Chapter 2. Encryption and Pseudo-Randomness

In this chapter, we discuss the oldest, most basic cryptographic mechanism: encryption using shared key cryptosystems (ciphers). We describe few early (‘classical’) ciphers, and some attacks on them, introducing the major attack models, e.g. known plaintext. We then show the difficulty of defining secure encryption, present the indistinguishability test, a conservative criteria for secure encryption. We then discuss deterministic block ciphers, where encryption and decryption are stateless (functions), showing simple, secure but impractical implementation as random permutations. This motivates our discussion of Pseudo-Random Permutations (PRP), an important, useful cryptographic primitive, often implemented using practical deterministic block cipher modules. We describe practical deterministic block ciphers, particularly the DES, Triple-DES and AES standards, and their modes of operation for long messages and randomization. We show that the standard ‘modes of operation’ transform the deterministic block ciphers into a secure randomized block cipher, passing the indistinguishability test. We then discuss stream ciphers, where we allow encryption and decryption to use memory (state). We describe the unconditionally-secure one-time pad stream cipher, where each plaintext bit is XOR-ed with a corresponding bit of the key. We finally present Pseudo-Random Generators (PRG), which produce bit stream indistinguishable from random; when XOR-ed with the plaintext, it provides computationally-secure stream cipher.

Encryption is apparently the earliest cryptographic primitive. An encryption scheme, also called cryptosystem or a cipher, has two dual processes: encryption and decryption. The goal of encryption is to compute a special encoding, called ciphertext, of some secret information, called plaintext. The ciphertext is produced by applying the encryption process to the plaintext, and the plaintext is retrieved by applying the decryption process to the ciphertext. The ciphertext should ‘appear random’, or more precisely not expose anything about the plaintext, until properly decrypted.

In very early cryptosystems, security relied on secret decryption algorithm. However, following Kerckhoffs’ principle, we only consider cryptosystems where the adversary knows the encryption and decryption algorithms, and the intended recipients use a secret decryption key (or just secret key) to decipher the ciphertext back to plaintext. The encoding of the plaintext into ciphertext also uses a key, the encryption key.

The encryption key may be identical to the decryption key, in which case it is a shared secret key (or just shared or secret key) used for both encryption and decryption. We call
such cryptosystems ciphers or shared key cryptosystems and illustrate them in Figure 2.1. Alternatively, encryption may use a public encryption key (or just public key), from which it is infeasible for the adversary to compute the secret decryption key. We call these cryptosystems public key cryptosystems and illustrate them in Figure 2.2. In this chapter, we focus on shared key cryptosystems (ciphers); public key cryptosystems are discussed in Chapter ???, dedicated to public key cryptography.

We define a cipher by a pair of functions, \(<Encrypt, Decrypt>\). Both functions have two inputs, the key and the plaintext (for Encrypt) or ciphertext (for Decrypt). The encryption process may be randomized, namely Encrypt may have also random bits as a third input, in which case the same plaintext may be encrypted in multiple ways (to different ciphertexts). In principle decryption can also be randomized, but this is unusual, since we always require decryption to recover the plaintext. Namely, for every key \(k\), plaintext \(p\) and (optionally) random bits for encryption \(r_e\) and for decryption \(r_d\), we have:

\[
p = Decrypt_k(Encrypt_k(p, r_e), r_d)
\]

And since decryption usually does not use random bits, we can usually write:

\[
p = Decrypt_k(Encrypt_k(p, r))
\]

If encryption is also deterministic, we have a deterministic cipher:

\[
p = Decrypt_k(Encrypt_k(p))
\]

---

1 Some authors use the term 'symmetric cryptosystem'.
2 Some authors use the term 'asymmetric cryptosystem'.
Ciphers often use fixed-length plaintext and ciphertext, e.g. \( \{0,1\}^n \). The standard term for such fixed-length ciphers is \textit{block ciphers}. In contrast, \textit{stream ciphers} are ciphers with variable-length input and output. Stream ciphers usually process their plaintext and produce ciphertext incrementally, block by block, but maintain some internal state, modified during encryption/decryption of each block; blocks may be very short, e.g. one byte or even one bit.

2.1. Early Ciphers and Attack Models

We begin by discussing two simple, classical ciphers. These ciphers are very simple, but insecure; we present them just to explain the concepts and introduce common attack models.

2.1.1. Caesar Cipher and Exhaustive Key-Search Attack

We first describe the \textit{Caesar cipher}, a very simple cipher which was used by Julius Caesar. The Caesar cipher replaces each character of the plaintext message \( p_i \) with the \( k^{th} \) letter\(^3\) following it in the alphabet (cyclic, i.e. \( z \) is followed by \( a \)). Namely: \( c_i = p_i + k \). Figure 2.3 shows a possible physical implementation of a Caesar cipher, as two rings each containing the alphabet, with one of them rotating and the other fixed. The rotating wheel is turned only initially, to select \( k \). To encrypt a plaintext letter, it is identified on the outer ring, and the corresponding ciphertext letter is the one next to it in the internal ring. For simplicity, the figure uses only the letters A to L.

![Figure 2.3: Caesar Cipher](image)

The Caesar cipher is very easy to attack. In particular, the number of possible different keys is one less than the number of different characters. Therefore, an attacker, provided with some encrypted message, can easily decrypt it using each of the possible keys (e.g. 25 keys if using English alphabet); a sensible decryption will imply the correct key was found. This attack, in which we simply try out all possible keys, is called \textit{Exhaustive key search} (or \textit{brute force}). To prevent this attack, clearly cryptosystems must use very large key space. For example, we will see later that the DES standard uses keys which are 56 random

\(^3\) Actually, Caesar appears to have used this cipher with a fixed key \( k=3 \). The cipher we describe may therefore be called \textit{generalized} or \textit{keyed} Caesar cipher.
bits, namely there are $2^{56}$ different keys. Still, with the increase in computation costs, it is now feasible to exhaustively search all of these $2^{56}$ different keys, and new systems should use ciphers with longer keys as described later on.

