1)})^k m$ (all operations mod$n$). But m is relatively prime to n (because $n = pq$ and p, q are primes), and so m is in Z_n^*. By the theorem in Section 3.3, m to the power of the size of Z_n^* is 1, so $(m^{(p-1)(q-1)})^k m = 1^k m = m$. In practice, we pick small e to make m^e cheaper.

The performance figures of RSA (relative to block ciphers) are fairly unimpressive. Running on Sparc 2 with 512-bit n, 8-bit e, the encryption throughput is 0.03 secs/block, which translates into 16 Kbits/sec. The decryption throughput is even worse with 0.16 secs/block (4 Kbits/sec). Such speeds are too slow even for a modem. With special-purpose hardware (circa 1993), we can boost the throughput to 64 Kbits/sec which is fast enough for a modem, but is slow for pretty much everything else. In comparison, IDEA block cipher implemented in software can achieve 800 Kbits/sec. Because of its slowness, RSA is typically used to exchange secret keys for block ciphers that are then used to encode the message.

In the "real world", the RSA algorithm is used in X.509 (ITU standard), X9.44 (ANSI proposed standard), PEM, PGP, Entrust, and many other software packages.

5.3 Factoring

The security of RSA is based on the difficulty of factoring large numbers. The state of the art in factoring is as follows:

- Quadratic sieve (QS): $T(n) = e^{(1+O(1))(\ln(n))^{\frac{1}{2}}(\ln(\ln(n)))^{\frac{1}{2}}}$. This algorithm was used in 1994 to factor 129-digit (428-bit) number. It took 1600 Machines 8 months to complete this task.
- Number field sieve (NFS): $T(n) = e^{(1.923+O(1))(\ln(n))^{\frac{1}{3}}(\ln(\ln(n)))^{\frac{2}{3}}}$. This algorithm is about 4 times faster than QS on the 129-digit number and is better for numbers greater than 110 digits. It was used in 1995 to factor a 130 digit (431-bit) number with approximately the same amount of work as taken by QS on 129-digits.

  More recently (August 1999) it was be used to factor a 155 digit (512 bit) number. This is significant since 512 bits is the number of bits used by many implementations of RSA, therefore arguing these implementations are no longer secure. However, it took 35.7 CPU years (about 8000 MIPS years) to compute spread across about 200 machines.

5.4 ElGamal Algorithm

The ElGamal encryption scheme is based on the difficulty of calculating discrete logarithms. The following are the steps of ElGamal:

1. Pick a prime p.
2. Pick some g, such that g is a generator for Z_p.
3. Pick random x and calculate y, such that $y = g^x \pmod{p}$. Now, distribute p, g, and y as the public key. The private key will be x.
4. To encrypt, choose a random number k, such that k is relatively prime to $p-1$. Calculate $a = g^k \pmod{p}$ and $b = y^k m \pmod{p}$. The cipher is the pair (a, b).
5. To decrypt, calculate $m = \frac{b}{a^x} \pmod{p}$.

Why does ElGamal work? Because $m = \frac{b}{a^x} = \frac{y^k m}{g^{kx}} = \frac{g^{kx} m}{g^{kx}} = m \pmod{p}$. To break ElGamal, one would need to either figure out x from y or k from a. Both require calculating discrete logs which is difficult.

Performance-wise, ElGamal takes about twice the time of RSA since it calculates two powers. It is used by TRW (trying to avoid the RSA patent).

6 Probabilistic Encryption

One of the problems with RSA encryption is that there is a possibility of gaining some information from public key. For instance, one could use public key E_{public} to encrypt randomly generated messages to see if it is possible to get some information about the original message.

This problem can be remedied by mapping every message m to multiple c randomly. Now, even if cryptoanalist has c, m, and E_{public}, she cannot tell whether $c = E_k(m)$ (i.e. k was used to encrypt m).

One such probabilistic encryption scheme, Blum-Goldwasser, is based on generating strings of random bits.

6.1 Generation Random Bits

This method is called Blum-Blum-Shub (BBS) and is based on the difficulty of factoring: given a single bit, predicting the next bit is as hard as factoring. Here are the steps:

1. Pick primes p and q, such that $p \bmod 4 = q \bmod 4 = 3$.
2. Calculate $n = pq$ (called Blum integer).
3. For each m we want to encrypt, choose random seed x that is relatively prime to n ($gcd(x, n) = 1$).
4. The state of the algorithm is calculated as follows:
   - $x_0 = x^2 \pmod n$
   - $x_i = x_{i-1}^2 \pmod n$
5. The i^{th} bit is then calculated by taking the least-significant bit of x_i.

Note that now we have $x_i = x_0^{2^i \bmod (p-1)(q-1)} \pmod n$ and $x_0 = x_i^{-2^i \bmod (p-1)(q-1)} \pmod n$. In other words, given x_0 and th prime factors p and q of n it is not hard to generate any i^{th} bit and vice versa. We shall now see how these properties can be used to construct an encryption scheme.

6.2 Blum-Goldwasser

Blum-Goldwasser is a public key probabilistic encryption scheme using BBS. The public key is n, the private key is the factorization of n, i.e. p and q. The following are the steps to encrypting a message:

1. For each bit m_i of the message ($0 \leq i < l$) compute $c_i = b_i \otimes m_i$ where b_i is i^{th} bit of BBS (as in usual stream cipher).
2. Append x_l to the end of message, where l is the length of ciphertext in bits.

To decrypt, we use the private key p and q to generate x_0 from x_l ($x_0 = x_l^{-2^l \bmod (p-1)(q-1)}$) and then use the BBS formula to regenerate x_i, and hence b_i. The security of this scheme is based on difficulty of calculating x_0 from x_l without knowing the factorization of n.

Note that this encryption scheme achieves the desired probabilistic property: each time we code the same message with the same key we get different cipher (provided x is chosen randomly for each message). Furthermore the extra information we need to send (x_l) is minimal.

7 Quantum Cryptography

[This section has been expanded based on the material from `http://eve.physics.ox.ac.uk/NewWeb/Research/crypto.html`]

Quantum cryptography makes use of the fact that measuring quantum properties can change the state of a system, e.g. measuring the momentum of a particle spreads out its position. Such properties are called *non-commuting observables*. This implies that someone eavesdropping on a secret communication will destroy the message making it possible to devise a secure communication protocol.

Photon polarization is one such property that can be used in practice. A photon may be polarized in one of the four polarizations: 0, 45, 90, or 135 degrees. According to the laws of quantum mechanics, in any given measurement it is possible to distinguish only between rectilinear polarizations (0 and 90), or diagonal polarizations (45 and 135), but not both. Furthermore, if the photon is in one of the rectilinear polarizations and someone tries to measure its diagonal polarization the process of measurement will destroy the original state. After the measurement it will have equal likelihood of being polarized at 0 or 90 independent of its original polarization.

These properties lead to the following is a key exchange protocol based on two non-commuting observables (here, rectilinear and diagonal polarizations):

1. Alice sends Bob photon stream randomly polarized in one of 4 directions.

2. Bob measures photon polarization, choosing the kind of measurement (diagonal or rectilinear) at random. Keeping the results of the measurement private, Bob then tells Alice (via an open channel), the kinds of measurement that he used.

3. Alice tells Bob which of his measurements are correct (also via an open channel).

4. Bob and Alice keep only the correct values, translating them into 0's and 1's.

Although this scheme does not prevent eavesdropping, Alice and Bob will not be fooled, because any effort to tap the channel will be detected (Alice and Bob will end up with different keys) and the eavesdropper will not have the key either (since she is very unlikely to have made the same measurements as Bob).

While practical uses of quantum cryptography are yet to be seen, a working quantum key distribution system has been built in a laboratory at the IBM T. J. Watson research center.