2.1.2. Monoalphabetic Substitution Cipher and Statistical Attack

We next describe the Monoalphabetic Substitution cipher. This block cipher looks at the plaintext one character at a time, and uses a fixed permutation (substitution) from each plaintext character to a corresponding ciphertext character. For example, the cipher may use mapping of characters as in Figure 2.4 (for simplicity, we again use only A to L). Namely, $C_i = K[P_i]$, where $K$ is a fixed permutation, which is really the key. Decryption is simply by applying the inverse permutation, i.e. $P_i = K^{-1}[C_i]$. The key is defined by the permutation $K$. For a physical pair of disks as in Figure 2.4, different positions of the internal disk give different (shifted) keys.

![Figure 2.4: Monoalphabetic Substitution cipher](image)

Exhaustive search is not a viable attack against the monoalphabetic substitution cipher; there are simply too many keys; a key is simply a permutation on the characters, and with (e.g.) 26 characters, this gives 26! possible keys, or about $2^{80}$. However, this cipher is also very weak, as it is easy to break by simple cryptoanalytical techniques, i.e. techniques which try to break the cipher in shorter time than trying all possible keys. Different techniques can be used, depending on the attack model, i.e. the abilities of the adversary.

The weakest attack model is the statistical attack\(^4\). In this attack, the adversary is given the encryptions (ciphertexts) of one or more plaintexts. The only additional ability of the adversary is to select random plaintexts, from the same probability distribution.

In particular, when the plaintext is a typical message in a known natural language, e.g. English, then the attacker can use statistics for the relative frequency of occurrence of different letters. For examples, in typical text in English, about 13% of the letters is the most common letter, $E$. Therefore, when an attacker detects that about 13% of the ciphertext is the same letter, say $A$, she can deduce that $A$ probably substitutes $E$. Using this technique, and additional knowledge about the language, the adversary can find the entire key from quite limited ciphertext.

\(^4\) Also called Ciphertext only attack or Known plaintext distribution attack.
2.1.3. Alberti’s Polyalphabetic Cipher and Known Plaintext Attack

In Figure 2.3 and Figure 2.4 we illustrated the Caesar and Monoalphabetic Substitution ciphers as a pair of disks, but the position of the disks remained fixed during encryption and decryption and was considered a part of the key. Polyalphabetic ciphers also operate character by character, but the encryption process is frequently changed, e.g. each letter or once every few words. Therefore, polyalphabetic ciphers are stream ciphers.

There are many possible polyalphabetic ciphers, but we describe only two. The first and simplest polyalphabetic ciphers is Alberti’s cipher, described by Alberti in 1466 [K66]. Alberti suggested to use two disks as in Figure 2.4, but to rotate the internal disk once every few words. To allow decryption, the ciphertext will indicate the shift used, for example by writing the plaintext letter corresponding to the ciphertext letter \(k\) (using special font to identify this as a rotation indicator rather than regular ciphertext). For example, to begin encryption in the position in Figure 2.4, we use the special font to indicate the letter \(K\) (as by chance, this happens to be the plaintext corresponding to the ciphertext letter \(k\)). If we later rotate the inner ring by 2 cells clockwise, we use the special font to indicate the letter \(A\).

Alberti’s cipher corresponds to the formula \(C_i = K[P_i - R_i]\), where \(P_i\) is the \(i^{th}\) plaintext letter, \(C_i\) is the \(i^{th}\) ciphertext letter, and \(R_i\) is the number of clockwise rotations (in cells) of the inner ring from the initial position (e.g. where \(K\) in the inner rink is under \(K\) in the external ring).

Statistical attacks on polyalphabetic ciphers are possible, although more complex than on the monoalphabetic substitution cipher. In Alberti’s cipher, the difficulty is due to the use of different positions \(R_i\) of the inner ring. However, let us show very simple attacks on Alberti’s cipher which are possible when the attacker has additional abilities, i.e. when using attack models stronger than the statistical attack model.

In particular, Alberti’s cipher is easily broken with a known plaintext attack, where the attacker knows the encryption of some random plaintext messages. Here the adversary has a random plaintext message and its encryption (ciphertext). Recall that \(C_i = K[P_i - R_i]\), where \(R_i\) is known to the attacker (as it is sent in the clear, i.e. without encryption). Therefore, each pair of plaintext letter and the corresponding ciphertext letter gives one entry in the \(K/F\) transposition table. After few entries, the attacker can reconstruct the entire table.

2.1.4. Vigenère’s Polyalphabetic Cipher and Chosen Plaintext Attack

To improve security, more advanced polyalphabetic ciphers try to hide the amount of rotation used for encrypting each letter. A typical example is the repeating keyword cipher, attributed to (and often named after) Vigenère, in the late 16th century [K66]. This cipher uses a secret transposition, e.g. using two rings as in Figure 2.4, but also a secret keyword (or phrase). The \(i^{th}\) plaintext letter is encrypted using the \(i^{th}\) letter of the keyword (repeated). If \(K[f]\) denotes the transposition (e.g. of Figure 2.4), \(l\) denotes the length of the keyword and \(keyword\) denotes the keyword itself, then the repeating keyword cipher is defined by
\[ C_i = K[P_i \cdot \text{keyword}(i \mod l)] \]

Equation 2.1

Again, it is feasible to mount statistical and known plaintext attacks against the repeating keyword cipher, but these will be more complex than the attacks described previously (against the simpler ciphers) and we will not describe them. Instead, we show that this cipher is easily broken with a chosen ciphertext attack, where the attacker is able to see the encryption (ciphertext) of a message (plaintext) of her choice.

In particular, assume that the attacker is able to receive the encryption of the plaintext ‘AAAA…’, i.e. a long stream consisting of the single letter A. Substituting this in Equation 2.1 we get: \[ C_i = K[A \cdot \text{keyword}(i \mod l)] \], which repeats every \( l \) characters. Clearly, the attacker easily learns the period \( l \) (assuming the message is longer than 2l). Furthermore, the attacker learns the encryption of the letter A when placed in arbitrary position in the plaintext. The attacker can repeat this attack with plaintext messages of length \( l \) for each of the letters in the alphabet, to find the encryption of every letter in every position. With slightly more refined analysis, the attacker can find the keyword and table \( K / \) with much fewer chosen plaintexts; readers may find this as an exercise.

2.1.5. Summary of Attack Models

We now summarize the attack models described in the previous sections. Following Kerckhoff’s principle, we always assume that the attacker knows the algorithms in use. We also always assume that the adversary knows the exact probability distribution of the inputs (plaintext). On the other hand, in this book (and most of applied cryptography) we assume that the computational resources of the attacker are limited (time and storage). Attack models define additional capabilities of the attacker. The most common attack models include:

- **Statistical attack** (also called ciphertext-only): Adversary knows the encryptions (ciphertexts) corresponding to one or more plaintext messages, selected by a known, efficient random process. For example, when encrypting text in English or other natural language, the attacker may have statistics on the probability of different letters and their combinations. We compare different attacks based on the number of ciphertexts required (the less the better). For stream ciphers, we prefer attacks which require minimal number of plaintext blocks.

- **Known plaintext attack**: Same as above, but the adversary also knows some of the randomly chosen plaintext values. Again, we compare attacks based on the number of plaintext-ciphertext pairs required, and their length (for stream ciphers); the less the better.

- **Chosen plaintext attack**: Adversary chooses some plaintext messages, and receives their encryptions. Security against chosen plaintext implies security against known plaintext. Also, notice that when the attacker has the encryption key, as we normally assume for public key cryptosystems, then she can easily compute encryption (ciphertext) for chosen plaintext.
We compare attacks based on the number of chosen and known plaintext-ciphertext blocks required (the less the better).

- **Chosen ciphertext attack**: Adversary chooses some strings, which are decrypted (i.e., the adversary’s strings are considered ciphertext). If decryption is successful, the adversary receives the decrypted plaintext; otherwise the adversary is informed that the string was not valid ciphertext. Intuitively, for an attack in this model to be successful, the attacker must then decrypt a different ciphertext block (or message).

### 2.2. Defining security for ciphers and the indistinguishability test

Defining security for encryption schemes, and in particular for ciphers, is surprisingly tricky. An intuitive attempt at defining secure encryption may be ‘without access to the secret decryption key, it is infeasible for an attacker to learn anything meaningful about the plaintext, given the ciphertext’. This fuzzy definition leaves several issues vague, such as the attack model discussed above. A more difficult problem is the definition of what constitutes ‘meaningful information’ that the attacker should not learn.

The definition of ‘meaningful information’ is difficult, since the meaning and significance of information varies among different applications. For example, in some applications, we do not care if the adversary learns the first few bits of the message, but in others these bits may be critical. In yet another application, we do not care about any particular bit, but we may care that the number of bits which are 1 will be secret. Significance and sensitivity depend on the specific application. Since we are unaware of the specific application and its requirements, we prefer to use a conservative definition that will hide all information. The indistinguishability test, which we illustrate in Figure 2.5, provides such a conservative definition. The notation $r \in_R R$ denotes that $r$ is chosen uniformly from the finite set $R$. 
Definition 1  Define the indistinguishability test for cipher $<\text{Encrypt}_k, \text{Decrypt}_k>$ and an arbitrary attack model $\text{AM}$, as follows. First, the attacker attacks $<\text{Encrypt}_k, \text{Decrypt}_k>$ under $\text{AM}$. Then, the attacker chooses two different plaintext blocks, $x[0]$ and $x[1]$. The attacker is given $\text{Encrypt}_k(x[b], r)$, the encryption of $x[b]$, where $b \in \mathbb{R} \{0,1\}$ and $r \in \mathbb{R} \{0,1\}^n$, and outputs $b'$. The attacker is successful if $b' = b$. We say that the cipher $<\text{Encrypt}_k, \text{Decrypt}_k>$ securely ensures indistinguishability under attack model $\text{AM}$, if the success probability is bounded by half, plus a negligible value (a function which diminishes faster than any polynomial in the length of the key).

Notice that the definition of the indistinguishability test allows for randomized and deterministic ciphers, and for block ciphers as well as stream ciphers. However, with a deterministic cipher, $\text{Encrypt}(x, r) = \text{Encrypt}(x)$ (no randomness) and therefore it is easy for the adversary to distinguish between any $x[0], x[1]$ whose encryptions were revealed. Therefore, we sometimes consider indistinguishability for unused plaintexts, where the adversary has to select $x[0], x[1]$ whose encryptions were not exposed yet.

Prudent cryptographic design attempts to design encryption schemes which securely ensure indistinguishability under strong attack models, while designing the protocols and system to limit the attacker to weaker attack models, e.g. prevent chosen plaintext or ciphertext attacks. For example, by checking for incorrectly formatted decryptions, or (better) encrypting messages with an integrity code (e.g. CRC) and checking it after decryption, we can detect attempts to provide chosen ciphertext and raise alert, making this attack unattractive.
2.3. Deterministic block cipher

A deterministic block cipher is a pair of functions, \(\text{Encrypt}: \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n\), \(\text{Decrypt}: \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n\), such that for every plaintext message \(p\) and key \(k\) holds:

\[
p = \text{Decrypt}_k(\text{Encrypt}_k(p))
\]

Equation 2.2

For simplicity, we assumed above that the plaintext and ciphertext sets are both the set of all binary strings of some fixed length \(n\). In this case Encrypt and Decrypt, for any given key, are permutations over this set.