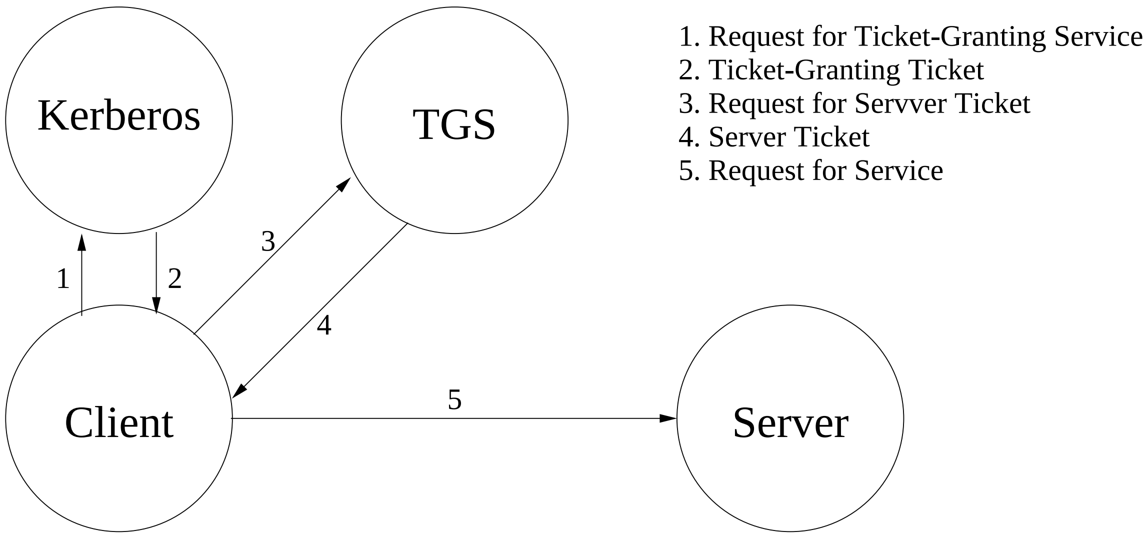


Figure 18: Kerberos authentication steps.

This prototype implementation can transmit over admittedly modest length of 30cm at rate it 10 bits/second. Nevertheless, as new photon sources, new photo-detectors and better optical fibers are being built, we shall see quantum cryptography becoming more practical.

8 Kerberos (A case study)

Kerberos is a trusted authentication protocol. A Kerberos service, sitting on the network, acts as a trusted arbitrator. Kerberos provides secure network authentication, allowing a person to access different machines on the network. Kerberos is based on private-key cryptography (DES).

8.1 How Kerberos Works

The basic Kerberos authentication steps are outlined in Figure 18. The client, a user or an independent software program, requests a *Ticket-Granting Ticket (TGT)* from Kerberos server located on a trusted and secure machine. Kerberos replies with this ticket, encrypting it with client's secret key. To use the requested service, the client requests a ticket for that server from *Ticket-Granting Sever (TGS)*. TGS replies to client with the server ticket, which the client uses along with authenticator to request services from the server.

8.2 Tickets and Authenticators

Kerberos uses two kinds of credentials: tickets and authenticators. The tickets have the following format (refer to Table 1 for notation):

$$T_{c,s} = s, \{c, a, v, K_{c,s}\} K_s \tag{1}$$

c	client
s	server
a	client's network address
v	beginning and ending validity time for ticket
t	timestamp
K_x	x's secret key
$K_{x,y}$	session key for x and y
$\{m\}K_x$	m encrypted in x's secret key
$T_{x,y}$	x's ticket to use y
$A_{x,y}$	authenticator from x to y

Table 1: Kerberos Notation

A ticket is good for a single server and a single client. It contains the client's name and network address c, a, the server's name s, a timestamp v, and a session key $K_{c,s}$. Once the client gets this ticket, he can use it multiple times to access the server – until the ticket expires.

The authenticator takes this form:

$$A_{c,s} = \{c, t, key\}K_{c,s} \tag{2}$$

The client generates an authenticator each time he wants to use service on a server. The authenticator contains the clients name c, a timestamp t, and an optional additional session key. An authenticator can only be used once, but the client can regenerate authenticators as needed.

8.3 Kerberos messages

There are five Kerberos messages:

1. Client to Kerberos: c, tgs. This message is sent whenever the client wants to talk to Ticket-Granting Server. Upon receipt of this message, Kerberos looks up client in its database and generates the session key to be used between the client and TGS.

2. Kerberos to client: $\{K_{c,tgs}\}K_c, \{T_{c,tgs}\}K_{tgs}$. There are two parts in this message, one containing session key to be used between client and TGS ($K_{c,tgs}$) encrypted with clients key (K_c), and the other containing TGT ($T_{c,tgs}$) encrypted with TGS's secret key (K_{tgs}). The client cannot decrypt this latter part, but uses it to talk to TGS. Note that Kerberos does not communicate with TGS directly and the tickets have limited lifetime.

3. Client to TGS: $\{A_{c,s}\}K_{c,tgs}\{T_{c,tgs}\}K_{tgs}$. This is the message sent by the client to TGS. The first part of this message contains information that proves client's identity to TGS. The second part is precisely the TGT Kerberos sent to client.

4. TGS to client: $\{K_{c,s}\}K_{c,tgs}\{T_{c,s}\}K_s$. This message is similar to Kerberos' response to clients initial request. It contains the session key to be used between client and the

server as well as the ticket for the client to use when requesting service. These also have limited lifetimes, depending on the kind of service requested.

5. Client to server: $\{A_{c,s}\}K_{c,s}\{T_{c,s}\}K_s$. With this message the client authenticates itself to the server as well as presents it with the ticket it received from TGS.