All deterministic block ciphers are vulnerable even under the weak statistical attack model, for scenarios where the probability distribution of the plaintext is significantly biased, or where the number of possible plaintexts is limited. Therefore, we assume that the length \(n\) of the plaintext is sufficiently large, to prevent the attacker from seeing a significant fraction of all possible ciphertexts or identifying repetitions. For example, when the block length \(n\) is very small, e.g. \(n=5\), the block cipher is essentially a monoalphabetic substitution cipher, which is very weak; see Section 2.1.2. Therefore, we must use longer blocks, to prevent the attacker from using statistics or collections of plaintext-ciphertext pairs. In practice, most systems, e.g. the Data Encryption Standard [DES], have block length of at least 64 bits.

2.3.1. Simple substitution cipher (random permutation)

Denote by \(N\) the set of all permutations of binary strings of length \(n\). The simple substitution cipher of length \(n\) is defined by picking as key \(k\) a random element (permutation) in \(N\). Encryption is simply application of the key (permutation) \(k\) to the plaintext, and decryption is simply the application of the inverse permutation. Clearly:

\[
p = k^{-1}(k(p))
\]

Equation 2.3

A simple substitution cipher is unconditionally secure as long as plaintext is if used to encrypt a single plaintext block (message of length \(n\)). If the block length is sufficient (as we assumed), and the plaintext distribution is uniform, then it is easy to see that the simple substitution cipher securely ensures indistinguishability for unused plaintext, as is unconditionally secure even for repeated encryptions.

Claim 1 A simple substitution cipher securely ensures indistinguishability for unused plaintext, under chosen plaintext and chosen ciphertext attacks.

Argument: Let \(K \subseteq N\) be the set of all permutations which agree with all the queries made by the adversary. Namely, if the adversary gave plaintext \(p\) and received ciphertext \(c\), or vice verse, then for all \(k \in K\) holds \(c=k(p)\). Clearly, each of the permutations \(k \in K\) is equally likely. Let \(x[0], x[1]\) be the plaintext submitted by the attacker, and let \(y\) be the ciphertext corresponding to encryption of one of them. Let \(K_0\) be the subset of \(K\) where \(y=k(x[0])\), and \(K_1\) be the subset of \(K\) where \(y=k(x[1])\). Clearly for every \(k_0 \in K_0\) there is a
corresponding $k_1 \in K_1$ which is identical except for the pre-image of $y$ ($y[0]$ or $y[1]$). The sets are therefore the same size and the probability is therefore exactly $\frac{1}{2}$. ■

The disadvantage of the simple substitution cipher is its key length, which is the index of the randomly chosen permutation in $\mathbb{N}$. The number of keys (permutations) is $2^n!$, which grows very fast (in $n$); even the minimal key length, $\log_2(2^n!)$, grows very fast. Even for very short (and therefore insecure) block length e.g. $l=5$, the number of keys is huge, $2^5! = 32! \approx 2.63 \times 10^{35}$, and requires keys of roughly 35 digits or 118 bits.

Therefore, we rarely use simple substitution ciphers in practice. Instead, ‘practical’ substitution ciphers often attempt to be a pseudo-random permutation, i.e. indistinguishable from a random permutation (i.e., simple substitution cipher) for computationally bounded adversary.

2.3.2. Pseudo-random permutations and functions

Practical block ciphers are good candidates for a very powerful cryptographic functions, namely pseudo-random permutations.

**Definition 2**  A pseudo-random permutation is a collection of pairs of a permutation and its inverse: $\{<E_k,D_k>\}$, with domain and range $\{0,1\}^n$ such that:

- **Efficient evaluation and inversion:** For a given key $^5 k$, the pseudo-random permutation can be computed efficiently, i.e. given any message $m$ it is easy to compute both $E_k(m)$ and $D_k(m)$.
- **Pseudo-randomness:** an adversary cannot efficiently distinguish between $E_k$ and a random permutation on $\{0,1\}^n$, for randomly chosen key $k$.

Namely, if $k$ is unknown to the adversary, then pseudo-random permutations are indistinguishable from random permutations, even if the attacker has ‘black box’ access to $E_k$ (chosen plaintext attack). Hence,

**Claim 2**  A pseudo-random permutation securely ensures indistinguishability under chosen plaintext attack for unused plaintext, for uniformly distributed plaintext. ♦

A super pseudo-random permutation is a pseudo-random permutation which remains indistinguishable from a random permutation, even when the attacker is allowed ‘black box’ access to $D_k$ as well as to $E_k$.

**Claim 3**  A super pseudo-random permutation securely ensures indistinguishability under chosen ciphertext attack for unused plaintext, for uniformly distributed plaintext. ♦

Both claims follow from Claim 1, since the adversary cannot distinguish when we use a random permutation; any advantage therefore indicates the use of the pseudo-random permutation, allowing us to identify it.

---

5The index $k$ is the key (or the seed) of the permutation (more precisely, of the pair of permutation and its inverse).
We believe that many designers of well-known practical deterministic block ciphers tried, in fact, to make them (super) pseudo-random permutations. Similarly, we believe that cryptanalysts considered any proof that a specific block cipher is not a (super) pseudo-random permutation, as success in breaking the block cipher. Therefore, practical deterministic block ciphers are usually good candidates to use when the design requires a (super) pseudo-random permutation. And in fact, as we show below, pseudo-random permutations are the basis for simple secure (randomized) ciphers.

Pseudo-random permutations have several applications in addition to their use for ciphers. In particular, a very important use for pseudo-random permutations is as a mechanism to derive multiple secret values (e.g. keys) \( v_1, v_2, \ldots \) from a single secret value \( v \). The derived values are the result of a pseudo-random permutation with key \( v \) computed over different values, e.g. \( v_i = f(v_i) \). Whenever you need secret, pseudo-random values (e.g. shared keys) for more than one purpose, e.g. for encryption and for authentication, one can use the above mechanism (or similar) to derive independent keys from a single shared secret value.

**Claim 4** Assume a system is secure when some parties share secret values \( v_1, v_2, \ldots \). Then this system is secure, against computationally bounded adversary, also if the parties share only one secret value \( v \) and use \( v_i = f_i(v) \), where \( f \) is a pseudo-random permutation.

Since pseudo-random permutation are such an important cryptographic primitive, we may have several candidates. For improved security, we may want to compose several candidate pseudo-random permutations, hoping to maintain security even if some of them turns out to be not very secure (i.e., distinguishable from random permutation). This is a special case of the ‘multiple lines of defense’ principle. We now show one simple way to compose two candidate pseudo-random permutations, by cascade execution.

**Claim 5** Let \( h_{k,k'}(x) = f_k(g_{k'}(x)) \). Then if either \( f \) or \( g \) is a PRP, then \( h \) is a PRP.

Proof sketch: Suppose \( Adv \) can distinguish \( h \) from random permutation. Then to distinguish \( g \), select \( k \) and use \( Adv \) on \( f_k(g_k(x)) \). Similarly for \( f \). □

This construction depends on the use of different, independent keys for \( f \) and for \( g \). If we allow the same key to be used, the construction would not be secure; for example, if \( f \) is the inverse of \( g \), then \( h_{k,k} \) will be the identity function.

We conclude that in order to increase the security of pseudo-random permutations (i.e., deterministic block ciphers), we can cascade multiple candidate functions, using different (preferably independent) keys. We now discuss the design of practical block ciphers, which often use this technique.

### 2.3.3. Practical Deterministic Block Ciphers

In the previous sections we found that deterministic block ciphers do not protect confidentiality when plaintext blocks may repeat (as in many applications). However,
practical deterministic block ciphers are good candidate (super) pseudo-random permutations, which in turn are useful for many cryptographic constructions and in particular secure (randomized) ciphers. Deterministic block ciphers are widely implemented and deployed in practice, and available in most cryptographic software packages. We now briefly describe popular designs, in particular, the Data Encryption Standard [DES], Triple-DES (DES with longer keys), and the Advanced Encryption Standard [AES].

Most practical block ciphers use an iterated, multiple-round design, composed by cascade as motivated in Claim 5. Such designs involve the sequential application of an internal round function, plus possibly some special additional processing of the input to the first round and the output from the last round. Round functions should be efficient, simple permutations, which are as close to pseudo-random permutations as possible; they should also be reversible (for decryption). The round function operates on the output from previous round, and a sub-key – a function of the key to the cipher and of the round number. Figure 2.6 illustrates this structure.

![Figure 2.6: Iterative block cipher](image)

One motivation for this design is from Claim 5 above, showing that cascading candidate pseudo-random permutations can only increase security. With this motivation, the round function is a candidate pseudo-random permutation, and by cascading it, we hope to obtain a secure pseudo-random permutation. An advantage of such iterative designs is the we can cryptoanalyze a weak version, say with four rounds, and use a stronger version, say with 16 rounds. The sub-keys used in the cascading should be different and preferably independent, as required by Claim 5. We can achieve this by using again a pseudo-random permutation, to generate the sub-keys, for example $SubKey_i = PRP_k(i)$. 
In many applications, there may value in using an intentionally complex and slow sub-key generator. A slow sub-key generator makes it harder to use exhaustive search attacks; in normal use, sub-keys will be computed relatively rarely (and then stored). Of course, slow sub-key generation slows down applications where keys change rapidly, and should only be used when key set-up overhead is not significant.

2.3.3.1. Feistel Cipher and DES

Round functions, following the iterative design, should be simple, efficient and reversible permutations, which are as close to pseudo-random permutations as possible. These goals are somewhat conflicting. In particular, it seems much easier to design candidate pseudo-random functions, which are not reversible or permutations. A common approach is to use a round function design that takes an arbitrary (candidate pseudo-random) function \( f \) and defines an invertible (candidate pseudo-random) permutation based on it, to be used as round function.

A particularly common design for a round function is the Feistel cipher (see Figure 2.7). In the Feistel cipher \([F73]\), the input to each round, say round \( t \), is broken into two parts, \( L_t \) and \( R_t \). The outputs are defined as \( L_{t+1} = R_t \) and \( R_{t+1} = L_t + f(R_t, K_t) \), where \( f \) is the non-invertible candidate pseudo-random function and \( K_t \) is the sub-key of round \( t \). If \( f \) is indeed a pseudo-random function, and the sub-keys are independent, then the result of three rounds of Feistel construction is a pseudo-random permutation. For details, see \([LR88]\).

![Figure 2.7: Feistel Round Function Design](image)

DES is a 16-round Feistel cipher whose input and output block sizes are 64 bits, and a key of 56 bits (or 64 bits out of which 8 bits are used for parity check). DES was designed in the late 1970’s, by a USA government standards body, NIST, and was often criticized as potentially containing an intentional weakness (e.g. the use of only 56 bit key, or an hidden ‘trapdoor’ allowing cryptoanalysis). One reason for this criticism was the fact that the design considerations were not published. In spite of these criticisms, many systems use DES, and in particular there are several hardware implementations. Indeed, probably since
DES was designed in the 1970’s, it is optimized for hardware implementations, with many ‘bit operations’ slowing down software implementations.

Substantial cryptoanalysis effort on DES revealed no trapdoors or substantial shortcut weaknesses. Indeed, DES appears relatively well protected even against differential and linear cryptoanalysis methods, published only in the 90’s; some of DES designers claimed that they were aware of these attacks (maybe motivating the decision not to publish the design considerations). However, the limited key size (56 bits) eventually made DES vulnerable to exhaustive, ‘brute-force’ key guessing attacks, which are feasible using computing resources which are widely available currently. Clearly, stronger ciphers are required.

2.3.3.2. Triple DES

One approach to deal with the exhaustive search attack against DES, is to use a mode of DES with longer key. A natural idea is to use two ‘regular’ DES keys, i.e. 112 bits instead of 56 bits. Using a key of 112 bits will definitely rule out the exhaustive search attack for the foreseeable future.

One may be tempted to use a construction such as ‘Double DES’, namely:
\[ c=DES^{2}_{k,k'}(m)=E_{k}(E_{k'}(m)), \]
where \( E \) is a single DES encryption. However, this construction, while using 112 bits of key, can be broken with only \( O(2^{56}) \) amount of computations and storage. The attack, called ‘meet in the middle’, proceeds as follows:

1. For \( x'=0^{56} \) to \( 1^{56} \): \( y[x']=E_{x}(m) \]
2. For \( x=0^{56} \) to \( 1^{56} \): \( z[x]=D_{x}(c) \]
3. Find all \( \langle x,x' \rangle \) s.t. \( y[x']=z[x] \)
   a. These are candidate keys \( (k=x, k'=x') \)
   b. At most \( 2^{56} \) such pairs (usually less)
   c. Test with another plain-ciphertext pair

From this attack we conclude that Double DES is vulnerable to an attack requiring \( O(2^{56}) \) amount of computations and storage; we say that it has effective key length of 56 bits. Since our goal is an effective key length of about 112 bits, we need a different design.

The most common design for extending DES key size is called triple-DES, also defined in [DES]. Triple-DES involves three applications of DES with 112 bit key (two 56 bit keys), or with 168 bit key (three 56 bit keys). Triple-DES with 112 bit key is defined as:
\[ DES^{3}_{k,k}(m)=E_{k}(D_{k'}(E_{k}(m))). \]

For compatibility with ‘regular’ DES, the triple-DES design uses DES encryption for the first and third application and DES decryption for the second; this makes triple-DES with two identical keys equivalent to DES with the same key, i.e. \( DES^{3}_{k,k}(m)=E_{k}(m). \)
With triple-DES, the key length is 112 bits, which seems sufficient to protect against exhaustive key search. This, together with the wide deployment of DES and Triple DES, their standardization and the huge effort of cryptoanalysis which so far failed to find substantial shortcut attacks, motivate the use of triple DES as block cipher (and pseudo-random permutation). However, notice that Triple-DES still uses block size of only 64 bits; we will see later applications of pseudo-random permutations, which require longer block size of e.g. 128 bits. This is one motivation for considering alternative block ciphers (and PRPs); another important motivation is performance. We mentioned above the DES is not very efficient in software; triple-DES is roughly three times slower. This motivates the use of faster, and possibly more secure, deterministic block ciphers (or PRPs).

### 2.3.3.3. AES and Rijndael

The recommended encryption scheme for new systems is the Advanced Encryption Standard [AES], defined recently by NIST, which is more secure as well as more efficient than DES (and certainly more efficient than Triple-DES). NIST selected AES after a lengthy public evaluation and selection process; the original proposal name was Rijndael [DR01]. The goals of AES are to improve security and efficiency compared to DES and Triple-DES.

AES has a variable block length and key length. The specifications cover keys with a length of 128, 192, or 256 bits. The block size is 128 bits, which is important for some applications (the original Rijndael proposal supported blocks with length of 128, 192 or 256 bits). AES allows efficient implementations in both software and hardware.

### 2.3.4. Modes of Operation

Deterministic block ciphers have two significant limitations:
- Being completely deterministic, the encryption of fixed blocks of plaintext results in corresponding fixed blocks of ciphertext. The adversary may exploit this, e.g. to identify frequently repeating blocks of plaintext.
- They operate over fixed-size, n-bit plaintext blocks.

We deal with these limitations by using *modes of operation* defining the use of the ciphers for longer messages and/or introducing randomness. The most commonly used modes of operations, defined in [DES], for producing the sequence of ciphertext blocks $c[1], c[2], \ldots$ from the plaintext blocks $p[1], p[2], \ldots$ are as follows. We use the symbol $\oplus$ to denote the bitwise exclusive OR operation.

- **Electronic code book (ECB) mode**: encrypt each plaintext block separately, i.e. for every $j$ we have $c[j]=E_k(p[j])$. This mode is trivial – in fact, it is the same as the block cipher, and does not add any solve either problems above. It is called a ‘mode of operation’ just so that every way of using the cipher can be called ‘mode of operation’.
- **Cipher Block Chaining (CBC) mode**: use an additional input of n-bits, called *Initialization Vector* (IV), and let $c[0]=\text{IV}$. The ciphertext is produced by computing $c[j]=E_k(c[j-1]\oplus p[j])$. The IV should be changed with every message (or at least frequently). See Figure 2.8.
• **Cipher Feedback (CFB) mode**: this mode also uses an initialization vector (IV). Let \( c[0] = IV \). The ciphertext is produced by computing \( c[j] = E_k(c[j-1]) \oplus p[j] \).

• **Output Feedback (OFB) mode**: this mode also uses an initialization vector (IV). It also uses an auxiliary sequence of blocks \( o[0], o[1], o[2], \ldots \), initializing the first block \( o[0] = IV \). The ciphertext is produced by computing \( o[j] = E_k(o[j-1]) \) and \( c[j] = o[j] \oplus p[j] \). See Figure 2.9. Notice that the auxiliary sequence may be computed in advance, before the message is received (‘offline’), and then encryption and decryption only involve exclusive OR operation.

![Figure 2.8: Cipher Block Chaining (CBC) mode](image)

![Figure 2.9: Output Feedback (OFB) mode](image)

### 2.4. Randomized Block Ciphers

Deterministic block ciphers can only be secure for specific distributions of the plaintext. In many cases, the designer cannot know the probability distribution of the plaintext in advance, or the probability distribution is such that a deterministic block cipher is not secure. For example, if there are only two plaintext messages, say “buy” and “sell”, then the attacker can quickly learn to identify the deterministic encryption and identify the plaintext. It is desirable to ensure security regardless of the probability distribution of the plaintext, and in particular maintain secrecy even if there are only two plaintext messages.
We can achieve this by adding randomization as part of the encryption process, using a *randomized cryptosystem*. In this section, we discuss *randomized block ciphers*, and in the following section, we explain how to apply randomization to stream ciphers.

Randomized block ciphers have three inputs: the plaintext, the key, and uniformly distributed random binary string (interpreted as coin tosses). The random input is combined with the plaintext during the encryption process, resulting in ciphertext that is longer than the plaintext; this overhead is acceptable in most applications. By combining the plaintext with the random inputs, randomized block ciphers attempt to ensure indistinguishability (secrecy), regardless of the plaintext distribution.

The ‘modes of operation’ defined in [DES] provide simple and secure constructions for randomized block ciphers, when used with random Initial Vector (IV). In particular, given a deterministic block cipher \(<E_k, D_k>\) with block length \(n\), to encrypt a message \(x \in \{0,1\}^n\) with key \(k\) we randomly select a random string of the same block length, \(r \in R\{0,1\}^n\), and construct the ciphertext as the pair:

- Using the Cipher Block Chaining (CBC) Mode: \(<r, E_k(x \oplus r)>\)
- Using the Output Feedback (OFB) Mode: \(<r, x \oplus E_k(r)>\)

The symbol \(\oplus\) denotes bit-wise exclusive-or operation. Decryption simply reverses this process (using \(D_k\) for CBC, or again \(E_k\) for OFB).

**Claim 6** If \(<E_k, D_k>\) is a pseudo-random permutation, then the CBC construction \(<r, E_k(x \oplus r)>\) ensures indistinguishability under chosen plaintext attack.

**Claim 7** If \(E_k\) is a pseudo-random permutation (or function), then the OFB construction \(<r, x \oplus E_k(r)>\) ensures indistinguishability under chosen plaintext attack.

**Proof sketch (both claims):** The input to \(E\) is always random. If \(E\) was a random permutation, claim follows from Claim 1. Therefore claim must follow, otherwise use adversary to distinguish \(E\) from random permutation. ■

However, these constructions (and modes) are insecure against chosen ciphertext attack. For example, consider the OFB construction. Suppose an attacker has ciphertext \(<r, x \oplus E_k(r)>\). The attacker flips the least significant bit, computing \(<r, x \oplus E_k(r) \oplus 1>\), and asks for the decryption of this. Let \(x'\) denote the decryption; clearly \(x = x' \oplus 1\). It is easy to find similar attack for CBC (Exercise 7). Since these modes are widely used in practice, we conclude that it is very important to design protocols and applications so that chosen ciphertext attacks will not be feasible.

### 2.5. Stream Ciphers and Pseudo-Random Generators

Randomized ciphers avoid assumptions about the probability distribution of their inputs, by ‘randomizing’ their inputs. However, in the constructions above, the block size must remain large to make it unlikely that the same randomness and plaintext pair will be repeated.
We could consider the randomization bits as part of the secret key, which allows us to use very short blocks. Taking this to the most extreme, we can encrypt the plaintext bit by bit, by exclusive-or with a secret, uniformly-random bit sequence serving as a key. Each bit of the key must be used with only one bit of the plaintext, and new key bits must be used for every encryption; hence we often refer to this scheme, illustrated in Figure 2.10, as one time pad.

![Figure 2.10: One Time Pad](image)

Shannon [S49] showed that the one time pad ensures unconditional security (secrecy). Namely, for any probability distribution of the plaintext (known to the attacker), the probability of any plaintext bit to be ‘1’ does not change by knowing that the corresponding ciphertext bit is ‘1’. Therefore, even an adversary with unlimited computational abilities cannot gain new information about the plaintext from the ciphertext.

Unfortunately, one time pad requires the parties to share a secret key of the same length as the plaintext. For most practical applications, this is unfeasible. One solution is to use a pseudo-random bit sequence as key, i.e. a bit sequence that is indistinguishable from uniformly distributed random sequence, for a computationally limited adversary. Notice, that this would not ensure security against a adversary with unlimited computational resources.

A pseudo-random generator is an algorithm PRG with input seed ∈ {0,1}^n and whose output is an infinite bit sequence, which is indistinguishable from a uniformly-distributed random bit sequence (and hence called pseudo-random bit sequence). We can implement a pseudo-random generator PRG from a pseudo-random permutation PRP by concatenating its outputs on the sequence of integers, i.e. PRP(seed)=PRG_{seed[1]}||PRG_{seed[2]}||…; the number of inputs is actually limited by the length of the block of the PRG, but we assume that we can use anyway only a polynomial number of bits.

We use the term pseudo-random generator based stream cipher for such cryptosystems, i.e. one-time-pad like cryptosystems with a pseudorandom sequence instead of a truly random key (see Figure 2.11). The Output Feedback (OFB) mode of operation presented in Section 2.3.4 above is an example of a pseudo-random generator based stream cipher (and of one possible construction of a pseudo-random generator, using a block cipher).
The security of a pseudo-random generator based stream cipher depends on the indistinguishability of the pseudo-random sequence from truly random bit sequence. Therefore, such ciphers are only computationally secure, but not secure against an computationally unlimited adversary. We caution the reader that quite often, intentionally or not, this point is blurred when vendors present pseudo-random generator based stream ciphers.

In general, a stream cipher is a cryptosystem that operates on unbounded length plaintext, and produces the ciphertext block-by-block (often, with one bit blocks, hence bit-by-bit). Each ciphertext block is a function of the corresponding plaintext block as well as of a state variable, as illustrated in Figure 2.12. The key is used as the initial state, and as an input to the block function.
Stream ciphers are mostly used, in practice, where very high speed encryption is needed; many of the designs use substantial pre-processing to enable very high speed ‘real time’ encryption. Many stream ciphers are designed for highly efficient implementations in simple, inexpensive hardware. Unfortunately, there is no widely adopted or standardized stream cipher, comparable to the DES and AES block cipher standards. Most ‘modes of operation’ of block ciphers, e.g. CBC and OFB, are actually stream ciphers constructions based on the block cipher. The modes of operations repeatedly apply the block cipher to plaintext blocks, combined with some state from the previous block. Of course, these constructions cannot be more efficient or simpler than the block cipher.

2.6. Extracting Randomness (to be completed)

Randomness is required for many cryptographic mechanisms, e.g. to pick random keys, or for randomizing the encryption process. Most of these applications require a uniformly distributed random string. However, computer systems are quite deterministic by design, and it is difficult to find a source of truly random bits, certainly with uniform distribution. In particular, the ‘random’ function calls, available in most programming languages, do not provide truly random bits.

The two main techniques to collect somewhat random bits are:

- Physical randomness: to sample some source of physical randomness. This usually requires the addition of a special hardware device to sample random bits, based on some random physical process. Such devices are available, however not deployed in standard computers.
- Sampling of highly unpredictable inputs: this is a software only solution. We often use the unpredictability of the timing of human actions, such as keystrokes or mouse movements (mouse movements also have unpredictability of exact locations). Other randomized inputs that software can sample are memory contents, disk delays, log files, etc.. Unfortunately, these sources of randomness are not always available and trustworthy, and even when available, produce limited amount of randomness.

Unfortunately, none of these sources is completely random and unpredictable, and certainly not uniform. Therefore, we collect such randomized inputs from one or more sources, and then apply some process to ‘improve randomness’. One way to ‘improve randomness’, discussed above, is to use a short random seed as input to a pseudo-random generator; the resulting sequence is pseudo-random, which is as good as random against computationally limited adversaries.

A more difficult problem is how to generate the random seed itself. The problem is that the available sources of randomness are not uniformly distributed, and may not be independent of each other. There have been considerable amount of theoretical works on extracting randomness from weakly independent random sources. We illustrate this in a very simple case, where we can sample independent random bits \(\{b_i\}\) from some unknown distribution. We can use such a source to generate a truly random bit sequence as follows.

For every \(i\), let \(c_i = \{0 \text{ if } b_i=0 \& b_{i+1}=1; 1 \text{ if } b_i=1 \& b_{i+1}=0; \text{otherwise null}\}\). It is easy to see that the sequence \(\{c_i\}\) is a truly random bit sequence.
2.7. Encryption and Compression

Messages often contain a large amount of redundancy, namely their probability distribution is far from uniform. Many communication systems use compression to remove some of that redundancy and reduce the size of the communicated message.

A good encryption algorithm produces output (ciphertext) that is indistinguishable from uniformly random string. Therefore, in particular, there should be no gain (no reduced length) by compressing the ciphertext. On the other hand, compressing the plaintext (before encryption) removes redundancies in it, possibly making cryptoanalysis harder. The correct order is therefore to first compress, then encrypt.

Note, however, that encryption does not hide the length of its input, which in fact is assumed to be constant (for block ciphers) or infinite (for stream ciphers). Therefore, when we encrypt after compression, the design should prevent information leakage from the length of the (compressed) plaintext.

2.8. Exercises

1. Demonstrate that all deterministic block ciphers are vulnerable to known plaintext distribution attack.
2. Show that the simple substitution cipher is unconditionally secure, when applied to a single block of plaintext.
3. Show that the simple substitution cipher is secure against chosen ciphertext attack, for sufficiently large block size and normally distributed plaintext.
4. Alice wants to encrypt communication with Bob and Charlie, but has storage for only one key. She decides to use the same key for both of them, but a different cryptosystem for each. Criticize this design, preferably by presenting a counterexample – two cryptosystems which are secure if used alone, but easily broken if used with the same key. Suggest a simple and secure alternative solution. Hint: consider external adversary (Eve) as well as possible exposure by Charlie of encrypted messages sent between Alice and Bob.
5. Show how to generate a pseudo-random sequence, given a pseudo-random permutation.
6. Let \( \langle E, D \rangle \) be a pseudo-random permutation on \( \{0,1\}^n \). Consider the function \( E'(x) = E_d(x[1..n]) \| E_d(E_d(x[1..n])) \| x[(n+1)..2n] \), namely the CBC mode applied to input of length \( \{0,1\}^{2n} \). Is this \( E' \) a permutation? A pseudo-random permutation?
7. Show that CBC mode, when used with random IV, is not secure against chosen ciphertext attack, for any deterministic block.